ARCoSS

LNCS 8097

Felix Wolf
Bernd Mohr
Dieter an Mey (Eds.)

Euro-Par 2013 Parallel Processing

19th International Conference
Aachen, Germany, August 2013
Proceedings

Springer

Felix Wolf   Bernd Mohr   Dieter an Mey (Eds.)

Euro-Par 2013 Parallel Processing

19th International Conference
Aachen, Germany, August 26-30, 2013
Proceedings

Springer

Volume Editors

Felix Wolf
RWTH Aachen University and
German Research School for Simulation Sciences GmbH
Schinkelstr. 2a, 52062 Aachen, Germany
E-mail: f.wolf@grs-sim.de

Bernd Mohr
Forschungszentrum Jülich GmbH
Jülich Supercomputing Centre
Station 22, 52425 Jülich, Germany
E-mail: b.mohr@fz-juelich.de

Dieter an Mey
RWTH Aachen University
Center for Computing and Communication
Seffenter Weg 23, 52074 Aachen, Germany
E-mail: anmey@rz.rwth-aachen.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40046-9 e-ISBN 978-3-642-40047-6
DOI 10.1007/978-3-642-40047-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013944322

CR Subject Classification (1998): D.1.3, D.3.3-4, C.1.4, D.4, C.4, C.2, G.1.0, C.3, H.3, I.6, I.2.6

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Euro-Par is an annual series of international conferences dedicated to the promotion and advancement of all aspects of parallel and distributed computing. It covers a wide spectrum of topics from algorithms and theory to software technology and hardware-related issues, with application areas ranging from scientific to mobile and cloud computing. Euro-Par provides a forum for the introduction, presentation, and discussion of the latest scientific and technical advances, extending the frontier of both the state of the art and the state of the practice.

The main audience of Euro-Par are the researchers in academic institutions, government laboratories, and industrial organizations. Euro-Par's objective is to be the primary choice of such professionals for the presentation of new results in their specific areas. As a wide-spectrum conference, Euro-Par fosters the synergy of different topics in parallel and distributed computing. Of special interest are applications that demonstrate the effectiveness of the main Euro-Par topics.

In addition, Euro-Par conferences provide a platform for a number of accompanying technical workshops. Thus, smaller and emerging communities can meet and develop more focused topics or as yet less established topics.

Euro-Par 2013 was the 19th conference in the Euro-Par series, and was organized in Aachen, Germany, by the German Research School for Simulation Sciences (GRS), Forschungszentrum Jülich, and RWTH Aachen University in the framework of the Jülich Aachen Research Alliance. Previous Euro-Par conferences took place in Stockholm, Lyon, Passau, Southampton, Toulouse, Munich, Manchester, Paderborn, Klagenfurt, Pisa, Lisbon, Dresden, Rennes, Las Palmas, Delft, Ischia, Bordeaux, and Rhodes. Next year, the conference will be held in Porto, Portugal. More information on the Euro-Par conference series and organization is available on the website at http://www.europar.org.

Euro-Par 2013 covered 16 topics. The paper review process for each topic was managed and supervised by a committee of at least four people: a global chair, a local chair, and two members. Some specific topics with a high number of submissions were managed by a larger committee with more members. The final decisions on the acceptance or rejection of the submitted papers were made at a meeting of the conference co-chairs and local chairs of the topics.

The call for papers attracted 261 full-paper submissions, representing 45 countries. A total of 1,016 review reports were collected, which is an average of 3.9 review reports per paper. The Program Committee members hailed from 22 different countries. We selected 70 papers to be presented at the conference and included in the conference proceedings, representing 26 countries from all continents, and resulting in an acceptance rate of 26.8%.

Euro-Par 2013 was very pleased to present three invited speakers of high international reputation, who discussed important developments in very interesting areas of parallel and distributed computing:

1. Alok Choudhary (Northwestern University, USA)
2. Arndt Bode (Leibniz Supercomputing Centre and Technische Universität München, Germany)
3. Timothy G. Mattson (Intel Corporation, USA)

As part of Euro-Par 2013, three tutorials and 13 workshops were held prior to the main conference. The three tutorials were:

1. Tools for High-Productivity Supercomputing
2. Introduction to OpenACC Programming on GPUs
3. Advanced OpenMP

The 13 workshops were:

1. Big Data Management in Clouds (BigDataCloud)
2. Dependability and Interoperability in Heterogeneous Clouds (DIHC)
3. Federative and Interoperable Cloud Infrastructures (FedICI)
4. Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar)
5. High Performance Bioinformatics and Biomedicine (HiBB)
6. Large-Scale Distributed Virtual Environments on Clouds and P2P (LSDVE)
7. Middleware for HPC and Big Data Systems (MHPC)
8. On-chip Memory Hierarchies and Interconnects (OMHI)
9. Parallel and Distributed Agent Based Simulations (PADABS)
10. Productivity and Performance (PROPER)
11. Resiliency in High-Performance Computing with Clusters, Clouds, and Grids (Resilience)
12. Runtime and Operating Systems for the Many-core Era (ROME)
13. UnConventional High-Performance Computing (UCHPC)

Workshop papers will be published in a separate proceedings volume.

The 19th Euro-Par conference in Aachen would not have been possible without the support of many individuals and organizations. We owe special thanks to the authors of all the submitted papers, the members of the topic committees, and the reviewers in all topics for their contributions to the success of the conference. We would also like to express our gratitude to the members of the Organizing Committee. Moreover, we are indebted to the members of the Euro-Par Steering Committee for their trust, guidance, and support. Finally, a number of institutional and industrial sponsors contributed to the organization of the conference. Their names and logos appear on the Euro-Par 2013 website: http://www.europar2013.org.

It was a pleasure and an honor to organize and host Euro-Par 2013 in Aachen. We hope that all participants enjoyed the technical program and the social events organized during the conference (despite the fact that the conference building did not overlook the beach, as it did last year).

August 2013

Felix Wolf
Dieter an Mey
Bernd Mohr

Organization

Euro-Par Steering Committee

Chair

Christian Lengauer (Chair)	University of Passau, Germany

Vice-Chair

Luc Bougé	ENS Cachan, France

European Representatives

José Cunha	Universidade Nova de Lisboa, Portugal
Marco Danelutto	University of Pisa, Italy
Emmanuel Jeannot	LaBRI-INRIA, Bordeaux, France
Christos Kaklamanis	Computer Technology Institute, Greece
Paul Kelly	Imperial College, UK
Thomas Ludwig	University of Hamburg, Germany
Emilio Luque	Universitat Autònoma de Barcelona, Spain
Tomàs Margalef	Universitat Autònoma de Barcelona, Spain
Wolfgang Nagel	Dresden University of Technology, Germany
Rizos Sakellariou	University of Manchester, UK
Henk Sips	Delft University of Technology, The Netherlands
Domenico Talia	University of Calabria, Italy

Honorary Members

Ron Perrott	Oxford e-Research Centre, UK
Karl Dieter Reinartz	University of Erlangen-Nuremberg, Germany

Observers

Fernando Silva	University of Porto, Portugal
Felix Wolf	GRS and RWTH Aachen University, Germany

Liaison with CCP&E

Luc Moreau	University of Southampton, UK

Euro-Par 2013 Organization

Chair

Felix Wolf	GRS and RWTH Aachen University, Germany

Proceedings

Bernd Mohr	Forschungszentrum Jülich, Germany

Workshops

Dieter an Mey	RWTH Aachen University, Germany

Local Organization

Vera Kleber	GRS, Germany
Elisabeth Altenberger	GRS, Germany
Michaela Bleuel	Jülich Aachen Research Alliance, Germany
Andre Dortmund	GRS, Germany
Beate Pütz	GRS, Germany
Andrea Wiemuth	Jülich Aachen Research Alliance, Germany

Program Committee

Topic 1: Support Tools and Environments

Chair

Bronis R. de Supinski	Lawrence Livermore National Laboratory, USA

Local Chair

Bettina Krammer	MoRitS, Bielefeld University and University of Applied Sciences Bielefeld, Germany

Members

Karl Fürlinger	Ludwig-Maximilians-Universität, Germany
Jesus Labarta	Barcelona Supercomputing Center, Spain
Dimitrios S. Nikolopoulos	Queen's University Belfast, UK

Topic 2: Performance Prediction and Evaluation

Chair

Adolfy Hoisie	Pacific Northwest National Laboratory, USA

Local Chair

Michael Gerndt	Technische Universität München, Germany

Members

Shajulin Benedict	St. Xavier's Catholic College of Engineering, India
Thomas Fahringer	University of Innsbruck, Austria
Vladimir Getov	University of Westminster, UK
Scott Pakin	Los Alamos National Laboratory, USA

Topic 3: Scheduling and Load Balancing

Chair

Zhihui Du	Tsinghua University, China

Local Chair

Ramin Yahyapour	Göttingen University, Germany

Members

Yuxiong He	Microsoft, USA
Nectarios Koziris	National Technical University of Athens, Greece
Bilha Mendelson	IBM Haifa Research Lab, Israel
Veronika Sonigo	Université de Franche-Comté, France
Achim Streit	Karlsruhe Institute of Technology, Germany
Andrei Tchernykh	Center for Scientific Research and Higher Education at Ensenada, Mexico

Topic 4: High-Performance Architectures and Compilers

Chair

Denis Barthou	INRIA, France

Local Chair

Wolfgang Karl	Karlsruhe Institute of Technology, Germany

Members

Ramón Doallo	University of A Coruña, Spain
Evelyn Duesterwald	IBM Research, USA
Sami Yehia	Intel, USA

Topic 5: Parallel and Distributed Data Management

Chair

Maria S. Perez-Hernandez	Universidad Politecnica De Madrid, Spain

Local Chair

André Brinkmann	Johannes Gutenberg University of Mainz, Germany

Members

Stergios Anastasiadis	University of Ioannina, Greece
Sandro Fiore	Euro Mediterranean Center on Climate Change and University of Salento, Italy

Members

Carlos Baquero	INESC TEC and Universidade do Minho, Portugal
Paul Ezhilchelvan	University of Newcastle, UK
Lars Lundberg	Blekinge Institute of Technology, Karlskrona, Sweden

Topic 9: Parallel and Distributed Programming

Chair

José Cunha	Universidade Nova de Lisboa, Portugal

Local Chair

Michael Philippsen	Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Members

Domenico Talia	University of Calabria and ICAR-CNR, Italy
Ana Lucia Varbanescu	Delft University of Technology, The Netherlands

Topic 10: Parallel Numerical Algorithms

Chair

Julien Langou	University of Colorado Denver, USA

Local Chair

Matthias Bolten	University of Wuppertal, Germany

Members

Laura Grigori	INRIA, France
Marian Vajteršic	University of Salzburg, Austria

Topic 11: Multicore and Manycore Programming

Chair

Luiz DeRose	Cray Inc., USA

Local Chair

Jan Treibig	Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Members

David Abramson	Monash University, Australia
Alastair Donaldson	Imperial College London, UK

William Jalby	University of Versailles Saint-Quentin-en-Yvelines, France
Alba Cristina M.A. de Melo	University of Brasilia, Brazil
Tomàs Margalef	Universitat Autňoma de Barcelona, Spain

Topic 12: Theory and Algorithms for Parallel Computation

Chair

Giuseppe F. Italiano	Università di Roma 2 Tor Vergata, Italy

Local Chair

Henning Meyerhenke	Karlsruhe Institute of Technology, KIT, Germany

Members

Guy Blelloch	Carnegie Mellon University, USA
Philippas Tsigas	Chalmers University of Technology, Sweden

Topic 13: High-Performance Networks and Communication

Chair

Olav Lysne	Simula Research Laboratory, Norway

Local Chair

Torsten Hoefler	Swiss Federal Institute of Technology Zurich, Switzerland

Members

Pedro López	Universitat Politècnica de València, Spain
Davide Bertozzi	University of Ferrara, Italy

Topic 14: High-Performance and Scientific Applications

Chair

Turlough P. Downes	Dublin City University, Ireland

Local Chair

Sabine Roller	University of Siegen, Germany

Members

Ari P. Seitsonen	University of Zurich, Switzerland
Sophie Valcke	CERFACS, France

Topic 15: GPU and Accelerator Computing

Chair

Naoya Maruyama	RIKEN Advanced Institute for Computational Science, Japan

Local Chair

Leif Kobbelt	RWTH Aachen University, Germany

Members

Pavan Balaji	Argonne National Laboratory, USA
Nikola Puzovic	Barcelona Supercomputing Center, Spain
Samuel Thibault	University of Bordeaux, France
Kun Zhou	Zhejiang University, China

Topic 16: Extreme-Scale Computing

Chair

David Keyes	King Abdullah University of Science and Technology, Saudi Arabia

Local Chair

Marie-Christine Sawley	Intel Exascale Lab Paris, France

Members

Thomas Schulthess	ETH Zurich, Switzerland
John Shalf	Lawrence Berkeley National Laboratory, USA

Euro-Par 2013 Reviewers

Laeeq Ahmed
Dong Ahn
Sadaf R Alam
Marco Aldinucci
Ferdinando Alessi
Francisco Alfaro
Paulo Sérgio Almeida
Lluc Alvarez
Alba Amato
Flora Amato
Nikos Anastopoulos
Diego Andrade
Cosmin Arad
Francisco Argüello
Athanasia Asiki
Ioannis Assiouras
Thomas Auckenthaler
Olivier Aumage
Rocco Aversa
Mauricio Ayala-Rincon
Luis F. Ayuso
Shahaan Ayyub
Hussein Aziz
Grey Ballard
Mehmet Balman
Daniel Balouek
Kevin Barker
Aritz Barrondo
Elvira Baydal
Olivier Beaumont
Anca Berariu
Tobias Berka

Alberto Ghiribaldi
Roland Glantz
Olivier Gluck
Håkan Grahn
Maria Gomez
Ricardo Gonçalves
Jose Luis Gonzalez Garcia
Dave Goodell
Georgios Goumas
José Gracia
Hakan Grahn
Thomas Grass
Ivan Grasso
Matthias Grawinkel
Philipp Gschwandtner
Ruirui Gu
Abdou Guermouche
Inge Gutheil
Georg Hager
Houssam Haitof
Mahantesh Halappanavar
Moritz Hanke
Laurent Hascoet
Manuel Hasert
Jiahua He
Sylvain Henry
Dora Heras
Pieter Hijma
Adan Hirales Carbajal
Andreas Hollmann
Katja Hose
Tobias Hossfeld
Qiming Hou
Haowei Huang
Andra-Ecaterina Hugo
Mauro Iacono
Shadi Ibrahim
Aleksandar Ilić
Sotiris Ioannidis
Thomas Jablin
Mathias Jacquelin
Julien Jaeger
Matthias Janetschek
Paulo Jesus
Chao Jin
Josep Jorba
Foued Jrad
Peter Kacsuk
Jürgen Kaiser
Vasileios Karakasis
Gabor Kecskemeti
Rainer Keller
Jeroen Ketema
Fahad Â Khalid
Mario Kicherer
Csaba Kiraly
Ioannis Kitsos
Daniel Klan
Harald Klimach
Nicholas Knight
Harald Koestler
Klaus Kofler
Ioannis Konstantinou
Luis Kornblueh
Christos Kotselidis
Kornilios Kourtis
Dorian Krause
Moritz Kreutzer
Jens Krüger
Tomas Kupka
Uwe Küster
Bruno Lang
Josep L. Larriba-Pey
Alessandro Leite
Daniele Lezzi
Changhua Li
Yaohang Li
Loredana Liccardo
Daniel Liew
Lin Lin
Li Liu
Björn Lohrmann
Nuno Lopes
Francesc-Josep Lordan Gomis
João Lourenço
Drazen Lucanin
Emilio Luque
Spyros Lyberis
Eamonn Maguire
Michael Maier

Tareq Malas
Fredrik Manne
Loris Marchal
Fabrizio Marozzo
Stefano Marrone
Claudio Martella
Patrick Martineau
José Martínez
Ficco Massimo
Toni Mastelic
Oliver Mattes
Matthias Meinke
Dirk Meister
Bunjamin Memishi
Robert Menzel
Andre Merzky
Marcel Meyer
Ulrich Meyer
Jean-Christophe Mignot
Jose Miguel-Alonso
Robert Mijaković
Ugljesa Milic
Hamid Mohammadi Fard
Kathryn Mohror
Jesús Montes
Adam Moody
Andreu Moreno
Graham Morgan
Francesco Moscato
Juan Carlos Moure
Vlad Nae
Lars Nagel
Anastassios Nanos
Tobias Neckel
Christian Neuhaus
Dang Nguyen
Hoang Nguyen
Jean-Marc Nicod
Bogdan Nicolae
Konstantinos Nikas
Ramon Nou
Fabian Nowak
Ana Nunes
Ryan Okuinghttons
Pablo Oliveira
Róbert Ormándi
Per-Olov Östberg
Simon Ostermann
Christian Pagé
Cosimo Palazzo
Francesco Palmieri
Jia Pan
Antonis Papaioannou
Hervé Paulino
Kevin Pedretti
Susanna Pelagatti
Antonio J. Peña
Chao Peng
Chengbin Peng
Manuel Perez Malumbres
Eric Petit
Ventsislav Petkov
Andreas Petlund
Judit Planas
Kassian Plankensteiner
Dirk Pleiter
Andrew Porter
Louis-Noël Pouchet
Suraj Prabhakaran
Polyvios Pratikakis
Bertrand Putigny
Ariel Quezada Pina
Rejitha R.S.
Roger Rafanell
M. Mustafa Rafique
Mustafa Rafique
Massimiliano Rak
Luca Ramini
Sven-Arne Reinemo
Zhong Ren
Michael Rezny
Liz Ribe-Baumann
Pedro Ribeiro
Alejandro Rico
Graham Riley
Hendirk Rittich
Francisco Rocha
Keith Rochford
Rodrigo Rodrigues
Casiano Rodriguez-Leon

Table of Contents

Topic 3: Scheduling and Load Balancing

Energy to Solution: A New Mission for Parallel Computing

Arndt Bode

Leibniz-Supercomputing Centre, Garching bei München
and
Lehrstuhl für Rechnerorganisation und Rechnertechnik
Technische Universität München
Germany
bode@lrz.de

Abstract. For a long period in the development of computers and computing efficient applications were only characterized by computational – and memory complexity or in more practical terms elapsed computing time and required main memory capacity. The history of Euro-Par and its predecessor-organizations stands for research on the development of ever more powerful computer architectures that shorten the compute time both by faster clocking and by parallel execution as well as the development of algorithms that can exhibit these parallel architectural features. The success of enhancing architectures and algorithms is best described by exponential curves regarding the peak computing power of architectures and the efficiency of algorithms. As microprocessor parts get more and more power hungry and electricity gets more and more expensive, "energy to solution" is a new optimization criterion for large applications. This calls for energy aware solutions.

Components of Energy Aware Computing

In order to reduce the power used to run an application, four components have to be optimized, three of them relate to the computer system and the programs to be extended, one relates to the infrastructure of the computer system:

- **Energy aware infrastructure:** This parameter relates to the fact, that computers need climate, cooling, uninterruptable voltage supply, building with light, heating and additional infrastructure components that consume power. Examples for measures to reduce energy are: Use of liquid cooling, direct cooling, free cooling, waste heat reuse, adsorption machines, monitoring and optimizing of energy consumption and infrastructure control, coupling of infrastructure power requirements with behavior of computers and the application execution.
- **Energy aware system hardware:** This parameter describes all mechanisms in new hardware to reduce power in the system itself: sleep modes of inactive parts, clock control of the parts, fine grain hardware-monitoring of

system consumption, any hardware that relates to accumulating, processing and using consumption data for power reduction in the part and its relation to other parts of the system, autonomous optimizing and learning behavior.
- **Energy aware system software:** This parameter describes all sorts of automatic, semi-automatic, or online and offline user controlled tools, that do monitor, analyze and control the execution of application software on the system: support for optimal pinings of threads, finding the optimal clock based on previous runs and data on execution behavior looking at relations between processing, memory, interconnection and storage/I/0 boundedness.
- **Energy aware algorithms:** In most cases, the fastest algorithm consumes the minimum power. But there are exceptions, if algorithms use redundancy (sometimes in applications with super linear speedup). This is a very broad research area in its own.

Experiences with SuperMUC

Energy-aware HPC solutions are tried out at Leibniz Supercomputing Center with SuperMUC a system with 160.000 cores of XEON (IBM iDataPlex). Energy consumption is measured and controlled at many different levels starting at the course grain level with the overall infrastructure control down to very fine grain tools on the individual chip level. Some of the tools are under user control, other were controlled by the datacenter management team, some are even fully automatic. Clocking control is offered by LRZ to the general user of SuperMUC as a tool that supports measuring in detail the execution of a program at a first run, puts data into a database and uses this for subsequent runs to optimize the clocking of SuperMUC.

These measures implemented so far present a first step and will be enhanced in the future. Further development and research is needed to couple and optimize the effects of the various tools. We also advocate for better energy awareness indicators (such as PUE) that do measure the total consumption including the entire infrastructure and take into account that the percentage of peak performance used in highly parallel applications is rather poor and varies with the system architecture, in order to allow for fair evaluations and comparison.

If we want to afford the electricity for an Exascale system, energy awareness of computing has to be approved by orders of magnitude. The methods and tools that have to be developed to do so, present an interesting new field of research for the Euro-Par community.

Topic 1: Support Tools and Environments (Introduction)

Bronis R. de Supinski, Bettina Krammer,
Karl Fürlinger, Jesus Labarta, and Dimitrios S. Nikolopoulos

Topic Committee

Despite an impressive body of research, parallel and distributed computing remains a complex task prone to subtle software issues that can affect both the correctness and the performance of the computation. This track focuses on tools and techniques to tackle that complexity. Submissions covered a wide range of challenges of parallel and distributed computing, including, but not limited to, scalability, programmability, portability, correctness, reliability, performance and energy efficiency. This topic brings together tool designers, developers, and users to share their concerns, ideas, solutions, and products for a wide range of platforms.

We received numerous submissions on these important topics. The submissions were subjected to a rigorous review process, drawing on experts from across parallel computing to assess their novelty, correctness and importance. Through this process we selected two papers for publication.

One of the accepted papers provides a novel mechanism to detect synchronization in shared memory programs. The technique controls thread execution in order to determine when a specific thread waits for the actions of another, specific thread. This technique can simplify automatic analysis of the correctness of shared memory programs.

The other accepted paper details advances in system simulation. The paper specifically discusses simulation of the BlueGene/Q processor. The paper presents techniques to reduce the time per simulated instruction while still providing an accurate timing. model. Overall, these techniques support evaluation of system software and application performance prior to fabrication of the actual processor chips.

We thank the authors of these papers and all submissions. In addition, we thank our many reviewers, all of who provided detailed evaluations of the submissions that ensured that the accpeted papers in this topic are of the highest quality.

Synchronization Identification through On-the-Fly Test

Xiang Yuan[1,2], Zhenjiang Wang[1], Chenggang Wu[1,⋆], Pen-Chung Yew[3,4], Wenwen Wang[1,2], Jianjun Li[1], and Di Xu[5,⋆⋆]

[1] SKL of Computer Architecture, Institute of Computing Technology, CAS
[2] University of Chinese Academy of Sciences Beijing, China
{yuanxiang,wangzhenjiang,wucg,wangwenwen,lijianjun}@ict.ac.cn
[3] Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minneapolis, USA
yew@cs.umn.edu
[4] Institute of Information Science, Academia Sinica, Taiwan
[5] IBM Research - China
xudi@cn.ibm.com

Abstract. Identifying synchronizations could significantly improve testing and debugging of multithreaded programs because it could substantially cut down the number of possible interleavings in those tests. There are two general techniques to implement synchronizations: modularized and ad-hoc. Identifying synchronizations in multi-threaded programs could be quite challenging. It is because modularized synchronizations are often implemented in an obscure and implicit way, and ad-hoc synchronizations could be quite subtle. In this paper, we try to identify synchronizations from a new perspective. We found that if a thread is waiting for synchronizations, the code it executes during the wait is very different from that after the completion of the synchronization. Based on such an observation, we proposed an effective method to identify synchronizations. It doesn't depend on the understanding of source codes or the knowledge of semantics of library routines. A system called SyncTester is developed, and experiments show that SyncTester is effective and useful.

Keywords: synchronization identification, concurrency testing, multithreading.

1 Instruction

Many debugging and testing algorithms use guided or unguided interleaving among threads as the basis of such tests, e.g. Eraser [1], FastTrack [3] and CTrigger [4]. However, the number of interleaves in multi-threaded programs could be enormous. It will make those algorithms very complicated and time consuming. To reduce the number of possible interleavings, and thus reducing the time complexity, these algorithms try to exploit synchronizations. It can also reduce

⋆ To whom correspondence should be addressed.
⋆⋆ This work was done when Di Xu attended Institute of Computing Sciences, CAS.

false positives in the test results. Previous work [12] shows that synchronizations could reduce 43-86% false data races found by Valgrind. Therefore, identifying synchronizations in multithreaded programs is very desirable and important.

In general, there are 2 types of synchronizations: modularized [12] and ad-hoc. Modularized synchronization could be identified by their semantics [5,3,7]. And most approaches identify ad-hoc ones by pattern-matching [7,8,12].

Identifying modularized synchronizations using their semantics could be tedious and error prone. There exist hundreds of libraries. Some of them have many routines. E.g. GLIBC 2.16 has about 1200 routines [14]. Some routines provide implicit synchronization functions (e.g. read()/write()). Moreover, some synchronization pairs could be from different libraries (e.g. pthread_kill/sigwait). Therefore, this work could be quite challenging for programmers.

To identify ad-hoc synchronizations, ISSTA08 [7] focuses on synchronizations that consist of a spinning read and a corresponding remote store. It is dynamic. Helgrind+ [8] and SyncFinder [12] focus on loops whose exit condition cannot be satisfied in their loop bodies. SyncFinder is a static method, while Helgrind+ searches for such loops statically, but identifies the remote stores dynamically. They are all based on the synchronization patterns summarized from various application codes. Because of the complexity and the large amount of codes, this process is time consuming and may miss or misjudge some patterns. To identify complex patterns, elaborate inter-procedural pointer analysis may be needed. However, there are still many cases pointer analysis cannot handle.

In this paper, we try to identify synchronizations from a different perspective. We leverage the essential feature of a synchronization that it forces a thread to wait when the thread may violate the intended order imposed by the programmer. Our scheme depends on neither the patterns nor the knowledge of library routines, thus labor-intensive pattern collection/recognition and learning of library routines are avoided. Moreover, our proposed scheme works on binary executable, which is an advantage when source code is not available. Overall, our work makes the following contributions:

(1) Propose a synchronization identification scheme from a new perspective. It can be used to identify both modularized and ad-hoc synchronizations in multi-threaded programs regardless of their implementation.

(2) Implement a prototype system, SyncTester. Experimental results on real programs show that it is very effective and useful.

(3) Introduce helper processes to do contrast test. With their help, SyncTester can avoid perturbation to the program execution.

2 Synchronization Testing

Our scheme tries to identify synchronization by monitoring the execution of a program. A multi-threaded program is tested for several executions. In each execution, we select a different thread as the *testing thread* and all other are *executing threads*. We propose two algorithms: a *forward test* and a *backward test*. They identify synchronizations according to the intended order between the testing thread and the executing threads. Fig.1 gives an overview of it.

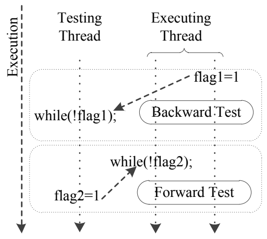


Fig. 1. Test Scheme Overview

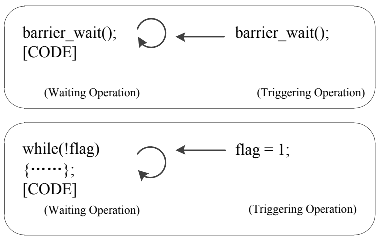


Fig. 2. Behavior of waiting operations

2.1 Forward Test

Synchronizations are used to control the orders among operations in different threads. A synchronization involves at least two threads: one is waiting (*waiting thread*) for a certain operation from another (*triggering thread*). We refer to the waiting action in the waiting thread as *waiting operation*, and the operation that wakes up the waiting thread as *triggering operation.* A waiting operation and its corresponding triggering operation form a *synchronization pair (sync pair).*

Waiting operations have two forms, as shown in Fig.2. The waiting thread may be blocked by a library call or spinning on a code segment. The triggering operation should be a library call or a shared memory store instruction respectively. So we take all the library calls and shared memory store instructions as *potential triggering operations* (PTO). When a thread is waiting for a certain operation, we say that it is in a *suspended state*, and such a thread is executing a *potential waiting operation* (PWO). When the corresponding triggering operation occurs, waiting thread will exit the suspended state and go on to execute its subsequent codes, as [CODE] shown in Fig.2. The code it executes during a suspended state is different from that after it exits this state.

Then we design a sync-pair identification scheme. Algorithm 1 shows its details. When testing thread encounters a PTO, we suspend its execution until all executing threads enter suspended states (Line 14-16). Then we execute the PTO. If it makes an executing thread exit a suspended state, we probably find a sync pair (Line 23-29).The italic light-grey codes are the specific techniques to improve the scheme.

This algorithm is “safe” in the sense that we allow only one testing thread in each test. Hence, if all the PTOs in the testing thread are not real triggering operations, executing threads will continue running to their completion without being suspended. If none of the suspended executing threads exits its suspended state after a PTO is executed, the testing thread will continue executing until a real triggering operation releases an executing thread.

It is important to identify whether a thread is in a suspended state or not. A basic block vector (BBV) [9] records the executing frequencies of all basic blocks in a time quantum. Every thread builds its own BBVs during execution. When a thread is in a suspended state, the BBVs of its continuous quanta should be very similar. When a threads BBVs in continuous quanta have small differences [9] and contain same blocks, this thread is marked as in a suspended state.

Algorithm 1 Forward Test

```
 1: Input: Set<Thread> Threads, including all threads in a multi-
    threaded program
 2: Map<Thread, PWO> pwOp := Φ;
 3: Map<Thread,Process> helpProc := Φ;
 4: Set<Thread> testingThreads = Threads;
 5: Set<Thread> execThreads;
 6: //Algorithm begins
 7: testingThreads = ThreadGroup(testingThreads);
 8: for testThread∈testingThreads do
 9:   //Each iteration of this for loop is a pass of execution
10:   while testThread is not finished do
11:     execThreads:= Threads \ {testThread}
12:     //execThreads are executing freely.
13:     nextOp := NextPTO(testThread);
14:     while not AllThreadSuspended(execThreads) do
15:       sched_yield();
16:     end while
17:     if not RepeatOp(nextOp) then
18:       execThreads:= TypeMatching(execThreads, nextOp);
19:       for t∈ execThreads do
20:         pwOp[t] := t's current PWO;
21:         helpProc[t] := CreateHelperProc(t);
22:       end for
23:       Execute(testThread, nextOp);
24:       for t∈ execThreads do
25:         if not Suspended(t) && Suspended(helpProc[t]) then
26:             RecordSyncPair(nextOp, pwOp[t]);
27:         end if
28:         Terminate(helpProc[t]);
29:       end for
30:     else
31:       Execute(testThread, nextOp);
32:     end if
33:   end while
34: end for
```

Algorithm 2 Backward Test

```
 1: Input: Set<Thread> Threads, including all threads in a multi-
    threaded program
         Set<SyncPair> SyncPairs, including sync pairs
    found by forward test
 2: Set<Thread> targetThreads = Threads;
 3: Set<Thread> execThreads;
 4: targetThreads = ThreadGroup(targetThreads);
 5: for testingThread∈targetThreads Do
 6:   //Each iteration of this for loop is a pass of execution
 7:   execThreads := Threads \ {testingThread}
 8:   //execThreads are executing freely
 9:   while testingThread is not finished do
10:     nextOp := NextReadOp(testingThread);
11:     if not AllThreadSuspended(execThreads) then
12:       WaitForSuspending(execThreads);
13:     end if
14:     wrOp = LastWriteOp(nextOp);
15:     if MatchHappenBefore(nextOp, wrOp, SyncPairs) &&
    not RepeatedOp(nextOp, wrOp) && not LockOp(nextOp,
    wrOp) && wrOp is not in testingThread then
16:       CreateHelperProcBT(nextOp, wrOp);
17:     end if
18:     Execute(testingThread, nextOp);
19:   end while
20: end for
21: //Define <rdOp, wrOp> as a Backward Test Pair
22: CreateHelperProcBT(OP rdOp, OP wrOp) begin
23:   Fork(); //Create Helper Process
24:   if this is helper process then
25:     //The following codes execute concurrently with main
    process
26:     RestoreToPreviousValue(this, wrOp);
37:     if Suspended(this) then
28:       helperProc2 = CreateHelperProcess(this);
29:       ResetToCurrentValue(helperProc2, wrOp);
30:       if Suspended(this) != Suspended(helperProc2) then
31:         RecordSyncPair(wrOp, rdOp);
32:       endif
33:     end if
34:     Terminate(this, helperProc2);//Kill helper processes
35:   end if
36: end
```

In the following cases, the differences of a threads continuous BBVs will be small. The thread is (1) blocked by a system call, (2) has exited or (3) spinning on a code segment. In the first two cases, the thread is no longer executing, and will not build BBVs, and all the elements of its BBVs are 0.The differences of its BBVs are 0.

When a thread exits its suspended state, it starts to execute other code segments. So, if we find a thread executes blocks different from those in the suspended state, it has exited its previous suspended state.

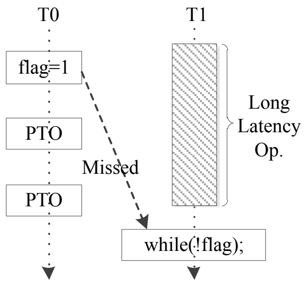


Fig. 3. Motivation of Backward test

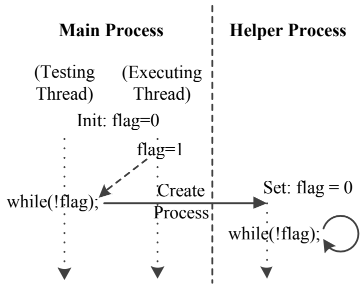


Fig. 4. Schematic of Backward Test

2.2 Backward Test

Algorithm 1 can identify many sync pairs. However, in some special cases, it may miss some sync pairs. For example, in Fig.3, "flag=1" and "while(!flag)" form a sync pair. Assume there is a *long latency operation (LLO)* in T1. We refer to an operation as a LLO if it lasts for several time quanta and can continue to execute without being triggered by other threads. When T1 is executing a LLO, Algorithm 1 may mistake T1 as entering a suspended state and miss this sync pair. To counter this difficulty, we propose a *backward test.*

Note that the *forward test* suspends the testing thread at each potential triggering operation. It allows executing threads to run ahead of the testing thread and enter suspended states. However, LLOs can distort it. The proposed *backward test* targets sync pairs whose triggering operations execute before the waiting operations. The sync pair shown in Fig.3 can be identified if we use a *backward test* and select T1 as the testing thread.

If triggering operation executes before waiting operation, the waiting operation will not spin or be blocked. This is because the triggering operation has removed the wait condition. Therefore, when a *shared variable* is read by the main program or library, we restore the shared variable to its *previous value.* If this thread is blocked or spinning on a shared variable, we treat the corresponding library routine (or code segment) and the operation that performs the previous store to the shared variable as a sync pair.

However, restoring a shared variable to its previous value will cause errors to the programs execution. Therefore, we perform such a test in a separate process, called *helper process.* Helper processes are created (by *fork()* syscall in Linux) when we encounter read operations to shared variables or library calls. They communicate with the main process via pipes. Because they need to inherit the current context of the main process, they can't be created in advance.

This is called a *backward test.* Fig.4 is its schematic, and Algorithm 2 is its details. Note that Algorithm 2 defines the backward test pair in Line 21.

Helper Process Isolation. Helper processes inherit the context of the main process. They share some resources with the main process, such as file descriptors and shared memory. If they access these resources, they will interfere with the execution of main process. Hence we need to handle system calls and shared memory in helper processes appropriately.

We classify system calls into 3 types and handle them in helper processes differently. (Type 1) System calls that don't access such shared resources, invoke them as usual, e.g. *gettimeofday()*, *futex()*; (Type 2) System calls that get data from shared resources only, e.g. *read()*, get these data from the main process. (Type 3) System calls that store data or change status of shared resources, ignore them and get their return values from main process, e.g. *open()*, *write()*.

Processes may use shared memory to communicate with other processes. We transform shared memory in the helper process into private. We back up the data in shared memory, detach it and then map private one. The range of shared memory can be got by system call instrumentation in main process.

2.3 Perturbation Elimination

Long Latency Operations (LLOs). Take the forward testing as an example. When an executing thread enters a LLO, our scheme may misidentify the executing thread as entering a suspended state. When the testing thread executes a potential triggering operation, if this executing thread happens to exit its misidentified suspended state, the forward test will mistakenly identify the LLO as a waiting operation. A false sync pair will be identified, i.e. a *false positive.*

The key to distinguish a LLO from a *true waiting operation* is the manner it exits the suspended state. A LLO can exit the suspended state by itself after finishing its work. But a waiting operation cannot exit the suspended state until another thread executes its corresponding triggering operation.

Fig.5 shows how to distinguish LLOs in forward test (Line 21 & 25 Algorithm 1). When testing thread (T0) encounters a potential triggering operation (flag=1;), a helper process is created by the executing thread (T1). Note that the helper process does not interact with the testing thread (i.e. 'flag' is always 0 in the helper process). After the testing thread has executed "flag=1;", we compare the state of T1 and helper process. If T1 and the helper process both exits the suspended state (Fig.5(b)), it means that the operation that made T1 enter a suspended state is a LLO. Otherwise we find a real sync pair (Fig.5(a)).

In the backward testing (Line 28-30 Algorithm 2), we take a similar approach to handle LLOs. When the helper process enters a suspended state, we create a second helper process using the first helper processs context. In the second helper process, we reset the value of the shared variables restored by the first helper process to the same as that in main process. If one helper process exits the suspended state while the other dose not, it means that there is a helper process waiting for the new values of these shared variables. We could treat this backward test pair as a sync pair.

Lock/Unlock Operations. When the helper process tests a lock operation in backward test, it restores the values of the lock variables to those before an unlock operation. These values make the helper process have held the lock and will be blocked by this lock operation. In order to avoid identifying lock/unlock operations as sync pairs, when we encounter a library routine at the first time, we create a helper process to test whether it is a lock operation or not. This helper process creates two threads and each of them invokes this library routine with the same arguments. From the definitions of ABI, we will know where the arguments are (e.g. register or stack). If one of these two threads is blocked and the other is not, this library routine is treated as a lock operation and will not form sync pairs. (LockOp() in Line 15 Algo.2)

2.4 Efficiency Issues

Efficiency is always a concern in testing multi-threaded programs. There are two ways to improve it. One is to reduce the number of testing threads, and the other is to reduce the number of testing points in testing threads. Testing points include potential triggering operations and backward test pairs.

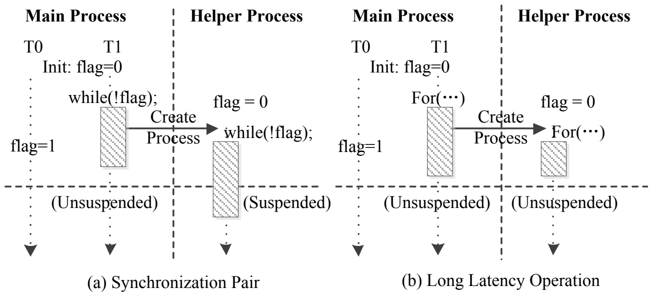


Fig. 5. Handling LLOs in forward test

Reducing Testing Threads. To reduce the number of testing threads, we divide the threads in a program into groups. If two threads are similar in their executions, the sync pairs they executed are likely to be the same. Hence, we put such two threads in a group. In each group, we select only one thread as testing thread (ThreadGroup() in Line 7 Algo.1 and Line 4 Algo.2).

In order to group threads, we calculate the difference between the BBVs of their executions. If the difference is smaller than a threshold, the executions of the two threads are likely to be similar. They are put into the same group.

This is heuristic. To reduce the probability of missing sync pairs due to thread grouping, we build a sync-op set. It contains triggering and waiting operations of sync pairs we found and their calling contexts. After all testing threads are tested, if we find that a thread executes an operation in the *sync-op* set with a different calling context and this operation doesn't belong to a sync pair, we select this thread as testing thread and do another pass of test.

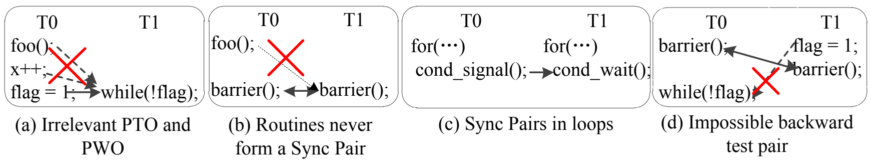


Fig. 6. Cases to reduce number of testing points

Reducing Testing Point. Fig.6 shows some cases, in which we can reduce the number of testing points. We propose the following schemes targeting them.

(1)Match the type/address of synchronization operations. (Type-Matching() in Line 18 Algo.1 and LastWriteOp() in Line 14 Algo.2)

In Fig.6(a), the waiting operation and triggering operation of a modularized sync pairs should be library routines, while an ad-hoc sync pair should consist of a spin loop and a store operation. So, when we encounter a library call, we can skip testing store operations to shared variable, and vice versa.

Furthermore, for an ad-hoc sync pair in the forward test, its waiting operation and triggering operation should access the same shared variable. According to the shared variables read by executing threads during a suspended state, we can skip store operations that access other shared variables in the testing thread.

(2)Use history of routines. (RepeatOp() in L.17 Algo.1 & L.15 Algo.2)

After analyzing various popular library routines, we found that a library routine with blocking function can only be unblocked by a few specific library routines. If a pair of library routines is tested for several times and they never form a sync pair, as shown in Fig.6(b), it is most likely that they are not a sync pair at all. So, when we encounter the same library pair, skip the test on them.

(3)Accelerate test in loops. (RepeatOp() in L.17 Algo.1 & L.15 Algo.2)

There exist sync pairs that are in loops as shown in Fig.6(c). Such sync pairs appear in every iteration and we need not to test them every time.

If a sync pair is found repeatedly, we assume it is in a loop. In forward test, we record the potential triggering operations (excluding its triggering operation) appear between two occurrences of its waiting operation, and skip them when the waiting operation appears again. In the backward test, we skip testing it.

(4)Use results of the forward test. (MatchHappenBefore() in L.15 Algo.2)

Sync pairs define happens-before relations between code segments. We can utilize sync pairs identified in a forward test to reduce testing points in a backward test. The happen-before relation defined by barrier() in Fig.6(d) ensures that the store (flag=1;) in a backward test pair executes before its corresponding read (while(!flag);). Then the read can't get the old value before the store and we don't need to test such cases in a backward test.

3 Evaluation

SyncTester is implemented using Pin [16]. It uses Pins API to instrument instructions that may access shared variables, code blocks, library routines and system calls. It then collects information and controls the execution of target programs. For example, SyncTester instruments store instructions and syscalls. It then records a shared variables previous value for the backward test.

We evaluated SyncTester on a series of multi-threaded programs. The test programs are from benchmark suites, such as SPLASH2 [10] and STAMP [11], or applications, such as PBZIP2, PFSCAN, and Apache Httpd. We con-figured SPLASH2 using POSIX thread library and configured STAMP with a simple software TM provided by its web site. Our experiments are run on a server with two 2.27GHz Intel Xeon E5520 quad-core processors and 8GB DRAM.

3.1 Effectiveness

In order to evaluate SyncTesters effectiveness, we compare it with two existing dynamic test schemes, ISSTA08 and Helgrind+. We implement ISSTA08's algorithm [7] and use Helgrind+'s newest version. These two approaches are designed to identify ad-hoc sync pairs. So, we compare them only for ad-hoc sync pairs. The results show that SyncTester found more sync pairs than ISSTA08 and Helgrind+ did, and it introduced very few false positives as shown in Table 1. In fact, SyncTest covered all sync pairs found by ISSTA08 and Helgrind+.

Table 1. 'All' column shows the number of identified sync pairs. 'FP' column shows the number of false positives and we verify them manually. The results of SyncTester are in the form of X(modularized)/Y(ad-hoc). FT and BT columns show the results of forward test and backward test, respectively.

Benchmarks	ISSTA08		Helgrind+		SyncTester				Pruned LLOs		Pruned Lock/ Unlock
	All	*FP*	*All*	*FP*	*All*	*FP*	*FT*	*BT*	*FT*	*BT*	
BARNES	1	0	2	2	5/2	3	5/2	5/1	4	235	4
FMM	4	0	8	2	10/8	1	10/4	10/5	12	197	23
OCEAN-C	0	0	0	15	20/0	0	20/0	20/0	0	0	6
OCEAN-NC	0	0	0	40	19/0	0	19/0	19/0	0	0	6
RADIOSITY	0	0	0	7	3/2	0	3/2	3/0	3	11	10
RAYTRACE	0	0	0	9	1/0	1	1/0	1/0	0	0	3
VOLREND	2	0	2	5	5/2	0	5/2	5/1	5	16	5
WATER-S	0	0	0	15	7/0	0	7/0	7/0	0	0	7
WATER-N	0	0	0	5	9/0	0	9/0	9/0	0	0	9
INTRUDER	1	1	2	5	3/2	0	3/0	0/2	3	0	3
LABYRINTH	0	0	1	3	2/1	0	2/0	0/1	2	0	3
PFSCAN	0	0	1	3	3/2	0	3/0	0/2	0	0	4
PBZIP2	0	0	0	0	5/5	0	5/0	0/5	17	0	13
Apache Httpd	0	0	*	*	1/7	1	1/2	0/5	0	0	21

* Maybe the version of Apache HTTPD is not fit for Helgrind+, Helgrind+ exits unexpectedly during our test to Apache Httpd. So we don't get such data.

FPs in Helgrind+ are probably because it searches for spinning loops on binary codes statically. And it is not accurate enough. For SyncTester, backward test restores a shared variable's value and don't know the relationship among different shared variables. This inconsistency brings FPs in some regular loops. And no FP is found in forward test.

The last 3 columns of Table 1 shows the results of perturbation elimination. In those experiments, we found that LLOs include some long loops and library routines. We can prune both of them no matter how they are implemented.

However, because we can't restore the states of system kernel, the backward test may miss some modularized sync pairs due to system calls, e.g. pthread_kill() and sigwait(). Such cases can be found by the forward test.

3.2 Efficiency Issues

Reducing the Number of Testing Threads. We set a threshold on the difference of BBVs in thread grouping. Different thresholds will result in different thread groupings, as shown in Table 2. "Real Thread Groups" column shows the best groupings found manually. It gives the minimum number of testing threads. We check the results in Table 2 and find that if the number of thread group is no fewer than this amount, the testing threads selected will contain all threads in the "Real Thread Groups". This happens to be the most prevalent case. It shows that our thread grouping scheme is quite effective. Finally, we choose 0.4 as the threshold. It is chosed as a tradeoff between accuracy and efficiency. In this case, labyrinth misses a sync pair, but it is found in the sync-op set.

The last 4 columns of Table 2 shows the results of thread grouping when the number of worker threads changes. For programs with many threads, it is likely that most threads execute similar codes, and this scheme will also reduce the number of testing threads. If we test a program with different testing threads, these tests can run concurrently. This will also save testing time.

Reducing the number of Testing Points. Table 3 is the results of testing point reduction. It shows that there are not many testing points in most programs. It also shows the number of testing points pruned by optimizations. For most benchmarks, more than 97% testing points can be pruned. If we have to test all of them, the efficiency of SyncTester will become quite unacceptable. In the backward test, we perform test in helper processes. Because helper processes can run concurrently, this can reduce the time overhead caused by them.

Table 2. Results of Thread Grouping

Benchmarks	**#ths**	**# thread groups under different threshold**						**Real Thread Groups**	**#total threads**	**# worker threads (threshold = 0.4)**		
		0.1	*0.2*	*0.3*	*0.4*	*0.5*	*1*			4	8	16
BARNES	4	1	1	1	**1**	1	1	1	N	1	1	1
FMM	4	4	4	2	**2**	2	1	1	N	2	6	4
OCEAN-C	4	2	1	1	**1**	1	1	1	N	1	1	1
OCEAN-NC	4	1	1	1	**1**	1	1	1	N	1	1	1
RADIOSITY	4	4	3	1	**1**	1	1	1	N	1	2	3
RAYTRACE	4	4	2	2	**1**	1	1	1	N	1	2	4
VOLREND	4	3	2	2	**2**	2	1	1	N	2	2	4
WATER-S	4	2	1	1	**1**	1	1	1	N	1	1	1
WATER-N	4	2	2	2	**1**	1	1	1	N	1	1	2
INTRUDER	4	3	3	2	**2**	1	1	2	N	2	3	3
LABYRINTH	4	1	1	1	**1**	1	1	2	N	1	1	1
PFSCAN	5	4	4	4	**4**	4	4	2	1+N	4	4	6
PBZIP2	8	6	6	6	**6**	6	5	5	4+N	6	5	5
Apache HTTPD	7	4	4	4	**4**	4	4	4	3+N	4	4	4

Table 3. Testing Points Reduction. Columns 2 and 3 are the real testing points during tests. Columns 4 to 7 are testing points pruned by our 4 schemes.

Benchmarks	**Testing Points**		**Pruned Testing Points**				**% Pruned**
	FT	***BT***	***Type Match***	***RTN History***	***Loop Accel***	***Use FTs Results***	
BARNES	36	412	160K	27K	153K	3.8M	99.9892%
FMM	63	463	39K	2363	53K	125K	99.7608%
OCEAN-C	105	250	360K	1798	1186	36K	99.9110%
OCEAN-NC	146	251	364K	808	1157	53K	99.9055%
RADIOSITY	23	74	25	47	31	0	51.5000%
RAYTRACE	2	4	102K	514	0	0	99.9941%
VOLREND	41	33	25K	34	39	26K	99.8557%
WATER-S	111	21	230K	352K	0	2.26M	99.9980%
WATER-N	48	75	70K	1828	14K	3.61M	99.9975%
INTRUDER	60	6	87K	158K	0	379	99.9731%
LABYRINTH	156	15	22K	461	0	4735	99.3833%
PFSCAN	53	11	41	15K	0	0	98.2511%
PBZIP2	151	38	7062	51	0	0	97.4117%
Apache HTTPD	90	224	1566	385	0	0	86.1369%

3.3 Running Time

Finally, we measured the running time of SyncTester, which is shown in Fig.7. Among these benchmarks, Apache HTTPD is a server. It is not easy to measure its running time. So we instead measure its throughput, i.e. the number of processed requests per second. On average, the slowdown factor is 32X.

ISSTA08 claims that its slowdown factor is 9X [7]. Although SyncTester is slower, it identifies more sync pairs. And SyncTester can identify ad-hoc sync pairs whose triggering operations execute before waiting operations and modularized sync pairs.Helgrind+ is a race detector. It identifies ad-hoc sync pairs to prune false races. For SPLASH2 benchmarks, its slowdown factor is more than 2000X. However, we don't compare it with our results because its main function is to detect data races, not sync pairs.

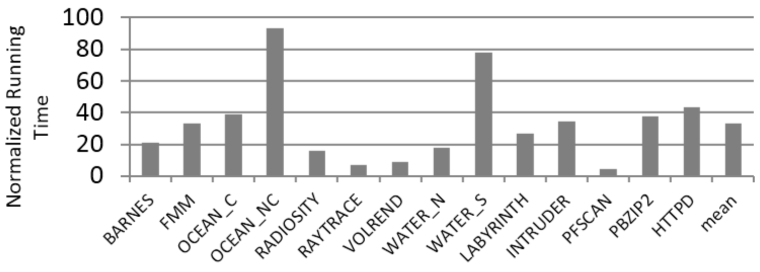


Fig. 7. Running Time of SyncTester

4 Related Work

ISSTA08[7] treats spinning reads and remote stores as a common pattern of ad-hoc synchronizations. However, if a remote store is executed before a spinning read, the spinning read will execute only once and it will miss such synchronizations. Helgrind+[8] tries to overcome such weaknesses of ISSTA08[7]. It searches for spinning loops whose exit conditions depend on loop invariant variables in the binary code and remote stores on the fly. However, in some complex cases, it may be difficulty to find spinning loops in binary codes. There is also a hardware scheme [15]. It uses some hardware buffers and detects spinning loops on the fly. SyncFinder [12] searches for loops with loop-invariant exit conditions. It is a static approach. All of its analysis is done on source code. It uses constant propagation to identify remote stores. Because the source codes are not always available, and pointer analysis is often not very precise, it may introduce some false positives and false negatives.

ATDetector [13] finds that address transfer (i.e. passing memory blocks' address between threads) also imposes implicit happens-before relation and could prune false races. Address transfer ensures that accesses to the memory block in the address sending thread happen before accesses in receiving thread.

5 Conclusion

In this paper, we showned that if a thread was held up in a synchronization by another thread, the code it executes during the wait is very different from that after the completion of the synchronization. From this observation, we propose a new approach to identify sync pairs in multi-threaded programs, called SyncTester. SyncTester can identify both modularized and ad-hoc sync pairs. It doesn't depend on the details of their codes and software implementation. Therefore, it has a great flexibility and is often more accurate than many existing approaches. Experimental results show that SyncTester is quite effective and practical.

Acknowledgements. This research is supported by the National High Technology Research and Development Program of China under the grant 2012AA010901, the National Natural Science Foundation of China (NSFC) under the grant No.61100011, the Innovation Research Group of NSFC under the grant 60921002, and the U.S. National Science Foundation under the grant CNS-0834599.

References

1. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. J. ACM TOCS 15(4), 391–411 (1997)
2. Zhang, W., Sun, C., Lu, S.: ConMem: detecting severe concurrency bugs through an effect-oriented approach. In: Proc. of 15th ASPLOS, pp. 179–192. ACM, NY (2010)
3. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: Proceedings of the PLDI, pp. 121–133. ACM, NY (2009)
4. Park, S., Lu, S., Zhou, Y.: CTrigger: exposing atomicity violation bugs from their hiding places. In: Proceedings of the 14th ASPLOS, pp. 25–36. ACM, NY (2009)
5. Cui, H., Wu, J., Tsai, C.C., Yang, J.: Stable deterministic multithreading through schedule memorization. In: Proceedings of the 9th OSDI. USENIX Association, Berkeley (2010)
6. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: PRES: probabilistic replay with execution sketching on multiprocessors. In: Proceedings of the 22nd SOSP, pp. 177–192. ACM, NY (2009)
7. Tian, C., Nagarajan, V., Gupta, R., Tallam, S.: Dynamic recognition of synchronization operations for improved data race detection. In: Proceedings of the ISSTA, pp. 143–154. ACM, NY (2008)
8. Jannesari, A., Tichy, W.F.: Identifying ad-hoc synchronization for enhanced race detection. In: Proceedings of the IPDPS, pp. 1–10. IEEE Press, NY (2010)
9. Sherwood, T., Perelman, E., Calder, B.: Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In: Proceedings of the PACT, pp. 3–14. IEEE Computer Society, Washington, DC (2001)
10. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd ISCA, pp. 24–36. ACM, NY (1995)
11. Minh, C.C., Chung, J.W., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: Proceedings of the 2008 IISWC, pp. 35–46. IEEE Press, NY (2008)
12. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: Ad hoc synchronization considered harmful. In: Proceedings of the 9th OSDI. USENIX Association, Berkeley (2010)
13. Zhang, J., Xiong, W., Liu, Y., Park, S., Zhou, Y., Ma, Z.: ATDetector: improving the accuracy of a commercial data race detector by identifying address transfer. In: Proceedings of the 44th MICRO, pp. 206–215. ACM, NY (2011)
14. The GNU C Library manual, `http://www.gnu.org/software/libc/manual/`
15. Li, T., Lebeck, A.R., Sorin, D.J.: Spin detection hardware for improved management of multithreaded systems. J. IEEE PDS 17(6), 508–512 (2006)
16. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the PLDI, pp. 190–200. ACM, NY (2005)

Fast Full-System Execution-Driven Performance Simulator for Blue Gene/Q

Diego S. Gallo[1,⋆], Jose R. Brunheroto[2], and Kyung Dong Ryu[2]

[1] IBM Research Brazil, Sao Paulo, SP 04007-900, Brazil
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
dsgallo@br.ibm.com,{brunhe,kryu}@us.ibm.com

Abstract. Full-system execution-driven simulators are essential tools in the development of supercomputers, such as IBM Blue Gene/Q. They enable software teams to develop and run code before the real system becomes available, typically a few years after the beginning of the project. Functional simulators support early software development. The addition of timing information allows for early performance assessment of applications to provide feedback on the hardware and system design. The techniques employed to implement a timing model for Blue Gene/Q, built on top of the functional model, are presented. Our simulator runs several orders of magnitude faster than traditional cycle-accurate simulators. The experiments with micro-kernels from the Sequoia Benchmark Suite demonstrate that our simulator provides the timing accuracy between 3 to 17% of the actual measurement from the real Blue Gene/Q machine. We also present some architecture design exploration and its performance implications.

1 Introduction

A full-system simulator is an essential tool in the development process of a supercomputer. Cycle-accurate models are important for hardware validation and verification of the architecture being designed before spending all the cost and time building a chip. Those models also help in power consumption and reliability analysis. Since the level of detail needed to be simulated is very high, cycle-accurate models are both time-consuming to develop and very slow to run in software-based simulators (often a mere 10-100 processor cycles per second [1]).

On the other hand, functional models are faster to develop, run faster, and can be built before all the hardware details are fully specified, providing support for the system software, compiler, and applications teams to start developing and testing their softwares long before any hardware prototype becomes available. That alleviates the dependencies among teams, allowing them to work in parallel, and consequently reducing the overall development cycle, although not much can be done using the purely-functional simulator to assure that the performance of those "components" (system software, compiler, and applications) will be

⋆ Work done while at IBM T.J. Watson Research Center.

satisfactory. Due to that, a large optimization effort usually takes place after the hardware becomes available, leading to either unnecessary delays to deliver the system or a poor performance when the system is initially delivered.

It is common to have a cycle-accurate model developed for software-based simulators as soon as the hardware is fully specified, as such simulation is used for hardware verification. Nevertheless, it does not provide much value for the software teams, because the time it takes just to boot the firmware and kernel is already prohibitive, and often it is not feasible to run any applications with minimally useful problem sizes. Consequently, the use of cycle-accurate software-based simulations in any optimization effort is extremely hard.

Alternatively, FPGA-based VHDL simulators can be used to provide reasonably fast cycle-accurate simulation. Although these simulators are important for running the validation tests faster, and even to run, test and optimize software, they take longer to be available due to the need for complete VHDL code, and are not as versatile as software-based simulators, since to change its behavior, one would have to alter the VHDL, which is usually non-parametrical. Additionally, FPGA simulators are expensive and typically in short supply.

Contrarily, software-based simulators are usually extremely flexible through parameterization, allowing a large variety of experiments to be performed through parameter exploration. A list of contributions from this work is itemized below:

- Methodology to build a timing model on top of a functional model, extending previous work [2] to model multi-threaded processors;
- Implementation of a timing model for the Blue Gene/Q (BG/Q) hardware, reasonably accurate for application performance analysis;
- Support architectural design decisions through parameter exploration, comparing the performance in different scenarios before writing any VHDL; and
- Fine-grained performance information about application execution without the need for instrumenting the applications.

In Section 2 the BG/Q system is briefly described, followed by a description of the Mambo simulator and the BG/Q functional model. Our proposed timing model is presented in Section 3, with validation of the model in Section 4. The simulation execution speeds of the different models (with increasing level of detail) are shown in Section 5, and some use cases of our full-system event-driven performance simulator, both in terms of profiling and what-if experiments, are shown in Section 6. Finally, Section 7 concludes this work.

2 Blue Gene/Q model on Mambo Simulator

This section briefly describes the BG/Q system itself, followed by a description of the functional model implemented on Mambo to simulate the system.

2.1 BlueGene/Q System Overview

The BG/Q system [8, 10] comprises compute racks with two midplanes. Each midplane has 16 node cards, with 32 nodes assembled on each card. That adds up to 1024 nodes per rack, providing a dense concentration of processing power. A five-dimensional (5D) torus network connects the compute nodes, providing node-to-node communication and integrating hardware-assisted barrier and collectives functionality onto the same network [4].

Each node contains 18 IBM PowerPC A2 cores, of which 16 are used to run application code, one is used to offload services provided by the lightweight kernel operating system CNK (Compute Node Kernel) [7], and one is a spare core used for increasing the yield during the chip manufacturing process. Figure 1 shows a block diagram of the A2 core components and the relationships among them.

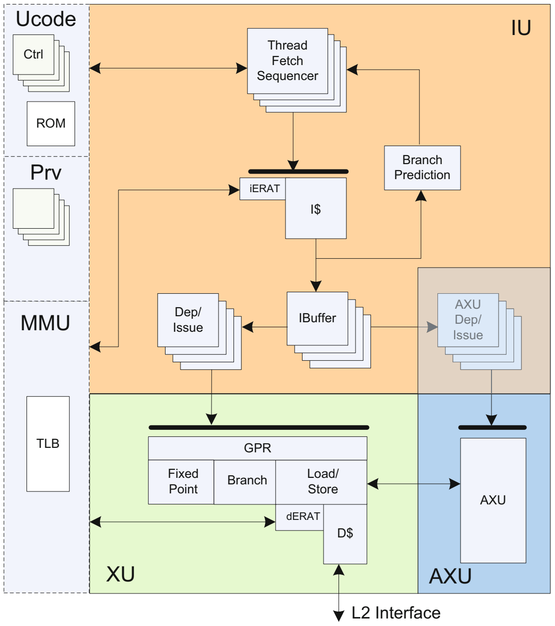


Fig. 1. BG/Q A2 core block diagram

These 1.6GHz cores have four in-order execution hardware threads each, that combined can issue up to two instructions per cycle: one integer instruction to the eXecution Unit (XU) and one floating-point instruction to the Auxiliary eXecution Unit (AXU). The Instruction Unit (IU) is responsible for fetching instructions into each thread instruction buffer, predicting branches direction, checking for register dependencies, and arbitrating between the threads that have an instruction ready to be issued.

The AXU is a 4-way Quad-vector Processing Unit (QPU), that implements the Quad Processing eXtension (QPX) to the Power ISA [6]. Due to the QPU, four multiply-and-add double-word floating-point operations can be executed in a SIMD (Single Instruction Multiple Data) way, providing 8 flops/cycle per core, summing up 204.8 Gflop/s of peak performance on each compute node.

As for the memory subsystem, each A2 core has its own private L1 instruction and data caches (16KB each). The node has a shared L2 cache of 32MB that is connected to 16GB of physical main memory. Additionally, each core has a prefetcher engine (L1P) with 32 entries of 128 bytes that sits between the L1 and L2 to prefetch data and to serve as a write-back buffer.

2.2 Functional Model of Blue Gene/Q on Mambo

The Mambo simulator [3] currently provides support for modeling and simulating the execution of a multitude of systems, including PowerPC, Cell, A2, and embedded processors, among others. Its modular and configurable design provides building blocks that can be re-used, as well as a scheduling mechanism that efficiently handles the context switch between any sort of tasks one can create (to handle processors, threads, and any other types of resources). This design allows focusing on the system characteristics instead of the simulation platform when modeling a system.

The purely functional model of BG/Q on Mambo comprises the A2 core model (its instruction set running on four hardware threads) combined with the instruction set provided by the QPX. All the instructions are implemented in an architecturally-accurate manner, providing bit-accurate results. Additionally, a memory subsystem with all the cache levels of the BG/Q system (private L1 cache and L1P prefetcher, shared L2 cache, and main memory) can be enabled, providing a first set of performance metrics in the form of hit/miss rates through the different cache levels. When the memory hierarchy model is enabled in the simulator, we say that it is running in *warmup* mode.

Even though the hardware threads are implemented in the functional model, each with its register file and instruction pointer, there is no notion of the execution pipelines or contention in instruction issue for resources that are shared among the threads. One instruction from each thread can be issued at every cycle, respecting only locks and barriers for the correctness of the execution.

Consequently, even though the memory subsystem with the cache hierarchy is implemented, providing statistics regarding hit/miss rates, these may be inaccurate since the execution time of the instructions is not respected. That can cause differences in cache behavior, since the four threads inside a core interfere with each other at the L1 and L1P level, and the seventeen cores inside a node share the L2 cache. The next section describes our model that not only solves this issue, but also provides a large variety of additional information about the simulated execution, allowing a better understanding of application performance.

3 Timing Model for Blue Gene/Q

On top of the purely functional model and the memory subsystem previously described, and based on previous work about tracking resource dependencies for a pseudo-cycle-accurate timing model for Blue Gene/L [2], a timing model for the BG/Q compute node was developed extending the previous work twofold:

First, extending the framework to support multi-threaded cores, since the A2 core has four hardware threads as opposed to the single-threaded PowerPC 440 on BG/L, and second, considering in our model the resources that mostly impact system performance, such as the thread instruction-fetch sequencer, the instruction buffer and the dynamic branch predictor from the Instruction Unit, and the Load Miss Queue (LMQ) and Store Queue (SQ) from the memory subsystem.

The proposed timing model relies on three main factors: time-stamping the most relevant resources (e.g. scalar, floating-point, and special-purpose register files; XU and QPU execution units); checking time-stamps of dependent resources before issuing instructions (waiting until all the resources are available); and updating time-stamps of all used resources after executing an instruction with the corresponding time when each resource becomes available.

This mechanism guarantees that an instruction will be issued if and only if the register dependencies are satisfied and the necessary execution unit is available, limiting the instruction execution rate by the two most restrictive requirements. The update of dependent resources after an instruction is executed is based on:

1. *Instruction latency*: Number of cycles needed between instructions that have register dependencies, i.e. that use a register that was a target in a previous instruction (for BG/Q, usually 1 or 2 cycles for XU instructions and 6 to 8 cycles for QPU instructions). Time-stamps of target registers are updated to reflect the instruction latencies.
2. *Instruction throughput*: Specified as the number of cycles in which a unit will be busy for the execution of a given instruction. Time-stamps of the XU or QPU execution unit will be updated accordingly (usually available in the next cycle, except for instructions that are not fully pipelined), e.g. some multiply instructions and micro-coded instructions).

The write-back latencies (i.e., the interval from when the instruction is issued until the resulting value is written back to the register) for arithmetic and logic instructions are usually directly dependent on the number of stages in the pipeline, except for instructions that re-enter the pipeline (e.g. multiply), and micro-coded instructions (e.g. divide), while the write-back latencies for load instructions depend on the memory subsystem. To avoid the need for a cycle-accurate memory subsystem, which would slow down the whole simulation and defeat the initial goal of having a really fast performance simulator, our model uses the average latencies to each cache level and to main memory. For every load, the average latency corresponding to the level of the hierarchy in which the data is located (i.e. L1, L1P, L2, or main memory) is used to estimate the load latency. The results in Section 4 show that using this approximation regarding the load latency still allows reasonable accuracy simply by modeling the cache hierarchy and the contention caused by the LMQ and SQ.

Additionally, since the execution units are shared among the four hardware threads, which may be competing for the same unit, their use have to be constrained respecting the round-robin policy for threads with the same priority. By incrementing the time-stamp of the XU or QPU according to the instruction throughput and using the Mambo simulator infrastructure to schedule the next events in two steps, firstly the threads that are waiting for dependencies, and secondly the threads that just executed, the round-robin policy is respected.

Moreover, the thread instruction-fetch sequencer and instruction buffer were implemented in our model to correctly handle the timing implications of fetching instructions ahead of time. In the A2 core, each thread has an instruction buffer that can hold up to eight instructions (32 bytes). At each cycle, the Instruction Unit can begin fetching instructions for one thread. A group of four instructions (16 byte-aligned bytes) is fetched, and instructions will be discarded in case the address to fetch is not at the beginning of this group. Additionally, instructions after a branch will also be discarded if the group contains a predicted-taken branch. The thread that will fetch instructions in a given cycle is chosen in a round-robin manner, and a thread has high priority if its instruction buffer is completely empty and there is no fetch request in flight.

Furthermore, the branch prediction mechanism was also implemented in the model to correctly reflect misprediction penalties in the execution time. On BG/Q, conditional branches with a hint are statically predicted via their hints, and all the other conditional branches are predicted using a gshare-like dynamic branch prediction mechanism that remembers prior branch directions using a Branch History Table (BHT). The BHT contains 1024 entries, 2 bits each, that are incremented for taken branches (saturating at three) and decremented for not-taken branches (saturating at zero). A branch is predicted as taken if the counter is two or three, and not-taken otherwise. Finally, to index the BHT, the lowest address bits of the instruction are XORed with a per-thread Global Branch History Register (GBHR), which helps in correctly predicting interleaved branches. In the event of a mispredicted branch, a flush is generated in the pipeline and there will be a minimum of 13 cycles from when the branch instruction is fetched (or issued) until the correct target is fetched (or issued).

Lastly, the LMQ and SQ are also important points of contention in the BG/Q node that had to be modeled for timing correctness, since they can cause threads to stall. Each core has an eight-entry LMQ, shared among the four threads, that holds load misses and non-cacheable loads while they are outstanding to the L2, allowing the thread to continue issuing instructions (provided that the target register from the load is not used). Additionally, a thread will stall if issuing a load request that misses the L1 when the LMQ is full.

Regarding store instructions, there are no store buffers in the A2 core. Stores are sent directly to the L1P, where they are queued and wait for arbitration in the L2 crossbar switch (having priority over load requests during the arbitration). Additionally, while in the L1P, two stores (8 bytes each) to adjacent and aligned memory locations can be combined to improve memory bandwidth. If the SQ is full, a thread issuing a store request will stall until one entry is successfully consumed by the crossbar switch. The SQ on BG/Q has 20 entries, modeled in our timing model to reflect contention caused due to store instructions.

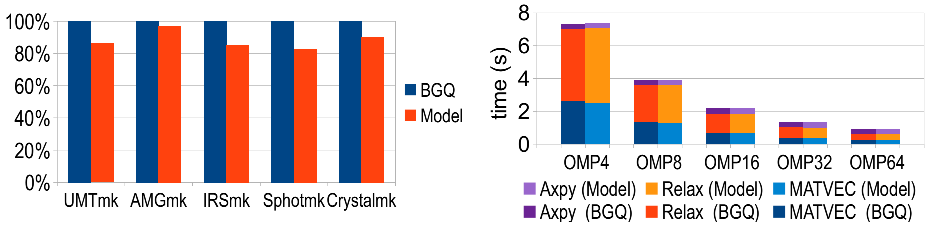
The parameters in the timing model (e.g. latency to caches and memory, depth of pipelines, number of execution pipelines, sizes of LMQ and SQ) can be configured, providing a way to quickly experiment with what-if scenarios. The next sections show the accuracy of the implemented model for BG/Q, its simulation speed compared to the purely functional model, and results from use cases, including advanced profiling and what-if experiments.

4 Timing Model Validation

In order to validate the model, we ran experiments with the microkernels (UMTmk, AMGmk, IRSmk, SPhotmk, and Crystalmk) from the Sequoia Benchmark suite [9] to cover multiple applications with different characteristics (e.g., different instruction mixes, memory access patterns, and SIMD utilization). These microkernels were developed at Lawrence Livermore National Lab to represent some of the main attributes of applications that make use of their supercomputers, providing meaningful performance numbers in more realistic scenarios than just the LINPACK (LINear algebra PACKage) benchmark [5] and others.

All the microkernels were used to validate the single-thread accuracy of the timing model. Additionally, since AMGmk has OpenMP support, we validated the accuracy of our model for the entire BG/Q node running it using up to 16 cores and 64 hardware threads. It is worth mentioning that for measuring the accuracy of our timing model, the absolute performance of the codes was not relevant. Thus, we simply used the reference benchmark codes without any optimization other than optimizations provided by the compiler.

Figure 2a shows the accuracy of our model for each of the five Sequoia microkernels. Time reported by our Mambo timing model for each of them is normalized to the time measured running the application using BG/Q hardware. Accuracy of the model for these benchmarks vary from -3% (AMGmk) up to -17% (SPhotmk). Additionally, Figure 2b illustrates the accuracy of our model when running the OpenMP version of the AMGmk while varying the number of threads. Across the different configurations, accuracy stays within 5% for each of the three phases of this microkernel (MATVEC, Relax, and Axpy).



(a) (b)

Fig. 2. Mambo timing model accuracy for the Sequoia microkernels (a) and time spent on each phase of AMGmk varying the number of OpenMP threads (b)

Note also that the AMGmk benchmark parallelizes loops both in the MATVEC and Relax phases through the use of OpenMP pragmas, decreasing the time spent in each phase when increasing the number of threads (i.e. 4, 8, 16, 32, and 64 threads). Contrarily, time spent in the Axpy phase is constant independently of the number of OpenMP threads, because the reference code does not specify any parallelism in that phase.

5 Timing Model Execution Performance

As mentioned and shown earlier in this paper, our timing model allows developers to obtain reasonably accurate performance information before all the hardware details are completely specified and any VHDL is written. Additionally, it is much faster than software-based cycle-accurate models, and more flexible than FPGA-based simulators.

Table 1 illustrates the average simulation execution speed for the Sphotmk benchmark running in a single thread (except for the MESA cycle-accurate value, which is a reference value). It compares the speed among the different models and different levels of detail, and also the real hardware.

It is worth mentioning that the row named "profiler enabled" shows the slowdown in the simulation when enabling our profiling functionality. Also, speed comparison should be done on an instruction-per-second basis, otherwise one would get the impression that our timing model is actually faster than the purely functional model, because it simulates more "cycles per second". That is only an artifact due to some instructions advancing the cycle counter by multiple cycles because of different dependencies and contention. Additionally, while the A2 core in the real BG/Q hardware runs at 1.6 GHz, all the other values are averages measured over the simulation executions, because different instructions take different amounts of time to simulate.

Table 1. Simulation execution speed for the different models

	Average simulation execution speed	
	cycles / sec	**instructions / sec**
Real BG/Q hardware	(1.6 GHz)	(375,434,609)
Purely functional	1,682,411	1,077,807
Warmup (cache hierarchy)	1,501,073	961,636
Proposed timing model	2,570,047	677,792
Profiler enabled	1,793,757	473,063
MESA cycle-accurate simulator [1]	(10-100)	(2-20)

Note that in the purely functional mode and the warmup mode, the difference between the number of cycles and the number of instructions is simply due to the fact that you can specify the maximum possible throughput for each instruction, with some instructions taking multiple cycles to execute. Nevertheless, it does not take into account any of the register and load dependencies, instruction buffering, branch misprediction penalties or anything else, basically allowing the issue of one instruction per thread per cycle at all times, thus leading to an extremely optimistic CPI (Cycles Per Instruction) value.

Additionally, when the application uses multiple cores in a single node, there is a near-linear slowdown in the simulation speed (cycles/sec), because there is more work to be simulated, except for the MESA cycle-accurate simulator, since it is always simulating the full node (all the cores), even when there are no instructions being issued on any of the other cores, because it is an accurate representation of the hardware based on the VHDL, simulating the propagation of every clock cycle through the entire node. Looking into the number of instructions executed per second in each mode, the warmup mode is verified to have slowed the simulation by 11%, while the slowdown due to the timing model was 37%. Enabling the profiler slowed the simulation a total of 56%. Nevertheless, the slowdown is minimal when compared with the simulation speed of the software-based cycle-accurate simulator (MESA), which is four to five orders of magnitude slower in this case (single-threaded application). Even comparing the speed when using all 64 threads, the cycle-accurate simulation is still 3 to 4 orders of magnitude slower than our timing model.

6 Use Cases for the Timing Model

In this section, two use cases enabled by the proposed timing model are illustrated: *application profiling* and *what-if experiments*. The first allows developers to obtain performance information as fine-grained as they would like, since the application execution is not disturbed by the collection of information. The second allows hardware architects to assess the impact of different architectural decisions, providing performance information for hypothetical scenarios.

6.1 Application Profiling

The functional model of the simulator provides some basic profiling capabilities, allowing the collection of the instruction mix from applications. Enabling the cache hierarchy model allows additionally collecting hit/miss rates at the different cache levels, but since each thread may execute one instruction per cycle in that mode without any contention due to the execution units, memory access, register dependencies, or anything else, the hit/miss rates might be inaccurate for some applications (e.g. unbalanced multi-threaded applications).

The timing model proposed herein adds new profiling capabilities to the simulator, therefore allowing the collection of fine-grained performance information without the need for instrumenting the application in ways that could alter its behavior (since fine-grained instrumentation often generates a prohibitive level of overhead and disturbance in the application).

To illustrate the profiling capabilities our timing model adds to the simulator, Figure 3a shows the IPC (Instructions Per Cycle) values for a section of the AMGmk benchmark, in both the XU and QPU execution units, and Figure 3b shows the percentage of time the application stalled due to a load request that misses the L1 when the LMQ is full (almost zero for this benchmark) and due to a store request when the SQ is full (as high as 16.7%, i.e., 16,700 cycles stalled due to SQ full in a 100,000 cycles sampling).

Developers can easily track the performance of an application with the information provided by our timing model, gaining insight into the parts of the

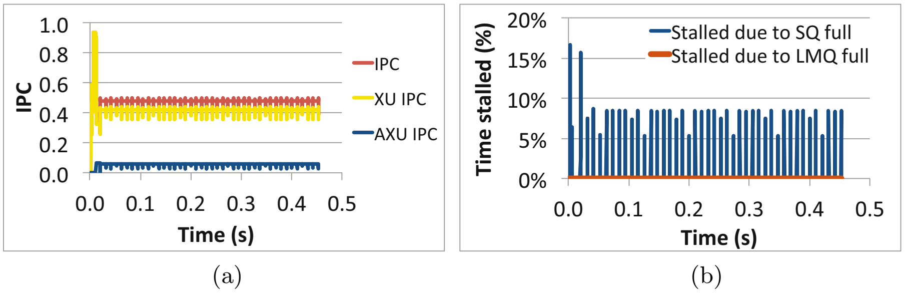


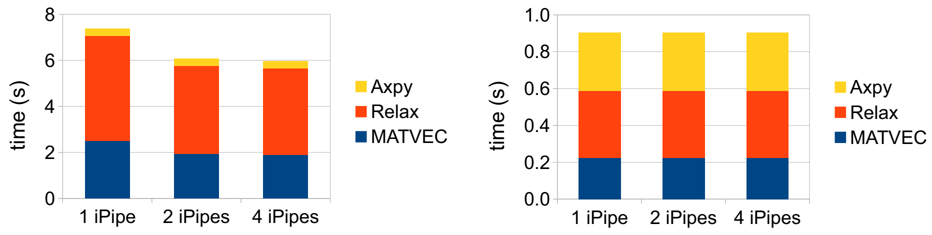
Fig. 3. Average IPC over each sample interval (a) and percentage of time stalled due to LMQ or SQ full (b) throughout the application execution

applications that should be optimized, and the causes that are leading to poor performance (e.g. what sections of the application saturate the memory, filling up the LMQ or SQ, and what sections saturate one of the execution pipelines).

6.2 What-if experiments

Another interesting use case for our timing model is the possibility to experiment with slightly different architectural designs, assessing their performance implications. This helps evaluating tradeoffs during the concept phase of the project. Since the compiler has not been changed to take advantage of the different architectural designs being explored, the performance results shown in this subsection might be underestimated. Nevertheless, they are still noteworthy.

Figure 4a evaluates the impact of having multiple integer execution pipelines (iPipes), simulating the execution of the AMGmk benchmark, using 4 OpenMP threads, configuring the A2 core model to have 1, 2 or 4 iPipes.



(a) AMGmk OMP4 timing (b) AMGmk OMP64 timing

Fig. 4. Impact on timing if A2 core had multiple integer pipelines

Note that adding a second iPipe in the A2 core would lead to a performance gain both in the ‘Relax’ and ‘MATVEC’ phases of AMGmk, achieving a 20% reduction in the total execution time. The reason is that a second iPipe would alleviate stalls allowing concurrent instruction issue by multiple threads. However, the experiment shows that more than 2 iPipes would not help. Nonetheless, it is also interesting to note that when using 16 cores for the application (having 64

OpenMP threads), the benefits of having multiple iPipes would disappear due to other bottlenecks, as shown in Figure 4b.

Another experiment evaluates the impact of different store queue sizes on AMGmk performance. Figure 5 shows that if a store queue with half the size of the current queue in the system (i.e. 10 entries instead of 20 entries) was used, the total execution time of the benchmark would increase by more than 50% and the time spent on the MATVEC phase alone would more than double. On the other hand, if we had twice as many entries in the store queue (i.e. 40 entries), the performance gain would be minimal (only 3%) for this benchmark.

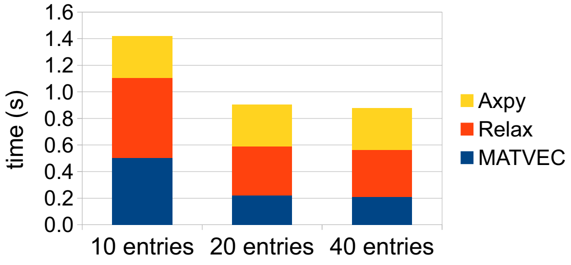


Fig. 5. Impact on AMGmk OMP4 timing if A2 had different store queue sizes

7 Conclusions

During the early phases of a supercomputer architecture definition, or any new processor architecture design, a full-system execution-driven performance simulator can provide fairly accurate information about the applications performance in such new hardware. Additionally, it allows design space exploration over different architectural parameters (e.g. number of execution units inside the core, size of queues and buffers, cache hierarchy and memory latencies), providing data to help evaluate tradeoffs (e.g. chip cost, area and power consumption vs. performance gain).

After the architecture is defined and the hardware starts to be produced, the timing model can provide valuable fine-grained information regarding the execution, allowing not only the analysis of subsections of the application without the need to instrument it, which would possibly alter its behavior, but also providing detailed information of where exactly the application is stalling due to busy execution units, register dependencies, load dependencies, or full load-miss or store queues. In that way, we augmented the capabilities provided by the BG/Q performance counters, allowing the simulator to collect performance information that cannot otherwise be collected by the hardware without interfering with the execution.

All the fine-grained information provided by our timing model, added to the reasonably fast simulation speed (tens of thousands times faster than the software-based cycle accurate simulator), makes it also a very helpful tool for optimizing application performance.

Acknowledgment. The authors would like to thank Martin Ohmacht for the detailed information about memory subsystem, Bob Walkup for pointing out the correct libraries and compilers, Fred Mintzer and Dave Singer for providing us access to the BG/Q system, and Bryan Rosenburg for all the help in improving this paper. The BG/Q project has been supported and partially funded by the Argonne National Laboratory and the Lawrence Livermore National Laboratory on behalf of the US Department of Energy, under Lawrence Livermore National Laboratory subcontract number B554331.

References

1. Asaad, S., Tierno, J., Bellofatto, R., Brezzo, B., Haymes, C., Kapur, M., Parker, B., Roewer, T., Saha, P., Takken, T.: A cycle-accurate, cycle-reproducible multi-FPGA system for accelerating multi-core processor simulation. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 153–161. ACM Press, New York (2012)
2. Bachega, L., Brunheroto, J., DeRose, L., Mindlin, P., Moreira, J.: The BlueGene/L pseudo cycle-accurate simulator. In: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 36–44. IEEE Computer Society, Washington, DC (2004)
3. Bohrer, P., Simpson, R., Speight, E., Sudeep, K., Van Hensbergen, E., Zhang, L., Peterson, J., Elnozahy, M., Rajamony, R., Gheith, A., Rockhold, R., Lefurgy, C., Shafi, H., Nakra, T.: Mambo - A Full System Simulator for the PowerPC Architecture. ACM SIGMETRICS Perform. Eval. Rev. 31(4), 8–12 (2004)
4. Chen, D., Eisley, N.A., Heidelberger, P., Senger, R.M., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D.L., Steinmacher-Burow, B., Parker, J.J.: The IBM Blue Gene/Q interconnection network and message unit. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–10. ACM Press, New York (2011)
5. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: past, present and future. Concurrency and Computation: Practice and Experience 15(9), 803–820 (2003)
6. Fox, T., Gschwind, M., Moreno, J.: QPX Architecture: Quad Processing eXtension to the Power ISA. Tech. rep., IBM Research, Yorktown Heights, NY (2012)
7. Giampapa, M., Gooding, T., Inglett, T., Wisniewski, R.W.: Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–10. IEEE Computer Society, Washington, DC (2010)
8. Haring, R.A., Ohmacht, M., Fox, T.W., Gschwind, M.K., Sugavanam, K., Coteus, P.W., Heidelberger, P., Blumrich, M.A., Wisniewski, R.W., Gara, A., Chiu, G.L.T., Boyle, P.A., Christ, N.H., Kim, C.: The IBM Blue Gene/Q Compute Chip. IEEE Micro 32(2), 48–60 (2012)
9. Lawrence Livermore National Laboratories: ASC Sequoia Benchmark Codes, `https://asc.llnl.gov/sequoia/benchmarks/`
10. The Blue Gene Team: Blue Gene/Q: by co-design. Computer Science - Research and Development (2012)

Topic 2: Performance Prediction and Evaluation (Introduction)

Adolfy Hoisie, Michael Gerndt, Shajulin Benedict,
Thomas Fahringer, Vladimir Getov, and Scott Pakin

Topic Committee

In recent years, a range of novel methodologies and tools have been developed for the purpose of evaluation, design, and model reduction of existing and emerging parallel and distributed systems. At the same time, the coverage of the term "performance" has constantly broadened to include reliability, robustness, energy consumption, and scalability in addition to classical performance-oriented evaluations of system functionalities. Indeed, the increasing diversification of parallel systems, from cloud computing to exascale, being fueled by technological advances, is placing greater emphasis on the methods and tools to address more comprehensive concerns. The aim of the Performance Prediction and Evaluation topic is to bring together system designers and researchers involved with the qualitative and quantitative evaluation and modeling of large-scale parallel and distributed applications and systems to focus on current critical areas of performance prediction and evaluation theory and practice.

The three papers selected for the topic area reflect the broadening perspective of parallel performance involving automatic comparison of performance traces, assessment of the uncertainty in performance prediction through simulation, and performance tuning for NUMA architectures.

The paper "Alignment-Based metrics for Trace Comparison" uses sequence-alignment algorithms to align events across two traces obtained from performance analysis tools to identify which events were added or removed or took a different amount of time to complete.

The uncertainty of Extreme-Scale HPC simulation is assessed in the paper "Validation and Uncertainty Assessment of extreme-Scale HPC Simulation through Bayesian Inference". The paper takes a statistical approach to quantify the uncertainty involved in predicting performance via simulation.

The third paper on "Dynamic thread Pinning for Phase-Based OpenMP Programs" investigate the impact of adjusting thread pinning (affinity) dynamically for each parallel region of an OpenMP program. The authors perform a dynamic analysis of memory accesses and calculate a gain function between pairs of threads, which is in turn used to generate a recursive partitioning of threads on shared memory spaces.

Alignment-Based Metrics for Trace Comparison*

Matthias Weber[1,2], Kathryn Mohror[2], Martin Schulz[2], Bronis R. de Supinski[2], Holger Brunst[1], and Wolfgang E. Nagel[1]

[1] Center for Information Services and High Performance Computing, Technische Universität Dresden, Germany
{matthias.weber,holger.brunst,wolfgang.nagel}@tu-dresden.de
[2] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, USA
{kathryn,schulzm,bronis}@llnl.gov

Abstract. Due to the complexity of today's architectures and applications, performance analysis and optimization are essential, and trace-based techniques have proven to be a powerful approach. However, a manual comparison of traces is difficult and time consuming because of the large volume of detailed data and the need to correctly line up trace events. Our solution is a set of techniques that automatically align traces so they can be compared, along with novel metrics that quantify the differences between traces, both in terms of differences in the event stream and timing differences across events. Further, we introduce visualization techniques that highlight and facilitate understanding of the sources of the differences. We demonstrate the effectiveness of our solution by showing automatically detected performance and code differences across different versions of two real-world applications.

1 Introduction

Today's complex architectures and applications make it challenging to fully exploit the performance of high performance computing (HPC) machines. Understanding the root cause of application performance problems requires sophisticated performance analysis techniques. A key operation required by nearly all these techniques is a comparison of performance data from multiple measurements, because there is not sufficient information to understand the performance properties of an application without baseline measurements for comparison. Typical examples for such comparisons are before/after comparisons when applying optimizations or changing code versions; comparisons of runs on different platforms to study performance portability; or contrasting the performance of different ranks in an MPI program to study load balance.

* This article has been authored by Lawrence Livermore National Security, LLC under Contract No. DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. LLNL-CONF-586852.

While such comparisons are straightforward for the aggregated data in performance profiles, no good solutions exist for comparing highly-detailed event traces. An event trace is a record of application behavior as a series of events, such as function entry and exit, or message passing. Each event consists of a timestamp along with relevant data, e.g., function name, or bytes of data transmitted. The information in traces can be used to detect many performance problems on HPC systems, e.g., the causes of synchronization delays. However, the level of detail in traces also presents a challenge because it implies a large amount of data from a single run, even exceeding hundreds of megabytes for a single process [13]. Thus, manually aligning event traces for comparison is extremely challenging and error prone, since the event stream may not be equal across application runs or MPI ranks, and events will not occur at exactly the same time across executions.

Although several efforts have been made for comparing performance across application runs [2,8,10,12,15–17], to the best of our knowledge, no tool supports automatic comparison of event traces. We address this gap with a technique for automatic trace comparison of two arbitrary event traces.[1] We use an hierarchical alignment algorithm that performs event-wise comparison and alignment of traces, from prior work [18]. In this paper, we make several contributions:

– We extend our alignment algorithm through an adaptive hybrid scheme, improving performance by nearly $8\times$.
– We develop novel metrics for comparative analysis of event traces.
– We integrate visualization support for trace comparison into the Vampir tracing tool [5] for intuitive understanding of the differences between traces.
– We present case studies of two real-world applications AMG [6] and ParaDiS [3], showcasing the viability of our method.

2 Alignment Algorithm for Trace Comparison

Our work builds on an hierarchical alignment algorithm presented in prior work [18]. In this work, we extend the algorithm and significantly reduce analysis times, which makes it feasible to use this approach for analyzing large traces.

2.1 Base Hierarchical Alignment Algorithm

In order to compare two traces, we borrow techniques from gene sequence alignment approaches in bioinformatics. We align traces with a dynamic programming algorithm that finds the optimal alignment for arbitrary sequences [4,7,14]. The algorithm separates the full pairwise alignment problem into independently optimizable sub-problems and then evaluates alignments with scores. It finds the optimal alignment of sub-problems with a recursive scoring scheme.

[1] Although we motivate and apply our technique to function call traces, our approach is general and can support any kind of event trace, e.g., memory traces or I/O traces.

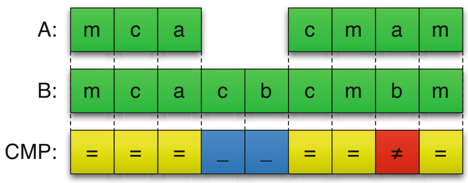


Fig. 1. Constructed alignment of sequence A and B

Although the dynamic programming approach is functional, the quadratic time complexity leads to long alignment times. We employ a modification proposed by Hirschberg [9] that computes the optimal alignment with quadratic time complexity but with only linear memory complexity with respect to the longest sequence. We also augment the algorithm with an hierarchical comparison approach based on the call tree structure of the execution that shortens the event sequence length for individual comparisons.

We show an example alignment of sequence A: $m\ c\ a\ c\ m\ a\ m$ with sequence B: $m\ c\ a\ c\ b\ c\ m\ b\ m$ in Figure 1. Using the algorithm, we mark portions of traces as *equal* (shown in yellow in Figure 1) if they have the same sequence of function calls in both traces, omitting for now the timing information associated with events.[2] We label portions of traces as *different* (red in Figure 1) if they contain different function calls at the same sequence position. For instance, if a call to function a in the first trace is replaced by a call to b in the second trace, these calls would be recognized as different. A *gap* (blue in Figure 1) is a missing section in one trace file, which occurs if some functions are executed during the generation of one trace and not the other, e.g., if a new code section is added to the application, or MPI ranks follow different execution paths.

2.2 Extended Hierarchical Alignment Algorithm

While this algorithm provides a suitable alignment, its performance and memory consumption prevents its use on larger event traces. To overcome this challenge, we extend the approach in this work by adaptively using both the Needleman-Wunsch [14] and the Hirschberg [9] algorithms for the alignment. Since the Hirschberg algorithm needs to recompute erased values it runs slower, but requires less memory. Thus, we use the faster Needleman-Wunsch algorithm, with quadratic memory complexity, for small alignments that fit into memory and use the Hirschberg algorithm, with linear memory complexity, only for larger ones that would otherwise not fit. This can speed up the alignment for large parts of the traces. In our experiments, using an Intel Xeon 5660 node running at 2.8 GHz with 24 GB RAM, the extended alignment algorithm ran nearly 8× faster

[2] Omitting timing information eliminates problems caused by the inherent jitter in timing measurements caused by timer inaccuracy, OS noise, or network traffic. The sequence of functions calls, on the other hand, is likely to be the same for large parts of the trace and hence can be used as anchors for the alignment process.

Table 1. Alignment Algorithm Performance

Application	Base Alignment	Extended Alignment	Performance Improvement
ParaDiS	$11min\ 38s\ 214ms$	$1min\ 30s\ 986ms$	7.67
AMG2006	$4min\ 56s\ 533ms$	$38s\ 667ms$	7.67

than the base algorithm. Table 1 shows the times needed to align the complete application traces shown in Section 5.

3 Visualization of Trace Comparison

We integrate support for visualizing compared traces into the Vampir tracing toolset [5] that provides users with a visualization of the similarities, differences, and gaps across traces, as well as visualizations of the timing differences. This goes beyond our prior work, where the visualization only showed a flat view of all traced events at any given time point, simply depicting the current state of the trace comparison at a time point — either equal, different, or gap.

In particular, we implement a hierarchical display of the trace differences based on the call tree. This hierarchical display facilitates understanding of trace differences, because it is easier to identify the root of those differences. For example, if additional function calls are made in one trace and not the other, it is easy to locate the enclosing function to further investigate the changes. Additionally, we now differentiate the origin of gaps. We introduce two gap states ($GAP\ A$ and $GAP\ B$), which indicate if a gap occurs in trace A or B.

4 Trace Comparison Metrics

Intuitive displays of aligned traces alone are not sufficient to help users understand the differences between traces. For this purpose, we introduce new trace comparison metrics. We design these metrics to aid the developer in identifying differences in function structure and timing behavior between multiple executions or event traces. While we assume that the user intends to compare two traces of similar behavior, our approach can be applied to a wide range of scenarios such as before/after comparison for optimizations, comparisons between MPI ranks or between threads, to study changes in code versions, or to understand the impact of OS-level runtime events on individual applications.

4.1 Similarity Metric

We base our metric for trace similarity on the alignment algorithm described in Section 2. The algorithm is based on the following scoring scheme, which we use to derive a metric stating the similarity of two traces:
Match Score: $\sigma_{equal} = 2$, Mismatch Score: $\sigma_{diff} = -1$, Gap Score: $\sigma_{gap} = -1$.

Using the raw score, however, is problematic. In an alignment, each function pair, dependent on its state—equal, different, or gap—represents the score defined in the scoring scheme. The sum of all scores, $Score_{actual}$, results in positive

Table 2. Trace comparison metrics

Metric	Definition
$Score_{max}$	$\sigma_{equal} * \max(M, N)$
$Score_{min}$	$\sigma_{diff} * \min(M, N) + \sigma_{gap} * \lvert M - N \rvert$
$Ratio$	$\frac{Score_{actual}}{Score_{max}}$
$Similarity$	$\frac{Ratio+0.5}{1.5}$

scores for equal areas and penalizes differences and gaps with negative scores. Thus, the higher the total score, the higher the similarity between the processes. However, the actual value of the total score also depends on the length of the compared sequences, M and N. This renders the total score impractical as a metric for the direct comparison of the similarity of multiple process pairs.

We note that in every comparison of two traces, there is a maximal and minimal total score, $Score_{max}$ and $Score_{min}$ in Table 2. Two completely equal traces achieve the maximal score. To compute the minimum score, we first insert gaps for missing events in the shorter trace. Then, using penalties of -1 for all gaps and differences in the traces, the total score decreases as an equal (positive) scoring pair is replaced by a difference/gap (negative) scoring pair. This results in a minimum score that aligns both sequences with differences and necessary gaps in case of unequal sequence length.

Using these minimum and maximum scores as a value range, we can now begin to derive our similarity metric. First, we compute how close the actual total score of an alignment is to its maximum, *Ratio* in Table 2. *Ratio* has a range of $[-0.5, 1]$. To define a more intuitive metric, we scaled the value to a range between $[0, 1]$. Thus, we define *Similarity* as given in Table 2. *Similarity* presents a means to objectively evaluate and compare the similarity of processes. $Similarity = 1$ means the processes are completely equal whereas $Similarity = 0$ means they are completely different.

4.2 Dissimilarity Timeline Metric

The *Dissimilarity Timeline* metric indicates how the similarity between two traces changes over time. This is useful for identifying regions of the trace that exhibit high dissimilarities for further inspection. Also, visualizing the metric along with the traces can help pinpoint periodic differences in the traces.

To compute the metric, we start the alignment process and then sample the alignment over time. In this process, we produce a series of data values at equal-distant time points indicating similarity for locations in the aligned traces. Once we have the alignment and data values, we apply a sliding window and sum up the score under the window. In our analysis, a window width of 10% of the entire alignment length produced good results for all experiments. We found that it balances temporal granularity with the ability to capture context information around points with increased changes avoiding misleading "spikes of differences".

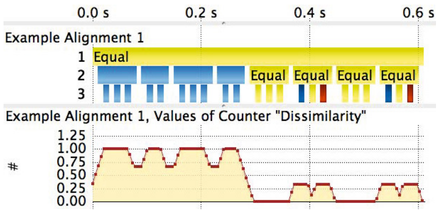


Fig. 2. Dissimilarity timeline

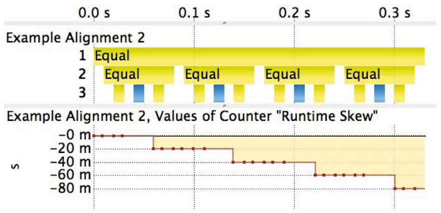


Fig. 3. Runtime skew timeline

Several strategies are possible for summing up the values in a window. For example, one could sum up the total score of all pairs, or could sum the score of all equal pairs. The latter would result in a metric representing the similarity of the sequences at the window position. Since we generally compare relatively similar applications and want to detect the dissimilarities, we use a slightly different approach. We sum the scores of all difference and gap pairs under the window to give a metric representing the dissimilarity of the traces over time. To make the metric comparable between alignments, we normalize it to a value range of $[0, 1]$. A dissimilarity of 1 means all pairs in the window are different or gaps, while a dissimilarity of 0 means all pairs are equal.

Figure 2 shows a *Dissimilarity Timeline* for a simple example. Sampled measurement points are depicted as red dots. The vertical axis at the top represents the function call depth. The timeline visualization makes it easy to detect areas with high dissimilarity. For example, at the beginning of the alignment shown there are four *Gap A* areas (blue) that result in high dissimilarity. However, the areas that are equal (all yellow) have a dissimilarity of 0. In Section 5, we will demonstrate the usefulness of this metric, as it is very challenging to pick out the dissimilarities of real application traces and alignments without it.

4.3 Runtime Skew Timeline Metric

For comparative performance analysis, it is important to understand the behavior of applications over time. Event traces are especially useful for this purpose, because they retain the time-stamp of each event occurrence. However, it is extremely challenging for users to gain this understanding manually. It involves attempting to line up iterations from traces and visually determine the timing differences between them. To aid in this process, we developed the *Runtime Skew Timeline* metric. In Section 5, we show that this metric is valuable for analyzing performance differences across traces, including the impact of code changes.

Note that our alignment algorithm introduces artificial virtual timings into the visualization of the traces; i.e., functions that appear to line up according to time in the visualization do not necessarily occur at the same time relative to the beginning of the execution. These artificial virtual timings arise because our alignment algorithm does not account for event timings, and only considers function names. For instance, the introduction of gap areas changes the apparent runtime of events. Also, aligning short running functions with long running functions alters the displayed runtime. In such cases, successive function calls are shifted back in time by the difference between the durations of the aligned functions. Figures 4 and 5 show examples of unaligned and aligned visualizations of the same traces. Hence, looking at an arbitrary pair of functions in an aligned trace does not allow one to draw conclusions about the real runtime behavior of the processes. Thus, we created the *Runtime Skew Timeline* metric, so that the user can understand the relative timing behavior across the aligned traces.

Figure 3 depicts a visualization of the *Runtime Skew Timeline* metric for an example alignment. In the alignment both processes are equal except for four functions only executed in process B, shown as gaps in blue in the figure. These four functions delay process B by $20ms$ per function. Thus, the value for the metric decreases with time, because process B is delayed $20ms$ in each iteration.

4.4 Function Time Difference Table

We give relative timing information in the *Function Time Difference* table. To generate the table, we sum the time differences for all invocations of each event. This shows the overall time that was gained or lost within each event. In general, it is unlikely that a particular event will exclusively either gain or lose time over the execution. It is more likely that the duration of the event will be sometimes faster or slower compared to the aligned event in the other trace. This table clearly presents this information. For each event, it shows the number of times that it was faster (No. $+$) or slower (No. $-$) in trace A than in trace B, and the overall time gained ($\Delta+$) or lost ($\Delta-$) (See Table 3).

5 Case Studies

Here, we demonstrate the effectiveness of our approach with two real-world applications, AMG and ParaDiS. We conducted our experiments on a Linux cluster with 864 quad-socket AMD Quad-Core Opteron nodes. The 13,824 cores run at 2.3 GHz and each node has 32 GB RAM. We used the Intel compiler version 12.1 and the MVAPICH2 MPI library. Our analysis and comparison tool is built on top of the OTF trace library [11]. We use it to capture the traces and then we compare two OTF traces by applying our alignment methods. The resulting differential trace is written in OTF format. We visualize the trace in Vampir [5].

5.1 AMG2006

AMG2006 [1,6] is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. We compared the default version of AMG with an optimized version that performs less coarsening. This new version does more overall work, but avoids a lot of expensive communication. We compared both versions solving a Laplace problem using 64 processes on 4 nodes.

Figure 4 shows the unaligned traces for rank 0 (first process of the 64 MPI processes) from both versions of AMG. The numbers on the left side indicate the

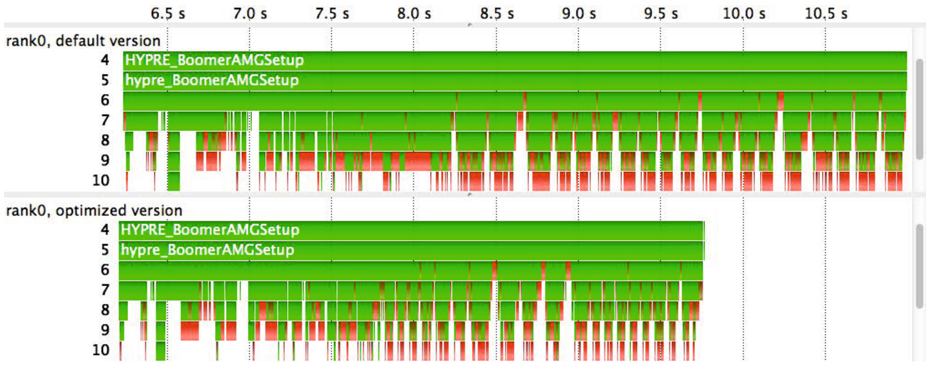


Fig. 4. Unaligned rank0 processes of the default and optimized AMG version

call level, i.e., sub-functions are drawn one level below the calling function, e.g., `HYPRE_BoomerAMGSetup` (level 4) calls `hypre_BoomerAMGSetup` (level 5). The optimized version is faster and finishes about 1.25 seconds earlier than the original version. Note that it is difficult to see the reasons for the performance differences from looking at the raw traces. Our analysis techniques make this task straightforward. In Figure 5, the first two timelines show the aligned traces for rank 0 for both versions. The third timeline shows the alignment state between both traces. The optimized version spends less time in initialization, identified by the blue gap area ($7.0s - 7.6s$) at the beginning of the run. Also, the optimized version saves work in each computation step, indicated by the repeating blue gap areas starting at the middle of the run ($9.0s - 12.7s$).

The *Dissimilarity Timeline* is represented by the set of colored bars in the fourth timeline of Figure 5. Blue/cold colors mean processes are equal, while red/hot colors indicate differences. The differences agree with the alignment of the rank 0 comparison. The red area indicates the big gap in the initialization and the green areas from the middle to the end indicate the smaller gaps occurring at each compute iteration. The differences are nearly the same throughout all processes, except for a few outliers indicated by horizontal green lines. The slightly higher dissimilarity in these processes is also highlighted using our *Similarity* metric (not shown). Most process pairs achieved similarity values between $0.45 - 0.53$, while the few outliers achieved similarity values between $0.28 - 0.40$.

The *Runtime Skew Timeline* is shown in the bottom two timelines in Figure 5. The upper graph depicts the runtime skew for rank 0 and the color-coded timeline depicts the runtime skew for all 64 process comparisons. The optimized version achieves a large speed gain in initialization. However, in the later stages of initialization ($7.5s - 8.5s$), it performs slower than the original version. Yet, the speed gain in the beginning overshadows this slow down. During the iterations of the main body of the code, the optimized version performs faster again. In this case the speed gains are not consistent from iteration to iteration. The gains level off in the middle of the execution and rise again at the end. Additionally, the color-coded *Runtime Skew Timeline* shows that this behavior is consistent

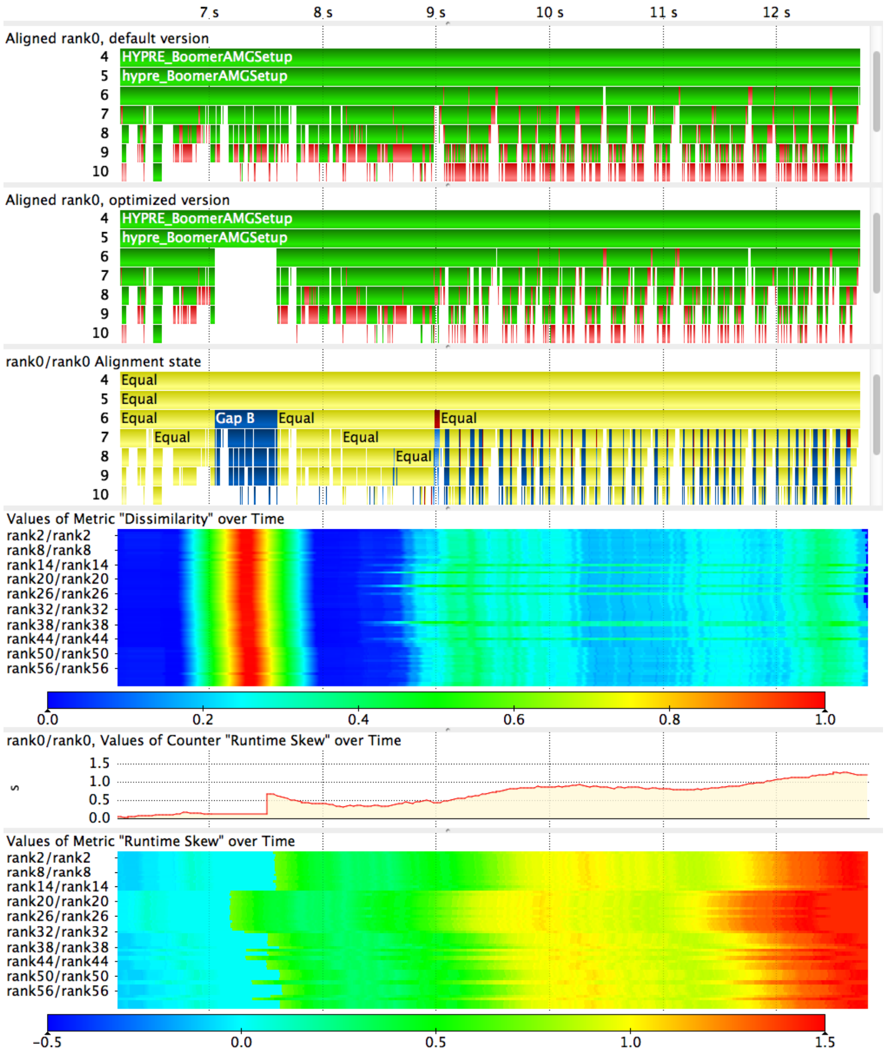


Fig. 5. Similarity and runtime skew between the default and optimized AMG versions

across all processes, but shows that performance on one node (second block of lines from the top) is slightly shifted.

5.2 ParaDiS

ParaDiS [3] models the dynamics of dislocation lines as they interact and move in response to the forces imposed by external stress and inter-dislocation interactions. We compared two versions of ParaDiS, v2.2.3 and v2.3.5.1, with release

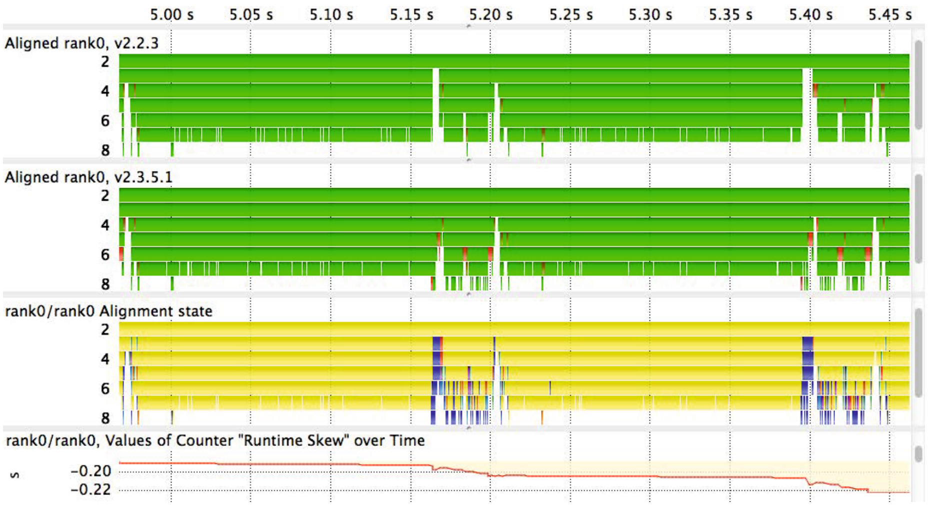


Fig. 6. Comparison of two iterations of ParaDiS versions 2.2.3 and 2.3.5.1

Table 3. ParaDiS function difference table for rank0

Function Name	No. +	Δ +	No. -	Δ -
MkTaylor	49817	41ms 542us	114230	71ms 586us
ComputeForces	315896	21ms 938us	46280	4ms 92us
MPI_Waitall	1035	5ms 274us	1178	3ms 517us

dates about two years apart. Version 2.3.5.1 includes bug fixes, improvements and corrections, as well as advanced load balancing. We ran both versions with 8 processes, solving the same example problem: "tests/fmm_8cpu.ctrl".

Figure 6 shows a comparison of two representative iterations of both versions. In the beginning of each iteration, the function structure is the same. However, at the end of each iteration, blue gap areas in the bottom timeline indicate new functions. These additional functions come from added capability in v2.3.5.1. The *Runtime Skew Timeline* at the bottom of Figure 6 shows that this change comes at the cost of higher runtime. The runtime difference for the whole traces is depicted in Figure 7. ParaDiS v2.3.5.1 runs consistently slower than v2.2.3.

Added functionality is not the only cause of differences. The bottom timeline in Figure 7 shows that dissimilarity varies across the processes. This variation is caused by the changes to the load balancer in v2.3.5.1. Figure 7 also shows that process pairs for ranks 1 and 6 are more similar than the other compared processes. This is reflected in the *Similarity* metric as well, with ranks 1 and 6 having similarity values of 0.40, while the other ranks have values of 0.29.

Table 3 is the *Function Time Difference* table for selected functions of rank 0. `MkTaylor` shows the most inconsistent behavior. Due to added functionality,

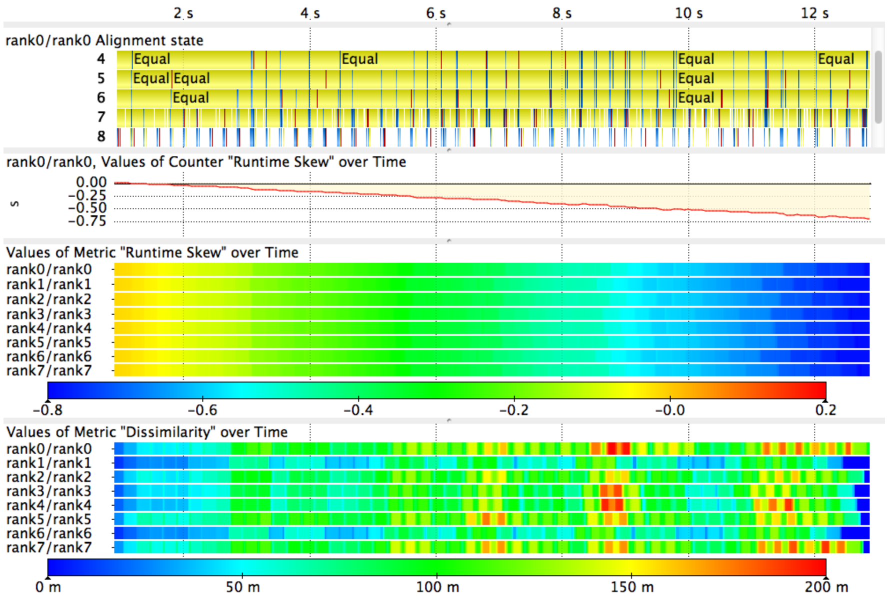


Fig. 7. Similarity and runtime skew analysis between ParaDiS versions 2.2.3 and 2.3.5.1

`MkTaylor` runs slower in v2.3.5.1 for 114,230 invocations, which adds $71ms\ 586\mu s$ to the execution time over v2.2.3. Yet, in 49,817 cases `MkTaylor` in v2.3.5.1 was faster than that of v2.2.3, reducing the execution time by $41ms\ 542\mu s$. In a normal profile these times would have been aggregated, resulting in a single potentially misleading reading of $30ms\ 044\mu s$ time lost in `MkTaylor` in v2.3.5.1.

6 Conclusions and Future Work

In this work we introduced a set of novel metrics and visualizations that help highlight the differences between traces. We applied our techniques to two applications and showed how they facilitate the identification of differences between code versions. Our metrics exposed differences that otherwise would have been hard or even impossible to find. Our techniques provide detailed insight into the performance of applications and will be of substantial help in optimizing them.

In the future, we plan to parallelize the alignment algorithm. Pairwise trace comparisons do not depend on other traces; thus, comparing runs with large process counts is embarrassingly parallel. Further, we plan to extend our approach to allow the alignments of more than two processes as well as to provide improved comparison visualizations and statistics. We will also evaluate the potential for automated detection of irregular performance behavior within traces.

References

1. ASC Sequoia Benchmark Codes, `https://asc.llnl.gov/sequoia/benchmarks/`
2. Intel Trace Analyzer, `http://software.intel.com/en-us/intel-trace-analyzer`
3. Arsenlis, A., Cai, W., Tang, M., Rhee, M., Oppelstrup, T., Hommes, G., Pierce, T.G., Bulatov, V.V.: Enabling Strain Hardening Simulations with Dislocation Dynamics. Modelling and Simulation in Materials Sci. and Eng. 15(6), 553 (2007)
4. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (2010)
5. Brunst, H., Hoppe, H.-C., Nagel, W.E., Winkler, M.: Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In: Alexandrov, V.N., Dongarra, J., Juliano, B.A., Renner, R.S., Tan, C.J.K. (eds.) ICCS 2001, Part II. LNCS, vol. 2074, pp. 751–760. Springer, Heidelberg (2001)
6. Gahvari, H., Baker, A.H., Schulz, M., Yang, U.M., Jordan, K.E., Gropp, W.: Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms. In: Proceedings of the International Conference on Supercomputing, ICS 2011, pp. 172–181 (2011)
7. Gusfield, D.: Algorithms on Stings, Trees, and Sequences. Computer Science and Computational Biology (1997)
8. Hauswirth, M.: Understanding Program Performance Using Temporal Vertical Profiles. PhD thesis, Boulder, CO, USA (2005)
9. Hirschberg, D.S.: A Linear Space Algorithm for Computing Maximal Common Subsequences. Commun. ACM 18(6), 341–343 (1975)
10. Karavanic, K.L., Miller, B.P.: Experiment Management Support for Performance Tuning. In: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (CDROM), Supercomputing 1997, pp. 1–10 (1997)
11. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the Open Trace Format (OTF). In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 526–533. Springer, Heidelberg (2006)
12. Krell. Open|SpeedShop (2012), `http://www.openspeedshop.org/`
13. Mohror, K., Karavanic, K.L.: Towards Scalable Event Tracing for High End Systems. In: Perrott, R., Chapman, B.M., Subhlok, J., de Mello, R.F., Yang, L.T. (eds.) HPCC 2007. LNCS, vol. 4782, pp. 695–706. Springer, Heidelberg (2007)
14. Needleman, S.B., Wunsch, C.D.: A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. Journal of Molecular Biology 48(3), 443–453 (1970)
15. Schulz, M., de Supinski, B.R.: Practical Differential Profiling. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 97–106. Springer, Heidelberg (2007)
16. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. Scientific Programming 16(2-3), 105–121 (2008)
17. Song, F., Wolf, F., Bhatia, N., Dongarra, J., Moore, S.: An Algebra for Cross-Experiment Performance Analysis. In: Proceedings of the 2004 International Conference on Parallel Processing, ICPP 2004, pp. 63–72 (2004)
18. Weber, M., Brendel, R., Brunst, H.: Trace File Comparison with a Hierarchical Sequence Alignment Algorithm. In: Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ISPA 2012, pp. 247–254 (2012)

Validation and Uncertainty Assessment of Extreme-Scale HPC Simulation through Bayesian Inference

Jeremiah J. Wilke, Khachik Sargsyan, Joseph P. Kenny, Bert Debusschere, Habib N. Najm and Gilbert Hendry

Sandia National Laboratories, Livermore, CA, USA
{jjwilke,ksargsy,jpkenny,bjdebus,hnnajm,ghendry}@sandia.gov

Abstract. Simulation of high-performance computing (HPC) systems plays a critical role in their development - especially as HPC moves toward the co-design model used for embedded systems, tying hardware and software into a unified design cycle. Exploring system-wide tradeoffs in hardware, middleware and applications using high-fidelity cycle-accurate simulation, however, is far too costly. Coarse-grained methods can provide efficient, accurate simulation but require rigorous uncertainty quantification (UQ) before using results to support design decisions. We present here SST/macro, a coarse-grained structural simulator providing flexible congestion models for low-cost simulation. We explore the accuracy limits of coarse-grained simulation by deriving error distributions of model parameters using Bayesian inference. Propagating these uncertainties through the model, we demonstrate SST/macro's utility in making conclusions about performance tradeoffs for a series of MPI collectives. Low-cost and high-accuracy simulations coupled with UQ methodology make SST/macro a powerful tool for rapidly prototyping systems to aid extreme-scale HPC co-design.

1 Introduction

Next-generation extreme-scale systems pose numerous challenges in exploiting massively increased parallelism and advancing in reliability, fault tolerance and power efficiency [1]. Despite intertwined design constraints, hardware and software are often developed independently. Only after next-generation systems are deployed does application software begin to adapt. These orthogonal development phases are not sustainable. System-wide tradeoffs in hardware and software must likely be considered in tandem to achieve exascale.

Simulation enables rapid evaluation of design considerations for both system architects and application programmers. Simulators span a range of fidelities. At low cost/low fidelity are analytic models which abstract away details, relying on parameterization to account for dynamic effects such as congestion [2]. Alternatively, high-fidelity, cycle-accurate treatments capture low-level details, but incur high computational cost [3,4,5]. A cost/accuracy compromise between these extremes is needed to rapidly and accurately prototype HPC systems.

Coarse-grained structural simulation can often achieve good accuracy with minimal cost [6]. However, some confidence interval on the approximate results must be established. This poses a difficult problem: how do you estimate simulation uncertainty for a machine that does not yet exist? In this work, we apply Bayesian inference to the coarse-grained simulator SST/macro (Structural Simulation Toolkit for Macroscale) [7]. We avoid the computationally infeasible task of expensive parameter sweeps on large (exascale) simulations to do uncertainty quantification (UQ). Instead, the simulator is exhaustively calibrated for small benchmarks against "correct" results from an actual machine or high-fidelity simulator. Once calibrated, we can predict confidence bounds for large simulations with manageable computational effort. Here we compare SST/macro simulations of modestly-sized (500 node) MPI collectives to Cray XE6 results. Propagating the UQ treatment through large simulations, we predict confidence bounds for SST/macro on large Cray XE6 runs. For future machines which do not yet exist, high-fidelity simulations can easily replace Cray XE6 results in the calibration step. SST/macro and corresponding UQ therefore provide flexible tools for exploring what-if scenarios in the HPC design space.

2 Related Work

The literature, dating back to at least the early 1990's, is replete with HPC simulators designed with various accuracy/cost tradeoffs. They span a spectrum from constitutive models based on simple bandwidth/latency formulas to cycle-

Table 1. Survey of analytic HPC simulators. Computation may be time-dependent trace, performance counter convolution (PerfCtr), direct execution, or coarse-grained.

Simulator	Ref no.	On/Off-line	Computation	Congestion	Language
LogGOPS	[2]	Trace	Model	Yes	DSL (GOAL)
BSIM	[6]	On-line	Coarse Model	Yes	Native
Mambo/Seshat	[8]	On-line	Cycle-Acc	No	Native
PSINS	[9]	Trace	Time-dependent	Yes	n/a
MPI-SIM	[10]	On-line	Direct	No	Native
Dimemas	[11]	Trace	PerfCtr	No	n/a
WARPP	[4]	Trace	PerfCtr	Yes	Native

Table 2. Survey of structural HPC simulators. Computation may be time-dependent trace, performance counter convolution (PerfCtr), or coarse-grained model.

Simulator	Ref no.	On/Off-line	Computation	Network	Language
BigSim	[12,13]	Both	PerfCtr/Model	Packet	Native
SIMGRID	[14,15]	Both	PerfCtr	Flow	Native
MARS	[16]	Trace	Time	Packet	n/a
MPI-NeTSim	[17]	On-line	Direct	Packet	Native
PACE	[18]	Both	Abstract	Abstract	DSL (CHIP^{3}S)
SST/macro	[19]	Both	PerfCtr/Model	Packet/Flow	Native

accurate simulation. We attempt a concise summary. Features for some (but not all) literature examples are given in Tables 1 and 2.

On-line simulators usually emulate an API such as the MPI [20] and link to application code, intercepting function calls to estimate elapsed time. Off-line simulators instead perform post-mortem analysis of application traces. Computation time modeling in on-line simulators span full cycle-accurate simulation [8] to simple coarse-grained analytic models [19]. SST/macro focuses on broader, system-level experiments, using coarse-grained models. For off-line simulation, time-dependent traces collect the compute time between communication events, but are limited to simulating a single architecture. Time-independent traces instead collect architecture-independent hardware counters that can be convolved with machine-parameters to replay on different node architectures [11,15]. For estimating communication time, simulators can generally be either structural or use a fixed analytic function. Structural simulators simulate discrete events on each switch and link as messages traverse the network. Analytic functions often ignore contention, but some formulas do incorporate congestion [2].

The closest simulators to SST/macro are BSIM, BigSim, and SIMGRID. While the BSIM work in [6] also emphasizes coarse-grained modeling, the communication is modeled via an analytic function. BigSim is structural and large simulations incorporating congestion and statistical computation modeling have been performed [21]. However, BigSim performs packet modeling via its BigNetSim module while SST/macro also provides flow-based congestion models (Section 4). Like SST/macro, BigSim can be an on-line network emulator, performing full execution of a parallel code, or on-line simulator estimating elapsed time without actually performing the computation [13]. SIMGRID is also designed primarily as an on-line simulator with various bindings (including MPI), but it is based on a flow congestion model [14].

3 SST/macro

SST/macro includes both a tracing utility (DUMPI) and MPI bindings for simulating native Fortran/C/C++ applications on-line. On-line simulation provides much greater flexibility in scaling to arbitrary machines and problem sizes. SST/-macro is both an emulator and a simulator. Payloads in matching send/recv calls are delivered, producing computation identical to a full MPI library. With minimal modification, any existing MPI code can be compiled as-is and run within SST/macro. To reduce memory overhead, payloads can be ignored, running SST/macro purely as an MPI simulator.

In reaching extreme-scale, both communication and computation must be coarse-grained. Compute-intensive code must be wrapped in function calls that can be intercepted by SST/macro to estimate elapsed time via performance models rather than actually computing. The “control-flow” portions of the code are usually lightweight and can be unmodified.

SST/macro is a structural simulator. Messages are explicitly simulated passing through NICs, switches and links. SST/macro provides a large set of default network topologies, including torus, dragonfly, or fat tree. It also provides flexibility in routing algorithms, congestion models, and RDMA vs TCP protocols.

4 Congestion Models

Both packet and flow models are common for network discrete event simulation. Packet models perform flow-control for an entire packet, ignoring flit-level details. Though packet size is often small in HPC systems (≈100 bytes for Hopper Cray XE6), coarse-grained models often use large packets (*e.g.* 1 KB), leading to a serialization latency error. In addition, packet simulations allow only one packet per physical link. In a real system, 1 KB of data might be multiplexed, containing 500 B from two distinct messages.

A common alternative is the flow model, which treats messages as a fluid flow between network endpoints. Network congestion is solved as a fluid mechanics problem based on sharing bandwidth between flows. Flow update events are generated when new flows begin or stop on congested links. Without congestion, the flow model is a constant cost regardless of message size. The flow model can rapidly become expensive with congestion, however. For packet simulation, discrete events are intrinsically local within a link or router. In contrast, flow congestion is not locally isolated. Flow updates are propagated across competing flows causing bandwidth changes on far-away links - a "ripple effect" that rapidly increases computational cost [22]. For exascale, this is problematic.

In SST/macro we employ a hybrid approximation that employs both discretized, packet-like modeling with a fluid-like approach. Messages are discretized into packets, but packets are arbitrated as a flow, allowing messages to be multiplexed on a shared link. Because messages are "packetized," flow updates occur only within a link and do not propagate into a ripple effect.

5 Hopper Cray XE6 and Validation Tests

We examine Hopper, a Cray XE6 with the Gemini interconnect [23]. Nodes use AMD MagnyCours processors with a non-uniform memory (NUMA) architecture with four dies each with six cores. Here we focus on the interconnect. The network interface controller (NIC) is connected via HyperTransport to the L3 cache of one die. The Gemini NIC provides two separate paths. The fast memory access (FMA) path is optimized for low-latency 8- to 64-byte puts/gets. FMA is intended for small messages as it still requires source-side synchronization. Special short message (SMSG) mailboxes can also be configured for very-low latency transfers on FMA. The block transfer engine (BTE) has higher latency, but provides fully asynchronous RDMA and is optimized for transferring large blocks directly to memory. The Gemini NIC shares a 48-port tiled YARC router with an adjacent (twin) Gemini NIC. Each router-router link provides 2.9 GB/s with two links in the X,Z directions. Packets are arbitrated both at the tile switch and output port. We therefore have four congestion points, each with associated latency and bandwidth: FMA, BTE, router switch, and router outport.

We limit experiments to rendezvous RDMA transactions involving the BTE. RDMA posts are explicitly coded in the native Cray GNI interface, ensuring that dimension-order, minimal routing is used. This not only removes adaptive routing as a variable but also worsens network contention, providing a more stringent test of our congestion models. Discrepancies between simulation and Hopper trials can therefore be mainly attributed to the coarse-graining rather than subtle differences between Hopper and SST/macro in, *e.g.*, use of eager buffers, adaptive routing, or RDMA memory registration.

We focus on MPI collectives (MPI_Allgather, MPI_Scatter, and MPI_Gather) to validate communication modeling. We avoid mixed computation-communication modeling since less insight would be gained from a compute-bound application. Collectives range from $N(node) = 128 - 4096$. We assume OpenMP parallelism within node and therefore report $N(node)$, not $N(cpu)$, which ranges from $N(cpu) = 3072 - 98304$. Collectives were reimplemented in Cray GNI to ensure that the SST/macro MPI and Hopper use the same algorithm and topology. MPI_Allgather uses a ring pattern while MPI_Scatter and MPI_Gather use a tree algorithm [24]. Simulation is performed on-line, intercepting MPI calls in a C++ code. No code modifications - only changes to include path and linkage flags - were made when compiling for Hopper or the simulator.

The Cray XE6 defines a potentially high-dimensional parameter space (Table 3). Small ping-pong benchmarks were performed to estimate bandwidth and latencies. Unexpectedly, the maximum bandwidth for a single message was not equal to the total link bandwidth. With minimal routing, only 1.8 GB/s was achieved despite a theoretical peak of 2.9 GB/s. Max BW (Table 3) therefore apples to single messages while Link BW affects congestion. In ping-pong benchmarks with congestion, large variance was observed for total link bandwidth (Table 3). For other parameters, literature values are given [23]. An initial sensitivity analysis (see Section 7) was performed to indicate important parameters. Changes in SMSG latency had little effect on simulated runtime, and we therefore fix it at a nominal value. Only Max BW, Link BW, and Inj BW were relevant for MPI_Scatter and MPI_Gather. For MPI_Allgather, Max BW,

Table 3. Parameters used in SST/macro models of MPI collectives

Parameter	Abbreviation	Nominal Value
Maximum single message bandwidth	Max BW	1.8 GB/s
Total link bandwidth	Link BW	2.0 – 2.9 GB/s
Hop latency	Hop Lat	100 ns
RDMA injection latency	Inj Lat	0.6 μs
RDMA injection bandwidth	Inj BW	7 GB/s
SMSG injection latency		0.5 μs
SMSG injection bandwidth		8 GB/s

Link BW, Inj Lat, and Hop Lat had significant effects on simulated runtime. Rather than 7-dimensions, we therefore solve more modest 3- and 4-dimensional problems.

6 Uncertainty Quantification

Model validation is viewed from a probabilistic perspective. We consider latency/bandwidth parameters as marginal probability distributions with certain values having a higher probability of being "correct." While uncertainty in model parameters often stems from experimental noise, here uncertainty derives largely from model imperfections. A narrow distribution indicates a tightly constrained parameter we are "certain of." A broad distribution indicates several parameter values of equivalent accuracy. In SST/macro, any *single* data point can be reproduced by tweaking parameters. Only an exact model with exact parameters reproduces *all* data points. In coarse-grained models, different effective bandwidths provide better results depending on the MPI collective $N(node)$ and buffer size. The probability distributions reflect this *model* uncertainty.

Given a model (SST/macro) and parameters (latency, bandwidth), we infer probability distributions for the parameters in light of data from Hopper tests described in Section 5. Then, armed with this data-informed, probabilistic representation of inputs, we propagate them through the SST/macro model and its associated statistical discrepancy model to produce *predictive* probability distributions for simulation results [25]. The discrepancies between simulation and Hopper trials are associated both with the background noise present during tests and SST/macro model *imperfection*. Since exhaustive UQ is not possible on extreme-scale simulations, we derive predictive probability distributions on small simulations assuming that they inform error estimates on larger simulations.

Model parameter inference relies on Bayes' formula [26]

$$p(\boldsymbol{\lambda}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\lambda})p(\boldsymbol{\lambda}), \tag{1}$$

which relates the *posterior* probability distribution $p(\boldsymbol{\lambda}|\mathcal{D})$ to the *prior* distribution $p(\boldsymbol{\lambda})$ for the input parameter vector $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_d)$ in light of experimental data $\mathcal{D} = \{T_{n,r}\}_{1\leq n\leq N, 1\leq r\leq R}$. Here we test three different MPI collectives where N labels the number of nodes $N(node)$ and buffer size. For MPI_Allgather, we have $N = 5 \times 4 = 20$ tests corresponding to $N(node) =$ 128, 192, 256, 384, 512 and buffer sizes 8 KB, 16 KB, 24 KB, and 32 KB. For MPI_Scatter and MPI_Gather, we have $N = 7 \times 4 = 28$ tests with $N(node) =$ 1024 and 2048 also included. Here the experimental data are Hopper runtimes with $R = 20$ replicas for each case.

The prior distribution $p(\boldsymbol{\lambda})$ encapsulates any prior knowledge about the parameters. We use uniform prior distributions across the range of "realistic" values for bandwidth/latency parameters. A priori there is no bias towards any specific value as long as it belongs to the given range. The key quantity in Bayes' formula (1) is the *likelihood* function $L_{\mathcal{D}}(\boldsymbol{\lambda}) = p(\mathcal{D}|\boldsymbol{\lambda})$ that measures an experiment's goodness-of-fit to model predictions. We use a Gaussian model (2) for this discrepancy, relying on the Central Limit Theorem, as both background noise in the system and the uncertainties associated with model deficiencies are contributed by many unknown sources.

$$\log L_{\mathcal{D}}(\boldsymbol{\lambda}) = -\frac{NR}{2}\log(2\pi\sigma^2) - \sum_{n=1}^{N}\sum_{r=1}^{R}\frac{(T_{n,k} - M_n(\boldsymbol{\lambda}))^2}{2\sigma^2} \tag{2}$$

Here $M_n(\cdot)$ is the model (SST/macro simulation time) of the n-th test. The variance parameter, σ^2 can either be fixed or inferred together with $\boldsymbol{\lambda}$ as a hyperparameter. External noise can result physically from many sources, including random differences in congestion arbitration, OS jitter, or small bandwidth variations with time. Hopper is also a shared machine, and, despite allocating most of the nodes, congestion from other jobs was sometimes observed. These points were usually severe outliers, though, and easily discarded.

The posterior probability distribution $p(\boldsymbol{\lambda}|\mathcal{D})$ from (1) is generally difficult to compute due to the high-dimensionality of the vector $\boldsymbol{\lambda}$. We therefore applied the adaptive Markov chain Monte Carlo (AMCMC) technique [27,28]. AMCMC travels the input parameter space of $\boldsymbol{\lambda}$, accepting or rejecting samples with probabilities according to the goodness-of-fit (2) and the prior distribution $p(\boldsymbol{\lambda})$ until a converged $p(\boldsymbol{\lambda}|\mathcal{D})$ is obtained. The SST/macro model, $M_n(\boldsymbol{\lambda})$ must be simulated at each AMCMC step. In addition, each step depends on the previous result, forcing AMCMC to proceed in serial. Each simulation takes only $5 - 60$ minutes, but many samples (thousands) are needed for a converged $p(\boldsymbol{\lambda}|\mathcal{D})$. We therefore prepared an analytic, polynomial $S_n(\boldsymbol{\lambda})$ that accurately estimates each SST/Macro model $M_n(\boldsymbol{\lambda})$. Multivariate Legendre polynomials $\Psi_k(\cdot)$ are used to approximate the model behavior as a function of input bandwidth/latency parameters, $\boldsymbol{\lambda}$, $S(\boldsymbol{\lambda}) = \sum_{k=0}^{K} c_k \Psi_k(\boldsymbol{\xi})$ in terms of rescaled parameters, $\boldsymbol{\xi}$, in the range $[-1, 1]$. Surrogates are constructed by performing a large but manageable parameter sweep, which "embarrassingly" parallelizes SST/macro across independent runs. We used a third order basis set and found the polynomial fit parameters c_k with a 1296-point (Allgather) and 216-point (Scatter/Gather) parameter sweep. The serial AMCMC is then quickly performed using the inexpensive but still accurate surrogate $S(\boldsymbol{\lambda})$. For 100 validation points, the surrogate was within approximately 1% relative error to the true model.

The polynomial surrogate allows efficient *global* sensitivity analysis of the model output. The variance-based sensitivity indices, or Sobol indices [29,30],

$$Se_i = \frac{\sum_{k\in I_i} c_k^2 \langle \Psi_k^2 \rangle}{\sum_{k=0}^{K} c_k^2 \langle \Psi_k^2 \rangle}, \text{ for } i = 1, \ldots, d, \tag{3}$$

represent the fraction of output variance due to the i-th input. Here I_i denotes basis terms involving the i-th input only. This decomposes the simulation uncertainty into contributions from individual latency and bandwidth terms.

To estimate uncertainties for simulations outside the calibration range - large $N(node)$ - we simulate 100 parameter combinations chosen randomly according to the parameter posterior distribution. Better converged results can be obtained with more simulations, but we suggest 100 is a good compromise for expensive (exascale) runs. Propagating input parameter distributions through the model, together with the model discrepancy size σ, yields the posterior predictive distribution [25], which provides a semi-quantitative error estimate for the large SST/macro simulation. After exhaustive calibration on small simulations, we can bracket the error on a large simulation by repeating only a modest number of times. We apply this technique on a simulation with 65,000 nodes.

7 Results

The simulated vs. actual runtimes are demonstrated in Figs 1–3 using the nominal parameters from Table 3. In general, SST/macro correctly reproduces the performance trends and even the correct congestion behavior.

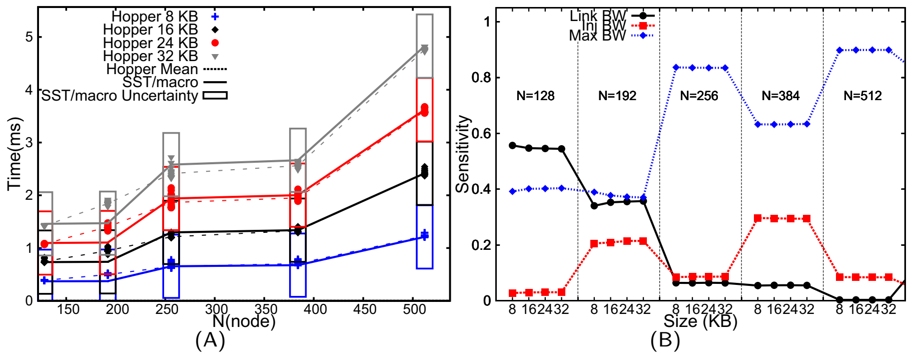


Fig. 1. (A) Hopper runtimes (N=20 replicates) and simulation runtimes for MPI_Scatter for $N(node) = 128 - 512$ (B) Sensitivity analysis of MPI_Scatter

We first consider MPI_Scatter (Figure 1). To isolate effects for individual nodes, we report the time needed by rank 0 (root) rather than the whole collective. The simulation times are shown against a scatter plot of 20 Hopper replicates for varying buffer sizes and $N(node)$. In general, the simulated runtimes lie within the observed range of Hopper runtimes. The sensitivity analysis partitions the total uncertainty into a percent contribution from each parameter. The line slope closely matches an ideal (uncongested) performance model with bandwidth 1.8 GB/s (Max BW), indicating only mild network contention. Max BW therefore shows a higher sensitivity value than Link BW. All data originates from the root node and performance is therefore partially limited by injection. The corresponding sensitivity value is therefore significant but small.

For MPI_Gather (Figure 2), excellent agreement is again observed. In contrast to MPI_Scatter, Inj BW has a negligible sensitivity value. Even though the root receives all data, the receives are staggered over a longer period of time. Again, Max BW dominates most cases. However, for $N = 192$, heavy congestion occurs on some links, making Link BW more significant.

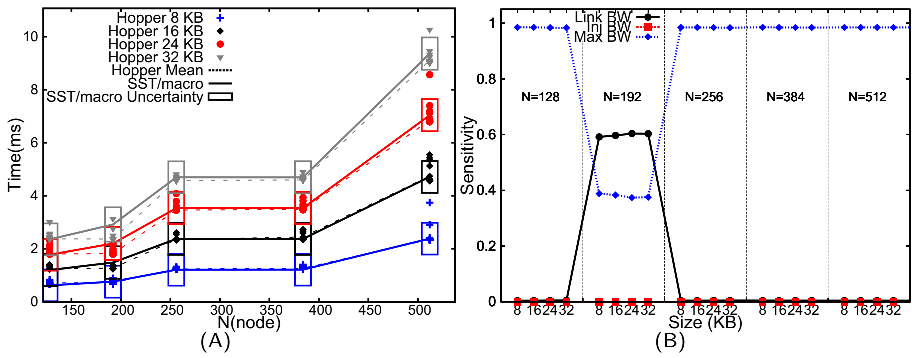


Fig. 2. (A) Hopper runtimes (N=20 replicates) and simulation runtimes for MPI_Gather for $N(node) = 128 - 512$ (B) Sensitivity analysis of MPI_Gather

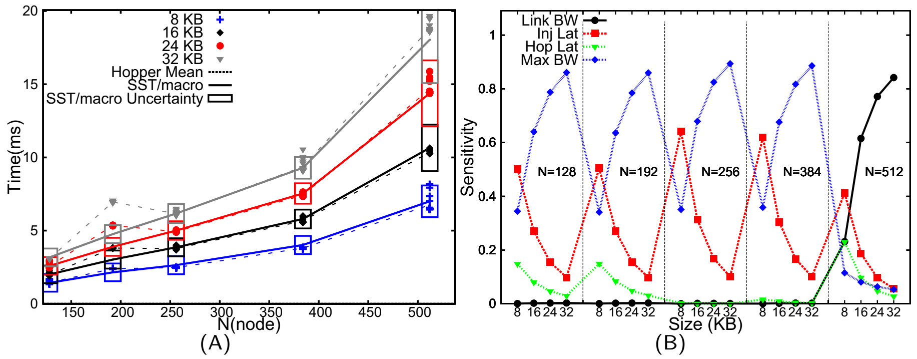


Fig. 3. (A) Hopper runtimes (N=20 replicates) and simulation runtimes for MPI_Allgather for $N(node) = 128 - 512$ (B) Sensitivity analysis of MPI_Allgather

The ideal performance model for the ring MPI_Allgather (Figure 3) is linear in $N(node)$ since the algorithm involves $N(node)-1$ neighbor exchanges of data of constant size [24]. Many medium-sized messages are sent, making both latency and bandwidth terms important. Sensitivity analysis again matches the expected physics since Max BW, Inj Lat, and Hop Lat all have significant values. As the buffer size increases, uncertainty shifts to the bandwidth terms. At $N = 512$, the curve slope in Figure 3A shifts upward, suggesting network contention starts to contribute - an effect reproduced by SST/macro. The sensitivity value for Link BW therefore becomes large.

From the extensive calibration for $N(node) \leq 512$, we can now propagate uncertainties. Validation jobs were run on Hopper and in SST/macro (Figure 4) outside the calibration range ($N = 1024$ for MPI_Allgather; $N = 4096$ for MPI_Scatter and MPI_Gather). SST/macro error bars for 90% confidence intervals are derived from the posterior predictive distributions (Section 6). For MPI_Allgather (Figure 4A), despite heavy congestion, SST/macro shows good agreement. While SST/macro underestimates slightly for large buffer sizes, the

Hopper results still lie within the SST/macro error bounds. Calibrating parameter uncertainties with modest benchmarks therefore works well in bracketing errors at larger problem sizes. For MPI_Gather (Figure 4B), the agreement is particularly good. The error bounds are too small to appear as distinct lines, suggesting (a priori) the simulation should be essentially exact.

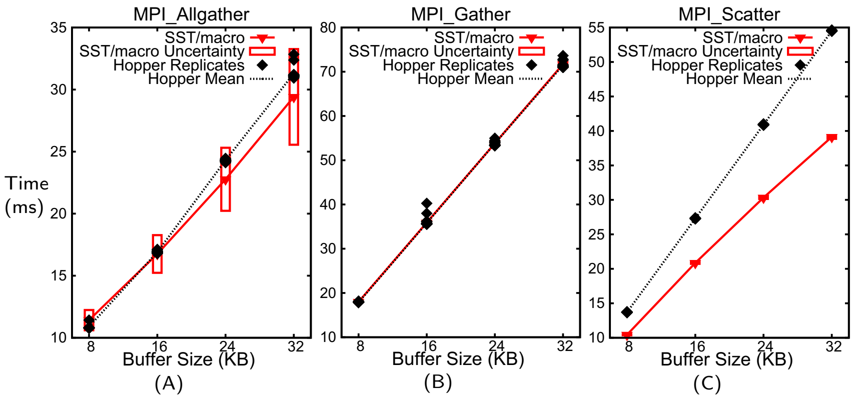


Fig. 4. Hopper runtimes (N=20 replicates) and simulation runtimes with propagated 90% confidence intervals. (A) MPI_Allgather at $N(node) = 1024$. (B) MPI_Gather at $N(node) = 4096$. (C) MPI_Scatter at $N(node) = 4096$.

For MPI_Scatter (Figure 4C), the agreement is quite poor, especially given the good agreement observed in calibration. Given the node layout on the Hopper torus, large messages sent from the root do not traverse the same links and should see little congestion. Small benchmarks suggest BTE (RDMA injection) performance may deteriorate with many simultaneous RDMA transactions. Our model assumes constant injection bandwidth, but better models may show gradually decreasing bandwidth with high offered load. $N = 4096$ may therefore represent a crossover to injection limited performance. External noise from other jobs may also contribute. Given the good agreement in calibration, estimated error bars are small (again barely visible). While MPI_Gather and MPI_Allgather show successes, MPI_Scatter shows the potential pitfall. Congestion effects not included in the calibration do not propagate to the uncertainty estimate. MPI_Scatter calibration did not stress RDMA injection, leaving error estimates too optimistic. Our UQ procedure is therefore not completely black-box since benchmarks should be prudently constructed to stress the coarse-grained models.

We repeated the uncertainty propagation for a theoretical machine with 65,536 nodes. Using the Hopper calibration, we propagate uncertainties for MPI_Gather. The simulation assumes a scaled-out version of Hopper, posing the question: if interconnects do not improve, what performance can be expected for MPI_Gather at larger scales? Running the 100 parameter samples produces Table 4. Given the excellent agreement observed in calibration, we predict SST/macro can estimate MPI_Gather times very precisely.

Buffer Size (KB)	Simulation Time (ms)
8	287.1 ± 0.97
16	574.0 ± 1.93
24	860.9 ± 2.90
32	1147.9 ± 3.87

Table 4. Simulated runtimes for single MPI_Gather on a theoretical scaled-out Hopper/XE6 with $N(node) = 65,536$ including estimated 90% confidence intervals

8 Conclusions

We have presented SST/macro, a coarse-grained simulator designed to explore macroscale performance tradeoffs in the co-design space of hardware, middleware and applications. High accuracy is observed for simulation of MPI collectives - even for runs exhibiting heavy network congestion. Most importantly, we have presented a rigorous UQ methodology based on Bayesian inference. The critical UQ step is calibrating simulation input parameter uncertainties against "correct" results from either actual machines or high-fidelity simulators. SST/macro can therefore simulate both existing and future extreme-scale machines. Although addition of parallel discrete event simulation to SST/macro is underway, the UQ methods are already highly parallel. Given the ease of compiling native codes directly into the simulator, we suggest SST/macro as a flexible tool in supporting co-design of extreme scale systems.

Acknowledgments. This work was supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing program and the DOE Office of Advanced Scientific Computing Research. SNL is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the DOE NNSA under contract DE-AC04-94AL85000.

References

1. Dongarra, J., Beckman, P., et al.: The International Exascale Software Project Roadmap. Int. J. High Perform. Comput. Appl. 25, 3 (2011)
2. Hoefler, T., Schneider, T., et al.: LogGOPSim: Simulating Large-Scale Applications in the LogGOPS Model. In: 19th ACM Int. Symp. On High Perform, p. 597 (2010)
3. Underwood, K.D., Levenhagen, M., et al.: Simulating Red Storm: Challenges and Successes in Building a System Simulation. In: Int. Parallel Distrib. Proc. Symp, p. 1 (2007)
4. Hammond, S.D., Mudalige, G.R., et al.: WARPP: A Toolkit for Simulating High-Performance Parallel Scientific Codes. In: 2nd Int. Conf. On Simul. Tools Tech., p. 1 (2009)
5. Rodrigues, A.F., Hemmert, K.S., et al.: The Structural Simulation Toolkit. ACM Sigmetrics Perform. Eval. Rev. 38, 37 (2011)
6. Susukita, R., Ando, H., et al.: Performance Prediction of Large-Scale Parallel System and Application Using Macro-Level Simulation. In: Supercomputing (2008)
7. Janssen, C.L., Adalsteinsson, H., et al.: A Simulator for Large-Scale Parallel Computer Architectures. Int. J. Distrib. Syst. Technologies 1, 57 (2010)
8. Leòn, E.A., Riesen, R., et al.: Instruction-Level Simulation of a Cluster at Scale. In: Supercomputing, p. 1 (2009)

9. Tikir, M.M., Laurenzano, M.A., Carrington, L., Snavely, A.: PSINS: An open source event tracer and execution simulator for MPI applications. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 135–148. Springer, Heidelberg (2009)
10. Adve, V.S., Bagrodia, R., et al.: Compiler-optimized Simulation of Large-Scale Applications on High Performance Architectures. J. Parallel Distrib. Comput. 62, 393 (2002)
11. Snavely, A., Carrington, L., et al.: A Framework for Performance Modeling and Prediction. In: Supercomputing, p. 1 (2002)
12. Zheng, G., Gunavardhan, K., et al.: BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In: 18th Int. Parallel Distrib. Proc. Symp., p. 26 (2004)
13. Zheng, G., Wilmarth, T., et al.: Simulation-Based Performance Prediction for Large Parallel Machines. Int. J. Parallel Program. 33, 183 (2005)
14. Casanova, H., Legrand, A., et al.: SimGrid: A Generic Framework for Large-Scale Distributed Experiments. In: Tenth Int. Conf. On Comput. Model. Simul, p. 126 (2008)
15. Desprez, F., Markomanolis, G.S., et al.: Improving the Accuracy and Efficiency of Time-Independent Trace Replay. In: 3rd Int. Workshop On Perform. Model. Benchmarking Simul. High Perform. Comput. Syst, PMBS 2012 (2012)
16. Denzel, W.E., Li, J., et al.: A Framework for End-to-End Simulation of High-Performance Computing Systems. In: 1st Int. Conf. On Simul. Tools Tech. Commun. Networks Syst. & Workshops, p. 1 (2008)
17. Penoff, B., Wagner, A., et al.: MPI-NeTSim: A Network Simulation Module for MPI. In: 15th Int. Conf. On Parallel Distrib. Syst. (ICPADS 2009), p. 464 (2009)
18. Nudd, G.R., Kerbyson, D.J., et al.: Pace–A Toolset for the Performance Prediction of Parallel and Distributed Systems. Int. J. High Perform. Comput. Appl. 14, 228 (2000)
19. Janssen, C.L., Adalsteinsson, H., et al.: Using Simulation to Design Extreme-Scale Applications and Architectures: Programming Model Exploration. ACM Sigmetrics Perform. Eval. Rev. 38, 4 (2011)
20. Gropp, W., Lusk, E.L., et al.: Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd edn. The MIT Press, Cambridge (1999)
21. Bhatele, A., Jain, N., et al.: Avoiding Hot-spots on Two-level Direct Networks. In: Supercomputing, p. 1 (2011)
22. Figueiredo, D.R., Liu, B., et al.: On the Efficiency of Fluid Simulation of Networks. Comput. Networks 50, 1974 (2006)
23. Alverson, R., Roweth, D., et al.: The Gemini System Interconnect. In: IEEE 18th Ann. Symp. On High Perform. Interconnects (HOTI), p. 83 (2010)
24. Thakur, R., Gropp, W.D.: Improving the performance of collective operations in MPICH. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840, pp. 257–267. Springer, Heidelberg (2003)
25. Lynch, S.M., Western, B.: Bayesian Posterior Predictive Checks for Complex Models. Sociological Methods Research 32, 301 (2004)
26. Sivia, D.S.: Data Analysis: A Bayesian Tutorial. Oxford Science, Oxford (1996)
27. Gamerman, D.: Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference. Chapman and Hall, London (1997)
28. Haario, H., Saksman, E., et al.: An Adaptive Metropolis Algorithm. Bernoulli 7, 223 (2001)
29. Sobol, I.M.: Sensitivity Estimates for Nonlinear Mathematical Models. Math. Model. Comput. Exp. 1, 407 (1993)
30. Saltelli, A., Tarantola, S., et al.: Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models. John Wiley and Sons, Chichester (2004)

Dynamic Thread Pinning for Phase-Based OpenMP Programs

Abdelhafid Mazouz[1], Sid-Ahmed-Ali Touati[2], and Denis Barthou[3]

[1] University of Versailles Saint-Quentin-en-Yvelines, France
[2] University of Nice Sophia Antipolis, France
[3] University of Bordeaux, France

Abstract. Thread affinity has appeared as an important technique to improve the overall program performance and for better performance stability. However, if we consider a program with multiple phases, it is unlikely that a single thread affinity produces the best program performance for all these phases. If we consider the case of OpenMP, applications may have multiple parallel regions, each with a distinct inter-thread data sharing pattern. In this paper, we propose an approach that allows to change thread affinity dynamically (thread migrations) between parallel regions at runtime to account for these distinct inter-thread data sharing patterns. We demonstrate that as far as cache sharing is concerned for SPEC OMP01, not all the tested OpenMP applications exhibit a distinct phase behavior. However, we show that while fixing thread affinity for the whole execution may improve performance by up to 30%, allowing dynamic thread pinning may improve performance by up to 40%. Furthermore, we provide an analysis about the required conditions to improve the effectiveness of the approach.

Keywords: OpenMP, thread level parallelism, thread affinity, multicores.

1 Introduction

Multicore architectures are nowadays the state of the art in the industry of processor design for desktop and high performance computing. With this design, multiple threads can run simultaneously exploiting thread level parallelism and consequently, improve overall program performance of the system. Unfortunately, the growing gap between processor performance and memory performance has led manufacturers to propose highly hierarchical machines to alleviate this problem. The common architectural design consists of two or more cores sharing some levels of memory caches, memory buses, prefetchers or memory nodes. As the memory hierarchy of these machines is becoming increasingly complex, achieving better program performance of parallel applications on these modern architectures is more challenging.

A hierarchy of memory caches allows to exploit data sharing between threads running on such platforms. Of course, to exploit that, a multi-threaded application has to meet two conditions. First, threads have to access common data. Second, the reuse distance has to be short enough to effectively exploit these shared data across multiple threads. In this context, thread affinity has appeared as an important technique to exploit data sharing and to accelerate program execution times [15,12,6,18,9]. Another advantage of fixing thread affinity is for better performance stability [9].

Using thread affinity enhances inter-thread data locality. If two threads make extensive accesses to common data in memory, it is better to place them on cores sharing the same L2/L3 cache, or the same NUMA node. Doing so, we decrease the number of cache misses. Indeed, if one thread brings a data element to some cache level, the second thread accessing the same data element will avoid unnecessary cache misses. Furthermore, binding threads to cores by considering the machine architecture may help hardware prefetching of frequently accessed shared regions.

In this paper, we focus on cache sharing, and study the impact of a phase-based or dynamic thread pinning. It it based on the *control flow graph* (CFG) of a parallel execution in a given program. A *node* in this CFG can be defined using different granularities: a sequence of some instructions, a function call, etc. Since OpenMP programs may implement multiple parallel regions which are called multiple times iteratively, we consider the CFG as a graph representing a sequence of calls to distinct parallel regions in an OpenMP program. This also means that we define an OpenMP phase as the execution of a parallel OpenMP region. In this study, we consider that the parallel region represents a good trade-off between a better sharing patterns identification accuracy and a low overhead incurred by a smaller number of thread migrations.

A previous research study [9] has showed that fixing thread affinity during the whole execution provides better performance improvements on NUMA machines than on SMP ones. Nevertheless, we think that it is possible to further enhance the performance gain using thread affinity by exploiting phase-based behavior in OpenMP programs. We made an extensive performance evaluation of multiple thread pinning strategies on four distinct machines. Among them, four strategies are application dependent: they rely on the characteristics (data sharing) of the application, and they set a distinct thread placement for each parallel region. Three strategies are application independent: they apply the same thread affinity for the whole execution and for each application. We show that dynamic thread pinning can improve performance by up to 40% compared to the Linux OS scheduler. Furthermore, we show the amount of inter-thread data sharing and the granularity of the parallel regions are main factors influencing most the effectiveness of dynamic thread pinning.

This article is composed as follows. Section 2 a synthetic example aiming to show the effectiveness of using per-parallel regions thread affinity within OpenMP programs. Section 3 presents the method we use to compute a distinct thread affinity for each parallel region. Section 4 describes our experimental setup and methodology (test machines, running methodology, statistical significance analysis). Section 5 shows our experimental results and analysis. Related work is presented in Sect. 6, then we conclude.

2 Motivation and Problem Description

We define an OpenMP phase as a unique and distinct OpenMP parallel region. In OpenMP, each structured code started by the construct `#pragma omp parallel` in C/C++ or `!$omp parallel` in Fortran is a new parallel region. That is, each OpenMP parallel region translates into distinct OpenMP phases.

To illustrate the benefit of changing thread pinning between consecutive OpenMP parallel regions, we use a synthetic micro-benchmark. This benchmark implements two

OpenMP parallel regions, each with a distinct sharing pattern. The benchmark uses a single large rectangular (the width is much greater than the height) matrix which is subdivided into equal parts among all the intervening threads. The benchmark is designed so that in parallel region 1, data sharing is between (T_1, T_5), (T_2, T_6), (T_3, T_7) and (T_4, T_8) thread pairs. Similarly, in parallel region 2, data sharing is between (T_1, T_2), (T_3, T_4), (T_5, T_6) and (T_7, T_8) thread pairs. Cache lines sharing between threads is implemented by allowing for each pair of threads to access common cells from the portion of the array that has been assigned to them. For each assigned portion from the array, each thread performs simple computations like additions and multiplications.

Each thread accesses to the same amount of data. Moreover, the amount of shared data blocks is equal between each pair of threads, of course with different sharing patterns across the two parallel regions. To analyze how the amount of inter-thread data sharing can influence the effectiveness of allowing thread migrations across parallel regions, we fix the same inter-thread data sharing in the first parallel region, and vary the amount of data sharing in the second. We consider in these experiments the 0%, 25%, 75% and 100% amounts of data sharing cases in the second parallel region.

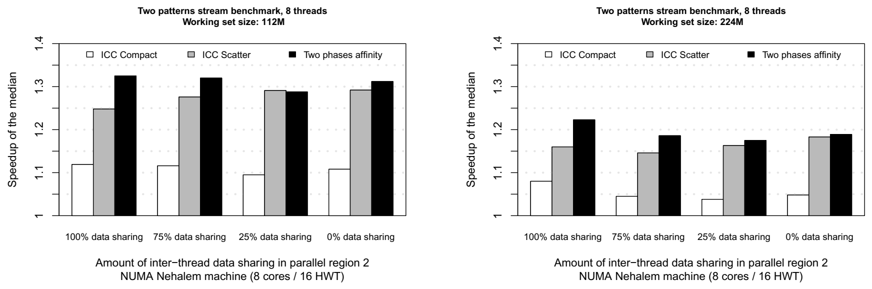


Fig. 1. Speedup of the median of the tested thread affinities for the synthetic benchmark using multiple matrix sizes and running with 8 threads on the `Nehalem` machine

We run the micro-benchmark multiple times and using multiple thread pinnings on top of an 8 cores Intel `Nehalem` machine[1]. Figure 1 shows the obtained speedups. We consider the `no affinity` strategy as the base comparison configuration. Speedups are reported according to the amount of data sharing in the second parallel region (four configurations) and the tested thread affinities. Reading from left to right, the first group represents the case of 100% data sharing, the second represents the case of 75% data sharing, the third represents the case of 25% data sharing and the last group represents the case of 0% data sharing.

First, it is clear that using a per parallel region thread affinity helps to improve performance compared to application-wide thread pinning strategies. Second, we observe that as the amount of data sharing in the second parallel region is reduced, the performance of the `icc scatter` and the per parallel region thread affinity strategies are close. This is due to two reasons: 1) since these two strategies are able to exploit the data sharing of the first parallel region, the obtained performance for that parallel region is

[1] More details about the machine can be found in Sect. 4.

similar, and 2) when there is no sharing at the second parallel region, the precise thread pinning is not important, so we obtain these observed program performance. From this simple experiment, we can conclude that changing the affinity between OpenMP parallel phases is beneficial and can lead to non negligible performance improvement over a fixed affinity for the whole program or a `no affinity` strategy.

3 Parallel OpenMP Phases Extraction and Thread Pinning

We focus on data sharing to compute effective thread pinnings. Instead of computing an *application-wide* thread affinity (apply the same affinity for the whole execution), we compute a thread pinning for each distinct parallel region in the OpenMP program. We use a profile guided method which consists of multiples steps that we detail below.

In OpenMP programs, computing a thread affinity for each parallel region requires to detect the entry and exit events of that region. For this purpose, we use the `OPARI` [10] instrumentation tool. The objective of `OPARI` is to provide a performance and measurement interface for OpenMP. It is a source-to-source translation tool which automatically adds function calls to a `POMP` runtime measurement library. This library is used to collect runtime performance data for OpenMP applications. `OPARI` supports C/C++ and Fortran programming languages. The idea behind the concept is to detect each OpenMP pragma/directive and add function calls to the `POMP` library. This method allows us to be compiler and runtime independent. In our approach, we do not use the `POMP` library for performance measurement. Instead, we have made changes in order to achieve dynamic thread pinning for each parallel region.

After the `OPARI` instrumentation, we make a memory tracing of OpenMP applications using the `PIN` [8] binary instrumentation framework. We fix a number of threads per application, and we collect for every thread and for each distinct parallel region (PR) all the accesses to all memory addresses (which are transformed to accesses to memory cache lines). In addition, we are able to deduce the parallel regions control flow graph `PRCFG`. It is a directed valued graph where the vertices represent the distinct PRs of the program and the edges represent the predecessor and the successor relationship between them. As reported before, an edge between a PR_i and PR_j is valued by the number of times the execution of the PR_i is followed by the execution of PR_j.

The collected memory trace profile is used to build an *affinity graph* for each parallel region in the program. Each *affinity graph* in the application is an undirected valued graph $G_p = (\mathcal{T}, \mathcal{E}, \alpha) \quad \forall p \in \mathcal{P}$. $\mathcal{T}$ is the set of application threads, $\mathcal{E} = \mathcal{T} \times \mathcal{T}$, $\alpha : \mathcal{E} \mapsto \mathbb{N}$ is a gain function applied to every pair of threads and $\mathcal{P}$ is the set of parallel regions implemented in the application.

The gain function $\alpha(T_i, T_j)$ models the attraction factor between two threads. Since we rely on data sharing between threads to compute an affinity graph, the gain function represents the number of common accesses to common memory caches lines accessed by both the T_i and T_j threads for a given parallel region. Let us precisely define α for an application with a fixed number of threads $n = \|\mathcal{T}\|$ and for a given parallel region $p \in \mathcal{P}$. The collected memory trace profile contains the information $A_p(T_i, b)$ which is the number of accesses of thread T_i to data block b at parallel region p. If we consider $B^p_{i,j}$ as the set of all data blocks accessed by the pair of threads (T_i, T_j) at parallel region p, then we can compute $\alpha(T_i, T_j)$ using Equation 1. We call this method the simple model or SM.

$$\alpha_p(T_i, T_j) = \sum_{b \in B^p_{i,j}} \min\left(A_p(T_i, b), A_p(T_j, b)\right) \tag{1}$$

We define another method to compute α. We call it the read/write model or RWM. We added this method because we consider that from the performance perspective, it is important to separate read and write accesses. The reason for that is that we consider a shared region of data wherein accesses are dominated by reads will have less impact on performance than a shared region of data wherein the read and write accesses are balanced. In fact, when the shared data are accessed only in a read mode, duplicating these data on multiple caches may not harm the performance in a great extent. Since we distinguish between reads and writes, then we exactly have $RD_p(T_i, b)$ and $WR_p(T_i, b)$ which is the number of reads and writes respectively performed by thread T_i to data block b and where $A_p(T_i, b) = RD_p(T_i, b) + WR_p(T_i, b)$. Given these constraints, Equation 2 defines the function $\alpha(T_i, T_j)$ for the read/write model.

$$\begin{aligned}\alpha_p(T_i, T_j) = \sum_{b \in B^p_{i,j}} \Big(&\min\left(RD_p(T_i, b), WR_p(T_j, b)\right) \; + \\ \min\left(WR_p(T_i, b), RD_p(T_j, b)\right) \; + \; &\min\left(WR_p(T_i, b), WR_p(T_j, b)\right) \Big)\end{aligned} \tag{2}$$

Once all the affinity graphs are constructed for an application and for a given number of threads, we can use them to investigate multiple thread pinning strategies. The idea is based on graph partitioning methods [5]. The affinity graphs must be decomposed into disjoint subsets, named a partition. A partition $V = \{V_1, V_2, \cdots, V_k\}$ has the property that $\bigcup_{1 \le l \le k} V_l = \mathcal{T}$ and $V_l \cap V_m = \emptyset$, where $l \neq m$ and $l, m \in [1, k]$. Every subset $V_l \in V$ contains a set of nodes representing threads that have to be placed on adjacent cores sharing the same cache level (L2 or L3, depending on the target machine). If we have k shared caches on the system, then we compute a partition with k subsets [5]. The global objective function is to maximize $\sum_{(T_i, T_j) \in V_l \times V_l} \alpha(T_i, T_j)$ the sum of the gains between threads belonging to the same partition. This optimization problem is a classical NP-complete problem, so we have to use heuristics such as in [5]. Fortunately, we have a special polynomial case. Indeed, if we are given a machine architecture where a cache level is shared between *two* adjacent cores, then the problem becomes to seek for partitions with a size equal to 2 ($\|V\| = 2$). It is easy to see that in the case of seeking partitions of size 2 the problem is equivalent to computing a set of thread pairs sharing a common cache while maximizing a global gain. In this special case, the optimization problem can be solved with a simpler maximum-weight matching in general graphs [2]. Precisely, it can be polynomially and optimally solved thanks to the algorithm of Edmonds in $O(\|\mathcal{T}\|^2 . \|\mathcal{E}\|)$ [2].

On a parallel machine with a memory hierarchy, the graph partitioning problem can be applied to reflect data reuse at each level of shared caches. We define two application dependent thread pinning strategies, corresponding to the application of heuristics for solving the graph k-partitioning problems:

1. `LPGP` strategy. After an initial step of optimal computation of thread pairs, we proceed by a graph k-partitioning [5]. It is a hierarchical strategy, where threads are

first paired and pinned on shared L2 or L3 cache then thread pairs are partitioned and placed on the different sockets according to their affinity.
2. `GPLP` strategy. It is a hierarchical strategy. It starts by an initial graph k-partitioning to fix threads on sockets, then perform an optimal polynomial algorithm to compute thread pairs sharing L2 or L3 cache levels.

In addition to the application dependent strategies presented above, we consider in our evaluation, the following application independent strategies:

1. `No affinity`. This strategy lets the OS decide about thread placement. This strategy allows thread migration between cores during application execution.
2. `icc compact`. This strategy assigns successive OpenMP threads to cores as close as possible in the topology map of the platform.
3. `icc scatter`. This strategy distributes OpenMP threads as evenly as possible across the entire sockets (one thread per socket if possible).

4 Experimental Setup and Methodology

Our experiments have been conducted using all SPEC OMP01 [14]. We used the `ref` data input with SPEC OMP01[2] whether for memory tracing or for performance evaluation. We tested multiple numbers of threads for every application according to the available number of cores. We tested various thread placement strategies for every application, thread number, input data set. For statistical significance, each measure was repeated 31 times and special care has been taken to limit any external interference on performance measures. The benchmarks have been compiled using Intel compiler (`icc` 11.1) with flag `-O3 -openmp`. To set a per parallel region thread affinity, we focus only on *hot parallel regions* which dominate the total execution time. This methodology helps to avoid setting a thread affinity for infrequently called or too short parallel regions, thus lowering the number of unnecessary thread migrations. No more than one application was executed at a time. The execution of each benchmark was repeated 31 times for each software configuration and machine. This high number of runs allows us to report statistics with a high confidence level [11,17]. The dynamic voltage scaling was disabled to avoid core frequency variation. Depending on the test machine, we run each benchmark with 8, 16 or 32 threads. When we plot speedups, only statistically significant ones are reported.

We conducted all our experiments on four platforms:

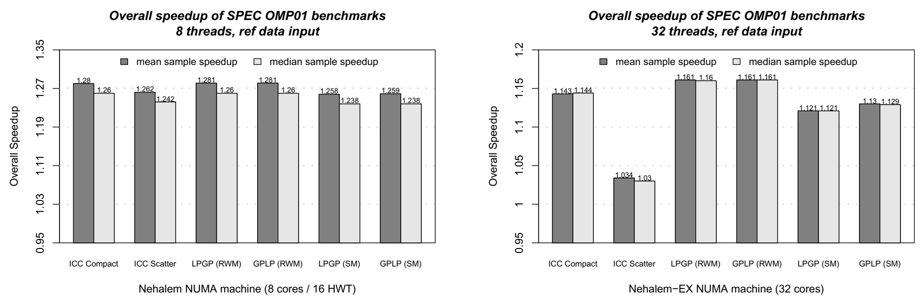
1. The `Nehalem` (8 cores) machine. It is an Intel `NUMA` machine with 2 processors. Each processor (`Nehalem` micro-architecture) has 4 cores (2 hardware threads per core) sharing an inclusive L3 cache of 8 MB. The core frequency is 2.93 GHz.
2. The `Nehalem-EX` (32 cores) machine. It is an Intel `NUMA` machine with 4 processors. Each processor (`Nehalem` micro-architecture) has 8 cores sharing an inclusive L3 cache of 18 MB. The core frequency is 2.0 GHz.
3. The `Shanghai` (8 cores) machine. It is an AMD `NUMA` machine with 2 `Opteron` processors. Each processor (`K10` micro-architecture) has 4 cores sharing an exclusive L3 cache of 6 MB. The core frequency is 2.4 GHz.

[2] We also tested NAS Parallel Benchmarks (NPB) [3]. For lack of space, we limit our analysis to OMP01.

4. The Barcelona (16 cores) machine. It is an AMD NUMA machine with 4 Opteron processors. Each processor (K10 micro-architecture) has 4 cores sharing an exclusive L3 cache of 2 MB. The core frequency is 1.9 GHz.

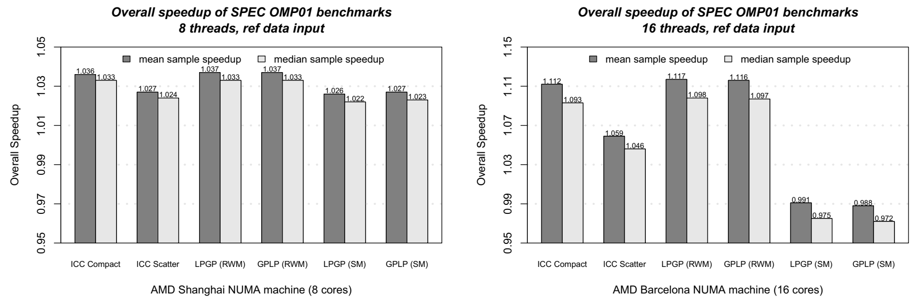
5 Experimental Evaluation of Phase-Based Thread Pinning

This section presents a performance evaluation and analysis about the effectiveness of the per parallel region thread affinity strategy for SPEC OMP01 benchmarks. We used four NUMA machines: Nehalem, Nehalem-Ex, Shanghai and Barcelona. Each benchmark has been executed with 8, 16 and 32 threads with respect to the maximal number of physical cores. We report the obtained speedups using the icc compact, icc scatter, LPGP(RWM), GPLP(RWM), LPGP(SM) and GPLP(SM) strategies compared to the default no affinity. Figures 2 and 3 show the overall sample speedups of every tested thread pinning strategy on the Nehalem, Nehalem-EX, Shanghai and Barcelona NUMA machines using bar plots. We report the speedups of the average and median execution times of all SPEC OMP01 applications running with 8, 16 and 32 threads.



(a) (b)

Fig. 2. Overall sample speedups of the tested thread affinities with SPEC OMP01 benchmarks running on the Intel Nehalem and Nehalem-EX NUMA machines



(a) (b)

Fig. 3. Overall sample speedups of the tested thread affinities with SPEC OMP2001 benchmarks running on the AMD Shanghai and Barcelona NUMA machines

On the `Barcelona` and `Nehalem-EX` machines, running OMP01 with 16 and 32 threads respectively with thread affinity enabled leads to non-negligible speedups and slowdowns (`Barcelona`). On the `Shanghai` machine, fixing thread affinity for OMP01 running with 8 threads leads to marginal speedups. On `Nehalem`, when running OMP01 with 8 threads, we observe non-negligible speedups for all the tested strategies. On `Nehalem`, the experiments were performed with HT enabled. This option increases the number of OS scheduling possibilities. Moreover, the number of involuntary thread migrations is increased which lead to the observed poor performance.

Even if the difference in terms of speedups is not significant, we observe that the `LPGP(SM)` and the `GPLP(SM)` perform slightly worse than `LPGP(RWM)` and `GPLP(RWM)` strategies. As a reminder, the SM strategies are computed from affinity graphs that do not consider the read/write model. Regarding the test machines, we do not observe any important difference, in terms of speedups, between the tested thread affinity strategies. This situation may suggest that there in no benefit of enabling a per parallel region thread affinity. Moreover, it is possible to conclude that this approach is not effective for SPEC OMP01 benchmarks.

As noticed earlier, from our experiments, we made two main observations which are highly related. First, the relative poor performance of strategies computed upon a model which does not consider the RWM. The observed overall sample speedup of the median for that strategies is in the range $[0.972 - 1.23]$. On the other hand, strategies that do consider the RWM have an overall sample speedup of the median in the range $[1.033 - 1.28]$. Second, the non clear benefit of using per parallel region thread affinity. If we compare the best overall sample speedup of the median obtained by application independent and application dependent strategies for each tested configuration (tested machine, number of threads), we observe that while application independent strategies have speedups in the range $[1.033 - 1.26]$, application dependent strategies have speedups in the range $[1.022 - 1.261]$. Indeed, regarding these speedups, it is possible to conclude that per parallel region affinity is not effective compared to application-wide (application independent) strategies.

To understand the presented experimental results, we first show in Table 1 the total number of times for which the computed per parallel region thread affinity (the `LPGP` and `GPLP` strategies computed using whether an aware or an unaware read/write model) consists of an application-wide thread affinity. This means, that the computed thread affinity is exactly the same for all the parallel regions in the program (Tables 2 reports the number of parallel regions in all the tested benchmarks), or at least for the detected most time consuming parallel regions. From Table 1, we can observe that using `LPGP(RWM)` and `GPLP(RWM)` strategies, at least 80% (on the `Nehalem-EX`, `apsi` and `equake` benchmarks run with a per parallel region thread affinity) of the benchmarks were executed with a single (application-wide) thread affinity. Moreover, the computed single application-wide strategies are similar to `icc compact`. On the other hand, the `LPGP(SM)` and `GPLP(SM)` strategies do not seem to reflect the same behavior. Indeed, for the SM strategies, we can observe that almost all the computed per parallel region thread affinity strategies have a thread affinity computed for at least two parallel regions.

In the light of the previous observations, we can say as a first conclusion, that thread affinity strategies computed from affinity graphs that do consider the RWM better

Table 1. Number of benchmarks where the computed per-parallel region thread affinity consists of setting a single-global-wide thread affinity. Each benchmark is executed using 8 16 and 32 threads on the `Nehalem`, `Nehalem-EX`, `Shanghai` and `Barcelona` machines.

#Threads	Machine	LPGP (RWM)	GPLP (RWM)	LPGP (SM)	GPLP (SM)
8	Nehalem	10/10	10/10	3/10	3/10
8	Shanghai	10/10	10/10	3/10	3/10
16	Barcelona	10/10	10/10	4/10	5/10
32	Nehalem-EX	8/10	9/10	4/10	4/10

Table 2. Number of parallel regions in SPEC OMP01 benchmarks running with the `ref` data input. For each benchmark, the number of iterations of the first hot parallel region is reported.

Benchmarks	#Parallel regions	#Iterations
wupwise	10	402
swim	8	1198
mgrid	12	18250
applu	22	50
galgel	32	117
equake	11	3334
apsi	24	50
fma3d	30	522
art	4	1
ammp	10	202

Table 3. Observed median execution times at each parallel region of the `swim` benchmark on the `Nehalem-EX` machine

	Median execution time (seconds)			
Parallel region	icc compact	icc scatter	LPGP (SM)	GPLP (SM)
PR 1	0.074634	0.074699	0.074282	0.07463
PR 2	0.069234	0.068776	0.068575	0.068885
PR 3	0.103967	0.103331	0.103097	0.103642
PR 4	18.08301	18.08452	19.33687	18.59277
PR 5	20.71075	20.72176	20.7149	23.32227
PR 6	4.972871	4.984532	4.974599	4.972682
PR 7	0.019681	0.019674	0.019683	0.030159
PR 8	23.53141	23.52575	43.52616	34.09414
Remote memory accesses ratio	3%	3%	55%	68%

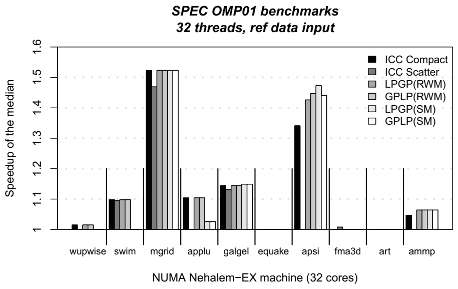


Fig. 4. The observed median speedups on the `Nehalem-EX` machine. Only statistically significant speedups are reported.

capture the sharing behavior of threads at the parallel region level than strategies that do not consider the RWM. Consequently, thread pinnings computed with the later strategies are likely to compute misleading thread affinity strategies which may hurt overall performance. Moreover, since almost all the per parallel region thread affinity strategies computed with a RWM tend to be application-wide strategies, explains why we observe that the performance of the RWM strategies is close to the performance of strategies like `icc compact` or `icc scatter`. Consequently, this observation suggests that SPEC OMP01 applications do not exhibit distinct phase behavior.

Unlike all the majority of the tested benchmarks, the `apsi` application does exhibit distinct inter-thread sharing patterns across parallel regions (more than 30% of data sharing (`ammp` has also more than 40% of shared accesses, but in a single region.). Figure 4 reports the statistically observed median speedups on the `Nehalem-EX` machine for OMP01 running with 32 threads. We can observe for the `apsi` benchmark that while application-wide strategies achieve up to 30% performance improvement compared to the Linux `no affinity` strategy, per parallel region thread affinity strategies achieve up to 45% performance improvement.

Now, we have to understand why strategies that exhibit distinct thread pinnings for distinct parallel regions are less effective compared to application-wide strategies for the tested applications. There are mainly two reasons for this performance behavior. First, the poor inter-thread data sharing exhibited by the distinct parallel regions for the tested benchmarks. Thus, applying a dynamic thread affinity technique on OMP01 benchmarks is not effective. Unfortunately, this is true because of: 1) the uniform distribution of the working set between running threads and 2) the presence of non-uniform data sharing patterns is rare. Second, the ratio between the number of times each parallel region is called, and the elapsed execution time in a single iteration of a given parallel region is very low (as noticed before in Tables 2). This means that threads are frequently migrated across too short parallel regions. Consequently, the small granularity of the selected *hot* parallel regions leads to lower the benefit from that migrations. Moreover, the small amount of inter-thread data sharing can exacerbate in a non-negligible extent the performance degradation due to NUMA effects: unnecessary remote memory accesses.

To illustrate the influence of poor inter-thread data sharing and unnecessary thread migrations on the overall performance, we report in Table 3 the observed execution times at each parallel region of the `swim` benchmark running with 32 threads on the `Nehalem-EX` machine (we do not report execution times for the `LPGP(RWM)` and `GPLP(RWM)) because these strategies compute a thread pinning similar to icc compact`. Even if the `LPGP(SM)` and `GPLP(SM)` compute a phase-based thread pinning strategy (for parallel regions 4,5 and 8), we observe that they behave poorly compared to `icc compact` or `icc scatter`. Moreover, while the later strategies exhibit at most 3% of remote memory accesses, the former exhibit more than 50% of remote memory accesses. If we run `swim` with an application-wide strategy by considering only parallel region 8 or 5, then we observed that the obtained performance is similar to `icc compact`. In fact, this benchmark does not exhibit an important amount of inter-thread data sharing (less than 1%), an exact thread affinity is not important. Consequently, applying a phase-based technique on this benchmark leads to frequent thread migrations impacting negatively the locality of data (NUMA accesses), thus the observed poor program performance.

6 Related Work and Discussion

Most of the thread affinity studies on multicores focus on data locality and cache sharing in parallel applications. Zhang *et al.* [18] conducted a measurement analysis to study the influence of CMP cache sharing on multi-threaded performance applications using the PARSEC [1] benchmarks. Through measurement they suggest that cache sharing has very limited influence on the performance of the PARSEC applications. However, they do not conclude that cache sharing has no potential to be explored for multi-threaded programs. Tam *et al.* [15] proposed threads clustering to schedule threads based on data sharing patterns detected on-line using hardware performance monitoring units. The mechanism relies on cross-chip communication performance impact.

Klug *et al.* [6] proposed `autopin`, a framework to automatically determine at runtime the thread pinning best suited for an application based on hardware performance counters information. The work is achieved by evaluating the performance of a set of different scheduling affinities and select the best one. The tool requires that the user provides an initial set of good thread placements. Terboven *et al.* [16] examined the programming possibilities to improve memory pages and thread affinity in OpenMP applications running on ccNUMA architectures. They provided a performance analysis of some HPC codes which may suffer from ccNUMA architectures effects.

Song *et al.* [13] proposed an affinity approach to compute application-wide thread affinity strategies. It relies upon binary instrumentation and memory trace analysis to find memory sharing relationships between user-level threads. Like us, they build an affinity graph to model the data locality relationship. Then, they use hierarchical graph partitioning to compute optimized thread placements. While their affinity graph is based on the number of addresses shared among threads, our affinity graphs are built upon the number of accesses to common cache lines reflecting real cache activity.

Some studies have addressed the data cache sharing at the compiler level. They focused on improving the data locality in multicores based on the architecture topology. Lee *et al.* [7] proposed a framework to automatically adjust the number of threads in an application to optimize system efficiency. The work assumes a uniform distribution of the data between threads. Kandemir *et al.* [4] discussed a compiler directed code restructuring scheme for enhancing locality of shared data in multicores. The scheme distributes the iterations of a loop to be executed in parallel across the cores of an on-chip cache hierarchy target.

Our work differs from the last efforts in two main points. First, we focus on the study of the impact on performance of dynamic thread pinning to exploit the inter-thread data sharing. Moreover, unlike other studies, we perform a statistical performance evaluation (running multiple times, we fix the experimental setup, data analysis through a rigorous statistical protocol [17]), we experiment multiple thread placement strategies and multiple machine architectures. Second, when it comes to compute a scheduling affinity, we rely on a profile-guided method. Using dynamic binary instrumentation, we fully analyze optimized binaries regardless of the compiler. Furthermore, we believe, that extracting all data dependencies and data sharing at compile time may not be sufficient, because these information depend on the working set which is known only at runtime.

7 Conclusion

We have presented an approach to exploit phase-based behavior in OpenMP programs using thread affinity. The presented technique relies on the *control flow graph* of the parallel OpenMP regions. The *control flow graph* gives for each parallel region its predecessor and successor in the execution flow. In other words, it is the graph representing the execution flow of distinct parallel regions. We have extended an existing tool to instrument the OpenMP constructs. Using a binary instrumentation tool, we build an *affinity graph* for each parallel region in the program. After that, we compute multiple thread pinning strategies for each parallel region.

References

1. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: Characterization and architectural implications. In: Proc. of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2008 (October 2008)

2. Edmonds, J.: Maximum matching and a polyhedron with 0-1 vertices. Journal Res. Nat. 69-B(1-22), 125–130 (1965)
3. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Tech. rep., NASA Ames Research Center (October 1999), `http://www.nas.nasa.gov/Resources/Software/npb.html`
4. Kandemir, M., Yemliha, T., Muralidhara, S., Srikantaiah, S., Irwin, M.J., Zhnag, Y.: Cache topology aware computation mapping for multicores. SIGPLAN Not. 45(6), 74–85 (2010)
5. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing 48, 96–129 (1998), `http://dx.doi.org/10.1006/jpdc.1997.1404`
6. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: autopin — automated optimization of thread-to-core pinning on multicore systems. In: Stenström, P. (ed.) Transactions on HiPEAC III. LNCS, vol. 6590, pp. 219–235. Springer, Heidelberg (2011)
7. Lee, J., Wu, H., Ravichandran, M., Clark, N.: Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In: Proc. of the Annual International Symposium on Computer Architecture, ISCA 2010, pp. 270–279. ACM, New York (2010)
8. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005), `http://doi.acm.org/10.1145/1065010.1065034`
9. Mazouz, A., Touati, S.A.A., Barthou, D.: Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In: Proc. of IEEE International Conference on High Performance Computing & Simulation, HPCS 2011, July 4-8, pp. 273–279. IEEE, Istanbul (2011)
10. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for openmp. The Journal of Supercomputing 23, 105–128 (2002), `http://portal.acm.org/citation.cfm?id=603339.603347`
11. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modelling. John Wiley and Sons (1991)
12. Song, F., Moore, S., Dongarra, J.: Feedback-directed thread scheduling with memory considerations. In: Proc. of the International Symposium on High Performance Distributed Computing, HPDC 2007, pp. 97–106. ACM, New York (2007), `http://doi.acm.org/10.1145/1272366.1272380`
13. Song, F., Moore, S., Dongarra, J.: Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In: Proc. of the IEEE International Conference on Cluster Computing, New Orleans, Louisiana, USA, August 31 - September 4. IEEE (2009)
14. Standard Performance Evaluation Corporation: SPEC CPU (2006), `http://www.spec.org/`
15. Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In: Proc. of theACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys 2007, pp. 47–58. ACM, New York (2007)
16. Terboven, C., An Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in OpenMP programs. In: Proc. of the Workshop on Memory Access on Future Processors, MAW 2008, pp. 377–384. ACM, New York (2008)
17. Touati, S.A.A., Worms, J., Briais, S.: The Speedup-Test: A Statistical Methodology for Program Speedup Analysis and Computation. To Appear in the Journal of Concurrency and Computation: Practice and Experience (2012), `http://hal.inria.fr/hal-00764454`
18. Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 203–212. ACM, New York (2010)

Topic 3: Scheduling and Load Balancing (Introduction)

Zhihui Du, Ramin Yahyapour, Yuxiong He, Nectarios Koziris,
Bilha Mendelson, Veronika Sonigo, Achim Streit, and Andrei Tchernykh

Topic Committee

Despite significant effort parallel and distributed systems available today are still not fully utilized and exploited. Scheduling and load balancing techniques remain crucial for implementing efficient parallel and distributed applications and for making best use of existing parallel and distributed systems. The need for such techniques intensifies with the foreseen advent of exa-scale computer systems with many core and accelerator architectures. Similarly, cloud computing became a viable paradigm for some applications. Scheduling includes planning and optimization of the resource allocation as well as coping with the dynamics of the systems. These topics have been subject for research for many decades but remain one of the core topics in parallel and distributed computing.

The scheduling techniques play a role either at the application level or at the system level, and both scenarios are of interest for this topic area at the Euro-Par conference series. At the application level, we consider the mapping of distributed and parallel applications onto infrastructures and the development of dynamic load balancing algorithms which are able to exploit particular characteristics of the underlying system. At the system level, particular areas of interest were the support of modern many-core architectures as well as virtual systems like Cloud infrastructures. The optimization goals are getting more sophisticated by including criteria beyond the common minimization of execution times and increasing utilization.

Euro-Par has considered this topic for several years. This year's iteration gathered again many submissions and the selection process for this topic area was highly competitive. All papers were reviewed by at least four independent reviewers. Eventually, eight papers have been selected. These papers provide a very good coverage of these different perspectives. In the following, you will find contributions which focus on theoretical aspect and on practical implications. Similarly, we can see scheduling on application and on system level; locally, clustered, and distributed. We see the trend to include optimization goals like energy efficiency. Constraints like network communication are taken into consideration. Especially workflow scheduling has been tackled by two paper contributions.

We would like to thank all the reviewers, for their valuable time and effort, who helped us in the selection process. At the same time, we would like to thank all authors who helped to maintain Euro-Par as one of the premier scientific conferences at which innovative scheduling concepts for parallel and distributed systems are presented.

Energy-Efficient Scheduling with Time and Processors Eligibility Restrictions

Xibo Jin[1,2], Fa Zhang[1], Ying Song[1], Liya Fan[3], and Zhiyong Liu[1]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
[3] IBM China Research Laboratory, Beijing, China
{jinxibo,zhangfa,songying,zyliu}@ict.ac.cn, fanliya@cn.ibm.com

Abstract. While previous work on energy-efficient algorithms focused on assumption that tasks can be assigned to any processor, we initially study the problem of task scheduling on restricted parallel processors. The objective is to minimize the overall energy consumption while speed scaling (SS) method is used to reduce energy consumption under the execution time constraint (Makespan C_{max}). In this work, we discuss the speed setting in the continuous model that processors can run at arbitrary speed in $[s_{min}, s_{max}]$. The energy-efficient scheduling problem, involving task assignment and speed scaling, is inherently complicated as it is proved to be NP-Complete. We formulate the problem as an Integer Programming (IP) problem. Specifically, we devise a polynomial time optimal scheduling algorithm for the case tasks have an uniform size. Our algorithm runs in $O(mn^3logn)$ time, where m is the number of processors and n is the number of tasks. We then present a polynomial time algorithm that achieves an approximation factor of $2^{\alpha-1}(2 - \frac{1}{m^\alpha})$ (α is the power parameter) when the tasks have arbitrary size work.

1 Introduction

Energy consumption has become an important issue in the parallel processor computational systems. Dynamic Speed Scaling (SS) is a popular approach for energy-efficient scheduling to reduce energy consumption by dynamically changing the speeds of the processors according to the work they need to perform. The well-known relationship between speed and power is the cube-root rule, more precisely, that is the power of a processor is proportional to s^3 when it runs at speed s [1, 2]. Most research literatures [3, 4, 5, 6, 7, 8, 9, 10] have assumed a more general power function s^α, where $\alpha > 1$ is a constant power parameter. Note that it is a convex function of the processor's speed. Obviously, energy consumption is the power integrated over duration time. Higher speeds allow for faster execution, at the same time, result in higher energy consumption. In the past few years, energy-efficient scheduling has received much attention from single processor to parallel processors environment. In algorithmic, the approaches can (in general) be categorized into the following two classes for reducing the energy usage [5, 7]: (1) *Dynamic speed scaling* and (2) *Power-down management*. Our paper focuses on energy-efficient scheduling via dynamic speed scaling strategy.

In this policy, the goals of scheduling are either to minimize the total energy consumption or to trade off the conflicting objectives of energy and performance. The main difference is the former reduces the total energy consumption as long as the timing constraint is not violated, while the later seeks the best point between the energy cost and performance metric (such as makespan and flow time).

Speed scaling has been widely studied to save energy consumption initiated by Yao et al. [3]. The previous work consider that a task can be assigned to any processor. But it is natural to consider the restricted scheduling in modern computational systems. The reason is that the systems evolve over time, such as cluster, then the processors of the system differ from each other in their functionality (For instance, the processors have different additional components). This leads to the task can only be assigned to the processors, which has the task's required component. I.e., it leads to different affinities between tasks and processors. In practice, certain tasks may have to be allocated for certain physical resources (such as GPU) [11]. It is also pointed out that some processors whose design is specialized for particular types of tasks, then tasks should be assigned to a processor best suited for them [12]. Furthermore, when considering tasks and input data, tasks need to be assigned on the processors containing their input data. In other words, a part of tasks can be assigned on processors set A_i, and a part of tasks can be assigned on processors set A_j, but $A_i \neq A_j, A_i \cap A_j \neq \emptyset$. Another case in point is the scheduling with processing processor restrictions aimed at minimizing the makespan has been studied extensively in algorithmic (See [13] for an excellent survey). Therefore, it is significant to study the scheduling with processor restrictions from both of practical and algorithmic requirements.

Previous Work: Yao et al. [3] were the first to explore the problem of scheduling a set of tasks with the smallest amount of energy on single processor environment via speed scaling. They proposed an optimal offline greedy algorithm and two bounded online algorithms named *Optimal Available* and *Average Rate*. Ishihara et al. [4] formulated the minimization-energy of dynamical voltage scheduling (DVS) as an integer linear programming problem when all tasks were ready at the beginning and shared common finishing time. They showed that in the optimal solution a processor only runs at two adjacent discrete speeds when it can use only a small number of discrete processor speeds.

Besides studying variant of the speed scaling problems on single processor, researchers also carried out studies on parallel processors environment. Chen et al. [6] considered energy-efficient scheduling with and without task migration over multiprocessor. They proposed approximation algorithm for different settings of power characteristics where no task was allowed to migrate. When task migration is allowed and migration cost is assumed being negligible, they showed that there is an optimal real-time task scheduling algorithm. Albers et al. [7] investigated the basic problem of scheduling a set of tasks on multi-processor settings with an aim to minimize the total energy consumption. First they studied the case that all tasks were unit size and proposed a polynomial time algorithm for agreeable deadlines. They proved it is NP-Hard for arbitrary release time and deadlines and gave a $\alpha^\alpha 2^{4\alpha}$-approximation algorithm. For scheduling tasks with arbitrary processing size, they developed constant factor approximation algorithms. Aupy et al. [2] studied the minimization of energy on a set of processors for which the tasks assignment had been given. They investigated different speed scaling models. Angel et al. [10] consider the multiprocessor migratory and preemptive scheduling problem with the objective of minimizing the energy consumption. They proposed an optimal algorithm in the case where the jobs have release dates, deadlines and the power parameter $\alpha > 2$.

There were also some literatures to research the performance under an energy bounded. Pruhs et al. [8] discussed the problem of speed scaling to optimize makespan under an energy budget in a multiprocessor environment where the tasks had precedence constraints ($Pm|prec, energy|C_{max}$, m is the number of processors). They reduced the problem to the $Qm|prec|C_{max}$ and obtained a poly-log(m)-approximation algorithm assuming processors can change speed continuously over time. Greiner et al. [9] studied the trade off between energy and delay, i.e., their objective was to minimize the sum of energy cost and delay cost. They suggested a randomized algorithm $\mathcal{RA}$ for multiple processors: each task was assigned uniformly at random to the processors, then a single processor algorithm $\mathcal{A}$ was applied separately by each processor. They proved that the approximation factor of $\mathcal{RA}$ was βB_α without task migration when $\mathcal{A}$ was a β-approximation algorithm (B_α is the α-th Bell number). They also showed that any β-competitive online algorithm for a single processor yields a randomized βB_α-competitive online algorithm for multiple processors without migration. Using the method of conditional expectations, the results could be transformed to a derandomized version with additional running time. Angel et al. [10] also extended their algorithm, which considered minimizing the energy consumption, to obtain an optimal algorithm for the problem of maximum lateness minimization under a budget of energy.

However, all of these results were established without taking into account the restricted parallel processors. More formally, let the set of tasks $\mathcal{J}$ and the set of processors $\mathcal{P}$ construct a bipartite graph $G = (\mathcal{J} + \mathcal{P}, E)$, where the edge of E denotes a task can be assigned to a processor. The previous work study G is a *complete* bipartite graph, i.e., for any two vertices, $v_1 \in \mathcal{J}$ and $v_2 \in \mathcal{P}$, the edge $v_1 v_2$ is in G. We study the energy-efficient scheduling that G is a *general* bipartite graph, i.e., $v_1 v_2$ may be not an edge of G.

Our Contribution: In this paper, we address the problem of task **S**cheduling with the objective of **E**nergy **M**inimization on **R**estricted **P**arallel **P**rocessors (SEMRPP). It assumes all tasks are ready at time 0 and share a common deadline (a real-time constraint) [2, 4, 6, 7]. In this work, We discuss the continuous speed settings that processors can run at arbitrary speed in $[s_{min}, s_{max}]$. In Section 2, we provide the formal description of model. Section 3 discusses some preliminary results and reformulate the problem as an Integer Programming (IP) problem. In Section 4, we devise a polynomial time optimal scheduling algorithm in the case where the tasks have an uniform size. For the general case that the tasks have non-uniform computational work, in Section 5, we present a $2^{\alpha-1}(2-\frac{1}{m^\alpha})$-approximation algorithm, where α is the power parameter and m is the number of processors. Finally we conclude the paper in Section 6. To the best of our knowledge, our work may be the initial attempt to study energy optimization on the restricted parallel processors.

2 Problem and Model

We model the SEMRPP problem of scheduling a set $\mathcal{J} = \{J_1, J_2, ..., J_n\}$ of n independent tasks on a set $\mathcal{P} = \{P_1, P_2, ..., P_m\}$ of m processors. Each task J_j has an amount of computational work w_j which is defined as the number of the required CPU cycles for the execution of J_j [3]. We refer to the set $\mathcal{M}_j \subseteq \mathcal{P}$ as eligibility processing set of the task J_j, that is, J_j needs to be scheduled on one of its eligible processors $\mathcal{M}_j(\mathcal{M}_j \neq \phi)$. We also say that J_j is allowable on processor $P_i \in \mathcal{M}_j$, and is not allowed to migrate after it is assigned on a processor. A processor can process at most one task at a time and all processors are available at time 0.

At any time t, the speed of J_j is denoted as s_{jt}, and the corresponding processing power is $P_{jt} = (s_{jt})^\alpha$. The amount of CPU cycles w_j executed in a time interval is the speed integrated over duration time and energy consumption E_j is the power integrated over duration time, that is, $w_j = \int s_{jt}dt$ and $E_j = \int P_{jt}dt$, following the classical models of the literature [2, 3, 4, 5, 6, 7, 8, 9, 10]. Note that in this work we focus on speed scaling and all processors are alive during the whole execution, so we do not take static energy into account [2, 8]. Let c_j be the time when the task J_j finishes its execution. Let x_{ij} be an $0-1$ variable which is equal to one if the task J_j is processed on processor P_i and zero otherwise. We note that $x_{ij} = 0$ if $P_i \notin \mathcal{M}_j$. Our goal is to schedule the tasks on processors to minimize the overall energy consumption when each task could finish before the given common deadline C and be processed on its eligible processors. Then the SEMRPP problem is formulated as follows:

$$(\mathbf{P_0}) \qquad min\sum_{j=1}^{n}\int P_{jt}dt$$

$$\begin{aligned} s.t. \qquad & c_j \leq C \quad \forall J_j, \\ & \sum_{i=1}^{m} x_{ij} = 1 \quad \forall J_j, \\ & x_{ij} \in \{0,1\} \quad \forall J_j, P_i \in \mathcal{M}_j, \\ & x_{ij} = 0 \quad \forall J_j, P_i \notin \mathcal{M}_j. \end{aligned}$$

3 Preliminary Lemma

We start by giving preliminary lemmas for reformulating the SEMRPP problem.

Lemma 1. *If S is an optimal schedule for the SEMRPP problem in the continuous model, it is optimal to execute each task at a unique speed throughout its execution.*

Proof. Suppose S is an optimal schedule that some task J_j does not run at a unique speed during its execution. We denote J_j's speeds by $s_{j1}, s_{j2}, ..., s_{jk}$, the power of each speed i is $(s_{ji})^\alpha, i = (1, 2, ..., k)$, and the execution time of the speeds are $t_{j1}, t_{j2}, ..., t_{jk}$, respectively. So, its energy consumption is $\sum_{i=1}^{k} t_{ji}(s_{ji})^\alpha$. We average the k speeds and keep the total execution time unchanged, i.e., $\bar{s}_j = (\sum_{i=1}^{k} s_{ji}t_{ji})/(\sum_{i=1}^{k} t_{ji})$. Because the power function is a convex function of speed, according to convexity [14] (In the rest of paper, it will use convexity in many place but will not add reference [14]), we have

$$\begin{aligned}\sum_{i=1}^{k} t_{ji}(s_{ji})^\alpha &= (\sum_{i=1}^{k} t_{ji})(\sum_{i=1}^{k} \frac{t_{ji}}{\sum_{i=1}^{k} t_{ji}}(s_{ji})^\alpha) \\ &\geq (\sum_{i=1}^{k} t_{ji})(\sum_{i=1}^{k} \frac{t_{ji}s_{ji}}{\sum_{i=1}^{k} t_{ji}})^\alpha = (\sum_{i=1}^{k} t_{ji})(\bar{s}_j)^\alpha = \sum_{i=1}^{k} t_{ji}(\bar{s}_j)^\alpha\end{aligned}$$

So the energy consumption by unique speed is less than a task run at different speeds. I.e. , if we do not change J_j's execution time and its assignment processor (satisfying restriction), we can get a less energy consumption scheduling, which is a contradiction to that S is an optimal schedule.

Corollary 1. *There exists an optimal solution for SEMRPP in the continuous model, for which each processor executes all tasks at a uniform speed, and finishes its tasks at time C.*

All tasks on a processor run at a unique speed can be proved like *Lemma 1.* If some processor finishes its tasks earlier than C, it can lower its speed to consume less energy without breaking the time constraint and the restriction. Furthermore there will be no gaps in the schedule [8].

Above discussion leads to a reformulation of the SEMRPP problem in the continuous model as following:

$$(\mathbf{P_1}) \qquad min \sum_{i=1}^{m} \frac{(\sum_{j=1}^{n} x_{ij}w_j)^\alpha}{C^{\alpha-1}}$$

$$s.t. \qquad \sum_{j=1}^{n} x_{ij}w_j \leq s_{max}C \quad \forall P_i, \tag{1}$$

$$\sum_{i=1}^{m} x_{ij} = 1 \quad \forall J_j, \tag{2}$$

$$x_{ij} \in \{0, 1\} \quad \forall J_j, P_i \in \mathcal{M}_j, \tag{3}$$

$$x_{ij} = 0 \quad \forall J_j, P_i \notin \mathcal{M}_j. \tag{4}$$

The objective function is from that a processor P_i runs at speed $\frac{\Sigma_{J_j on P_i} w_j}{C} = \frac{\Sigma_{j=1}^{n} x_{ij} w_j}{C}$, that is each task on P_i will run at this speed, and P_i will complete all the tasks on it at time C (It assumes that, in each problem instance, the computational cycles of the tasks on one processor is enough to hold the processor will not run at speed $s_i < s_{min}$. Otherwise we are like to turn off some processors). Constraint (1) follows since a processor can not run at a speed higher than s_{max}. Constraint (2) relates to that if a task has assigned on a processor it will not be assigned on other processors, i.e, non-migratory. Constraint (3) and (4) are the restrictions of the task on processors.

Lemma 2. *Finding an optimal schedule for SEMRPP problem in the continuous model is NP-Complete in the strong sense.*

Lemma 3. *There exists a polynomial time approximation scheme (PTAS) for SEMRPP problem in the continuous model, when $\mathcal{M}_j = \mathcal{P}$ and s_{max} is fast enough.*

Note that we give detailed proofs (Due to the space limit, we omit the proof. See our report [15] for details) of Lemma 2 and Lemma 3 that were similarly stated as observations in the work [7], and we mainly state the conditions when they are established in the restricted environment. (such as the set of restricted processors and the upper speed s_{max} that we discuss below in the paper)

4 Uniform Tasks

We now propose an optimal algorithm for a special case of SEMRPP problem in which all tasks have equal execution cycles (uniform) (denoted as ECSEMRPP_Algo algorithm). We set $w_j = 1, \forall J_j$ and set $C = C/w_j$ in ($\mathbf{P_1}$) without loss of generality. Given the set of tasks $\mathcal{J}$, the set of processors $\mathcal{P}$ and the sets of eligible processors of tasks $\{\mathcal{M}_j\}$, we construct a network $G = (V, E)$ as follow: the vertex set of G is $V = \mathcal{J} \cup \mathcal{P} \cup \{s, t\}$ (s and t correspond to a source and a destination, respectively), the edge set E of G consists of three subsets: (1)(s, P_i) for all $P_i \in \mathcal{P}$; (2)(P_i, J_j) for $P_i \in \mathcal{M}_j$; (3)(J_j, t) for all $J_j \in \mathcal{J}$. We set unit capacity to edges (P_i, J_j) and (J_j, t), (s, P_i) have capacity c (initially we can set $c = n$). Define $L^* = min\{max\{L_i\}\}(i = 1, 2, ..., m)$, L_i is the load of processor P_i and it can be achieved by ***Algorithm 1***.

Lemma 4. *The algorithm BS_Algo solves the problem of finding minimization of maximal load of processor for restricted parallel processors in $O(n^3 logn)$ time, if all tasks have equal execution cycles.*

Its proof can mainly follow from the Maximum-flow in [16]. The computational complexity is equal to the time $O(n^3)$ to find Maximum-flow multiple $logn$ steps, i.e, $O(n^3 logn)$.

We construct our ECSEMRPP_Algo algorithm (***Algorithm 2***) through finding out the min-max load vector $\boldsymbol{l}$ that is a strongly-optimal assignment defined in [17, 18].

Algorithm 1. BS_Algo(G, n)

```
input : $(G, n)$
output: $L^*, P_i$ that have the maximal load, the set $\mathcal{J}_i$ of tasks that load on $P_i$
1: Let variable $l = 1$ and variable $u = n$;
2: If $l = u$, then the optimal value is reached: $L^* = l$, return the $P_i$ and $\mathcal{J}_i$, stop;
3: Else let capacity $c = \lfloor \frac{1}{2}(l + u) \rfloor$. Find the Maximum-flow in the network $G$. If
the value of Maximum-flow is exact $n$, namely $L^* \leq c$, then set $u = c$ and keep
$P_i$, $\mathcal{J}_i$ by the means of the Maximum-flow. Otherwise, the value of
Maximum-flow is less than $n$, namely $L^* > c$, we set $l = c + 1$. Go back to 2.
```

Algorithm 2. ECSEMRPP_Algo

```
1: Let $G_0 = G(V, E)$, $\mathcal{P}^H = \phi$, $\mathcal{J}^H = \{\phi_1, ..., \phi_m\}$;
2: Call $BS\_Algo(G_0, n)$;
3: Set maximal load sequence index $i = i + 1$. According to the scheduling
returned by step 2, we denote the processor that has actual maximal load as
$P_i^H$ and denote the tasks set assigned on it as $\mathcal{J}_i^H$. $\mathcal{E}_i^H$ corresponds to the
related edges of $P_i^H$ and $\mathcal{J}_i^H$. We set $G_0 = \{V \backslash P_i^H \backslash \mathcal{J}_i^H, E \backslash \mathcal{E}_i^H\}$,
$\mathcal{P}^H = \mathcal{P}^H \cup \{P_i^H\}$, $\phi_i = \mathcal{J}_i^H$. If $G_0 \neq \phi$, go to step 2;
4: We assign the tasks of $\mathcal{J}_i^H$ to $P_i^H$ and set all tasks at speed $\frac{\Sigma_{J_j \in \mathcal{J}_i^H} w_j}{C}$ on
$P_i^H$. Return the final schedule $H$.
```

Definition 1. *Given an assignment H denote by S_k the total load on the k most load of processors. We say that an assignment is strongly-optimal if for any other assignment H' (S_k' accordingly responds to the total load on the k most load of processors) and for all $1 \leq k \leq m$ we have $S_k \leq S_k'$.*

Theorem 1. *Algorithm ECSEMRPP_Algo finds the optimal schedule for the SEMRPP problem in the continuous model in $O(mn^3 logn)$ time, if all tasks have equal execution cycles.*

Proof. First we prove the return assignment H of ECSEMRPP_Algo is a strongly-optimal assignment. We set $H = \{L_1, L_2, ..., L_m\}$, L_i corresponds to the load of processor P_i in non-ascending order. Suppose H' is another assignment that $H' \neq H$ and $\{L_1', L_2', ..., L_m'\}$ corresponds to the load. According to the ECSEMRPP_Algo algorithm, we know that H' can only be the assignment that P_i moves some tasks to $P_j (j < i)$, because P_i can not move some tasks to $P_{j'} (j' > i)$ otherwise it can lower the L_i which is a contradiction to ECSEMRPP_Algo algorithm. We get $\Sigma_{k=1}^i L_i \leq \Sigma_{k=1}^i L_i'$, i.e., H is a strongly-optimal assignment by the definition. It turns out that there does not exist any assignment that can reduce the difference between the loads of the processors in the assignment H. I.e., there are not other assignment can reduce our aim as it is convexity. So the optimal scheduling is obtained.

Every time we discard a processor, so the total cost time is $m \times O(n^3 logn) = O(mn^3 logn)$ according to *Lemma* 4, which completes the proof.

5 General Tasks

As it is NP-Complete in the strong sense for general tasks (*Lemma* 2), we aim at getting an approximation algorithm for the SEMRPP problem. First we relax the equality (3) of ($\mathbf{P_1}$) to

$$0 \le x_{ij} \le 1 \qquad \forall J_j, P_i \in \mathcal{M}_j \tag{5}$$

After relaxation, the SEMRPP problem transforms to a convex program. The feasibility of the convex program can be checked in polynomial time to within an additive error of ϵ (for an arbitrary constant $\epsilon > 0$) [19], and it can be solved optimally [14]. Suppose x^* be an optimal solution to the relaxed SEMRPP problem. Now our goal is to convert this fractional assignment to an integral one $\bar{x}$. We adopt the dependent rounding introduced by [18, 20].

Define a bipartite graph $G(x^*) = (V, E)$ where the vertices of G are $V = \mathcal{J} \cup \mathcal{P}$ and $e = (i, j) \in E$ if $x^*_{ij} > 0$. The weight on edge (i, j) is $x^*_{ij} w_j$. The rounding iteratively modifies x^*_{ij}, such that at the end x^*_{ij} becomes integral. There are mainly two steps as following:

i. Break cycle:

1.While($G(x^*)$ has cycle $C = (e_1, e_2, ..., e_{2l-1}, e_{2l})$)

2.Set $C_1 = (e_1, e_3, ..., e_{2l-1})$ and $C_2 = (e_2, e_4, ..., e_{2l})$.

Find minimal weight edge of C, denoted as e^C_{min} and its weight $\epsilon = min_{e \in C_1 || e \in C_2} e$;

3.If $e^C_{min} \in C_1$ then every edge in C_1 subtract ϵ and every edge in C_2 add ϵ;

4.Else every edge in C_1 add ϵ and every edge in C_2 subtract ϵ;

5.Remove the edges with weight 0 from G.

ii. Rounding fractional tasks:

1.In the first rounding phase consider each integral assignment if $x^*_{ij} = 1$, set $\bar{x}_{ij} = 1$ and discard the corresponding edge from the graph. Denote again by G the resulting graph;

2.While($G(x^*)$ has connected component C)

3.Choose one task node from C as root to construct a tree Tr, match each task node with any one of its children. The resulting matching covers all task nodes;

4.Match each task to one of its children node (a processor) such that $P_i = argmin_{P_i \in \mathcal{P}} \Sigma_{\bar{x}_{ij}=1} \bar{x}_{ij} w_j$, set $\bar{x}_{ij} = 1$, and $\bar{x}_{ij} = 0$ for other children node respectively.

We denote above algorithm as Relaxation-Dependent-Rounding. Next we analyse the approximation factor it can find.

Theorem 2. (i) *Relaxation-Dependent-Rounding finds a* $2^{\alpha-1}(2 - \frac{1}{p^\alpha})$*-approximation to the optimal schedule for the SEMRPP problem in the continuous model in polynomial time, where* $p = max_{\mathcal{M}_j} |\mathcal{M}_j| \le m$. ($ii$) *For any processor* P_i, $\Sigma_{\mathcal{J}} \bar{x}_{ij} w_j < \Sigma_{\mathcal{J}} x^*_{ij} w_j + max_{\mathcal{J}: x^*_{ij} \in (0,1)} w_j$, x^*_{ij} *is the fractional task assignment at the beginning of the second phase. (i.e., extra maximal execution cycles of linear constraints are violated only by* $max_{\mathcal{J}: x^*_{ij} \in (0,1)} w_j$*)*

Proof. (i) Denote the optimal solution for the SEMRPP problem as OPT, H^* as the fractional schedule obtained after breaking all cycles and $\bar{H}$ as the schedule returned by the algorithm. Moreover, denote by H_1 the schedule consisting of the tasks assigned in the first step, i.e., $x^*_{ij} = 1$ right after breaking the cycles and by H_2 the schedule consisting of the tasks assigned in the second rounding step, i.e., set $\bar{x}_{ij} = 1$ by the matching process. We have $\|H_1\|_\alpha \leq \|H^*\|_\alpha \leq \|OPT\|_\alpha$[1], where the first inequality follows from H_1 is a sub-schedule of H^* and the second inequality results from H^* being a fractional optimal schedule compared with OPT which is an integral schedule. We consider $\|H_1\|_\alpha \leq \|H^*\|_\alpha$ carefully. If $\|H_1\|_\alpha = \|H^*\|_\alpha$, that is all tasks have been assigned in the first step and the second rounding step is not necessary, then we have $\|H_1\|_\alpha = \|H^*\|_\alpha = \|OPT\|_\alpha$. Such that the approximation is 1. Next we consider $\|H_1\|_\alpha < \|H^*\|_\alpha$, so there are some tasks assigned in the second rounding step, w.l.o.g., denote as $\mathcal{J}_1 = \{J_1, ..., J_k\}$. We assume the fraction of task J_j assigned on processor P_i is f_{ij} and the largest eligible processor set size $p = max_{\mathcal{M}_j} |\mathcal{M}_j| \leq m$. Then we have

$$\begin{aligned}
(\|H^*\|_\alpha)^\alpha &= \sum_{i=1}^{m} (\Sigma_{J_j : x^*_{ij}=1} w_j + \Sigma_{J_j \in \mathcal{J}_1} f_{ij})^\alpha \\
&\geq \sum_{i=1}^{m} (\Sigma_{J_j : x^*_{ij}=1} w_j)^\alpha + \sum_{i=1}^{m} (\Sigma_{J_j \in \mathcal{J}_1} f_{ij})^\alpha \\
&= (\|H_1\|_\alpha)^\alpha + \sum_{i=1}^{m} (\Sigma_{J_j \in \mathcal{J}_1} f_{ij})^\alpha \geq (\|H_1\|_\alpha)^\alpha + \sum_{i=1}^{m} \sum_{j=1}^{k} (f_{ij})^\alpha \\
&= (\|H_1\|_\alpha)^\alpha + \sum_{j=1}^{k} \sum_{i=1}^{m} (f_{ij})^\alpha \geq (\|H_1\|_\alpha)^\alpha + \sum_{j=1}^{k} (\frac{\sum_{i=1}^{m} f_{ij}}{p})^\alpha \\
&= (\|H_1\|_\alpha)^\alpha + \frac{1}{p^\alpha} \sum_{j=1}^{k} (w_j)^\alpha
\end{aligned} \tag{6}$$

From the fact that H_2 schedules only one task per processor, thus optimal integral assignment for the subset of tasks it assigns and certainly has cost smaller than any integral assignment for the whole set of tasks. In a similar way we have

$$(\|H_2\|_\alpha)^\alpha = \sum_{j=1}^{k} (w_j)^\alpha \leq (\|OPT\|_\alpha)^\alpha \tag{7}$$

So the inequality (6) can be reduced to

$$(\|H^*\|_\alpha)^\alpha \geq (\|H_1\|_\alpha)^\alpha + \frac{1}{p^\alpha} (\|H_2\|_\alpha)^\alpha \tag{8}$$

[1] In H_1 schedule, when the loads of m processors is $\{l_1^{h1}, l_2^{h1}, ..., l_m^{h1}\}$, $\|H_1\|_\alpha$ means $((l_1^{h1})^\alpha + (l_2^{h1})^\alpha + ... + (l_m^{h1})^\alpha)^{\frac{1}{\alpha}}$.

then

$$\begin{aligned}(\|\bar{H}\|_\alpha)^\alpha &= (\|H_1+H_2\|_\alpha)^\alpha \leq (\|H_1\|_\alpha + \|H_2\|_\alpha)^\alpha \\ &= 2^\alpha(\frac{\|H_1\|_\alpha + \|H_2\|_\alpha}{2})^\alpha \leq 2^\alpha(\frac{1}{2}(\|H_1\|_\alpha)^\alpha + \frac{1}{2}(\|H_2\|_\alpha)^\alpha) \\ &\leq 2^{\alpha-1}((\|H^*\|_\alpha)^\alpha - \frac{1}{p^\alpha}(\|H_2\|_\alpha)^\alpha + (\|H_2\|_\alpha)^\alpha) \\ &\leq 2^{\alpha-1}(2-\frac{1}{p^\alpha})(\|OPT\|_\alpha)^\alpha\end{aligned}$$

So

$$\frac{(\|\bar{H}\|_\alpha)^\alpha}{(\|OPT\|_\alpha)^\alpha} \leq 2^{\alpha-1}(2-\frac{1}{p^\alpha})$$

Which concludes the proof that the schedule $\bar{H}$ guarantees a $2^{\alpha-1}(2-\frac{1}{p^\alpha})$-approximation to optimal solution for the SEMRPP problem and can be found in polynomial time.

(ii) Seen from above, we also have

$$\Sigma_{J_j\in\mathcal{J}}\bar{x}_{ij}w_j < \Sigma_{J_j\in\mathcal{J}}x^*_{ij}w_j + max_{J_j\in\mathcal{J}:x^*_{ij}\in(0,1)}w_j, \forall P_i$$

Where the inequality results from the fact that the load of processor P_i in $\bar{H}$ schedule is the load of H^* plus the weight of task matched to it. Because we match each task to one of its child node, i.e., the execution cycle of the adding task $\bar{w}_j < max_{J_j\in\mathcal{J}:x^*_{ij}\in(0,1)}w_j$.

Now we discuss the s_{max}. First we give Proposition 1 to feasible and violation relationship.

Proposition 1. *If* $(\mathbf{P_1})$ *has feasible solution for the SEMRPP problem in the continuous model, we may hardly to solve* $(\mathbf{P_1})$ *without violating the constraint of the limitation of the maximal execution cycles of processors.*

Obviously, if $(\mathbf{P_1})$ has a unique feasible solution, i.e., the maximal execution cycles of processors is set to the OPT solution value. Then if we can always solve $(\mathbf{P_1})$ without violating the constraint, this means we can easily devise an exact algorithm for $(\mathbf{P_1})$. But we have proof that $(\mathbf{P_1})$ is NP-Complete in the strong sense. Next, we give a guarantee speed which can be regarded as fast enough on the restricted parallel processors scheduling in the dependent rounding.

Lemma 5. *Dependent rounding can get the approximation solution without violating the maximal execution cycles of processors constraint when* $s_{max}C \geq max_{P_i\in\mathcal{P}}L_i + max_{J_j\in\mathcal{J}}w_j$*, where* $L_i = \Sigma_{J_j\in\mathcal{J}_i}\frac{1}{|\mathcal{M}_j|}w_j$*,* $\mathcal{J}_i$ *is the set of tasks that can be assigned to processor* P_i*.*

Proof. First we denote a vector $\boldsymbol{H} = \{H_1, H_2, ..., H_m\}$ in non-ascending sorted order as the execution cycles of m processors at the beginning of the second step. We also denote a vector $\boldsymbol{L} = \{L_1, L_2, ..., L_m\}$ in non-ascending sorted order as the execution of m processors that $L_i = \Sigma_{J_j\in\mathcal{J}_i}\frac{1}{|\mathcal{M}_j|}w_j$. Now we need to prove

$H_1 \leq L_1$. Suppose we have $H_1 > L_1$, w.l.o.g., assume that the processor P_1 has the execution cycles of H_1. We denote the set of tasks assigned on P_1 as $\mathcal{J}_1^H$. Let $\mathcal{M}_1^H$ be the set of processors to which a task, currently fractional or integral assigned on processor P_1, can be assigned, i.e., $\mathcal{M}_1^H = \bigcup_{J_j \in \mathcal{J}_1^H} \mathcal{M}_j$. Similarly we denote the set of tasks can process on $\mathcal{M}_1^H$ as $\mathcal{J}^H$ and the set of processors $\mathcal{M}^H$ for every task in $P_i \in \mathcal{M}_1^H$ can be assigned. We have $\mathcal{M}^H = \bigcup_{J_j \in \mathcal{J}^H} \mathcal{M}_j$. W.l.o.g, we denote $\mathcal{M}^H$ as a set $\{h_1, h_2, ..., h_k\}(1 \leq k \leq m)$ and also denote a set $\{l_1, l_2, ..., l_k\}(1 \leq k \leq m)$ as its corresponding processors set in $\boldsymbol{L}$. According to the convexity of the objective, we get $H_{h_1} = H_{h_2} = ... = H_{h_k}$. By our assumption, $H_{h_p} > L_{l_q}, \forall p, \forall q$. Then

$$\Sigma_p H_{h_p} > \Sigma_q L_{l_q} \tag{9}$$

Note that each integral task (at the beginning of the second step) in the left part of inequality (9) can also have its respective integral task in the right part, but the right part may have some fractional task. So $\Sigma_q L_{l_q} - \Sigma_p H_{h_p} \geq 0$, i.e., $\Sigma_p H_{h_p} \leq \Sigma_q L_{l_q}$, a contradiction to inequality (9). The assumption is wrong, we have $H_1 \leq L_1$. By Theorem 2's the maximal execution cycles of dependent rounding $\bar{H}_{max}$, we have following process to finish the proof:

$$\begin{aligned}\bar{H}_{max} &< H_1 + max_{J_j \in \mathcal{J}: x^*_{ij} \in (0,1)} w_j \leq L_1 + max_{J_j \in \mathcal{J}: x^*_{ij} \in (0,1)} w_j \\ &\leq L_1 + max_{J_j \in \mathcal{J}} w_j = max_i L_i + max_{J_j \in \mathcal{J}} w_j \quad .\end{aligned}$$

6 Conclusion

In this paper we explore algorithmic instruments leading to reduce energy consumption on restricted parallel processors. We aim at minimizing the sum of energy consumption while the speed scaling method is used to reduce energy consumption under the execution time constraint (C_{max}). We first assess the complexity of scheduling problem under speed and restricted parallel processors settings. We present a polynomial-time approximation algorithm with a $2^{\alpha-1}(2 - \frac{1}{p^\alpha})$-approximation ($p = max_{\mathcal{M}_j} |\mathcal{M}_j| \leq m$) factor for the general case that the tasks have arbitrary size of execution cycles. Specially, when the tasks have an uniform size, we propose an optimal scheduling algorithm with time complexity $O(mn^3 logn)$. (We omit the evaluation results here due to the space limit, see our report [15] for details.)

Acknowledgement. This work was supported by grants of National Natural Science Foundation for China (No. 61020106002, No. 61161160566) and Spanish MINECO/MICINN grant TEC2011-29688-C02-01.

References

1. Mudge, T.: Power: A first-class architecture design constraint. Journal of Computer 34(4), 52–58 (2001)
2. Aupy, G., Benoit, A., Dufossé, F., Robert, Y.: Reclaiming the energy of a schedule: Models and algorithms. INRIA Research report RR-7598 (April 2011); Short version appeared in SPAA 2011

3. Yao, F., Demers, A., Shenker, S.: A scheduling model for reduced CPU energy. In: Proceedings of the IEEE Symposium on Foundation of Computer Science (FOCS 1995), pp. 374–382 (1995)
4. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: Proceeding of the International Symposium on Low Power Electronics and Design (ISLPED 1998), pp. 197–202 (1998)
5. Irani, S., Shukla, S., Gupta, R.: Algorithms for power savings. In: ACM-SIAM Symposium on Discrete Algorithms (SODA 2003), pp. 37–46 (2003)
6. Chen, J., Kuo, W.: Multiprocessor energy-efficient scheduling for real-time jobs with different power characteristics. In: International Conference on Parallel Processing (ICPP 2005), pp. 13–20 (2005)
7. Albers, S., Müller, F., Schmelzer, S.: Speed scaling on parallel processors. In: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2007), pp. 289–298 (2007)
8. Pruhs, K., van Stee, R., Uthaisombut, P.: Speed scaling of tasks with precedence constraints. Theory of Computing System 43(1), 67–80 (2008)
9. Greiner, G., Nonner, T., Souza, A.: The bell is ringing in speed-scaled multiprocessor scheduling. In: Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2009), pp. 11–18 (2009)
10. Angel, E., Bampis, E., Kacem, F., Letsios, D.: Speed scaling on parallel processors with migration. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 128–140. Springer, Heidelberg (2012)
11. Srikantaiah, S., Kansal, A., Zhao, F.: Energy aware consolidation for cloud computing. In: Proceedings of the Conference on Power Aware Computing and Systems (Hotpower 2008) (2008)
12. Gupta, A., Im, S., Krishnaswamy, R., Moseley, B., Pruhs, K.: Scheduling heterogeneous processors isn't as easy as you think. In: ACM-SIAM Symposium on Discrete Algorithms (SODA 2012), pp. 1242–1253 (2012)
13. Leung, J., Li, L.: Scheduling with processing set restrictions: A survey. International Journal of Production Economics 116(2), 251–262 (2008)
14. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press (2004)
15. Jin, X., Zhang, F., Song, Y., Fan, L., Liu, Z.: Energy-efficient Scheduling with Time and Processors Eligibility Restrictions. CoRR-abs/1301.4131 (2013), `http://arxiv.org/abs/1301.4131`
16. Lin, Y., Li, W.: Parallel machine scheduling of machine-dependent jobs with unit-length. European Journal of Operational Research 156(1), 261–266 (2004)
17. Alon, N., Azar, Y., Woeginger, G., Yadid, T.: Approximation schemes for scheduling. In: ACM-SIAM Symposium on Discrete Algorithms (SODA 1997), pp. 493–500 (1997)
18. Azar, Y., Epstein, L., Richter, Y., Woeginger, G.: All-norm approximation algorithms. Journal of Algorithms 52(2), 120–133 (2004)
19. Nesterov, Y., Nemirovskii, A.: Interior-point polynomial algorithms in convex programming. SIAM Studies in Applied Mathematics. SIAM (1994)
20. Gandhi, R., Khuller, S., Parthasarathy, S., Srinivasan, A.: Dependent rounding in bipartite graphs. In: Proceedings of the IEEE Symposium on Foundation of Computer Science (FOCS 2002), pp. 323–332 (2002)

A $(2+\varepsilon)$-Approximation for Scheduling Parallel Jobs in Platforms[⋆]

Pierre-François Dutot[1], Klaus Jansen[2], Christina Robenek[2], and Denis Trystram[1]

[1] LIG, Grenoble Institute of Technology INPG
51 rue Jean Kuntzmann F-38330 Montbonnot St. Martin, France
{pfdutot,trystram}@imag.fr

[2] Department of Computer Science, University of Kiel
Christian-Albrechts-Platz 4, 24098 Kiel, Germany
{kj,cot}@informatik.uni-kiel.de

Abstract. We consider the problem of Scheduling parallel Jobs in heterogeneous Platforms: We are given a set $\mathcal{J} = \{1, \ldots, n\}$ of n jobs, where a job $j \in \mathcal{J}$ is described by a pair (p_j, q_j) of a processing time $p_j \in \mathbb{Q}_{>0}$ and the number of processors required $q_j \in \mathbb{N}$. We are also given a set $\mathcal{B}$ of N heterogeneous platforms $P_1, \ldots, P_N$, where each P_i contains m_i processors for $i \in \{1, \ldots, N\}$. The objective is to find a schedule for the jobs in the platforms minimizing the makespan. Unless $\mathcal{P} = \mathcal{NP}$ there is no approximation algorithm with absolute ratio strictly better than 2 for the problem. We give a $(2+\varepsilon)$-approximation for the problem improving the previously best known approximation ratio.

1 Introduction

This paper considers the problem of Scheduling parallel Jobs in heterogeneous Platforms (SPP): We are given a set $\mathcal{J} = \{1, \ldots, n\}$ of n jobs, where a job $j \in \mathcal{J}$ is described by a pair (p_j, q_j) of a processing time $p_j \in \mathbb{Q}_{>0}$ and the number of processors $q_j \in \mathbb{N}$ that are required to execute j. We are also given a set $\mathcal{B}$ of N platforms $P_1, \ldots, P_N$, where each P_i contains a set M_i of $|M_i| = m_i$ processors for $i \in [N] := \{1, \ldots, N\}$. In general we assume that the numbers m_i may be different, that are *heterogeneous platforms.* If all values m_i are equal we have *identical platforms.* For heterogeneous platforms we may assume $m_1 \geq \ldots \geq m_N$. A schedule is an assignment $a : \mathcal{J} \to \bigcup_{i=1}^{N} 2^{M_i} \times \mathbb{Q}_{\geq 0}$, that assigns every job j to a starting time t_j and to a subset $A_j \subset M_i$ of the processors of a platform P_i with $|A_j| = q_j$. Obviously, a job j can only be scheduled in platform P_i if $m_i \geq q_j$. A schedule is feasible if every processor in every platform executes at most one job at any time. The objective is to find a feasible schedule with minimum makespan $\max_{i \in [N]} C_{\max}^{(i)}$, where $C_{\max}^{(i)} = \max_{\{j | A_j \subset M_i\}} t_j + p_j$ denotes the local makespan for platform P_i. We denote with $\mathrm{OPT}_{\mathrm{SPP}}(\mathcal{J}, \mathcal{B})$

[⋆] Research supported by German Research Foundation (DFG) project JA612/12-2.

the optimum value for the makespan of a schedule for the jobs in $\mathcal{J}$ into the platforms in $\mathcal{B}$.

By reduction from 3-Partition it follows that SPP is strongly $\mathcal{NP}$-hard even for identical platforms. Moreover, there exists no approximation algorithm with absolute ratio strictly better than 2, unless $\mathcal{P} = \mathcal{NP}$.

For $N = 1$ the problem is equal to Scheduling parallel Jobs, in the relevant literature denoted with $P|size_j|C_{\max}$. This problem is strongly $\mathcal{NP}$-hard even for a constant number of processors $m \geq 5$ [7]. By reduction from Partition it can be shown that there is no approximation algorithm for $P|size_j|C_{\max}$ with ratio strictly less than 1.5, unless $\mathcal{P} = \mathcal{NP}$. If we constrain the co-domain of the assignment a further and assume identical platforms the problem is equivalent to Strip Packing (for $N = 1$) and Multiple Strip Packing($N \geq 2$): In addition to $A_j \in \bigcup_{\ell=1}^{N} 2^{M_\ell}$ we postulate that A_j is equal to a set of consecutively numbered processors for every job $j \in \mathcal{J}$. Every job then corresponds to a rectangle of width q_j and height p_j. In general because of this contiguity constraint, algorithms for SPP cannot be directly applied to Multiple Strip Packing, since rectangles may be cut. But the optimal value for Multiple Strip Packing is an upper bound for the optimal value for the corresponding SPP problem with identical platforms. Interestingly, fractional versions of both problems coincide and therefore a solution of fractional (Multiple) Strip Packing gives a fractional solution for SPP with identical platforms.

1.1 Related Work

There are several approximation algorithms for Scheduling parallel Jobs.

If the number of processors is bounded by a constant, the problem admits a PTAS [1]. In case that the number of machines is polynomially bounded in the number of jobs, a $(1.5+\varepsilon)$-approximation for the contiguous problem (where a job has to be executed on processors with consecutive adresses) and a $(1+\varepsilon)$-approximation for the non-contiguous problem were given in [12]. Recently, for an arbitrary number of processors a tight approximation algorithm with absolute ratio $1.5+\varepsilon$ was achieved [10].

Also for an arbitrary number of processors the contiguous case of $P|size_j|C_{\max}$ is closely related to Strip Packing. A vast number of approximation algorithms for Strip Packing have been developed during the last decades.

One of the most powerful results for Strip Packing is an asymptotic fully polynomial time approximation scheme given by Kenyon and Rémila based on a linear program relaxation for Bin Packing [13]. For any accuracy $\varepsilon > 0$ their algorithm produces a $(1+\varepsilon)$-approximative packing plus an additive height of $\mathcal{O}(1/\varepsilon^2)h_{\max}$, where $h_{\max}$ denotes the height of the tallest rectangle. Recently, we showed that the additive term can be reduced to $\mathcal{O}(1/\varepsilon \log(1/\varepsilon))h_{\max}$ using a more sensitive rounding technique [5]. We will use the algorithm in [13] as a subroutine and refer to it as the KR algorithm.

Multiple Strip Packing was first considered by Zhuk [23], who also showed that there is no approximation algorithm with absolute approximation ratio better than 2. Meanwhile, several approximation algorithms for Multiple Strip Packing and Scheduling parallel Jobs in Platforms have been developed. Some of them can be applied to both problems achieving the same approximation ratio. However, due to different underlying techniques used for designing those algorithms, some of them are only suitable for one of the problems or require even more constraints on the jobs (rectangles) and platforms. In Table 1 we give an overview about the different kinds of algorithms and their approximation ratios.

Table 1. Known approximation algorithms and their ratios

		Platforms	Jobs	Rect.	Ratio	Constraints
Tchernykh et al.	[20] 2005	het.	✓	✓	10	none
Schwiegelshohn et al	[18] 2008	het.	✓	no	3	non-clairvoyant
Ye et al.	[22] 2009	ident.	✓	✓	2ρ	ρ makespan of $P\|\|C_{\max}$
Pascual et al.	[16] 2009	ident.	✓	no	4	none
Tchernykh et al.	[21] 2010	het.	✓	✓	$2e+1$	release dates
Quezada-Pina et al.	[17] 2012	het.	✓	no	3	$q_j \leq \min_i m_i$
Bougeret et al.	[5] 2009	ident.	no	✓	2	none
	[3] 2010	ident.	✓	no	2.5	none
	[4] 2010	het.	✓	no	2.5	$q_j \leq \min_i m_i$
	[5] 2011	het.	✓	✓	AFPTAS	none
	[6] 2012	ident.	✓	no	2	$\max_j q_j \leq m/2$

1.2 New Result

Currently, the best known absolute ratio for an approximation algorithm directly applicable to SPP is 3 given in [18]. Remark that in [18], processing times are not available to the scheduler. In this article we present a polynomial time algorithm with absolute ratio $(2+\varepsilon)$. Moreover, we nearly close the gap between the inapproximability bound of 2 and the currently best absolute ratio.

Theorem 1. *For any accuracy $\varepsilon > 0$ there is an algorithm that for a set $\mathcal{J}$ of n parallel jobs and a set $\mathcal{B}$ of heterogeneous platform generates a schedule for $\mathcal{J}$ into the platforms in $\mathcal{B}$ with makespan at most $(2+\varepsilon)\mathrm{OPT}_{SPP}(\mathcal{J},\mathcal{B})$. The algorithm has running time $g(1/\varepsilon) \cdot n^{\mathcal{O}(f(1/\varepsilon))}$ for some functions g, f with $g(1/\varepsilon), f(1/\varepsilon) = 2^{\mathcal{O}(1/\varepsilon \log(1/\varepsilon))}$.*

1.3 Methods and Overview

To obtain a simpler structure for the set of platforms $\mathcal{B}$ we use a new technique to group and round the platforms by the number of processors: Initially, we partition the platforms into a set $\mathcal{B}_0$ containing a constant number of the largest platforms, and a set $\mathcal{B}_1$ containing the remaining smaller platforms with less processors. For a certain number L the platforms in $\mathcal{B}_1$ are grouped and rounded obtaining a set $\tilde{\mathcal{B}}_1$ that contains L groups $\widetilde{B}_1, \ldots, \widetilde{B}_L$ of equal constant cardinality, so that the platforms in each group $\widetilde{B}_\ell$ are identical, see Section 2.1. Later we convert a solution for the rounded platforms $\mathcal{B}_0 \cup \tilde{\mathcal{B}}_1$ into one for the original ones in $\mathcal{B} = \mathcal{B}_0 \cup \mathcal{B}_1$, see Figure 1.

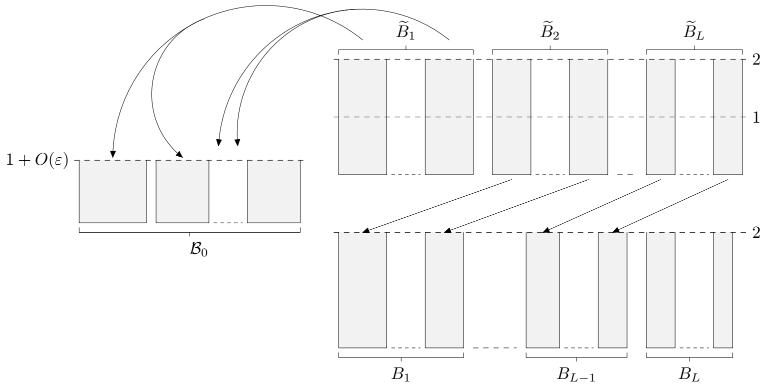


Fig. 1. Converting the schedule

Using gap creation [11] we simplify the structure of an optimum solution in $\mathcal{B}_0$, see Section 2.2 and Figure 2. Then we allocate a subset of jobs with large processing time jobs in $\mathcal{B}_0$. The main difficulty here is to place the correct subset of large narrow jobs, that have large processing time and require only few processors, since we cannot enumerate an assignment for them in polynomial time. Instead we guess an approximate gap structure for them.

With a skillful linear program relaxation (refer to Section 2.6) we fractionally assign a subset of large narrow jobs to the guessed gaps in $\mathcal{B}_0$, subsets of jobs with small and medium processing time to $\mathcal{B}_0$, and the remaining jobs to $\tilde{\mathcal{B}}_1$. In this new approach we have both, horizontal and vertical fractions of large narrow jobs, which are related by a nice covering constraint. Interestingly, we can apply a result for scheduling unrelated machines [15] to round those fractions to integral jobs producing only a small error even though there are different kinds of fractions. The Linear Program in 2.6 also produces a fractional schedule for $\tilde{\mathcal{B}}_1$. Here, the crucial part is to round the fractional schedule to an integral one without loosing too much. Therefore, the jobs involved in that fractional schedule have harmonically rounded processing times, see Section 2.5. That is, relatively large processing times are rounded up to the next value of the form $1/q$, $q \in \mathbb{N}$. We use the harmonically rounded processing times for rounding the fractional schedule in $\tilde{\mathcal{B}}_1$ to an integral one using an idea by Bansal et al. [2] based on the fact that any integer can be represented as a multiple of $1/q$, see [8]. Again the large narrow jobs are difficult as for one large narrow job we may produce fractions referring to different processing times in $\mathcal{B}_0$ and $\tilde{\mathcal{B}}_1$. This problem is also cleverly modelled and solved in our LP-relaxation. An overview about the algorithm is given in Algorithm 1.

1.4 Principles and Notations

First we define some notations and recall some well-known packing and scheduling principles. For $j \in \mathcal{J}$ we define the *size* of a job as $q_j p_j$ and $SIZE(\mathcal{J}) := \sum_{j\in\mathcal{J}} q_j p_j$ for a set of jobs. With $p_{\max} := \max_{j\in\mathcal{J}} p_j$ we denote the largest processing time of a job. A rectangle is a pair $r = (w_r, h_r)$ of width $w_r \in \mathbb{Q}_{>0}$ and height $h_r \in \mathbb{Q}_{>0}$. The *size* of r is defined as $w_r h_r$. The size of a set of rectangles $\mathcal{R}$ is $SIZE(\mathcal{R}) := \sum_{r\in\mathcal{R}} w_r h_r$. A *two-dimensional bin* of width x and height y will be denoted with $b(x, y)$. In this context a *strip* is a bin of width 1 and infinite height $b(1,\infty)$. We also use the notation $b(x,\infty)$ for a strip of width x. If $x \in \mathbb{N}$ a strip $b(x,\infty)$ corresponds to a platform with x processors.

Geometric Rounding: For a set $\mathcal{R}_{wide}$ of rectangles $r = (w_r, h_r)$ we obtain the geometrically rounded set $\mathcal{R}_{sup}$ with only M different widths in the following way: Order the rectangles by non-increasing width and stack them left aligned on top of each other, starting with the widest rectangles. Let H denote the height of the stack. Then draw horizontal lines at heights $(iH)/M$ for $i = 0, 1, \ldots, M$ through the stack. For $i = 0, 1, \ldots, M-1$ group together those rectangles that lie completely with their interior between the ith and $(i+1)th$ line or intersect with their interior the $(i+1)th$ line. In every group round up the width of every rectangle to the width of the widest rectangle contained in this group.

Fractional Strip Packing: For a set of rectangles $\mathcal{R}$ with $w_r \in (0, w]$ for $r \in \mathcal{R}$ a fractional strip packing of height $h > 0$ into a strip $b(w,\infty)$ corresponds to a feasible solution of a linear program of the form $\min\{\sum_i x_i | \sum_{i:C_i(r)=1} x_i \geq h_r \;\; r \in \mathcal{R}, x_i \geq 0\}$ with cost at most h. The variable x_i denotes the height (or length) of a configuration $C_i : \mathcal{R} \to \{0,1\}$, that is a function that represents a subset of rectangles that can be placed next to each into the strip $b(w,\infty)$, i.e. $\sum_{\{r\in\mathcal{R}|C_i(r)=1\}} w_r \leq w$. If $x_i > 0$, for every rectangle with $C_i(r) = 1$ a fraction of height x_i and width w_r is placed into the strip. If for $\mathcal{R}$ there exists a fractional strip packing of height h, we say $\mathcal{R}$ *fits fractionally into* $b(w,h)$. The content of the following Lemma is given in [13].

Lemma 1. *Let $\mathcal{R}$ be a set of rectangles $r = (w_r, h_r)$ with width $w_r \in (0, w]$ and heights $h_r \in (0, 1]$. Let $\varepsilon' > 0$ and $M := 1/\varepsilon'^2$ and let $\mathcal{R}_{wide} := \{r \in \mathcal{R} | w_r > \varepsilon' w\}$ and $\mathcal{R}_{narrow} := \mathcal{R} \backslash \mathcal{R}_{wide}$. If $\mathcal{R}_{wide}$ fits fractionally into a bin $b(w, h)$, then $\mathcal{R}_{sup}$ fits fractionally into bin $b(w, h(1+\varepsilon'))$. Moreover, $\mathcal{R}$ can be packed integrally into a strip $b(w,\infty)$ with height at most $\frac{(1+\varepsilon')h}{1-\varepsilon'} + (4M + 1)\max_{r\in\mathcal{R}} h_r$.*

2 Algorithm

Our algorithm considers two main scenarios for the shape of the platforms given by the input. For $\varepsilon > 0$ with $3/18 \geq \varepsilon$ and $\gamma = \frac{8}{3}N_1$, where $N_1 = \mathcal{O}(1/\varepsilon^4)$ (specified in the end of Section 2.6) we distinguish:

1. For all $i \in [N]$ we have $\frac{m_1}{m_i} \leq \gamma$.
2. There is a number $K \in [N]$ with $\frac{m_1}{m_i} \leq \gamma$ for all $i \leq K$ and $\frac{m_1}{m_i} > \gamma$ for all $i > K$.

In this section we give a detailed description of the algorithm for the first scenario from which the algorithm for the second scenario is derived. Details for the second scenario can be found in our technical report [8].

2.1 Platform Rounding

For $N_0 = 2(2N_1 + 1)$ we partition the set of platforms $\mathcal{B}$ into $L+1$ groups $B_0, B_1, \ldots, B_L$ by $L-$times collecting the N_1 smallest platforms where $L := \max\left\{0, \lfloor \frac{N-N_0}{N_1} \rfloor\right\}$. Let $\mathcal{B}_0 = B_0 := \{P_1, \ldots, P_{N-LN_1}\}$ and for $\ell \in [L]$ define $B_\ell := \{P_{N-(L-(\ell-1))N_1+1}, \ldots, P_{N-(L-\ell)N_1}\}$ and $\mathcal{B}_1 = \bigcup_{\ell=1}^{L} B_\ell$. Therefore, group $\mathcal{B}_1$ is further partitioned into several groups B_ℓ of equal constant cardinality. Each group $B_\ell \subseteq \mathcal{B}_1$ contains exactly N_1 platforms. Group $\mathcal{B}_0$ contains a constant number of platforms, moreover we have $5N_1+2 = N_0+N_1 > |\mathcal{B}_0| \geq N_0$. In each group B_ℓ, $\ell \in [L]$, we round the number of processors of each platform up to the number of processors $\widetilde{m}_\ell := m_{N-(L-(\ell-1))N_1+1}$ of the largest platform $P_{N-(L-(\ell-1))N_1+1}$ contained in this group and denote with $\widetilde{B}_\ell$ the group of rounded platforms. We compute a schedule for $\mathcal{B}_0 \cup \widetilde{\mathcal{B}}_1$, where $\widetilde{\mathcal{B}}_1 = \bigcup_\ell \widetilde{B}_\ell$, and later convert this solution into a solution for $\mathcal{B}_0 \cup \mathcal{B}_1$ applying a shifting argument, see Figure 1.

2.2 Simplifying the Structure of an Optimum Solution in $\mathcal{B}_0$

Via binary search in the interval $\left[SIZE(\mathcal{J})/(\sum_{i=1}^{N} m_i), np_{\max}\right]$ we find a candidate T for the makespan. By scaling we may assume $T = 1$. Now consider an optimum solution with makespan 1 and denote with $\mathcal{J}^\star(\mathcal{B}_0)$ the set of jobs that are scheduled in $\mathcal{B}_0$ by the optimum solution. In the following we use the gap creation technique [11] to find a subset of jobs with medium processing time $\mathcal{J}^\star_{medium}(\mathcal{B}_0) \subseteq \mathcal{J}^\star(\mathcal{B}_0)$ and small total load. We can remove these medium jobs from the instance and schedule them later on top only slightly increasing the makespan. Define $\sigma_0 = \frac{\varepsilon}{20}$ and $\sigma_{k+1} = \sigma_k^5$ and sets $\mathcal{J}_k = \{j \in \mathcal{J} | p_j \in (\sigma_k, \sigma_{k-1}]\}$ for $k \geq 1$. Let $\mathcal{J}_k^\star(\mathcal{B}_0)$ and $\mathcal{J}_k^\star(\mathcal{B}_1)$ denote those jobs in $\mathcal{J}_k$ that are scheduled by an optimum algorithm in $\mathcal{B}_0$ and $\mathcal{B}_1$, respectively. We have $\sum_{k\geq 1}\sum_{j\in\mathcal{J}_k(\mathcal{B}_0)} p_j q_j \leq \sum_{i=1}^{|\mathcal{B}_0|} m_i \leq |\mathcal{B}_0| m_1$. Using the pigeonhole principle we proof the existence of a set $\mathcal{J}_\tau^\star(\mathcal{B}_0)$ with $\tau \in \{1, \ldots, \frac{|\mathcal{B}_0|}{\varepsilon}\}$ so that $\sum_{j\in\mathcal{J}_\tau(\mathcal{B}_0)} p_j q_j \leq \varepsilon m_1$: If not, we have $\sum_{k\geq 1}^{|\mathcal{B}_0|/\varepsilon}\sum_{j\in\mathcal{J}_k(\mathcal{B}_0)} p_j q_j > |\mathcal{B}_0| m_1$ which is a contradiction. Then we choose $\delta = \sigma_{\tau-1}$ and may assume that ε^{-1} is an integer and thus $\delta^{-1} = \left(\frac{\varepsilon}{20}\right)^{-\left(5^{\tau-1}\right)} = \left(\frac{20}{\varepsilon}\right)^{5^{\tau-1}}$ is an integer (if not, we choose the next smaller value for ε). Furthermore, note that in the worst case $\delta^{-1} = \left(\frac{20}{\varepsilon}\right)^{\frac{|\mathcal{B}_0|}{\varepsilon}-1}$.

We partition the jobs into small jobs $\mathcal{J}_{small} := \{j \in \mathcal{J} | p_j \leq \delta^5\}$, medium jobs $\mathcal{J}_{medium} := \mathcal{J}_\tau = \{j \in \mathcal{J} | p_j \in (\delta^5, \delta]\}$ and large jobs with $\mathcal{J}_{large} := \{j \in \mathcal{J} | p_j \in (\delta, 1]\}$.

Algorithm 1. $(2+\varepsilon)$-Algorithm

```
Input: $\mathcal{J}$ , $\varepsilon > 0$
Output: A schedule of length $(2+\varepsilon)\mathrm{OPT}_{\mathrm{SPP}}(\mathcal{J})$
 1: For a certain constant $N_1 = \mathcal{O}(1/\varepsilon^4)$ partition the set of platforms into $L+1$
    groups $\mathcal{B}_0, B_1 \ldots, B_L$ and let $\mathcal{B}_1 := \bigcup_{\ell=1}^{L} B_\ell$.
 2: Round the number of processors of the platforms in each group $B_\ell$ and obtain $\widetilde{\mathcal{B}}_1$
    containing groups $\widetilde{B}_\ell$ of $N_1$ similar platforms
 3: for a candidate value for the makespan $T \in \left[\frac{SIZE(\mathcal{J})}{\sum_{i=1}^{N} m_i}, np_{\max}\right]$ do
 4:     for $k \in \{1, \ldots, \frac{|\mathcal{B}_0|}{\varepsilon}\}$ do
 5:         Let $\delta := \sigma_{k-1}$ where $\sigma_0 = \varepsilon/20$, $\sigma_{k+1} = \sigma_k^5$ for $k \geq 1$.
 6:         For $\delta$ distinguish small, medium, and large jobs
 7:         Round the processing times and possible starting times of large jobs to
    integral multiples $\delta^2$.
 8:         For $\alpha = \delta^4/16$ distinguish wide and narrow large jobs.
 9:         Enumerate an assignment vector $V$ of large wide jobs to $\mathcal{B}_0$ and let
    $\mathcal{J}_{la-wi}(\mathcal{B}_0)$ denote the selected jobs.
10:         for an assignment vector $V$ of large wide jobs do
11:             Approximately guess the total load $\Pi$ of large narrow jobs for each
    starting time and height in every platform of $\mathcal{B}_0$ and block corresponding gaps.
12:             for a guess $\Pi$ do
13:                 Compute free layers of height $\delta^2$ in $\mathcal{B}_0$.
14:                 Round the processing times $p_j$ of the jobs $\mathcal{J}' = \mathcal{J} \setminus \mathcal{J}_{la-wi}(\mathcal{B}_0)$
    harmonically.
15:                 Compute a solution of the LP in 2.6
16:                 if There is no feasible solution then
17:                     Discard the guess $\Pi$ and take another one and go back to Step
    13. If all guesses have failed discard $V$, take another and go back to Step 11. If all
    pairs $(V, \Pi)$ have failed, increase $k$ and go to Step 5.
18:                 end if
19:                 Round the solution of the LP using a result of Lenstra et al. [15] and
    obtain an almost integral assignment of
      – a subset of the small jobs to the free layers in $\mathcal{B}_0$
      – a subset of the large narrow jobs to the gaps $\Pi$ in $\mathcal{B}_0$
      – the remaining jobs to the groups $\widetilde{B}_\ell$ in $\widetilde{\mathcal{B}}_1$.
20:                 Pack small jobs with STRIP PACKING subroutine into the layers.
21:                 Schedule medium jobs in $\mathcal{J}_\tau(\mathcal{B}_0)$ in $P_1$.
22:                 for $\ell = 1, \ldots, L$ do
23:                     Pack the jobs assigned to $\widetilde{B}_\ell$ into at most $2N_1$ bins $b(\tilde{m}_\ell, 1)$
24:                 end for
25:             end for
26:         end for
27:     end for
28: end for
29: Convert the schedule for $\mathcal{B}_0 \cup \widetilde{\mathcal{B}}_1$ into a schedule for $\mathcal{B}_0 \cup \mathcal{B}_1$
```

Scheduling the medium jobs in $\mathcal{J}_\tau^\star(\mathcal{B}_0)$ in the end on top of the largest platform P_1 using List Schedule [9] increases the makespan by at most $2\max\left\{(1/m_1)\sum_{j\in\mathcal{J}_\tau(\mathcal{B}_0)} p_j q_j, \max_{j\in\mathcal{J}_\tau} p_j\right\} = 2\max\left\{\frac{\varepsilon m_1}{m_1}, \delta\right\} \leq 2\max\{\varepsilon,\delta\} \leq 2\varepsilon$.

For $\mathcal{B}_0$ we can now simplify the structure of the starting times and different processing times of large jobs. We round up the processing time of each job with processing time $p_j > \delta$ to $\bar{p}_j = h\delta^2$, the next integer multiple of δ^2 with $(h-1)\delta^2 < p_j \leq h\delta^2 = \bar{p}_j$, for $h \in \{\frac{1}{\delta}+1,\dots,\frac{1}{\delta^2}\}$. Since there can be at most $1/\delta$ jobs with height $>\delta$ on each processor within each platform this increases the makespan in $\mathcal{B}_0$ by only $\delta^2/\delta = \delta$. The number of different large jobs sizes H is bounded by $\frac{1}{\delta^2} - (\frac{1}{\delta}+1)+1 \leq \frac{1}{\delta^2}$. In a similar way we round the starting time of each large job in $\mathcal{B}_0$ to $a\delta^2$. This increases the makespan again by at most δ to $1+2\delta$. Therefore the large jobs have starting times $a\delta^2$ with $a \in \{0,1,\dots,\frac{1+2\delta}{\delta^2}-1\}$ and the number of different starting times is $A = \frac{1+2\delta}{\delta^2}$. An optimum schedule for $\mathcal{J}^\star(\mathcal{B}_0)\setminus\mathcal{J}_\tau^\star(\mathcal{B}_0)$ in $\mathcal{B}_0$ with rounded processing times $\bar{p}_j$ and rounded starting times for the large jobs has length at most $1+2\delta$.

Let $\tau \in \{1,\dots,\frac{|\mathcal{B}_0|}{\varepsilon}\}$ be the current iteration step for finding $\mathcal{J}_\tau$ with the desired properties and $\delta = \sigma_{\tau-1}$. We enumerate the set of large wide jobs and approximately guess the structure of large narrow jobs in $\mathcal{B}_0$ that correspond to a good solution for the jobs with rounded processing times $\bar{p}_j$. We distinguish between wide and narrow large jobs as follows. Assume that $m_N \geq 32/\delta^4$, otherwise the number of different platform sizes is a constant depending on γ and $1/\delta$. We choose $\alpha = \delta^4/16$. Then α satisfies $\alpha m_N \geq 2$, implying $\lfloor \alpha m_N \rfloor \geq \alpha m_N - 1 \geq \alpha m_N/2$. A job $j \in \mathcal{J}$ is called *wide* if $q_j \geq \lfloor \alpha m_N \rfloor$ and *narrow* otherwise. Furthermore distinguish large narrow jobs $\mathcal{J}_{la-na} := \{j \in \mathcal{J}_{large} | q_j \leq \lfloor \alpha m_N \rfloor\}$ and large wide jobs $\mathcal{J}_{la-wi} := \{j \in \mathcal{J}_{large} | q_j > \lfloor \alpha m_N \rfloor\}$.

2.3 Assignment of Large Wide Jobs in $\mathcal{B}_0$

The number of large wide jobs, that fit next to each other within one platform, is bounded by $\frac{m_1}{\lfloor \alpha m_N \rfloor} \leq \frac{m_1}{\alpha m_N - 1} \leq \frac{m_1}{(\alpha m_N)/2} \leq (2\gamma)/\alpha$. Since large jobs have processing times $>\delta$, at most $\frac{1+2\delta}{\delta}$ rounded large jobs can be finished on one processor before time $1+2\delta$. Therefore, the number of large wide jobs that have to be placed in every $P_i \in \mathcal{B}_0$ is bounded by $\frac{2\gamma}{\alpha}\cdot\frac{1+2\delta}{\delta}$. Furthermore, in every platform large jobs can have A different starting times. Each possible assignment of large wide jobs to platform and starting time can be represented by a tuple of vectors $V = (v_1,\dots,v_{|\mathcal{B}_0|}) \in \left(([n]\cup\{0\})^{A\cdot\frac{2\gamma}{\alpha}\cdot\frac{1+2\delta}{\delta}}\right)^{|\mathcal{B}_0|}$. The running time of a dynamic program to compute such an assignment is equal to the number of possible vectors which is bounded by $(n+1)^{|\mathcal{B}_0|\cdot A\cdot\frac{2\gamma}{\alpha}\cdot\frac{1+2\delta}{\delta}}$. Let $\mathcal{J}_{la-wi}(\mathcal{B}_0)$ denote the set of large wide jobs selected and let $\mathcal{J}' := \mathcal{J}\setminus\mathcal{J}_{la-wi}(\mathcal{B}_0)$.

2.4 Gaps for Large Narrow Jobs in $\mathcal{B}_0$

In every platform $P_i \in \mathcal{B}_0$ we approximately guess the total load $\Pi^\star_{i,a,h}$ of jobs with height $h\delta^2$ starting at time $a\delta^2$. Note that we only need to consider those triples (i, a, h) with $h\delta^2 + a\delta^2 \leq (1 + 2\delta)$. Therefore we compute a vector $\Pi = (\Pi_{i,a,h})$ with $\Pi_{i,a,h} = b \cdot \lfloor \alpha m_N \rfloor$, $b \in \{0, 1, \ldots, \frac{2\gamma}{\alpha}\}$ and $\Pi_{i,a,h} \leq \Pi^{\star}_{i,a,h} \leq \Pi_{i,a,h} + \lfloor \alpha m_N \rfloor$. Here the condition $\alpha m_N - 1 \geq \alpha m_N/2$ guarantees that $\frac{2\gamma}{\alpha} \cdot \lfloor \alpha m_N \rfloor \geq \frac{2\gamma}{\alpha} \cdot (\alpha m_N - 1) \geq m_1$. There is only a constant number $(1 + \frac{2\gamma}{\alpha})^{|\mathcal{B}_0| \cdot A \cdot H}$ of different vectors Π. For every triple (i, a, h) we block a gap of $\Pi_{i,a,h} + \lfloor \alpha m_N \rfloor$ (not necessary contiguous) processors for large narrow jobs with $\bar{p}_j = h\delta^2$. Later we will place large narrow jobs with $\bar{p}_j = h\delta^2$ total width $\geq \Pi^{\star}_{i,a,h}$ into them. This will be done using linear programming and subsequent rounding. Let G denote the total number of gaps, clearly $G \leq |\mathcal{B}_0| \cdot A \cdot H$. Since $\gamma, |\mathcal{B}_0| = \mathcal{O}(N_1) = \mathcal{O}(1/\varepsilon^3 \log(1/\varepsilon))$ and $\delta^{-1} = 2^{\mathcal{O}(1/\varepsilon \log(1/\varepsilon))}$, the steps described above take time $g(1/\varepsilon) \cdot n^{\mathcal{O}(f(1/\varepsilon))}$ for some function g and $f(1/\varepsilon) = 2^{\mathcal{O}(1/\varepsilon \log(1/\varepsilon))}$.

In Figure 2 an allocation of the enumerated large wide jobs and a guess Π for the gaps reserved for the large narrow jobs in $\mathcal{B}_0$ are illustrated. We compute the free layers of height δ^2 that correspond to the empty space between and next to the gaps and the large wide jobs. Let $L_1, \ldots, L_F$ denote the free layers, each having m_f processors for $f \in [F]$.

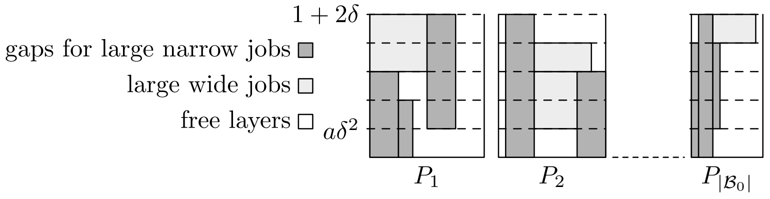


Fig. 2. Simplified structure of large jobs in $\mathcal{B}_0$

2.5 Rounding Jobs in $\widetilde{\mathcal{B}}_1$

Let $\mathcal{J}^{\star}(\mathcal{B}_1) \subset \mathcal{J}'$ be the subset of jobs scheduled in $\mathcal{B}_1$ in an optimum solution. Let $k := \frac{20}{\varepsilon}$. We assign to every job $\mathcal{J}^{\star}(\mathcal{B}_1)$ its harmonically rounded processing time $\tilde{p}_j := h_k(p_j) \in [0, 1]$, where $h_k : [0, 1] \longrightarrow [0, 1]$ is defined as in [2] via $h_k(x) = 1/q$ for $x \in (1/(q + 1), 1/q], q = 1, \ldots, k - 1$ and $h_k(x) = x$ for $x \leq 1/k$. Since $\varepsilon \leq 1/6$, we have $k = \frac{20}{\varepsilon} \geq 120$. In fact, we only modify the processing times of large jobs in $\mathcal{J}^{\star}(\mathcal{B}_1)$, because the small and medium jobs have processing times $p_j \leq \delta \leq \varepsilon/20 = 1/k$. Consequently, for all small and medium jobs we have $\widetilde{p_j} = p_j$. It might also be possible that there are large jobs with processing time $1/k \geq p_j > \delta$ for which we have $p_j = \widetilde{p_j}$. The following Lemma can be derived from the fact that for a sequence of numbers $x_1, \ldots, x_n$ with values in $(0, 1]$ and $\sum_{i=1}^{n} x_i \leq 1$ we have $\sum_{i=1}^{n} h_k(x_i) \leq T_\infty$ [2,14] together with a result of Seiden and van Stee [19] (where T_∞ is a constant $\simeq 1.691$, see [2]).

Lemma 2. *If the processing times of the jobs in $\mathcal{J}^{\star}(\mathcal{B}_1)$ are rounded harmonically, an optimum schedule of the rounded jobs into $\mathcal{B}_1$ (and therefore in $\widetilde{B}_1$) has makespan at most T_∞.*

2.6 Linear Program for the Remaining Jobs $\mathcal{J}'$

We give a linear programming relaxation for the following problem:

– place a set of small jobs $\mathcal{J}_{small}(\mathcal{B}_0) \subset \mathcal{J}_{small}$ into the layers $L_1, \ldots, L_F$
– select large narrow jobs $\mathcal{J}_{la-na}(\mathcal{B}_0) \subset \mathcal{J}_{la-na}$ to be placed into the gaps Π,
– fractionally place the remaining jobs into $\widetilde{\mathcal{B}}_1$.

We think of every group $\widetilde{B}_\ell$ as a single strip $b(\widetilde{m}_\ell, \infty)$ and introduce a set $\mathcal{C}^\ell$ of feasible configurations $C^\ell : \mathcal{J}' \to \{0,1\}$. Let $q(\ell)$ denote the number of different configurations for $\widetilde{B}_\ell$. In the LP below the variable x_i^ℓ indicates the length of configuration C_i^ℓ for $i \in [q(\ell)]$. In a similar way, we think of each layer L_f in $\mathcal{B}_0$ as a strip $b(\widetilde{m}_f, \infty)$ and introduce a set $\mathcal{C}^f$ of feasible configurations $C^f : \mathcal{J}_{small} \to \{0,1\}$ of small jobs and denote with $q(f)$ the number of different configurations for L_f. The variable x_i^f indicates the length of configuration C_i^f for $i \in [q(f)]$. For every job $j \in \mathcal{J}_{la-na}$ we introduce variables $y_j^{i,a,h} \in [0,1]$, that indicate the vertical fraction of job j (with $\bar{p}_j = h\delta^2$) that is assigned to a gap $\Pi_{i,a,h}$ in $\mathcal{B}_0$. For every group $\tilde{B}_\ell$ we need a constraint that guarantees that the length of the fractional schedule in $b(\widetilde{m}_\ell, \infty)$ corresponding to a feasible LP-solution does not exceed length NT_∞. Since we have N platforms in $\tilde{B}_\ell$ this gives a fractional schedule of length T_∞ in every platform. In a similar way we have one constraint for every layer L_f. For each gap $\Pi_{i,a,h}$ a constraint guarantees that the total load of large narrow jobs (fractionally) assigned to the gap does not exceed $\Pi_{i,a,h} + \lfloor \alpha m_N \rfloor$. To guarantee that all jobs are scheduled we have covering constraints.

For every small job we have a covering constraint combined from heights of configurations in L_f and in $\widetilde{B}_\ell$. Furthermore, we have a covering constraint for each large wide job that is not placed in $\mathcal{B}_0$, i.e. $j \in \mathcal{J}_{la-wi} \setminus \mathcal{J}_{la-wi}(\mathcal{B}_0)$. Every large narrow job $j \in \mathcal{J}_{la-na}$ is covered by a clever area constraint: The total fractional width of job j assigned to $\mathcal{B}_0$ multiplied with its height $\tilde{p}_j$ in $\widetilde{\mathcal{B}}_1$ plus the fraction of the area of this job covered by configurations in $\widetilde{\mathcal{B}}_1$ should be at least $\tilde{p}_j q_j$. For the medium jobs $\mathcal{J}_\tau$ the last constraint ensures that the total area of uncovered medium jobs is small, i.e. less than εm_1. Finally we add $x_i^f, x_i^\ell \geq 0$ and $y_j^{i,a,h} \in [0,1]$.

$$\begin{array}{ll}
\sum_{i=1}^{q(\ell)} x_i^\ell \leq N_1 T_\infty & \ell \in [L] \\
\sum_{i=1}^{q(f)} x_i^f \leq \delta^2 & f \in [F] \\
\sum_{\{j \in \mathcal{J}_{la-na} | \bar{p}_j = h\delta^2\}} y_j^{i,a,h} \cdot q_j \leq \Pi_{i,a,h} + \lfloor \alpha m_N \rfloor & i \in [|\mathcal{B}_0|], a \in [A], h \in [H] \\
\sum_{f=1}^{F} \sum_{\{i \in [q(f)] | C_i^f(j)=1\}} x_i^f + \sum_{\ell=1}^{L} \sum_{\{i \in [q(\ell)] | C_i^\ell(j)=1\}} x_i^\ell \geq \widetilde{p}_j (= p_j) & j \in \mathcal{J}_{small} \\
\sum_{\ell=1}^{L} \sum_{\{i \in [q(\ell)] | C_i^\ell(j)=1\}} x_i^\ell \geq \widetilde{p}_j & j \in \mathcal{J}_{la-wi} \setminus \mathcal{J}_{la-wi}(\mathcal{B}_0) \\
\sum_{\ell=1}^{L} \sum_{\{i \in [q(\ell)] | C_i^\ell(j)=1\}} x_i^\ell \cdot q_j + \tilde{p}_j \cdot \sum_{i,a,h : \bar{p}_j = h\delta^2} y_j^{i,a,h} \cdot q_j \geq \tilde{p}_j \cdot q_j & j \in \mathcal{J}_{la-na} \\
\sum_{j \in \mathcal{J}_\tau} p_j q_j - \sum_{j \in \mathcal{J}_\tau} \sum_{\ell=1}^{L} \sum_{\{i \in [q(\ell)] | C_i^\ell(j)=1\}} x_i^\ell q_j \leq \varepsilon m_1 & \\
x_i^f, x_i^\ell \geq 0, \quad y_j^{i,a,h} \in [0,1] &
\end{array}$$

If the LP has no feasible solution either the enumerated set $\mathcal{J}_{la-wi}(\mathcal{B}_0)$ was not correct, the choice of Π does not fit or the choice of δ, moreover the choice of τ, was not correct. We can compute an approximate solution of the linear program above by solving approximately a Max-Min Resource Sharing problem.

A solution $((x^f), (x^\ell), (y_j^{i,a,h}))$ of the LP can be transformed into a fractional solution of a general assignment problem. This assignment problem corresponds to scheduling n jobs on $|\mathcal{B}_0| \cdot A \cdot H + (F + L)(M + 1) + 1$ unrelated machines, for $M = 1/\varepsilon'^2$, $\varepsilon' = \varepsilon/(4 + \varepsilon)$. Using a result by Lenstra et al. [15] a fractional solution of this problem can be rounded to an almost integral one with only one fractionally assigned job per machine.

Different job manipulations are then described in [8] to assign those fractionally assigned jobs integrally to parts of $\mathcal{B}_0$ or in some gaps, without increasing the makespan. Then using a rounding technique for strip packing with harmonically rounded rectangles presented in [2], we show in [8] using $N_1 = \frac{(3M(k+1)+2)k}{2k-(k+1)(1+\varepsilon)T_\infty}$, how to produce a schedule with makespan $2 + \mathcal{O}(\varepsilon)$ of all jobs in $\mathcal{J}$ in the platforms of $\mathcal{B}_0 \cup \widetilde{\mathcal{B}}_1$. The schedule is finally converted into one for $\mathcal{B}_0 \cup \mathcal{B}_1$ with a shifting procedure illustrated in Figure 1.

3 Conclusion

We have obtained an Algorithm that constructs a schedule of a set $\mathcal{J}$ of n parallel jobs into a set $\mathcal{B}$ of N heterogeneous platforms with makespan at most $(2 + \varepsilon)\text{OPT}(\mathcal{J}, \mathcal{B})$. We assume that it is also possible to find an algorithm that packs a set of n rectangles into N strips of different widths. Many of the techniques used also apply to rectangles. The main difficulties will be the selection and packing process of the large narrow rectangles for $\mathcal{B}_0$ as the gaps provided by our algorithm might contain non-contiguous processors.

References

1. Amoura, A.K., Bampis, E., Kenyon, C., Manoussakis, Y.: Scheduling independent multiprocessor tasks. Algorithmica 32(2), 247–261 (2002)
2. Bansal, N., Han, X., Iwama, K., Sviridenko, M., Zhang, G.: Harmonic algorithm for 3-dimensional strip packing problem. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007), pp. 1197–1206 (2007)
3. Bougeret, M., Dutot, P.F., Jansen, K., Otte, C., Trystram, D.: Approximating the non-contiguous multiple organization packing problem. In: Calude, C.S., Sassone, V. (eds.) TCS 2010. IFIP AICT, vol. 323, pp. 316–327. Springer, Heidelberg (2010)
4. Bougeret, M., Dutot, P.-F., Jansen, K., Otte, C., Trystram, D.: A fast 5/2-approximation algorithm for hierarchical scheduling. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part I. LNCS, vol. 6271, pp. 157–167. Springer, Heidelberg (2010)
5. Bougeret, M., Dutot, P.-F., Jansen, K., Robenek, C., Trystram, D.: Approximation algorithms for multiple strip packing and scheduling parallel jobs in platforms. Discrete Mathematics, Algorithms and Applications 3(4), 553–586 (2011)

6. Bougeret, M., Dutot, P.-F., Jansen, K., Robenek, C., Trystram, D.: Tight approximation for scheduling parallel jobs on identical clusters. In: IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 878–885 (2012)
7. Du, J., Leung, J.Y.-T.: Complexity of scheduling parallel task systems. SIAM Journal of Discrete Mathematics 2(4), 473–487 (1989)
8. Dutot, P.-F., Jansen, K., Robenek, C., Trystram, D.: A $(2+\varepsilon)$-approximation for scheduling parallel jobs in platforms. Department of Computer Science, University Kiel, Technical Report No. 1217 (2013)
9. Garey, M.R., Graham, R.L.: Bounds for multiprocessor scheduling with resource constraints. SIAM Journal on Computing 4(2), 187–200 (1975)
10. Jansen, K.: A $(3/2+\varepsilon)$ approximation algorithm for scheduling moldable and non-moldable parallel tasks. In: 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2012), pp. 224–235 (2012)
11. Jansen, K., Solis-Oba, R.: Rectangle packing with one-dimensional resource augmentation. Discrete Optimization 6(3), 310–323 (2009)
12. Jansen, K., Thöle, R.: Approximation algorithms for scheduling parallel jobs. SIAM Journal on Computing 39(8), 3571–3615 (2010)
13. Kenyon, C., Rémila, E.: A near-optimal solution to a two-dimensional cutting stock problem. Mathematics of Operations Research 25(4), 645–656 (2000)
14. Lee, C.C., Lee, D.T.: A simple on-line bin-packing algorithm. Journal of the ACM 32(3), 562–572 (1985)
15. Lenstra, J.K., Shmoys, D.B., Tardos, É.: Approximation algorithms for scheduling unrelated parallel machines. Mathematical Programming 46, 259–271 (1990)
16. Pascual, F., Rzadca, K., Trystram, D.: Cooperation in multi-organization scheduling. Journal of Concurrency and Computation: Practice and Experience 21, 905–921 (2009)
17. Quezada-Pina, A., Tchernykh, A., Gonzlez-Garca, J., Hirales-Carbajal, A., Miranda-Lpez, V., Ramrez-Alcaraz, J., Schwiegelshohn, U., Yahyapour, R.: Adaptive job scheduling on hierarchical Grids. Future Generation Computer Systems 28(7), 965–976 (2012)
18. Schwiegelshohn, U., Tchernykh, A., Yahyapour, R.: Online scheduling in grids. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008), pp. 1–10 (2008)
19. Seiden, S.S., van Stee, R.: New bounds for multidimensional packing. Algorithmica 36(3), 261–293 (2003)
20. Tchernykh, A., Ramírez, J., Avetisyan, A., Kuzjurin, N., Grushin, D., Zhuk, S.: Two level job-scheduling strategies for a computational grid. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 774–781. Springer, Heidelberg (2006)
21. Tchernykh, A., Schwiegelsohn, U., Yahyapour, R., Kuzjurin, N.: On-line Hierarchical Job Scheduling on Grids with Admissible Allocation. Journal of Scheduling 13(5), 545–552 (2010)
22. Ye, D., Han, X., Zhang, G.: On-line multiple-strip packing. In: Du, D.-Z., Hu, X., Pardalos, P.M. (eds.) COCOA 2009. LNCS, vol. 5573, pp. 155–165. Springer, Heidelberg (2009)
23. Zhuk, S.N.: Approximate algorithms to pack rectangles into several strips. Discrete Mathematics and Applications 16(1), 73–85 (2006)

Scheduling Jobs with Multiple Non-uniform Tasks

Venkatesan T. Chakaravarthy, Anamitra Roy Choudhury, Sambuddha Roy, and Yogish Sabharwal

IBM Research - India, New Delhi
{vechakra,anamchou,sambuddha,ysabharwal}@in.ibm.com

Abstract. This paper considers the problem of maximizing the throughput of jobs wherein each job consists of multiple tasks. Consider a system offering a uniform capacity of a resource (say unit bandwidth). We are given a set of jobs, each consisting of a sequence of at most r tasks. Each task is associated with a window (specified by a release time and a deadline) within which it can be scheduled; each task also has a processing time and a bandwidth requirement. Each job has a profit associated with it. A feasible solution must choose a subset of jobs and schedule all the tasks for these jobs such that at any point of time, the total bandwidth requirement does not exceed the capacity of the resource; furthermore, the schedule must obey the precedence constraints (tasks of a job must be scheduled in order of the input sequence). The goal is to compute the feasible solution having maximum profit.

Prior work has studied the problem without the notion of windows; furthermore, the algorithms presented therein require that the bandwidths of all the tasks of a job are uniform. Under these two restrictions, $O(r)$-approximation algorithms are known. Our main result presents an $O(r)$-approximation algorithm for the general case wherein tasks can have windows and bandwidths of tasks within the same job may be non-uniform.

1 Introduction

Scheduling of jobs arises in diverse areas such as parallel and distributed computing, workforce management and energy management. In particular, consider a compute environment (such as a grid, cloud, etc.) offering resources as a service for executing jobs. The resources offered may be computational nodes, storage, network bandwidth, etc. The aim of the service provider owning the environment is to schedule jobs that maximize its profit subject to the availability of resources. Typically jobs do not require all the resources during their entire execution time and may have different requirements of the resources at different points in time. Suppose that the jobs can specify the time range and the durations during which they require the resources. This enables the service provider to schedule the jobs more optimally, thereby accommodating more jobs as well as increasing their profits. Motivated by this, we consider a setting in which a job is decomposed into multiple tasks, where each task specifies the time range, duration and quantity of the resource required. The problem also finds applications in computational biology, multimedia streaming and computational geometry (see Bar-Yehuda and Rawitz [4] for more details). We use the phrase *bandwidth* as a generic term for resources.

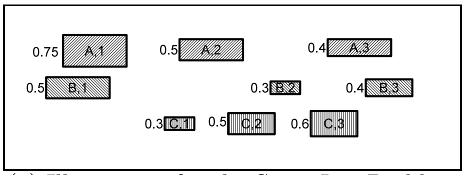
Illustration. Figure 1(a) illustrates the problem. Consider a system offering a uniform bandwidth of one unit. We have three jobs A, B and C, each containing 3 tasks. Each task has a requirement for the bandwidth, as shown in the figure. For example, the three tasks of the job A have requirements $0.75, 0.5$ and 0.4. A feasible solution must select a subset of jobs such that at any point of time, the sum of bandwidth requirements of the scheduled tasks must not exceed the bandwidth offered (i.e., one unit). We see that a feasible solution cannot pick both A and B, because the combined bandwidth requirement of the overlapping tasks $(A, 1)$ and $(B, 1)$ is 1.25. On the other hand, we can see that A and C can be picked together, since the combined bandwidth requirement does not exceed one unit at any point of time.

Problem Statement. Motivated by applications mentioned above, we first define the basic version, the SplitJob problem. Then we discuss a natural generalization of the problem.

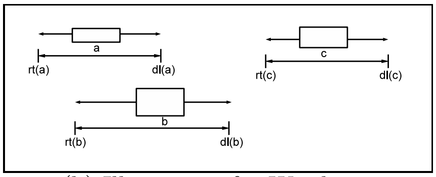
Basic SplitJob *Problem:* We assume that time is divided into discrete timeslots $\{1, 2, \ldots, T\}$. Consider a system offering a uniform bandwidth of say 1 unit throughout the span $[1, T]$. The input consists of a set of n jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$. Each job $J \in \mathcal{J}$ consists of a sequence of (at most) r tasks; each task is specified by a segment (or interval), given by a starting timeslot and an ending timeslot; Each task $a \in J$ also has a bandwidth requirement or *height.* We require that for any job, the r tasks constituting the job are non-overlapping. Every job $J \in \mathcal{J}$ is associated with a profit $p(J)$. A set of jobs $\mathcal{S} \subseteq \mathcal{J}$ is said to be a *feasible solution*, if at any timeslot $1 \leq t \leq T$, the sum of the heights for all the jobs selected by $\mathcal{S}$ does not exceed 1; we call this the *bandwidth constraint.* The profit of the solution $\mathcal{S}$ is defined to be the sum of profits of the jobs in $\mathcal{S}$. The SplitJob problem is to find a feasible solution $\mathcal{S}$ having the maximum profit.

SplitJob *Problem with Windows:* In the SplitJob problem, each task is specified by a fixed interval where it must be scheduled. However, in realistic applications, a task can specify a window within which it can be executed. To capture these scenarios we define a generalization of the SplitJob problem.

In this setup, each job J is specified by a sequence of tasks $a_1, a_2, \ldots, a_r$. Each input task a is specified by a *window* $[\mathrm{rt}(a), \mathrm{dl}(a)]$, where $\mathrm{rt}(a)$ and $\mathrm{dl}(a)$ are the release time and the deadline for the task, respectively. Each task is also associated with a processing time $\rho(a)$ and a height $h(a)$. The task can be scheduled on any segment of length $\rho(a)$ contained within its window. In addition to choosing a set of jobs, a feasible solution must also decide in which segment to schedule the tasks of the chosen jobs. Apart from satisfying the bandwidth constraint, we also require that the solution must satisfy the precedence constraint: the segment where a_i is scheduled must finish before the segment for a_{i+1} starts



(a) Illustration for the SplitJob Problem



(b) Illustration for Windows.

Fig. 1. Illustrations

(in other words, the execution of a_i should end before the execution of a_{i+1} starts). As before, the goal is to choose a feasible solution of maximum profit. We call this the SplitJob problem with windows. Notice that the windows of the tasks of a specific job may overlap, however a feasible solution must choose non-overlapping segments for them.

The SplitJob problem with windows includes as special case the following interesting version of the problem. In the new setup, the windows are associated with jobs, instead of tasks. Each job has a release time and deadline; each task is specified only by a processing time and a height. A task can be scheduled in any segment contained within the window of the job. A feasible solution must respect both the bandwidth constraints and the precedence constraints.

Special Cases. Prior work has addressed the following special cases of the basic SplitJob problem (without windows).

- *Single Task Case:* Here, each job consists of only one task (i.e., $r = 1$).
- *Unit Height Case:* All tasks of all jobs have height 1 (the bandwidth available). In this scenario, no two overlapping jobs can be scheduled.
- *Uniform Height Case:* In this case, for any job $J \in \mathcal{J}$, all the tasks of the job have the same height. Thus, the notion of height can be associated with the job itself, rather than with individual tasks. Note that different jobs are however allowed to have different heights.

All the above special cases also apply to the SplitJob problem with windows.

Prior Work. Bar-Noy et al. [2] studied the case of the single tasks ($r = 1$) and presented a 3-approximation algorithm using the local ratio technique (this algorithm can also handle the concept of windows and the approximation ratio becomes 5). Independently, Calinescu et al. [7] also designed a 3-approximation algorithm, via rounding linear program solutions. The problem has been generalized to the setup where the available bandwidth varies over time and it is known as the unsplittable flow problem on line (UFP), for which constant factor approximation algorithms are known (see [6]).

The unit height case of the basic SplitJob problem has been addressed in the context of finding maximum weight independent sets in r-interval graphs (e.g. [1,5]). Working under this framework, Bar-Yehuda et al. [3] presented a $(2r)$-approximation algorithm; in this context, they introduced the fractional local ratio paradigm. They also proved a hardness result: it is NP-hard to approximate the problem within a factor of $O(r/\log r)$. Thus, their approximation ratio is near-optimal.

Building on the techniques of Bar-Yehuda et al. [3], Bar-Yehuda and Rawitz [4] studied the uniform case of the basic SplitJob problem (without windows) and derived a $(6r)$-approximation algorithm. Their algorithm also utilizes the fractional local ratio technique.

Our Main Result. To the best of our knowledge, when $r \geq 2$, the prior work does not address two important aspects: (i) the concept of windows; (ii) non-uniform heights (i.e., the tasks of the same job may have different heights). The goal of this paper is to design an algorithm that handles both these aspects.

We present an approximation algorithm for a practically important special case of the problem, where no task requires more than half the bandwidth available; that is for any task a, its height $h(a) \leq 1/2$. Our main result is as follows:

Theorem 1. *There exists a randomized $(8r)$-approximation algorithm for the* SplitJob *problem with windows (with non-uniform tasks) when all the input tasks have height at most $1/2$. The running time of the algorithm is polynomial in n, T and r (n is the number of jobs and T is the number of timeslots).*

An interesting question here is whether we can design an algorithm having an constant approximation ratio (independent of the number of tasks r). However, this would imply NP=P, as discussed next. We can show that the basic SplitJob problem (without windows) includes as a special case the multi-dimensional knapsack problem, for which Chekuri and Khanna [9] derived certain hardness results. Using their results, we can prove that it is NP-hard to approximate the SplitJob problem (without windows) within factor of $O(r^{1/3-\epsilon})$, even when all the tasks have height at most $1/2$.

Other Results. We also prove these additional results.

– The main result can be generalized to the case where the task heights are bounded by a fixed constant. For any fixed constant $\alpha < 1$, we derive a randomized algorithm for the case where all the tasks have height at most α and the approximation ratio is $(4r)/(1-\alpha)$ (the main result corresponds to the value $\alpha = 1/2$).
– Our approach can also handle the case of uniform and unit height tasks and the approximation ratios obtained for these cases are $4r$ and $12r$ respectively.
– The algorithms claimed in the main result and the above two scenarios can handle the notion of windows and run in time polynomial in n, T and r. We note that in all the three cases, if we consider the corresponding basic versions without windows, then the algorithm can be made to run in time polynomial in n and r (i.e., the dependency on T can be removed).
– The main result deals with the case where all the tasks have height at most $1/2$. It is an interesting open problem to obtain an $O(r)$-approximation algorithm for the SplitJob problem (with or without windows), where the height of tasks can be arbitrary. In this context, we point out a difficulty in extending our algorithm for this general case. The $(8r)$-approximation

algorithm for the case of small tasks is based on rounding a natural linear program. We prove that this linear program has an integrality gap of $\Omega(2^r)$, even without windows. This shows that solving the open problem must involve a different strategy.

Due to lack of space, we will present only the main result in this paper and defer the details of the other results to the full version of the paper.

Proof Techniques and Discussion. Recall that Bar-Yehuda et al. [3] presented a $(2r)$-approximation algorithm for the scenario where all the tasks have unit height (unit height case). Extending their algorithm, Bar-Yehuda and Rawitz [4] presented a $(6r)$-approximation algorithm for the scenario where all the tasks of a job have the same height (uniform height case). Both these algorithms are based on the fractional local ratio paradigm, which involves rounding a linear program solution using the local ratio technique. Our goal is to design an algorithm than can solve the more general problem having two additional features: (i) the notion of windows; (ii) allow the tasks of the same job to have different heights (non-uniform case). We handle the notion of windows by considering an exponential sized linear program and solving it using a separation oracle. We note that the procedures of Bar-Yehuda et al. [3] and Bar-Yehuda and Rawitz [4] can be extended by incorporating our separation oracle to handle the concept of windows, as long as the tasks have unit or uniform heights, respectively. However, the notion of non-uniform heights poses more interesting challenges. To the best of our efforts, we could not extend their algorithms to handle the non-uniform scenario. In this paper, we overcome the issue by taking a different approach, namely randomized rounding. Thus, at a technical level, the main contribution of this paper is to show that randomized rounding offers an alternative method for dealing with scheduling multi-task jobs and furthermore, this approach can also deal with the case of non-uniform tasks.

Our algorithms are inspired by work of Chakrabarti et al. [8], who study the unsplittable flow problem (UFP) on line. Generalizing the work of Bar-Yehuda et al. [3] suitably for the case of non-uniform heights so as to apply the fractional local ratio technique is left as an interesting open question.

Remark: In the our problems, a job is allowed to have at most r tasks. However, we can assume without loss of generality that every job has exactly r tasks; this can be easily accomplished by introducing dummy tasks. So, in the rest of the paper, we assume that every job has exactly r tasks.

2 Main Result: LP Formulation and Solution

We say that a task a is *small*, if $h(a) \leq 1/2$. Our goal is to establish the main result of the paper, by designing a randomized $(8r)$-approximation algorithm for the special case of the SplitJob problem with windows, wherein all the tasks all small. Meaning, the algorithm outputs a solution $\mathcal{S}$ such that the expected profit of $\mathcal{S}$ is within a factor of $8r$ of the optimum solution $\texttt{Opt}$ (i.e., $\mathbf{E}[p(\mathcal{S})] \geq p(\texttt{Opt})/(8r)$). The algorithm goes via formulating a LP and rounding the fractional LP optimum solution. In this section, we present the LP formulation and discuss a duality based method for solving it. The rounding procedure is described in the next section. The following notations are useful for this purpose.

Notations. Let $\mathcal{J}$ be the set of n jobs, where each job $J \in \mathcal{J}$ consists of a sequence of r tasks. Each task a is specified by a window $[\mathrm{rt}(a), \mathrm{dl}(a)]$, a processing time $\rho(a)$ and a height $h(a)$. The task a can be scheduled in any *segment* $[s, e]$ of length $\rho(a)$ contained within the window $[\mathrm{rt}(a), \mathrm{dl}(a)]$. For each such segment u, its height is defined to be $h(u) = h(a)$. Such a segment u is said to be *active* at a timeslot t, if $t \in [s, e]$; this is denoted $u \sim t$. Let U be a set of segments (arising from multiple jobs/tasks) and let t be a timeslot. We define $h_t(U)$ to be the sum of heights of all segments from U active at the timeslot t: $h_t(U) = \sum_{u \in U \,:\, u \sim t} h(u)$.

LP Formulation. Let J be a job consisting of a sequence of tasks $a_1, a_2, \ldots, a_r$. For each task a_i with an associated window $[\mathrm{rt}(a_i), \mathrm{dl}(a_i)]$, the number of possible segments is $q(a_i) = \mathrm{dl}(a_i) - \rho(a_i) - \mathrm{rt}(a_i) + 2$. The total number of combinations for choosing segments for all the r tasks of the job J is $q = \Pi_{i=1}^{r} q(a_i)$. For a combination to be valid, it must satisfy the precedence constraint: namely, for $1 \leq i \leq r-1$, the segment chosen for a_i must end before the segment chosen for a_{i+1} starts. Discard the invalid combinations and let $\mathrm{Inst}(J)$ denote the set of remaining valid combinations. The number of valid combinations for the job J is at most T^r, where T is the total number of timeslots. We call each valid combination present in $\mathrm{Inst}(J)$ as a *job instance* of J. Each such job instance consists of a set of r segments each specified by a start time, end time and a height such that the segments are non-overlapping. Let $\mathcal{I}$ denote the union of job instances over all the jobs. For a job J and job instance $I \in \mathrm{Inst}(J)$, we define the profit of I to be $p(I) = p(J)$. We say that a job instance $I \in \mathcal{I}$ is *active* at a timeslot t, if one of its segments is active at the timeslot; we denote this as $I \sim t$. In this case, let $h_t(I)$ denote the height of the (unique) segment of I active at the timeslot (we call this the height of I at the timeslot t).

$$\max \quad \sum_{I \in \mathcal{I}} y(I) \cdot p(I)$$

$$\sum_{I \in \mathcal{I} \,:\, I \sim t} y(I) h_t(I) \leq 1 \qquad \text{for all time-slots } 1 \leq t \leq T \tag{1}$$

$$\sum_{I \in \mathrm{Inst}(J)} y(I) \leq 1 \qquad \text{for all jobs } J \in \mathcal{J} \tag{2}$$

$$y(I) \in \{0, 1\} \qquad \text{for all jobs } J \in \mathcal{J}$$

The integer program (IP) given above arises from the following equivalent formulation of a feasible solution. A feasible solution selects a subset of instances $\mathcal{F} \subseteq \mathcal{I}$ such that the following requirements are satisfied: (i) Bandwidth constraint: for any timeslot t, $\sum_{I \in \mathcal{F} \,:\, I \sim t} h_t(I) \leq 1$; (ii) For any job J, at most one job instance from $\mathrm{Inst}(J)$ is included in $\mathcal{F}$. Our goal is to choose a feasible solution having the maximum profit. In the IP, for each instance $I \in \mathcal{I}$, we introduce a variable $y(I)$ that denotes whether or not I is chosen in the solution. Constraints (1) and (2) encode the above requirements. We get a linear program by relaxing the integrality constraints as $y(I) \geq 0$, for all $I \in \mathcal{I}$.

The main issue with the above LP is that it has exponential number of variables. The LP has one variable for each job instance and so, the total number of variables is $|\mathcal{I}|$, which can be as large as T^r. In our setup, r is assumed to be an arbitrary input and so, the number of variables could be exponential. Hence, a polynomial time algorithm cannot even afford to explicitly write down the above LP and directly solve it. However, notice that number of constraints in the above LP is $T+n$, which is polynomial in the input length. This means that an optimal basic feasible solution (BFS) will set at most $T+n$ variables to non-zero values. Our goal is to find these non-zero variables and their values in time polynomial in T, n and r. We achieve this above goal by constructing a separation oracle for the dual LP, as discussed next.

Solving the LP. Consider the dual LP. We introduce dual variables $\alpha(t)$ corresponding to the set of constraints (1) and $\beta(J)$ corresponding to the set of constraints (2). The dual includes a constraint corresponding to each primal variable $y(I)$. For an instance $I \in \mathcal{I}$, let J_I denote the job to which it belongs. Then, the dual LP is as follows:

$$\min \qquad \sum_{t\in[1,T]} \alpha(t) + \sum_{J\in\mathcal{J}} \beta(J)$$

$$\beta(J_I) + \sum_{t\ :\ I\sim t} \alpha(t)\cdot h_t(I) \geq p(I) \qquad \text{for all job instances } I \in \mathcal{I}$$

The dual also includes non-negativity constraints: $\alpha(t) \geq 0$ and $\beta(J) \geq 0$. The dual has $T+n$ variables and $|\mathcal{I}|$ constraints (excluding the trivial non-negativity constraints); the number of variables is polynomial, whereas the number of constraints is exponential.

We shall construct a separation oracle for the dual. Recall that such a procedure takes as input a vector specifying values for all the dual variables and outputs whether or not the vector is a feasible solution; moreover, if the vector is not feasible, then the procedure must also output a constraint which is violated. Given such an oracle, the ellipsoid algorithm can solve the dual LP and find the optimum solution in polynomial time, even though the number of constraints is exponential. Our separation oracle procedure works using a dynamic programming approach.

The separation oracle is described next. Let $\alpha(\cdot)$ and $\beta(\cdot)$ be the input vectors specifying values assigned to the variables. We say that a job instance I is violated, if the dual constraint corresponding to I is violated by the input vectors. The goal is to find if there exists a job instance I which is violated. Towards that goal, we consider the jobs in $\mathcal{J}$ iteratively and for each job J, we determine if one of the job instances of J is violated.

Fix a job $J \in \mathcal{J}$. For a job instance $I \in \text{Inst}(J)$, let $\lambda(I)$ denote the sum $\sum_{t\,:\,I\sim t} \alpha(t)h_t(I)$. Let I^* be the job instance having the minimum value of $\lambda(I)$, among all the job instances in $\text{Inst}(J)$. All the instances $I \in \text{Inst}(J)$ have identical value $\beta(J_I)$ and $p(I)$. So, if there exists a instance $I \in \text{Inst}(J)$ which is violated, then the job instance I^* will also violated. Thus, it suffices if we find the job instance I^* and the value $\lambda(I^*)$.

We shall find I^* and $\lambda(I^*)$ using dynamic programming. Let $a_1, a_2, \ldots, a_r$ be the sequence of tasks contained in the job J. Fix an integer $1 \leq k \leq r$. By a *k-partial job instance* of J, we mean a sequence of segments $u_1, u_2, \ldots, u_k$ such that u_i is a segment of a_i and u_i finishes before u_{i+1} starts. The notion of $\lambda(\cdot)$ can be naturally extended to k-partial job instances P. Namely, P is said to be active at a timeslot t, if one of the segments of P is active at t and in this case, $h_t(P)$ is defined to be the height of the segment of P active at t; then, $\lambda(P) = \sum_{t\,:\,P\sim t} \alpha(t)h_t(P)$. For a timeslot $t \in [1,T]$ and an integer $1 \leq k \leq r$, let $M[t,k]$ denote the minimum value $\lambda(P)$ achieved by any k-partial job instance of J satisfying the property that all the segments of P are contained within $[1,t]$. Notice that due to the release time and deadline constraints of the tasks, no such k-partial job instance may exist; in this case, we define $M[t,k]$ be ∞. The value $\lambda(I^*)$ that we wish to compute is given by the entry $M[T,r]$.

The table $M[\cdot,\cdot]$ can be computed using the recurrence relation described below. We consider all possible segments of the task a_k which are contained within $[1,t]$ and for each such possibility, we consider the best way of selecting segments for $a_1, a_2, \ldots, a_{k-1}$. Then, among these possibilities we choose the one yielding the minimum $\lambda(\cdot)$ value. The recurrence relation is as follows:

$$M[t,k] = \min_{\text{rt}(a_k) \leq \widetilde{t} \leq t-\rho(a_k)+1} \left(M[\widetilde{t}-1, k-1] + \sum_{i=\widetilde{t}}^{\widetilde{t}+\rho(a_k)-1} \alpha(i)\cdot h(a_k). \right).$$

For the base case, we define $M[t,0] = 0$, for all timeslots $t \in [1,T]$. Using the above recurrence relation, we can compute all the entries of M. In particular, we can find I^* and $\lambda(I^*)$.

The separation oracle runs in time polynomial in n, T and r. Given the oracle, the ellipsoid algorithm can compute the optimum solution to the dual LP. Furthermore, it can also output the optimum solution to the primal LP. As mentioned earlier, only $n+T$ primal variables will have non-zero value in the primal (basic feasible) optimum solution. Using the ellipsoid algorithm, in conjunction with the separation oracle, we can find these non-zero variables and their values in time polynomial in n, T and r. We refer to the book by Grötschel et al. [10] for more details.

3 Rounding the LP Solution

The algorithm discussed in the previous section yields the optimum fractional solution to the primal LP, denoted by $\mathbf{y}$. In this section, we describe a procedure for rounding the solution. Let $\widetilde{\mathcal{I}}$ be the set of job instances that receive non-zero value under $\mathbf{y}$. Recall that only $n + T$ primal variables will have non-zero value in the primal (basic feasible) optimum solution. Thus, the number of instances in $\widetilde{\mathcal{I}}$ is at most $n + T$.

For a job J, let $\mathbf{x}(J)$ denote the sum of $\mathbf{y}(I)$ over all job instances of J that receive non-zero value under $\mathbf{y}$. Intuitively, this is the value assigned by the LP solution to the job J. Let $\widetilde{\mathcal{J}}$ denote the set of jobs having non-zero value for $\mathbf{x}(J)$. The profit of the LP solution is then given by $p(\mathbf{y}) = \sum_{J \in \widetilde{\mathcal{J}}} \mathbf{x}(J) p(J)$. Clearly, the optimum integral solution satisfies $p(\mathtt{Opt}) \leq p(\mathbf{y})$. We shall present a randomized rounding procedure which outputs a (integral) feasible solution $\mathcal{S}$ such that the expected profit satisfies $\mathbf{E}[p(\mathcal{S})] \geq p(\mathbf{y})/(8r)$.

The basic idea behind the rounding procedure is as follows. A natural rounding strategy is to select each job with probability $\mathbf{x}(J)$. But, it is difficult to argue that such a procedure will output a feasible solution with high profit. However, we shall show that if we "scale down" the selection probability by a factor $1/(cr)$, then we can get a solution with high profit (where c is a suitable constant). We note that the above idea of scaling down the probabilities has been successfully used in other contexts in prior work (see for example, [7], [8]). The rounding procedure is explained in detail next.

The rounding procedure proceeds in four phases:

– *Job Selection Phase:* Consider each job $J \in \widetilde{\mathcal{J}}$ and select it with probability $\mathbf{x}(J)/(4r)$. The jobs are selected independently at random. Let $\mathcal{J}_{\text{sel}}$ denote the set of selected jobs.
– *Job Instance Selection Phase:* Consider each selected job $J \in \mathcal{J}_{\text{sel}}$. Select exactly one job instance from $\text{Inst}(J)$, where an instance $I \in \text{Inst}(J)$ is chosen with probability $\mathbf{y}(I)/\mathbf{x}(J)$. Let $\mathcal{I}_{\text{sel}}$ be the set of job instances selected.
– *Segment Selection Phase:* Consider the set of all the segments belonging to the selected job instances. Arrange all these segments in the increasing order of their starting timeslots. Let $U = \emptyset$. For each segment u in the above ordering , select u if u can be added to U without violating the bandwidth constraint (i.e., $h(u) + h_t(U) \leq 1$, for all timeslots t in the span of u). Let $\mathcal{U}_{\text{sel}}$ denote the set of selected segments.
– *Output Phase:* Construct a set $\mathcal{I}_{\text{out}}$ as follows. For each job instance $I \in \mathcal{I}_{\text{sel}}$, include I in $\mathcal{I}_{\text{out}}$, if all the r segments of I are found in $\mathcal{U}_{\text{sel}}$. Consider a job $J \in \mathcal{J}_{\text{sel}}$ and let I be the unique job instance selected for J. Output the job J, if I is included in $\mathcal{I}_{\text{out}}$. Let $\mathcal{S}$ be the set of all jobs output. The set $\mathcal{S}$ is the solution output by the procedure.

Regarding the second phase, for any job $J \in \mathcal{J}_{\text{sel}}$, $\sum_{I \in \text{Inst}(J)} \mathbf{y}(I)/\mathbf{x}(J) = 1$. So, we will select exactly one job instance from $\text{Inst}(J)$. In the fourth phase, for any job $J \in \mathcal{S}$, the corresponding job instance I included in $\mathcal{I}_{\text{out}}$ specifies where the tasks of J must be scheduled. Thus, $\mathcal{S}$ constitutes the full description of a feasible solution. We next analyze the rounding procedure.

Lemma 1. *Suppose all the input tasks are small. Then,* $\mathbf{E}[p(\mathcal{S})] \geq p(\mathbf{y})/(8r)$.

Proof: Consider any job instance $I \in \widetilde{\mathcal{I}}$. The probability that I is output is :

$$\Pr[I \in \mathcal{I}_{\text{out}}] = \Pr[I \in \mathcal{I}_{\text{sel}}] \cdot \Pr[I \in \mathcal{I}_{\text{out}} \mid I \in \mathcal{I}_{\text{sel}}]. \quad (3)$$

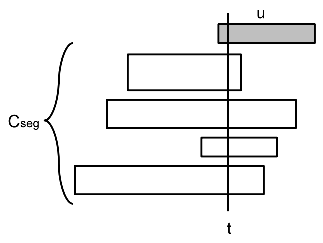


Fig. 2. Illustration for Proof of Lemma 1

Consider the first term in the RHS. Let J be the job to which I belongs. Then,

$$\Pr[I \in \mathcal{I}_{\text{sel}}] = \Pr[J \in \mathcal{J}_{\text{sel}}] \cdot \Pr[I \in \mathcal{I}_{\text{sel}} \mid J \in \mathcal{J}_{\text{sel}}] = \frac{\mathbf{x}(J)}{4r} \times \frac{\mathbf{y}(I)}{\mathbf{x}(J)} = \frac{\mathbf{y}(I)}{4r}. \quad (4)$$

Now consider the second term in the RHS of (3). Let the segments contained in I be $u_1, u_2, \ldots, u_r$. Then,

$$\begin{aligned}\Pr[I \in \mathcal{I}_{\text{out}} \mid I \in \mathcal{I}_{\text{sel}}] &= \Pr[\forall u \in I, u \in \mathcal{U}_{\text{sel}} \mid I \in \mathcal{I}_{\text{sel}}] \\ &= 1 - \Pr[\exists u \in I, u \notin \mathcal{U}_{\text{sel}} \mid I \in \mathcal{I}_{\text{sel}}] \geq 1 - \sum_{u \in I} \Pr[u \notin \mathcal{U}_{\text{sel}} \mid I \in \mathcal{I}_{\text{sel}}], \quad (5)\end{aligned}$$

where the last statement follows from the union bound. Let us derive a bound on each term of the summation in the last line.

We refer to Figure 2(a). Consider any segment $u \in I$. Let t be the starting timeslot of u. Let U be the set of segments that have already been selected when u was considered in the segment selection phase. Suppose u is not selected to be included in $\mathcal{U}_{\text{sel}}$. This implies that inclusion of u violates the bandwidth constraint at some timeslot t' in the span of u, meaning $h_{t'}(U) + h(u) > 1$. Recall that all segments are assumed to be small. In particular, $h(u) \leq 1/2$ and so $h_{t'}(U) \geq 1/2$. The segments are considered in the increasing order of their starting timeslots. Thus all segments of U active at the timeslot t' must also be active at the timeslot t. It follows that $h_t(U) \geq h_{t'}(U) \geq 1/2$. Hence,

$$\Pr[u \notin \mathcal{U}_{\text{sel}} \mid I \in \mathcal{I}_{\text{sel}}] \quad \leq \quad \Pr[h_t(U) \geq 1/2 \mid I \in \mathcal{I}_{\text{sel}}]. \quad (6)$$

We next derive a bound on the random variable $h_t(U)$.

Let $\mathcal{U}$ be the union of all segments[1]. For a segment $v \in \mathcal{U}$, let I_v be the job instance to which v belongs. Let C_{seg} be the set of all segments from $\mathcal{U}$ which are active at the timeslot t and considered earlier than u in the ordering considered in the segment selection phase (excluding u); we call C_{seg} as the *conflict segment set* of u. The expectation of the random variable $h_t(U)$ can be expressed as:

$$\mathbf{E}[h_t(U)] = \sum_{v \in C_{\text{seg}}} \Pr[v \in \mathcal{U}_{\text{sel}}] h(v) \leq \sum_{v \in C_{\text{seg}}} \Pr[I_v \in \mathcal{I}_{\text{sel}}] h(v),$$

[1] Notice that $\mathcal{U}$ is multi-set, since a segment v belonging to a task a of a job J' may be added multiple times in $\mathcal{U}$ by the different job instances of J'.

where the second statement follows from the fact that a segment v can belong to $\mathcal{U}_{\text{sel}}$, only if I_v belongs to $\mathcal{I}_{\text{sel}}$. A similar argument shows that

$$\mathbf{E}[h_t(U) \mid I \in \mathcal{I}_{\text{sel}}] \leq \sum_{v \in C_{\text{seg}}} \Pr[I_v \in \mathcal{I}_{\text{sel}} \mid I \in \mathcal{I}_{\text{sel}}] h(v).$$

For any job instance I' belong to the same job as I, $\Pr[I' \in \mathcal{I}_{\text{sel}} \mid I \in \mathcal{I}_{\text{sel}}] = 0$. On the other hand for any job instance I' belonging to a different job than that of I, the two events "$I' \in \mathcal{I}_{\text{sel}}$" and "$I \in \mathcal{I}_{\text{sel}}$" are independent (since jobs are includes in $\mathcal{J}_{\text{sel}}$ independently at random). It follows that

$$\mathbf{E}[h_t(U) \mid I \in \mathcal{I}_{\text{sel}}] \leq \sum_{v \in C_{\text{seg}}} \Pr[I_v \in \mathcal{I}_{\text{sel}}] h(v).$$

We can now appeal to Equation (4):

$$\begin{aligned} \mathbf{E}[h_t(U) \mid I \in \mathcal{I}_{\text{sel}}] = \sum_{v \in C_{\text{seg}}} \frac{\mathbf{y}(I_v)}{4r} h(v) \quad &= \quad \sum_{v \in C_{\text{seg}}} \frac{\mathbf{y}(I_v)}{4r} h_t(I_v) \\ \leq \sum_{I' \,:\, I' \sim t} \frac{\mathbf{y}(I')}{4r} h_t(I') \quad &\leq \quad 1/(4r). \end{aligned}$$

The first statement follows from Equation (4); the third statement follows from the fact that all the segments in C_{seg} are active at timeslot t; the last statement follows from the bandwidth constraint of the primal LP. By Markov's inequality, $\Pr[h_t(U) \geq 1/2 \mid I \in \mathcal{I}_{\text{sel}}] \leq (1/2r)$. Substituting in (6), we get that $\Pr[u \notin \mathcal{U}_{\text{sel}} \mid I \in \mathcal{I}_{\text{sel}}] \leq 1/(2r)$. Substituting in (5), we have that $\Pr[I \in \mathcal{I}_{\text{out}} \mid I \in \mathcal{I}_{\text{sel}}] \geq 1/2$. (since each job instance has r segments). It now follows from (3) and (4) that $\Pr[I \in \mathcal{I}_{\text{out}}] \geq \mathbf{y}(I)/(8r)$.

Consider any job J. The job J will be included in $\mathcal{S}$, if the job instance chosen for J is included in $\mathcal{I}_{\text{out}}$. We see that

$$\Pr[J \in \mathcal{S}] \quad = \quad \sum_{I \in \text{Inst}(J)} \Pr[I \in \mathcal{I}_{\text{out}}] \quad \geq \quad \left(\frac{1}{8r}\right) \sum_{I \in \text{Inst}(J)} \mathbf{y}(I) \quad = \quad \frac{\mathbf{x}(J)}{8r}.$$

We can now compute $\mathbf{E}[p(S)]$, by appealing to linearity of expectation.

$$\mathbf{E}[p(\mathcal{S})] \quad = \quad \sum_{J \in \mathcal{J}} \Pr[J \in \mathcal{S}] p(J) \quad \geq \quad \left(\frac{1}{8r}\right) \sum_{J \in \mathcal{J}} \mathbf{x}(J) p(J) \quad = \quad \frac{p(\mathbf{y})}{8r}.$$

This completes the proof of the lemma. □

We have established the main result of the paper (Theorem 1).

4 Conclusions and Open Problems

We presented a randomized $O(r)$-approximation algorithm for the SplitJob problem with windows, when all the tasks have height at most 1/2. We showed that when the tasks can have arbitrary heights, the natural LP has an integrality gap of $\Omega(2^r)$. Overcoming this issue and designing an $O(r)$-approximation algorithm is an interesting open problem.

Recall that in the introduction, we identified an interesting special case of the SplitJob problem with windows, wherein the windows are associated with jobs, rather than tasks. Clearly, our results imply $O(r)$-approximation algorithms for this problem. Designing an algorithm with better approximation ratio is an interesting avenue of research. We note that a constant factor approximation algorithm is not ruled out for this problem.

Recall that it is NP-hard to approximate the basic SplitJob problem (without windows) within a factor of $O(r/\log r)$, for the unit height case [3]. This hardness result also holds for the uniform height case. For the case of small tasks, we showed that it is NP-hard to approximate within $r^{1/3}$. Improving the hardness result to $O(r/\log r)$ would be of interest.

References

1. Bafna, V., Narayanan, B., Ravi, R.: Nonoverlapping local alignments (weighted independent sets of axis-parallel rectangles). Discrete Applied Math. 71(1-3), 41–53 (1996)
2. Bar-Noy, A., Bar-Yehuda, R., Freund, A., Naor, J., Schieber, B.: A unified approach to approximating resource allocation and scheduling. J. of the ACM 48(5), 1069–1090 (2001)
3. Bar-Yehuda, R., Halldórsson, M., Naor, J., Shachnai, H., Shapira, I.: Scheduling split intervals. SIAM Journal of Computing 36(1), 1–15 (2006)
4. Bar-Yehuda, R., Rawitz, D.: Using fractional primal-dual to schedule split intervals with demands. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 714–725. Springer, Heidelberg (2005)
5. Berman, P., DasGupta, B., Muthukrishnan, S.: Simple approximation algorithm for nonoverlapping local alignments. In: SODA (2002)
6. Bonsma, P., Schulz, J., Wiese, A.: A constant factor approximation algorithm for unsplittable flow on paths. In: FOCS (2011)
7. Calinescu, G., Chakrabarti, A., Karloff, H., Rabani, Y.: Improved approximation algorithms for resource allocation. In: Cook, W.J., Schulz, A.S. (eds.) IPCO 2002. LNCS, vol. 2337, pp. 401–414. Springer, Heidelberg (2002)
8. Chakrabarti, A., Chekuri, C., Gupta, A., Kumar, A.: Approximation algorithms for the unsplittable flow problem. Algorithmica 47(1), 53–78 (2007)
9. Chekuri, C., Khanna, S.: On multidimensional packing problems. SIAM J. Comput. 33(4), 837–851 (2004)
10. Grötschel, M., Lovász, L., Schrijver, A.: Geometric Algorithms And Combinatorial Optimization. Springer (1993)

Workflow Fairness Control on Online and Non-clairvoyant Distributed Computing Platforms

Rafael Ferreira da Silva[1], Tristan Glatard[1], and Frédéric Desprez[2]

[1] University of Lyon, CNRS, INSERM, CREATIS, Villeurbanne, France
{rafael.silva,glatard}@creatis.insa-lyon.fr
[2] INRIA, University of Lyon, LIP, ENS Lyon, Lyon, France
Frederic.Desprez@inria.fr

Abstract. Fairly allocating distributed computing resources among workflow executions is critical to multi-user platforms. However, this problem remains mostly studied in clairvoyant and offline conditions, where task durations on resources are known, or the workload and available resources do not vary along time. We consider a non-clairvoyant, online fairness problem where the platform workload, task costs and resource characteristics are unknown and not stationary. We propose a fairness control loop which assigns task priorities based on the fraction of pending work in the workflows. Workflow characteristics and performance on the target resources are estimated progressively, as information becomes available during the execution. Our method is implemented and evaluated on 4 different applications executed in production conditions on the European Grid Infrastructure. Results show that our technique reduces slowdown variability by 3 to 7 compared to first-come-first-served.

1 Introduction

The problem of fairly allocating computing resources to application workflows rapidly arises on shared computing platforms such as grids or clouds. It must be addressed whenever the demand for resources is higher than the offer, that is, when some workflows are slowed down by concurrent executions. In some cases, unfairness makes the platform totally unusable, for instance when very short executions are launched concurrently with longer ones. We define fairness as in [1,2,3], i.e. as the variability in a set of workflows of the *slowdown* $\frac{M_{multi}}{M_{own}}$, where M_{multi} is the makespan when concurrent executions are present, and M_{own} is the makespan without concurrent executions.

We consider a software-as-a-service platform where users can, at any time, launch application workflows that will compete for computing resources. Our two main assumptions are (i) that the problem is *online*: new workflows can be submitted at any time, and resources may also join or leave at any time, and (ii) that the problem is *non-clairvoyant*: the execution time of a task on a given computing resource is unknown. Non-clairvoyance comes from the lack of application models in the platform and from the lack of information about the performance of computing and network resources. We also assume a limited control on the scheduler, i.e. that only task priorities can be changed to influence scheduling. These conditions are representative of a large set of platforms, for instance the Virtual Imaging Platform (VIP [4]) and other science gateways [5,6,7] deployed on the European Grid Infrastructure (EGI[1]). These gateways offer applications deployed as workflows on shared computing platforms, but they have no information about when users will launch them and how long each task will last on a given resource.

Fairness among workflow executions has been addressed in several studies which, however, mostly assume clairvoyant conditions. For instance, the works in [2,1,3,8,9,10] either directly minimize the slowdown (which assumes that makespans can be predicted) or use heuristics assuming that task durations and resources are known. A notable exception is found in [11], where a non-clairvoyant algorithm is proposed: nevertheless, it is purely offline, assuming that the tasks and resources are known and do not vary.

In this work, we propose an algorithm to control fairness on non-clairvoyant online platforms. Based on a progressive discovery of applications' characteristics on the infrastructure, our method dynamically estimates the fraction of pending work for each workflow. Task priorities are then adjusted to harmonize this fraction among active workflows. This way, resources are allocated to application workflows relatively to their amount of work to compute. The method is implemented in VIP, and evaluated with different workflows, in production conditions, on the EGI. We use the slowdown as a *post-mortem* metric, to evaluate our method once execution times are known. Contributions of this paper are:

1. A new instantiation of our control loop [12] to handle unfairness, consisting of (i) an online, non-clairvoyant fairness metric, and (ii) a task prioritization algorithm.
2. Experiments demonstrating that this method improves fairness compared to a first-come-first-served approach, in production conditions, and using 4 different applications.

The next section details our fairness control process, and section 3 presents experiments and results.

2 Fairness Control Process

Workflows are directed graphs of activities spawning sequential tasks for which the executable and input data are known, but the computational cost and produced data volume are not. Workflow graphs may include conditional and loop operators . Algorithm 1 summarizes our fairness control process. Fairness is controlled by allocating resources to workflows according to their fraction of pending work. It is done by re-prioritising tasks in workflows where the unfairness degree η_u is greater than a threshold τ_u. This section describes how η_u and τ_u are computed, and details the re-prioritization algorithm.

Measuring Unfairness: $\boldsymbol{\eta_u}$. Let m be the number of workflows with an active activity; a workflow activity is active if it has at least one waiting (queued) or

[1] `http://www.egi.eu`

Algorithm 1. Main loop for fairness control

```
1: input: $m$ workflow executions
2: while there is an active workflow do
3:     wait for timeout or task status change in any workflow
4:     determine unfairness degree $\eta_u$
5:     if $\eta_u > \tau_u$ then
6:         re-prioritize tasks using Algorithm 2
7:     end if
8: end while
```

running task. The unfairness degree η_u is the maximum difference between the fractions of pending work:

$$\eta_u = W_{\max} - W_{\min}, \tag{1}$$

with $W_{\min} = \min\{W_i, i \in [1, m]\}$ and $W_{\max} = \max\{W_i, i \in [1, m]\}$. All W_i are in $[0, 1]$. For $\eta_u = 0$, we consider that resources are fairly distributed among all workflows; otherwise, some workflows consume more resources than they should. The fraction of pending work W_i of a workflow $i \in [1, m]$ is defined from the fraction of pending work $w_{i,j}$ of its n_i active activities:

$$W_i = \max_{j \in [1, n_i]} (w_{i,j}) \tag{2}$$

All $w_{i,j}$ are between 0 and 1. A high $w_{i,j}$ value indicates that the activity has a lot of pending work compared to the others. We define $w_{i,j}$ as:

$$w_{i,j} = \frac{Q_{i,j}}{Q_{i,j} + R_{i,j} P_{i,j}} \cdot \hat{T}_{i,j}, \tag{3}$$

where $Q_{i,j}$ is the number of waiting tasks in the activity, $R_{i,j}$ is the number of running tasks in the activity, $P_{i,j}$ is the performance of the activity, and $\hat{T}_{i,j}$ is its relative observed duration. $\hat{T}_{i,j}$ is defined as the ratio between the median duration $\tilde{t}_{i,j}$ of the completed tasks in activity j and the maximum median task duration among all active activities of all running workflows:

$$\hat{T}_{i,j} = \frac{\tilde{t}_{i,j}}{\max_{v \in [1,m], w \in [1, n_i^*]} (\tilde{t}_{v,w})} \tag{4}$$

Tasks of an activity all consist of the following successive phases: `setup`, `inputs download`, `application execution` and `outputs upload`; $\tilde{t}_{i,j}$ is computed as $\tilde{t}_{i,j} = \tilde{t}_{i,j}^{setup} + \tilde{t}_{i,j}^{input} + \tilde{t}_{i,j}^{exec} + \tilde{t}_{i,j}^{output}$. Medians are progressively estimated as tasks complete. At the beginning of the execution, $\hat{T}_{i,j}$ is initialized to 1 and all medians are undefined; when two tasks of activity j complete, $\tilde{t}_{i,j}$ is updated and $\hat{T}_{i,j}$ is computed with equation 4. In this equation, the max operator is computed only on $n_i^* \leq n_i$ activities with at least 2 completed tasks, i.e. for which $\tilde{t}_{i,j}$ can be determined. We are aware that using the median may be inaccurate. However, without a model of the applications' execution time, we must rely on observed task durations. Using the whole time distribution (or at least its few first moments) may be more accurate but it would complexify the method.

In Eq. 3, the performance $P_{i,j}$ of an activity varies between 0 and 1. A low $P_{i,j}$ indicates that resources allocated to the activity have bad performance for the activity; in this case, the contribution of running tasks is reduced and $w_{i,j}$ increases. Conversely, a high $P_{i,j}$ increases the contribution of running tasks, therefore decreases $w_{i,j}$. For an activity j with k_j active tasks, we define $P_{i,j}$ as:

$$P_{i,j} = 2\left(1 - \max_{u\in[1,k_j]}\left\{\frac{t_u}{\tilde{t}_{i,j} + t_u}\right\}\right), \tag{5}$$

where $t_u = t_u^{setup} + t_u^{input} + t_u^{exec} + t_u^{output}$ is the sum of the estimated durations of task u's phases. Estimated task phase durations are computed as the max between the current elapsed time in the task phase (0 if the task phase has not started) and the median duration of the task phase. $P_{i,j}$ is initialized to 1, and updated using Eq. 5 only when at least 2 tasks of activity j are completed.

If all tasks perform as the median, i.e. $t_u = \tilde{t}_{i,j}$, then $\max_{u\in[1,k_j]}\left\{\frac{t_u}{\tilde{t}_{i,j}+t_u}\right\} = 0.5$ and $P_{i,j} = 1$. Conversely, if a task in the activity is much longer than the median, i.e. $t_u \gg \tilde{t}_{i,j}$, then $\max_{u\in[1,k_j]}\left\{\frac{t_u}{\tilde{t}_{i,j}+t_u}\right\} \approx 1$ and $P_{i,j} \approx 0$. This definition of $P_{i,j}$, considers that bad performance results in a few tasks blocking the activity. Indeed, we assume that the scheduler doesn't deliberately favor any activity and that performance discrepancies are manifested by a few "unlucky" tasks slowed down by bad resources. Performance, in this case, has a relative definition: depending on the activity profile, it can correspond to CPU, RAM, network bandwidth, latency, or a combination of those. We admit that this definition of $P_{i,j}$ is a bit rough. However, under our non-clairvoyance assumption, estimating resource performance for the activity more accurately is hardly possible because (i) we have no model of the application, therefore task durations cannot be predicted from CPU, RAM or network characteristics, and (ii) network characteristics and even available RAM are shared among concurrent tasks running on the infrastructure, which makes them hardly measurable.

Thresholding Unfairness: τ_u. Task prioritisation is triggered when the unfairness degree is considered critical, i.e $\eta_u > \tau_u$. Thresholding consists in clustering platform configurations in two groups: one for which unfairness is considered acceptable, and one for which task re-prioritization is needed. We determine τ_u from execution traces, for which different thresholding approaches can be used. For instance, we could consider that x% of the platform configurations are unfair while the rest are acceptable. The choice of x, however, would be arbitrary. Instead, we inspect the modes of the distribution of η_u to determine a threshold with a practical justification: values of η_u in the highest mode of the distribution, i.e. which are clearly separated from the others, will be considered unfair.

In this work, the distribution of η_u is measured from traces collected in VIP between January 2011 and April 2012 [13]. The data set contains 680, 988 tasks (including resubmissions and replications) of 2, 941 workflow executions executed by 112 users; task average queueing time is about 36 min. Applications deployed in VIP are described as GWENDIA workflows [14] executed using the MOTEUR workflow engine [15]. Resource provisioning and task scheduling are provided by

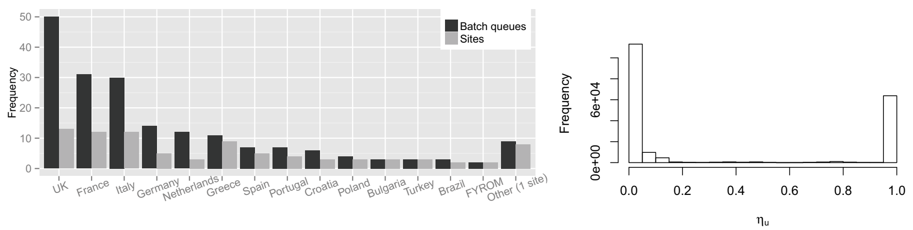


Fig. 1. Distribution of sites and batch queues per country in the biomed VO (January 2013) (*left*) and histogram of the unfairness degree η_u sampled in bins of 0.05 (*right*)

DIRAC [16]. Resources are provisioned online with no advance reservations. Tasks are executed on the biomed virtual organization (VO) of the European Grid Infrastructure (EGI) which has access to some 150 computing sites world-wide and to 120 storage sites providing approximately 4 PB of storage. Fig. 1 (left) shows the distribution of sites per country supporting the biomed VO.

The unfairness degree η_u was computed after each event found in the data set. Fig. 1 (right) shows the histogram of these values, where only $\eta_u \neq 0$ values are represented. This histogram is clearly bi-modal, which is a good property since it reduces the influence of τ_u. From this histogram, we choose $\tau_u = 0.2$. For $\eta_u > 0.2$, task prioritization is triggered.

Task Prioritization. The action taken to cope with unfairness is to increase the priority of $\Delta_{i,j}$ waiting tasks for all activities j of workflow i where $w_{i,j} - W_{\min} > \tau_u$. Running tasks cannot be pre-empted. Task priority is an integer initialized to 1. $\Delta_{i,j}$ is determined so that $\tilde{w}_{i,j} = W_{min} + \tau_u$, where $\tilde{w}_{i,j}$ is the estimated value of $w_{i,j}$ after $\Delta_{i,j}$ tasks are prioritized. We approximate $\tilde{w}_{i,j}$ as:

$$\tilde{w}_{i,j} = \frac{Q_{i,j} - \Delta_{i,j}}{Q_{i,j} + R_{i,j}P_{i,j}}\hat{T}_{i,j},$$

which assumes that $\Delta_{i,j}$ tasks will move from status queued to running, and that the performance of new resources will be maximal. It gives:

$$\Delta_{i,j} = Q_{i,j} - \left\lfloor \frac{(\tau_u + W_{\min})(Q_{i,j} + R_{i,j}P_{i,j})}{\hat{T}_{i,j}} \right\rfloor, \tag{6}$$

where $\lfloor\rfloor$ rounds a decimal down to the nearest integer value.

Algorithm 2 describes our task re-prioritization. *maxPriority* is the maximal priority value in all workflows. The priority of $\Delta_{i,j}$ waiting tasks is set to *maxPriority+1* in all activities j of workflows i where $w_{i,j} - W_{min} > \tau_u$. Note that this algorithm takes into account scatter among W_i although η_u ignores it (see Eq. 1). Indeed, tasks are re-prioritized in *any* workflow i for which $W_i - W_{\min} > \tau_u$.

The method also accommodates online conditions. If a new workflow i is submitted, then $R_{i,j} = 0$ for all its activities and $\hat{T}_{i,j}$ is initialized to 1. This leads to $W_{max} = W_i = 1$, which increases η_u. If η_u goes beyond τ_u, then $\Delta_{i,j}$ tasks of activity j of workflow i have their priorities increased to restore fairness. Similarly, if new resources arrive, then $R_{i,j}$ increase and η_u is updated accordingly. Table 1 illustrates the method on a simple example.

Algorithm 2. Task re-prioritization

```
input: $W_1$ to $W_m$ //fractions of pending works
maxPriority = max task priority in all workflows
for i=1 to m do
    if $W_i - W_{\min} > \tau_u$ then
        for j=1 to $a_i$ do
            //$a_i$ is the number of active activities in workflow $i$
            if $w_{i,j} - W_{\min} > \tau_u$ then
                Compute $\Delta_{i,j}$ from equation 6
                for p=1 to $\Delta_{i,j}$ do
                    if $\exists$ waiting task q in activity $j$ with priority $\leq$ maxPriority then
                        $q$.priority = maxPriority + 1
                    end if
                end for
            end if
        end for
    end if
end for
```

3 Experiments and Results

Experiments are performed on a production grid platform to ensure realistic conditions. Evaluating fairness in production by measuring the slowdown is not straightforward because M_{own} (see definition in the introduction) cannot be directly measured. As described in Section 3.1, we estimate the slowdown from task durations, but this estimation may be challenged. Thus, Experiment 1 evaluates our method on a set of identical workflows, where the variability of the measured makespan can be used as a fairness metric. In Experiment 2, we add a very short workflow to this set of identical workflow, which was one of the configurations motivating this study. Finally, Experiment 3 considers the more general case of 4 different workflows with heterogeneous durations.

3.1 Experiment Conditions

Fairness control was implemented as a MOTEUR plugin receiving notifications about task and workflow status changes. Each workflow plugin forwards task status changes and $\tilde{t}_{i,j}$ values to a service centralizing information about all the active workflows. This service then re-prioritizes tasks according to Algorithms 1 and 2. As no online task modification is possible in DIRAC, we implemented task prioritization by canceling and resubmitting queued tasks to DIRAC with new priorities. This implementation decision adds an overhead to task executions. Therefore, the timeout value used in Algorithm 1 is set to 3 minutes.

The computing platform for these experiments is the biomed VO used to determine τ_u in Section 2. To ensure resource limitation without flooding the production system, experiments are performed only on 3 sites of different countries (France, Spain and Netherlands). Four real medical simulation workflows are considered: `GATE` [17], `SimuBloch`, `FIELD-II` [18], and `PET-Sorteo` [19]; their main characteristics are summarized on Table 2.

Three experiments are conducted. Experiment 1 tests whether unfairness among *identical workflows* is properly addressed. It consists of three `GATE` workflows sequentially submitted, as users usually do in the platform. Experiment 2

Table 1. Example

Let's consider two identical workflows composed of one activity with 6 tasks, and assume the following values at time t:

i	$Q_{i,1}$	$R_{i,1}$	$\tilde{t}_{i,1}$	$P_{i,1}$	$\hat{T}_{i,1}$	$W_i = w_{i,1}$
1	1	3	10	0.9	1.0	0.27
2	6	0	-	1.0	1.0	1.00

Values unknown at time t are noted '-'. Workflow 1 has 2 completed and 3 running tasks with the following phase durations (in arbitrary time units):

u	t_u^{setup}	t_u^{input}	t_u^{exec}	t_u^{output}	t_u
1	2	2	4	1	9
2	1	2	3	2	8
3	2	3	5	-	-
4	2	2	-	-	-
5	1	-	-	-	-

, We have $\tilde{t}_{1,1}^{setup} = 2$, $\tilde{t}_{1,1}^{input} = 2$, $\tilde{t}_{1,1}^{exec} = 4$ and $\tilde{t}_{1,1}^{output} = 2$. Therefore, $\tilde{t}_{1,1} = 10$.

The configuration is clearly unfair since workflow 2 has not started tasks.
Eq. 1 gives $\eta_u = 0.73$. As $\eta_u > \tau_u = 0.2$, the platform is considered unfair and task re-prioritization is triggered.

$\Delta_{2,1}$ tasks from workflow 2 should be prioritized. According to Eq. 6:
$\Delta_{2,1} = Q_{2,1} - \left\lfloor \frac{(\tau_u + W_1)(Q_{2,1} + R_{2,1}P_{2,1})}{\hat{T}_{2,1}} \right\rfloor = 6 - \left\lfloor \frac{(0.2+0.27)(6+0\cdot 1.0)}{1.0} \right\rfloor = 4$

At time $t' > t$:

i	$Q_{i,1}$	$R_{i,1}$	$\tilde{t}_{i,1}$	$P_{i,1}$	$\hat{T}_{i,1}$	$W_i = w_{i,1}$
1	1	3	10	0.8	1.0	0.29
2	2	4	-	1.0	1.0	0.33

.

Now, $\eta_u = 0.04 < \tau_u$. The platform is considered fair and no action is performed.

tests if the performance of *very short workflow executions* is improved by the fairness mechanism. Its workflow set has three `GATE` workflows launched sequentially, followed by a `SimuBloch` workflow. Experiment 3 tests whether unfairness among *different workflows* is detected and properly handled. Its workflow set consists of a `GATE`, a `FIELD-II`, a `PET-Sorteo` and a `SimuBloch` workflow launched sequentially.

For each experiment, a workflow set using our fairness mechanism (`Fairness – F`) is compared to a control workflow set (`No-Fairness – NF`). No method from the literature could be included in the comparison because, as mentioned in the introduction, they are either non-clairvoyant or offline. `Fairness` and `No-Fairness` are launched simultaneously to ensure similar grid conditions. For each task priority increase in the `Fairness` workflow set, a task in the `No-Fairness` workflow set task queue is also prioritized to ensure equal race conditions for resource allocation. Experiment results are not influenced by the re-submission process overhead since both `Fairness` and `No-Fairness` experience the same overhead. Four repetitions of each experiment are done, along a time period of four weeks to cover different grid conditions. Grid conditions vary among repetitions because computing, storage and network resources are shared

Table 2. Workflow characteristics ($\rightarrow$ indicate task dependencies)

Workflow		#Tasks	CPU time	Input	Output
GATE	(CPU-intensive)	100	few minutes to one hour	~115 MB	~40 MB
SimuBloch	(data-intensive)	25	few seconds	~15 MB	< 5 MB
FIELD-II	(data-intensive)	122	few seconds to 15 minutes	~208 MB	~40 KB
PET-Sorteo	(CPU-intensive)	$1\rightarrow80\rightarrow1\rightarrow80\rightarrow1\rightarrow1$	~10 minutes	~20 MB	~50 MB

with other users . We use MOTEUR 0.9.21, configured to resubmit failed tasks up to 5 times, and with the task replication mechanism described in [12] activated. We use the DIRAC v6r5p1 instance provided by France-Grilles[2], with a first-come, first-served policy imposed by submitting workflows with decreasing priority values.

Two different fairness metrics are used. The unfairness μ is the area under the curve η_u during the execution:

$$\mu = \sum_{i=2}^{M} \eta_u(t_i) \cdot (t_i - t_{i-1}),$$

where M is the number of time samples until the makespan. This metric measures if the fairness process can indeed minimize its own criterion η_u. In addition, the slowdown s of a completed workflow execution is defined by:

$$s = \frac{M_{multi}}{M_{own}}$$

where M_{multi} is the makespan observed on the shared platform, and M_{own} is the estimated makespan if it was executed alone on the platform. In our conditions, M_{own} is estimated as:

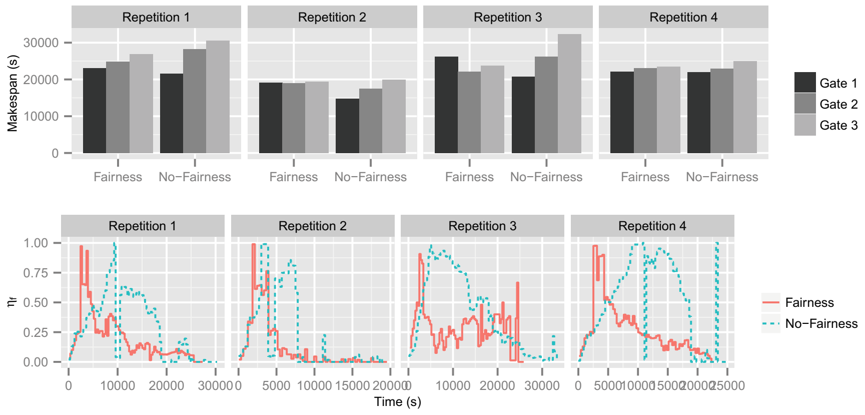
$$M_{own} = \max_{p \in \Omega} \sum_{u \in p} t_u,$$

where Ω is the set of task paths in the workflow, and t_u is the measured duration of task u. This assumes that concurrent executions only impact task waiting time, not performance. For instance, network congestion or changes in performance distribution resulting from concurrent executions are ignored. We use σ_s, the standard deviation of the slowdown to quantify the unfairness. In Experiment 1, the standard deviation of the makespan (σ_m) is also used.

3.2 Results and Discussion

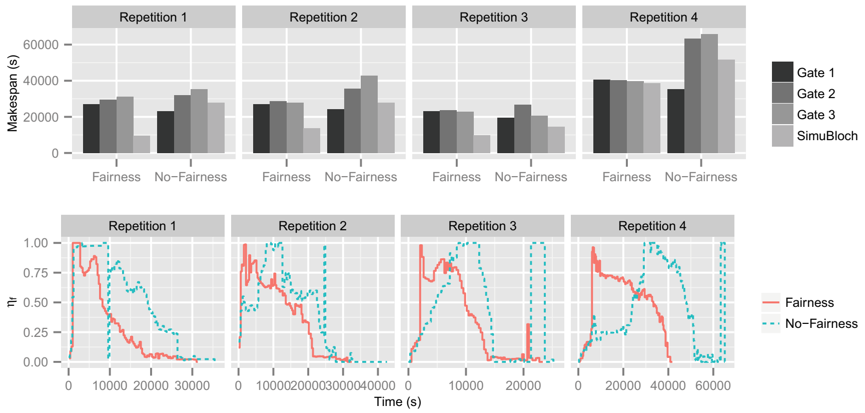
Experiment 1 (*identical workflows*): Fig. 2 shows the makespan, unfairness degree η_u, makespan standard deviation σ_m, slowdown standard deviation σ_s and unfairness μ for the 4 repetitions. The difference among makespans and unfairness degree values are significantly reduced in all repetitions of `Fairness`. Both `Fairness` and `No-Fairness` behave similarly until η_u reaches the threshold value $\tau_u = 0.2$. Unfairness is then detected and the mechanism triggers task prioritization. Paradoxically, the first effect of task prioritization is a slight increase of η_u. Indeed, $P_{i,j}$ and $\hat{T}_{i,j}$, that are initialized to 1, start changing earlier in `Fairness` than in `No-Fairness` due to the availability of task duration values to compute $\tilde{t}_{i,j}$. Note that η_u reaches similar maximal values in both cases, but reaches them faster in `Fairness`. The fairness mechanism then manages to decrease η_u back under 0.2 much faster than it happens in `No-Fairness` when tasks progressively complete. Finally, slight increases of η_u are sporadically observed towards the end of the execution. This is due to task replications performed by MOTEUR:

[2] `https://dirac.france-grilles.fr`



	Repetition 1				Repetition 2				Repetition 3				Repetition 4		
	$\sigma_m(s)$	σ_s	$\mu(s)$		$\sigma_m(s)$	σ_s	$\mu(s)$		$\sigma_m(s)$	σ_s	$\mu(s)$		$\sigma_m(s)$	σ_s	$\mu(s)$
NF	4666	1.03	8758	NF	2541	0.50	4154	NF	5791	2.10	13392	NF	1567	0.87	12283
F	1884	0.40	5292	F	167	0.07	2367	F	2007	0.84	7243	F	706	0.24	6070

Fig. 2. Experiment 1 (identical workflows). Top: comparison of the makespans; middle: unfairness degree η_u; bottom: makespan standard deviation σ_m, slowdown standard deviation σ_s and unfairness μ.



	Repetition 1			Repetition 2			Repetition 3			Repetition 4	
	σ_s	$\mu(s)$		σ_s	$\mu(s)$		σ_s	$\mu(s)$		σ_s	$\mu(s)$
NF	94.88	17269	NF	100.05	16048	NF	87.93	11331	NF	213.60	28190
F	15.95	9085	F	42.94	12543	F	57.62	7721	F	76.69	21355

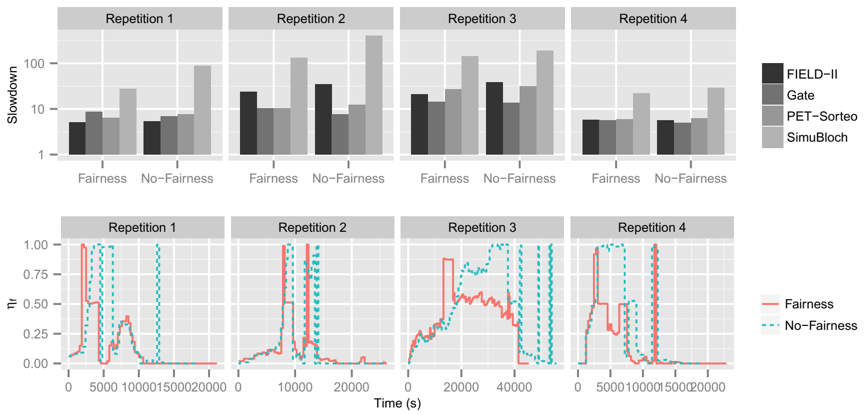
Fig. 3. Experiment 2 (very short execution). Top: comparison of the makespans; middle: unfairness degree η_u; bottom: unfairness μ and slowdown standard deviation.

when new tasks are created, the fraction of pending work W increases, which has an effect on η_u. Quantitatively, the fairness mechanism reduces σ_m up to a factor of 15, σ_s up to a factor of 7, and μ by about 2.

Experiment 2 (*very short execution*): Fig. 3 shows the makespan, unfairness degree η_u, unfairness μ and slowdown standard deviation. In all cases, the makespan of the very short `SimuBloch` executions is significantly reduced for

Table 3. Experiment 2: `SimuBloch`'s makespan, average wait time and slowdown

Run	Type	m (secs)	$\bar{w}$ (secs)	s
1	No-Fairness	27854	18983	196.15
	Fairness	9531	4313	38.43
2	No-Fairness	27784	19105	210.48
	Fairness	13761	10538	94.25
3	No-Fairness	14432	13579	182.68
	Fairness	9902	8145	122.25
4	No-Fairness	51664	47591	445.38
	Fairness	38630	27795	165.79



Repetition 1		
	σ_s	μ (s)
NF	40.29	4443
F	10.52	2689

Repetition 2		
	σ_s	μ (s)
NF	192.58	4004
F	58.83	2653

Repetition 3		
	σ_s	μ (s)
NF	81.81	25173
F	60.56	18537

Repetition 4		
	σ_s	μ (s)
NF	11.61	6613
F	8.10	3303

Fig. 4. Experiment 3 (different workflows). Top: comparison of the slowdown; middle: unfairness degree η_u; bottom: unfairness μ and slowdown standard deviation.

`Fairness`. The evolution of η_u is coherent with Experiment 1: a common initialization phase followed by an anticipated growth and decrease for `Fairness`. `Fairness` reduces σ_s up to a factor of 5.9 and unfairness up to a factor of 1.9.

Table 3 shows the execution makespan (m), average wait time ($\bar{w}$) and slowdown (s) values for the `SimuBloch` execution launched after the 3 `GATE`. As it is a non-clairvoyant scenario where no information about task execution time and future task submission is known, the fairness mechanism is not able to give higher priorities to `SimuBloch` tasks in advance. Despite that, the fairness mechanism speeds up `SimuBloch` executions up to a factor of 2.9, reduces task average wait time up to factor of 4.4 and reduces slowdown up to a factor of 5.9.

Experiment 3 (*different workflows*): Fig. 4 shows slowdown, unfairness degree, unfairness μ and slowdown standard deviation σ_s for the 4 repetitions. `Fairness` slows down `GATE` while it speeds up all other workflows. This is because `GATE` is the longest and the first to be submitted; in `No-Fairness`, it is favored by resource allocation to the detriment of other workflows. The evolution of η_u is similar to Experiments 1 and 2. σ_s is reduced up to a factor of 3.8 and unfairness up to a factor of 1.9.

In all 3 experiments, fairness optimization takes time to begin because the method needs to acquire information about the applications which are totally unknown when a workflow is launched. We could think of reducing the time of this information-collecting phase, e.g. by designing initialization strategies maximizing information discovery, but it couldn't be totally removed. Currently, the method works best for applications with a lot of short tasks because the first few tasks can be used for initialization, and optimization can be exploited for the remaining tasks. The worst-case scenario is a configuration where the number of available resources stays constant and equal to the number of tasks in the first submitted workflow: in this case, no action could be taken until the first workflow completes, and the method would not do better than first-come-first-served. Pre-emption of running tasks should be considered to address that.

4 Conclusion

We presented a method to address unfairness among workflow executions in an online and non-clairvoyant environment. We defined a novel metric η_u quantifying unfairness based on the fraction of pending work in a workflow. It compares workflow activities based on their ratio of queuing tasks, their relative durations, and the performance of resources where tasks are running. Performance is defined from the variability of task duration in the activity: good performance is assumed to lead to homogeneous task durations. To separate fair configurations from unfair ones, a threshold on η_u was determined from platform traces. Unfair configurations are handled by increasing the priority of pending tasks in the least performing workflows. This is done by estimating the number of running tasks that these workflows should have to bring η_u under the threshold value.

The method was implemented in the MOTEUR workflow engine and deployed on EGI with the DIRAC resource manager. We tested it on four applications extracted from VIP, a science gateway for medical simulation used in production. Three experiments were conducted, to evaluate the capability of the method to improve fairness (i) on identical workflows, (ii) on workflow sets containing a very short execution and (iii) on different workflows. In all cases, results showed that our method can very significantly reduce the standard deviation of the slowdown, and the average value of our metric η_u.

The work presented here is a step in our attempt to control computing platforms where very little is known about applications and resources, and where situations change over time. Our works in [12,20] consider similar platform conditions but they target completely different problems, namely fault-tolerance and granularity control. We believe that results of this paper are the first ones presented to control fairness in such conditions which are often met in production platforms. Future work could include task pre-emption in the method, and a sensitivity analysis on the influence of the relative task duration ($T_{i,j}$) and of the performance factor ($P_{i,j}$).

Acknowledgment. This work is funded by the French National Agency for Research under grant ANR-09-COSI-03 "VIP". The research leading to this publication has also received funding from the EC FP7 Programme under grant agreement 312579 ER-flow = Building an European Research Community through

Interoperable Workflows and Data, and the framework LABEX ANR-11-LABX-0063 of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR). Results obtained in this paper were computed on the biomed virtual organization of the European Grid Infrastructure (http://www.egi.eu). We thank the European Grid Infrastructure and its supporting National Grid Initiatives, in particular France-Grilles, for providing the technical support, computing and storage facilities.

References

1. N'Takpe, T., Suter, F.: Concurrent scheduling of parallel task graphs on multi-clusters using constrained resource allocations. In: IPDPS 2009, pp. 1–8 (2009)
2. Zhao, H., Sakellariou, R.: Scheduling multiple DAGs onto heterogeneous systems. In: IPDPS 2006, p. 159 (2006)
3. Casanova, H., Desprez, F., Suter, F.: On cluster resource allocation for multiple parallel task graphs. J. of Par. and Dist. Computing 70(12), 1193–1203 (2010)
4. Glatard, T., et al.: A virtual imaging platform for multi-modality medical image simulation. IEEE Trans. Med. Imaging 32, 110–118 (2013)
5. Shahand, S., et al.: Front-ends to Biomedical Data Analysis on Grids. In: Proceedings of HealthGrid 2011, Bristol, UK (June 2011)
6. Kacsuk, P.: P-GRADE Portal Family for Grid Infrastructures. Concurrency and Computation: Practice and Experience 23(3), 235–245 (2011)
7. Roberto: Supporting e-science applications on e-infrastructures: Some use cases from latin america. In: Grid Computing, pp. 33–55 (2011)
8. Hsu, C.C., Huang, K.C., Wang, F.J.: Online scheduling of workflow applications in grid environments. Fut. Gen. Computer Systems 27(6), 860–870 (2011)
9. Arabnejad, H., Barbosa, J.: Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems. In: ISPA 2012, pp. 633–639 (July 2012)
10. Sabin, G., Kochhar, G., Sadayappan, P.: Job fairness in non-preemptive job scheduling. In: ICPP 2004, pp. 186–194 (2004)
11. Hirales-Carbajal, A., et al.: Multiple workflow scheduling strategies with user run time estimates on a grid. Journal of Grid Computing 10, 325–346 (2012)
12. Ferreira da Silva, R., Glatard, T., Desprez, F.: Self-healing of operational workflow incidents on distributed computing infrastructures. In: CCGrid 2012, pp. 318–325 (2012)
13. Ferreira da Silva, R., Glatard, T.: A Science-Gateway Workload Archive to Study Pilot Jobs, User Activity, Bag of Tasks, Task Sub-Steps, and Workflow Executions. In: CoreGRID/ERCIM Workshop on Grids, Clouds and P2P Computing (2012)
14. Montagnat, J., et al.: A data-driven workflow language for grids based on array programming principles. In: WORKS 2009, Portland, USA, pp. 1–10. ACM (2009)
15. Glatard, T., et al.: Flexible and Efficient Workflow Deployment of Data-Intensive Applications on Grids with MOTEUR. IJHPCA 22(3), 347–360 (2008)
16. Tsaregorodtsev, A., et al.: DIRAC3. The New Generation of the LHCb Grid Software. Journal of Physics: Conference Series 219(6), 062029 (2009)
17. Jan, S., et al.: GATE V6: a major enhancement of the GATE simulation platform enabling modelling of CT and radiotherapy. Phys. Med. Biol. 56(4), 881–901 (2011)
18. Jensen, J., Svendsen, N.B.: Calculation of pressure fields from arbitrarily shaped, apodized, and excited ultrasound transducers. IEEE UFFC 39(2), 262–267 (1992)
19. Reilhac, A., et al.: PET-SORTEO: Validation and Development of Database of Simulated PET Volumes. IEEE Trans. on Nuclear Science 52, 1321–1328 (2005)
20. da Silva, R.F., Glatard, T., Desprez, F.: On-line, non-clairvoyant optimization of workflow activity granularity on grids. In: Wolf, F., Mohr, B., an Ney, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 255–266. Springer, Heidelberg (2013)

How to be a Successful Thief

Feudal Work Stealing for Irregular Divide-and-Conquer Applications on Heterogeneous Distributed Systems

Vladimir Janjic and Kevin Hammond

School of Computer Science, University of St Andrews, Scotland, UK
vj32@st-andrews.ac.uk, kh@cs.st-andrews.ac.uk

Abstract. Work Stealing has proved to be an effective method for load balancing regular divide-and-conquer (D&C) applications on heterogeneous distributed systems, but there have been relatively few attempts to adapt it to address *irregular* D&C applications. For such applications, it is essential to have a mechanism that can estimate dynamic system load during the execution of the applications. In this paper, we evaluate a number of work-stealing algorithms on a set of generic Unbalanced Tree Search (UTS) benchmarks. We present a novel *Feudal Stealing* work-stealing algorithm and show, using simulations, that it delivers consistently better speedups than other work-stealing algorithms for irregular D&C applications on high-latency heterogeneous distributed systems. Compared to the best known work-stealing algorithm for high-latency distributed systems, we achieve improvements of between 9% and 48% for irregular D&C applications.

Keywords: Irregular Parallelism, Work Stealing, Divide-and-Conquer, Heterogeneous Systems.

1 Introduction

Work stealing [4], where idle "thieves" steal work from busy "victims", is one of the most appealing load-balancing methods for distributed systems, due to its inherently distributed and scalable nature. Several good work-stealing algorithms have been devised for distributed systems with non-uniform communication latencies [2,3,14,15,13]. Most of these algorithms are, however, tailored to *regular* Divide-and-Conquer (D&C) applications. In scientific computations, for example, it is very common to encounter *irregular* D&C applications, where the structure of parallelism for different application tasks varies quite significantly.

We previously showed [10] that in order to obtain good speedups for irregular D&C applications, it is essential to use *dynamic system load information* as part of a work-stealing algorithm. In this paper, we consider several algorithms that can be used to obtain suitable load information. In particular, we compare *centralised* (where this information is kept on a fixed set of nodes), *distributed* (where this information is exchanged in a peer-to-peer way between all nodes)

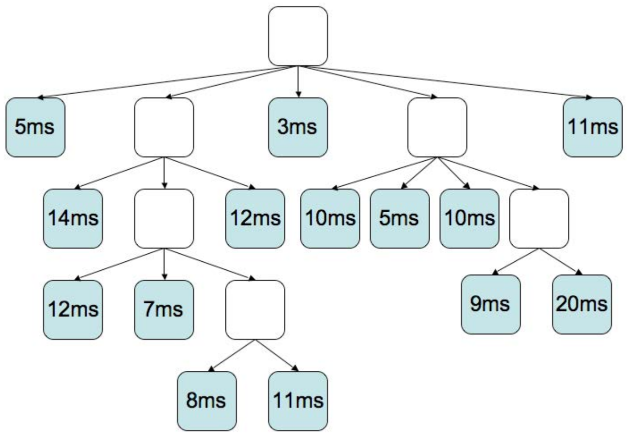


Fig. 1. Example task graph for an irregular D&C application

and *hybrid* (a combination of the two) methods for obtaining load information. We also describe a novel *Feudal Stealing* algorithm, a new algorithm that exploits dynamic load information, but which significantly outperforms other similar work-stealing algorithms in terms of the speedups that are obtained. This paper makes the following specific research contributions:

– We present a novel *Feudal Stealing* algorithm, which uses hybrid dynamic load information approach;
– We evaluate representative state-of-the-art work-stealing algorithms that use centralised, distributed and hybrid dynamic load information on the Unbalanced Tree Search (UTS) benchmark of irregular D&C applications;
– We show, using simulations, that our new Feudal Stealing algorithm delivers notably better speedups on irregular D&C applications than the other state-of-the-art algorithms we consider.

2 Work Stealing for Divide-and-Conquer Applications

Divide-and-conquer (D&C) applications can be represented by task trees, where nodes in the tree represent tasks, and edges represent parent-child relationships. One especially interesting class is that of *irregular* D&C applications, whose task trees are highly unbalanced (see Fig. 1), and which frequently occur in scientific computations (e.g. Monte-Carlo Photon Transport simulations [7], implementations of the Min-Max algorithm and Complex Polynomial Root Search [6]). They are also frequently used as benchmarks for evaluating load-balancing algorithms (e.g. the Unbalanced Tree Search [12] and Bouncing Producer-Consumer [5] benchmarks). We focus on load balancing irregular D&C applications on widely distributed systems, which comprise a set of *clusters* communicating over high-latency networks. Each cluster comprises a set of *processing nodes*, where nodes from the same cluster are connected by fast, local-area networks. Such systems correspond to grid-like or cloud-like server farms. They are typically highly heterogeneous both in terms of the characteristics of the individual nodes (processing power, memory characteristics etc.) and the networks that connect clusters (communication latency, bandwidth etc.). This paper focuses on how to deal with heterogeneous communication latencies in such systems.

Each processing node stores the tasks that it creates in its own *task pool.* Load balancing between nodes uses *work stealing.* In such a setting, when a node's task pool becomes empty, that node becomes a *thief.* The thief sends a *steal attempt* message to its chosen *target* node. If the target has more than one task in its task pool, it becomes a *victim* and returns one (or more) tasks to the thief. Otherwise, the target can either forward the steal attempt to some other target or, alternatively, it can send a negative response to the thief, which then deals with it in an appropriate way (either by initiating another steal attempt or by delaying further stealing). To make our discussion more focused, we assume that only one task is sent from the victim to the thief, and that any targets that do not have any work will always forward the steal attempt to some other target. In addition, following the usual practice in work stealing for divide-and-conquer applications, we assume that the victim always sends the *oldest* task from its task pool to the thief. In divide-and-conquer applications, this usually corresponds to the largest task, that also generates the most additional parallelism. Finally, we assume that execution starts with one node executing the main task (the root of the task tree), and all other nodes are thieves that start stealing immediately.

In order to hide the potentially high communication latencies that can occur in distributed systems, it is essential that work-stealing algorithms employ good methods for selecting targets. This serves to minimise the number of wasted messages. Work-stealing algorithms can be divided into two broad classes, according to the type of information that they use for the target selection: i) algorithms that only use static information about network topology; and ii) algorithms that also use dynamic load information.

2.1 Algorithms That Use Only Network Topology Information

The most important algorithms that use only network topology information are:

- *Random Stealing* [3], where a thief always selects a random target.
- *Hierarchical Stealing* [2], where nodes are organized into a tree, according to communication latencies. A thief first tries stealing from closer nodes (i.e. its descendants in the tree). If it fails to obtain any work there, it attempts to steal from its parent, which then repeats the same procedure.
- *(Adaptive) Cluster-Aware Random Stealing* (CRS) [14], where each node divides the set of all other nodes into two sets: i) nodes from the same cluster (*local nodes*), and ii) those outside of it (*remote nodes*). A thief attempts to steal *in parallel* from a random local node and a random remote node.In this way, thieves hope to obtain work locally, but remote stealing will prefetch work. In the adaptive variant, thieves prefer closer clusters.

We have previously shown [10] that these algorithms perform well for regular D&C applications, but that they run into problems for highly-irregular applications. In such applications, all the tasks may be concentrated on relatively few nodes, and the semi-randomised way of selecting targets that is employed by these algorithm can consequently perform badly. We have also shown that it would be beneficial to extend them with mechanisms for obtaining and using dynamic load information. Our main objective in the rest of the paper is to describe and compare different methods for obtaining and using load information in work-stealing, in order to discover the methods that give the best estimation of system load and the best overall speedups. We discuss existing methods in the next section, our novel Feudal Stealing method is described in Section 3.

2.2 Algorithms That Use Dynamic Load Information

Some work-stealing algorithms use dynamic load information to estimate the size of node task pools, and so inform the choice of target. In algorithms with *centralised* load information, a fixed set of nodes is responsible for managing load information. These nodes act as routers for steal attempts, forwarding them to targets. There are two basic ways to do this:

1. *Fully Centralised* methods, where all nodes periodically send their load information to a single *central* node. A thief sends a steal attempt to the central node, and this is forwarded to the victim that is *nearest* to the thief[1].
2. *Hierarchical Load-Aware Stealing (HLAS)* [11], analogous to Hierarchical Stealing. Each node periodically sends its load information to its parent. Based on its load information, a thief then attempts to steal from a child with non-zero load. If all children have zero load, the steal attempt is sent to the thief's parent, which then repeats the same procedure for finding work.

The main appeal of such algorithms is that load information is updated regularly, and it will therefore be relatively accurate. However, this also means that significant strain may be placed on the central nodes, since they have to communicate frequently with the rest of the system. For high-latency networks with fine-grained tasks, this information may also be inaccurate. In contrast, for algorithms using *distributed* load information, each node holds its own approximations of the load of all other nodes. A representative example is the *Grid-GUM Stealing* algorithm [1]. In this algorithm, timestamped load information is attached to each stealing-related message (steal attempts, forwarding of steal attempts etc.). The recipient of a message compares this information against its own load information, and updates both the information that is contained in the message (if the message is to be forwarded further) and its own load information. Provided that nodes frequently exchange load information, they will then obtain good approximations to the system load. In algorithms with distributed load information, work-stealing is still fully decentralised, and no significant overheads

[1] We also considered a variant of this algorithm where the thief forwards the steal attempt to a *random* target with work, rather than to the closest one. This variant, however, delivered consistently worse speedups, so we will not consider it further.

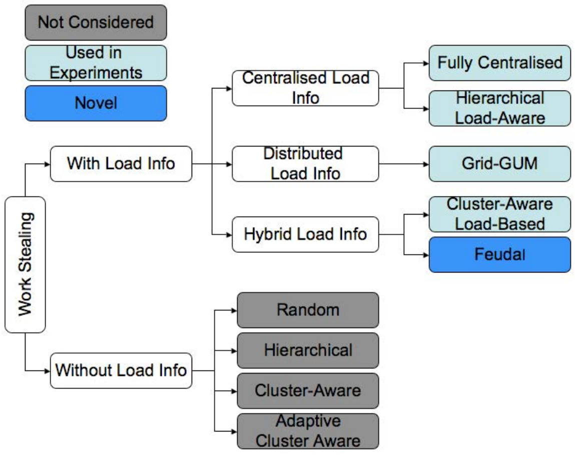


Fig. 2. The work-stealing algorithms that this paper considers

are introduced in approximating system load [1]. However, the accuracy of the load information that a node has (and, conversely, that the rest of the system has about that node) depends on how often it communicates with the rest of the system. An isolated node can easily have outdated load information, and the rest of the system may also have outdated information about its load. *Hybrid* algorithms combine both types of algorithms in order to overcome the disadvantages of each approach. We present one such method here, *Cluster-Aware Load-Based Stealing (CLS)* [14], before introducing our novel extension, *Feudal Stealing*. Like the CRS algorithm, the CLS algorithm considers only two levels of communication latencies (local and remote). In each cluster, one node is nominated as a central node, and every other node in the cluster periodically sends its load information to this node. All nodes in the cluster, apart from the central node, perform only (random) local stealing. When the central node determines that the load of the cluster has dropped below some threshold, it initiates remote stealing from a randomly selected remote node. This approach has several drawbacks. Firstly, tasks that are stolen remotely are always stored on central nodes, which means that additional messages are needed to distribute these tasks to their final destinations. Secondly, as with CRS, when all tasks are concentrated on a few nodes, random remote stealing may be unacceptable. Fig. 2 gives an overview of the work-stealing algorithms that we consider in this paper.

3 Feudal Stealing

Feudal Stealing represents an attempt to combine the best features of the CRS, CLS and Grid-GUM Stealing algorithms while avoiding their drawbacks. Its basic principle is similar to the CRS algorithm. A thief initiates both (random) local and remote stealing in parallel. Remote stealing is done via central nodes (one for each cluster, as in the CLS algorithm). The thief sends a *remote-steal*

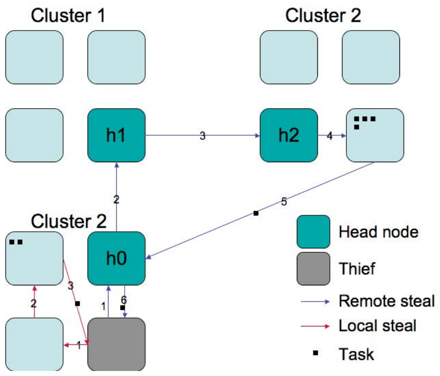


Fig. 3. Overview of Feudal Stealing

message to its central node, which then forwards it to some other, appropriately chosen, central node. When a central node receives a remote-steal message, it forwards it either to some node in its own cluster or to some other central node, depending on the cluster load. When a node is found that has work, one task from its task pool is returned to the original thief via the central nodes.

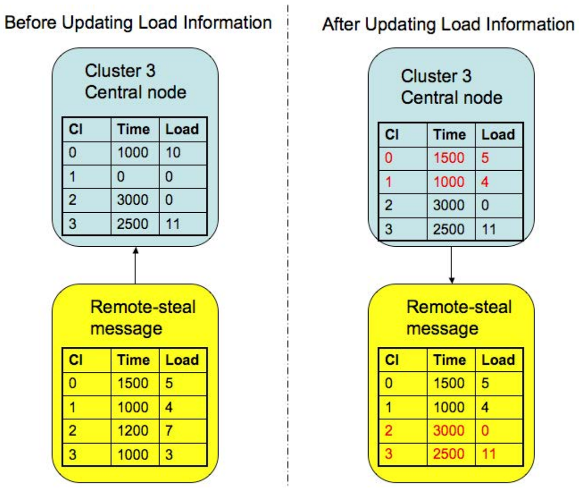
Fig. 3 gives an overview of Feudal Stealing. Here, the grey node is a thief, which initiates remote and local stealing attempts. The local stealing message, whose path is shown by red arrows, visits random local nodes, looking for work. In this case, a suitable local node is found in the second hop, and the work is immediately returned to the thief. The remote stealing message, whose path is shown by blue arrows, starts by visiting the central node of the local cluster, *h0*. It is then forwarded to the other central nodes, until one whose cluster has work is found (the third hop in the figure, *h2*). It is then forwarded to the node within the cluster that has work (hop 4), which returns the work to *h0* (hop 5). This forwards the work to the original thief (hop 6).

In Feudal Stealing, as with the CLS algorithm, central nodes keep load information for their cluster. However, they also keep load information for other clusters, and this information is used when a central node needs to decide where to forward a remote-steal message. The remote-steal message is forwarded to a random central node, with respect to estimated loads of central nodes. Load information is exchanged between central nodes in a similar way to Grid-GUM Stealing. The central node's load approximation is attached to each message that is sent (or forwarded) from that node. Each central node updates its own load information from the messages that it receives (see Fig. 4).

4 Simulation Experiments

For our simulation experiments, we use the publically-available highly-tunable SCALES[2] work-stealing simulator [8]. SCALES has been shown to accurately simulate realistic runtime systems [8,9]. It models irregular D&C applications

[2] Source code is available at `http://www.cs.st-andrews.ac.uk/jv/SCALES.tar.gz`


Before Updating Load Information
After Updating Load Information
Cluster 3 Central node
Remote-steal message

Before Updating Load Information — Cluster 3 Central node:

Cl	Time	Load
0	1000	10
1	0	0
2	3000	0
3	2500	11

Before Updating Load Information — Remote-steal message:

Cl	Time	Load
0	1500	5
1	1000	4
2	1200	7
3	1000	3

After Updating Load Information — Cluster 3 Central node:

Cl	Time	Load
0	1500	5
1	1000	4
2	3000	0
3	2500	11

After Updating Load Information — Remote-steal message:

Cl	Time	Load
0	1500	5
1	1000	4
2	3000	0
3	2500	11

Fig. 4. Exchanging load information between a central node and a message

on systems consisting of inter-connected groups of clusters. We have chosen to use simulations for our experiments, rather than implementation in a real runtime-system, solely in order to abstract from the details of specific runtime systems. This enables us to ignore some specific overheads (e.g. task and thread creation, message processing) that certain runtime systems impose, and which can obstruct the results we are trying to obtain.

Our main benchmark is *Unbalanced Tree Search (UTS)* [12], which simulates state space exploration and combinatorial search problems. UTS is commonly used as a benchmark to test how a system copes with load imbalance. It models a traversal of an implicitly constructed tree, parameterised by shape, depth, size and imbalance. In *binomial* UTS trees, which we denote by UTS(m,q, $rootDeg$, S), each node has m children with probability q, and has no children with probability $1 - q$. $rootDeg$ is the number of children of the root node. When $qm < 1$, this process generates a finite tree with an expected size of $\frac{1}{1-qm}$. The product qm also determines how unbalanced the UTS tree is. As this product approaches 1, the variation in the sizes of subtrees of the nodes increases dramatically. S denotes the cost (in miliseconds) of processing a node. In order to keep the number of experiments manageable, we have decided to model one specific distributed system consisting of 8 clusters, with 8 nodes in each cluster, giving a total of 64 nodes, as shown in Fig. 5. Each cluster is split into two *continental* groups of clusters, with an inter-continental latency of 80ms. Each continental group is further split into two *country* groups, with an inter-country latency of 30ms. The latency between clusters that belong to the same country group was set to be 10ms, and the latency between nodes from the same cluster to be 0.1ms This models the latencies of a system where clusters from different continents, countries and sites within countries are connected into one large supercomputer.

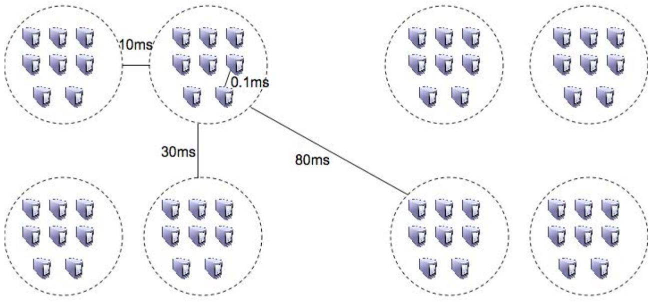


Fig. 5. The model of a system used in simulations

4.1 Results

This section evaluates the speedups that can be obtained by work-stealing algorithms that use dynamic load information. We define speedup to be the ratio of the simulated runtime of an application on a one-node system node to that on the distributed system. In our first set of experiments, we consider the $UTS(7, q, 3000, 5ms)$ applications, where q takes values $0.08, 0.09.0.1, 0.11, 0.12$, 0.13 and 0.14. This represents applications with a large number of tasks (approximately 7011, 8246, 9223, 13044, 18751, 33334 and 150000 respectively), and with increasing irregularity (where as q increases, so the tree becomes more unbalanced). Fig. 6 shows the speedups of these applications under all of the algorithms that we have considered. We observe that for less irregular applications, Feudal Stealing, Grid-GUM Stealing and Fully Centralised Stealing have approximately the same performance. However, for more irregular applications, Feudal Stealing notably outperforms all the other algorithms. We can also observe that algorithms that use distributed load information (Feudal Stealing and Grid-GUM Stealing) generally outperform those that mostly rely on centralised load information. The improvements in speedup that Feudal Stealing brings over the next best algorithm (Grid-GUM) vary from 9% for $UTS(7, 0.11, 3000, 5ms)$ ($qm = 0.77$) to 48% for $UTS(7, 0.14, 3000, 5ms)$ ($qm = 0.98$).

Fig. 7 helps explain these results. We focus here on the three algorithms that deliver the best speedups (Fully Centralised, Grid-GUM Stealing and Feudal Stealing), and on the highly-irregular applications. The left part of the figure shows the percentage of successful steal attempts (i.e. those that manage to locate work). As expected, we can see that, for more irregular applications, the fully-centralised algorithm has the highest success rate. Feudal Stealing also has a very high success rate, whereas Grid-GUM is noticeably worse than the other two. The right part of the figure shows the average time that it takes for a node to obtain work (i.e. the average time between the initiation of a successful steal attempt and the arrival of the work to the thief). We can see that the time it takes to locate work is greatest for fully centralised stealing. This is expected, since all steal attempts need to go via the central node. This makes the steal attempts quite expensive for nodes that are further away from the central node. In Feudal Stealing and Grid-GUM, nodes are able to obtain work much

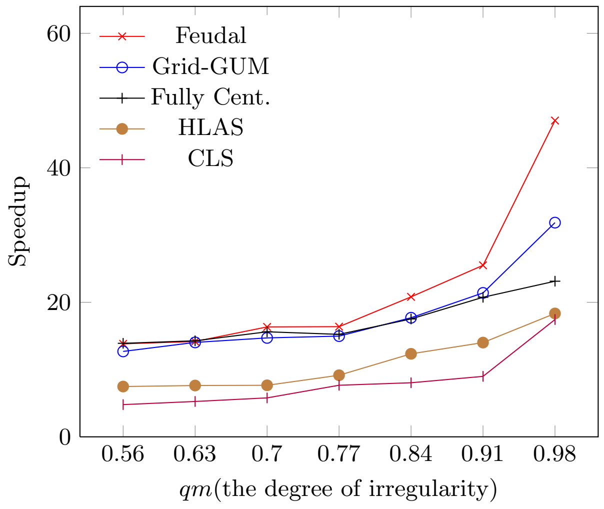


Fig. 6. Speedups for the UTS(q,7,3000,5ms) applications

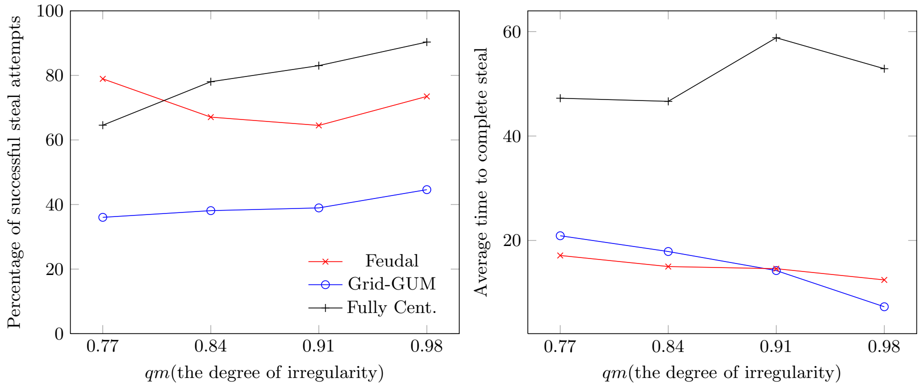


Fig. 7. Percentage of successful steal attempts (left) and average time it takes to successfully complete a steal (right) for the UTS(q,7,3000,5ms) applications

faster. Together, these two figures show why Feudal Stealing outperforms other algorithms. Central nodes are able to obtain relatively accurate load information, resulting in good selection of stealing targets, and the stealing messages are routed in such way that they quickly reach the targets with work, resulting in rapid response to the steal attempts. Grid-GUM and Fully Centralised stealing sacrifice one of these two features (accurate load information for Grid-GUM and rapid response to steal attempts for the Fully Centralised stealing) in order to make the other as good as possible.

Similar conclusions can be obtained when we look at the other examples of the UTS applications. For example, Fig. 8 shows the speedups of the UTS(15,q,3000, 10ms) applications, for $q \in \{0.062, 0.063, 0.064.0.065.0.066\}$. This represents

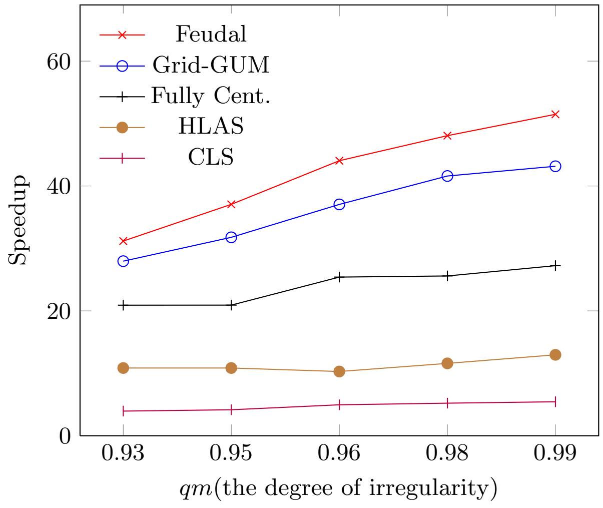


Fig. 8. The speedups of the UTS(q,15,3000,10ms) applications

applications with larger number of more coarse-grained tasks, with the high degree of irregularity ($qm \in \{0.93, 0.945, 0.96, 0.975, 0.99\}$). In these applications, the probability of a node having children is lower than for the UTS(7,q,3000,5ms) applications; however, each such node generates more children. Also, the sequential tasks are larger. We can see from the figure that Feudal Stealing and Grid-GUM give the best speedups (with Feudal Stealing performing best), and that both algorithms significantly outperform all other algorithms. The improvements in speedup of Feudal Stealing over the next best algorithm (Grid-GUM Stealing) range from 11% for $UTS(15, 0.062, 3000, 10ms)$ ($qm = 0.93$) to 20% for $UTS(15, 0.066, 3000, 10ms$ ($qm = 0.99$).

5 Conclusions and Future Work

In this paper, we have proposed a novel Feudal Stealing work stealing algorithm that uses a combination of centralised and distributed methods for obtaining dynamic system load information. We compared, using simulations, the performance of Feudal Stealing against the performance of algorithms that use centralised, fully-distributed, and a combination of both methods for obtaining dynamic load information. We have shown that Feudal Stealing outperforms all of these algorithms on heterogeneous distributed systems for the Unbalanced Tree Search benchmark, which models irregular divide-and-conquer (D&C) applications. Our previous results (reported in Janjic's PhD thesis [8]) have demonstrated that Feudal Stealing also outperforms state-of-the-art work-stealing algorithms that do not use load information on heterogeneous systems, and that it delivers comparable speedups to them on homogeneous systems and for regular D&C applications. Collectively, these results show that Feudal Stealing is the method of choice for load balancing the D&C applications on various different classes of systems (high-performance clusters of multicore machines, grids, clouds, skies etc.). Furthermore, we believe that this algorithm is also applicable to other classes of parallel applications (e.g. applications with nested data-parallelism and master-worker applications).

As future work, we plan to test the implementation of Feudal Stealing in a realistic runtime system (e.g. the Grid-GUM runtime system for Parallel Haskell [1], or the Satin system for distributed Java [16]) and to evaluate its performance on larger scale parallel applications. We also plan to incorporate information about the sizes of parallel tasks in the definition of the load for each node. This information has been shown to be important for selecting the task to offload in response to a steal attempt [9],. We envisage it can further improve stealing target selection, and hence speedup. Finally, we plan to introduce a measure of heterogeneity of a computing system, and to investigate "how heterogeneous" a system needs to be for Feudal Stealing to outperform other state-of-the-art work-stealing approaches.

Acknowledgments. This research is supported by European Union grants ADVANCE (IST-248828) and ParaPhrase (IST-288570), EPSRC grant HPC-GAP (EP/G 055181 and EU COST Action IC1202: Timing Analysis On Code-Level (TACLe).

References

1. Al Zain, A.D., et al.: Managing Heterogeneity in a Grid Parallel Haskell. Scalable Computing: Practice and Experience 7(3), 9–25 (2006)
2. Baldeschwieler, J.E., Blumofe, R.D., Brewer, E.A.: ATLAS: An Infrastructure for Global Computing. In: Proc. 7th Workshop on System Support for Worldwide Applications, pp. 165–172. ACM (1996)
3. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM 46(5), 720–748 (1999)
4. Burton, F.W., Sleep, M.R.: Executing Functional Programs on a Virtual Tree of Processors. In: Proc. FPCA 1981: 1981 Conf. on Functional Prog. Langs. and Comp. Arch., pp. 187–194. ACM (1981)
5. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable Work Stealing. In: Proc. SC 2009: Conf. on High Performance Computing Networking, Storage and Analysis, pp. 1–11. ACM (2009)
6. García Zapata, J., Díaz Martín, J.: A Geometrical Root Finding Method for Polynomials, with Complexity Analysis. Journal of Complexity 28, 320–345 (2012)
7. Hammes, J., Bohm, W.: Comparing Id and Haskell in a Monte Carlo Photon Transport Code. J. Functional Programming 5, 283–316 (1995)
8. Janjic, V.: Load Balancing of Irregular Parallel Applications on Heterogeneous Computing Environments. PhD thesis, University of St Andrews (2012)
9. Janjic, V., Hammond, K.: Granularity-Aware Work-Stealing for Computationally-Uniform Grids. In: Proc. CCGrid 2010: IEEE/ACM Intl. Conf. on Cluster, Cloud and Grid Computation, pp. 123–134 (May 2010)
10. Janjic, V., Hammond, K.: Using Load Information in Work-Stealing on Distributed Systems with Non-uniform Communication Latencies. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 155–166. Springer, Heidelberg (2012)

11. Neary, M.O., Cappello, P.: Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. In: Proc. JGI 2002: Joint ACM-ISCOPE Conference on Java Grande, pp. 56–65 (2002)
12. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.-W.: UTS: An Unbalanced Tree Search Benchmark. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007)
13. Ravichandran, K., Lee, S., Pande, S.: Work Stealing for Multi-Core HPC Clusters. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 205–217. Springer, Heidelberg (2011)
14. Van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In: Proc. PPoPP 2001: 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog., pp. 34–43 (2001)
15. Van Nieuwpoort, R.V., et al.: Adaptive Load Balancing for Divide-and-Conquer Grid Applications. J. Supercomputing (2004)
16. Van Nieuwpoort, R.V., et al.: Satin: A High-Level and Efficient Grid Programming Model. ACM TOPLAS: Trans. on Prog. Langs. and Systems 32(3), 1–39 (2010)

Scheduling HPC Workflows for Responsiveness and Fairness with Networking Delays and Inaccurate Estimates of Execution Times

Andrew Burkimsher, Iain Bate, and Leandro Soares Indrusiak

Department of Computer Science, University of York, York, YO10 5GH, UK

Abstract. High-Performance Computing systems (HPCs) have grown in popularity in recent years, especially in the form of Grid and Cloud platforms. These platforms may be subject to periods of overload. In our previous research, we found that the Projected-SLR list scheduling policy provides responsiveness and a starvation-free scheduling guarantee in a realistic HPC scenario. This paper extends the previous work to consider networking delays in the platform model and inaccurate estimates of execution times in the application model. P-SLR is shown to be competitive with the best alternative scheduling policies in the presence of network costs (up to 400% computation time) and where execution time estimate inaccuracies are within generous error bounds (<1000%) while still giving starvation-free behaviour.

1 Introduction

High-Performance Computing systems (*HPCs*) made up of a large number of parallel processors have become increasingly popular. To increase the available computing power, geographically-distributed networks of such clusters have been created, and these are known as *grids* [1]. These grids are often heterogeneous, with machines of varying capacity and architecture. The geographic distribution of the clusters in the grid gives rise to network delays when transferring data.

The pieces of work run on grids are rarely run in isolation, but instead form part of workflows, with dependencies between different computational tasks [2,3]. Each self-contained workflow is known as a *job*, made up of non-pre-emptible multicore *tasks* with dependencies.

Organisations that own grid or HPC capacity as well as cloud providers have an interest in providing Quality of Service (QoS) to their users. A particularly important aspect of QoS for many users is responsiveness. It is almost inevitable for grids and clouds to experience significant variations in demand, which can lead to transient periods of overload where some jobs have to wait. Industrial users interviewed by the authors indicated their desire for response times of jobs to be proportionate to the jobs' execution times. Furthermore, users desire fair treatment of their jobs. An example of a particularly unfair situation is if some jobs experience starvation (unbounded waiting time) under overload.

Previous work on fair scheduling for workflows with dependencies has been performed for offline [4] and batching [5] schedulers. These are ineffective when there is a wide variation in runtimes [6,7,8] because the response times required of the smallest tasks (hours) are orders of magnitude smaller than the execution times of the largest tasks (months), so no batch size will suit both. Effective prioritisation by the scheduler is required to keep the system responsive for the smallest tasks but avoid starvation for the largest ones.

In previous work [6], the authors developed an online list scheduling policy for dependent tasks called P-SLR that achieves responsiveness for small tasks and also provides a guarantee that no task will ever starve. P-SLR gave statistically indistinguishable responsiveness and fairness when compared to the best alternative scheduler, Shortest Remaining Time First (SRTF), even though SRTF is not starvation-free.

The P-SLR scheduler requires an estimate of the task execution time. The previous work [6] assumed that these times were known exactly. However, it is an ongoing field of research to accurately predict task execution times before they run, and they will always have their limitations [9]. Although users can provide hints about their task execution times, these are also far from accurate [10]. This paper describes how the models of the original work were extended to include these. The performance of the P-SLR scheduler is evaluated as to the impact of these inaccurate estimates.

Network delays also have an effect on scheduling. These can be described using the communication to computation ratio (CCR) value [11]. In the original analysis, only a single value of CCR was considered, using a network with a single central router. We extend the network model to a hierarchical architecture, and investigate the impact of changing CCR on the responsiveness and fairness of the P-SLR scheduling policy.

The context and models which define the scenario considered are presented in section 2. The considerations of measuring responsiveness and fairness, along with the P-SLR scheduling policy are defined in Section 3. Section 4 will describe the experimental method used to evaluate P-SLR against the alternative scheduling policies, and Section 5 will present the results of this evaluation.

2 Models

2.1 Application Model

A non-preemptible piece of work to be executed on one or more processors concurrently will be known as a task, denoted T^i. A set of tasks with dependencies is a job, denoted J^k. A set of jobs will be known as a workload W. Dependencies will be in the form of a directed acyclic graph (DAG), following [2]. Each task has an associated *architecture*, which defines which resources in the grid are available for it to execute on. The following terms define the attributes of tasks and jobs.

- Task execution time : $T^i_{\texttt{exec}} \in \mathbb{N}^\star$
- Task cores required : $T^i_{\texttt{cores}} \in \mathbb{N}^\star$
- Task start time: $T^i_{\texttt{start}} \in \mathbb{N}^0$

- Task finish time: $T^i_{\texttt{finish}} = T^i_{\texttt{start}} + T^i_{\texttt{exec}}$
- Task dependendents/successors: $T^i_{\texttt{succ}}$
- Task upward rank: $T^i_{\texttt{R}} = T^i_{\texttt{exec}} + \max(T^j_{\texttt{R}}) \bullet \forall T^j \in T^i_{\texttt{succ}}$
- Job arrival time (not necessarily the same as start time): $J^k_{\texttt{arrive}} \in \mathbb{N}^0$
- Job start time: $J^k_{\texttt{start}} = \min\left(T^i_{\texttt{start}}\right) \bullet \forall T^i \in J^k$
- Job finish time: $J^k_{\texttt{finish}} = \max\left(T^i_{\texttt{finish}}\right) \bullet \forall T^i \in J^k$
- Job response time: $J^k_{\texttt{response}} = J^k_{\texttt{finish}} - J^k_{\texttt{arrive}}$
- Job total execution time: $J^k_{\texttt{exec}} = \sum\left(T^i_{\texttt{exec}} \times T^i_{\texttt{cores}}\right) \bullet \forall T^i \in J^k$
- Job critical path: $J^k_{\texttt{CP}} = \max\left(T^i_R\right) \bullet \forall T^i \in J^k$

2.2 Inaccurate Estimates of Execution Times

In a realistic system, it is assumed that an estimate of execution time, albeit inaccurate, will be available from the user or from an automated job profiler. In simulation, however, the exact execution times are known in advance, so inaccuracies need to be introduced into the model. In this work, two possible ways are considered to convert exact execution times ($e_{\texttt{orig}}$) into inaccurate estimates, in order to evaluate the impact these inaccuracies have on the schedule quality:

Normal Error

This creates an estimate by sampling a normal distribution, shown in Equation 1, with a parameter N to vary the standard deviation, and hence the inaccuracy, of the estimate. The evaluation considers a wide range of values for N, between 1 % and 10^8 %.

$$e_{\texttt{est}} = \left\lceil \texttt{normal}(\mu = \mathrm{e}_{\texttt{orig}}, \sigma = e_{\texttt{orig}} \times \frac{N}{100}) \right\rceil \tag{1}$$

Log Rounding

This form of inaccuracy (Equation 2) reflects the expertise of users in knowing whether a job will take minutes, hours or days, but without much greater precision. M is the base of the logarithm used, with smaller values giving larger numbers of possible classes. The evaluation considers bases between 1 (no rounding) and 10^7 (all are in the same class).

$$e_{\texttt{est}} = M^{\lceil \log_M(e_{\texttt{orig}}) \rceil} \tag{2}$$

2.3 Platform Model

The resources in the grid are grouped into homogeneous clusters. These are connected together in a tree structure with a router at each node and a cluster at each leaf. Network delays are only considered between clusters, as delays inside a cluster are assumed to be negligible.

Jobs are submitted to the root of the tree and are randomly cascaded down the tree until a cluster is reached. Tasks from a single job that share the same architecture are kept together and are allocated to the same cluster, to avoid unnecessary network costs. The detailed scheduling decisions are then made by a list scheduler executing on the clusters.

2.4 Network Delay Model

The network model is considered to be that of a thin tree [12]. This is where nodes at lower levels of the tree have greater communication speed between them than nodes higher in the tree. This reflects what is seen in geographically distributed networks, where nodes further apart tend to have slower connections. This can approximate a real network, because all fully connected networks possess a spanning tree [13].

The aim of this network model is to provide an acceptable model to investigate the effects of network delays on scheduling while adding minimal computational overhead. The network speed is calculated by using the fact that the network is tree structured. Therefore, any two clusters will share a common parent node, and the number of nodes in between a cluster and the common parent is measured in levels. The speed equation takes a parameter p to vary how much slower the higher levels of the network become.

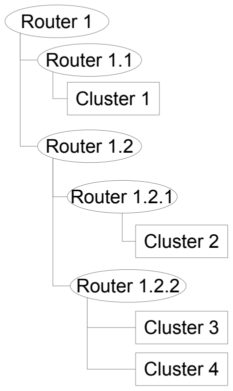


Fig. 1. Thin Tree Network Diagram

$$N_{\mathtt{speed}} = (\mathtt{max\_levels\_to\_common}(C_1, C_2))^p \tag{3}$$

To find the data volume to transfer, the communication to computation ratio parameter (CCR) is used [11], along with the execution time of the task $T^i_{\mathtt{exec}}$, as shown in Equation 4.

$$T^i_{\mathtt{data}} = \frac{T^i_{\mathtt{exec}}}{CCR} * (1 - CCR) \tag{4}$$

The time taken to transfer data between two tasks is determined by dividing the data volume required by the speed of the network between them.

$$T^i_{\mathtt{net\_delay}} = \frac{T^i_{\mathtt{data}}}{N_{\mathtt{speed}}} \tag{5}$$

3 Metrics and the P-SLR Scheduler

In order to evaluate the responsiveness and fairness achieved by scheduling policies, appropriate metrics are required. We suggested in [6] that the most informative metric for measuring responsiveness is the *Schedule Length Ratio* (SLR) [3], when applied to each job in a workload. The *critical path* of a job is the longest path through the job's dependency graph and defines the minimum time

Algorithm 1. Projected SLR ordering algorithm

$$\texttt{projected\_slr}(T^i, J^k, \texttt{curr\_time}, Q) =$$

$$\frac{\left(T^i_R + \texttt{curr\_time} + 1\right) - J^k_{\texttt{arrive}}}{J^k_{\texttt{CP}}} + \left\lfloor \frac{\texttt{curr\_time} - J^k_{\texttt{arrive}}}{\max\left(J^n_{\texttt{CP}} \bullet J^n \in Q\right)} \right\rfloor^2$$

that the job can be executed in even if the number of processors was unbounded [14]. The SLR of a job is its response time relative to the length of its critical path.

We proposed using the distribution of SLR values to measure fairness [6], of which three kinds can be described (Figure 2). Class 1 is where the responsiveness of longer jobs is prioritised over that of short jobs, with Class 2 being the opposite case. Class 3 is where there is equal prioritisation of responsiveness with respect to execution times.

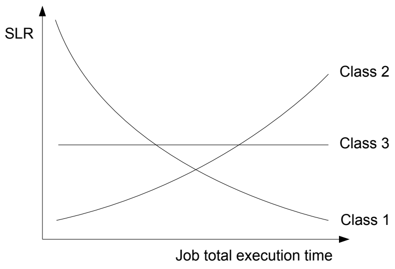


Fig. 2. Classes of prioritisation by execution time

The common scheduling policy First In First Out (FIFO) falls into Class 1 because on average, each job will wait in the queue for the same length of time. This waiting time is proportionately larger relative to execution time for smaller tasks, penalising the SLR of short-running jobs. This pattern is true for any policy not considering execution times. The Longest Remaining Time First (LRTF) scheduler also falls into Class 1. The Shortest Remaining Time First (SRTF) scheduler is of Class 2. The authors designed the P-SLR scheduler to exhibit Class 3 behaviour, and demonstrated this in [6].

In order to make use of execution time estimates, LRTF, SRTF and P-SLR use the concept of *Upward Rank* introduced by [3]. Upward Rank is defined for each task, and is the length of the critical path that remains to be completed after the task has executed. LRTF and SRTF sort the list of tasks by decreasing and increasing Upward Rank, respectively. These policies can suffer from starvation under overload, because the shortest (LRTF) or longest (SRTF) tasks may never reach the head of the queue (Q).

P-SLR uses the upward rank to calculate a forward projection of the job finish time and hence SLR if the considered task was run immediately (Algorithm 1). The task where the P-SLR is largest (is most 'late') is run first, letting small jobs 'jump' the queue as their SLRs are more sensitive to waiting time. This is distinct from the approach used by [5] which uses the downward rank (looks backward) to calculate a partial value for SLR based on the tasks that have already completed.

P-SLR is starvation-free because the projected SLR rises for all jobs as they wait, which means all jobs will eventually run, as long as overloads are transient. In the case of extreme overload where the work queue continually grows unboundedly, the waiting time term (the second part of the equation in Algorithm 1) comes to dominate, reverting the ordering to that of FIFO, thus avoiding starvation in all cases.

4 Evaluation Method

The evaluation method will seek to investigate two principal hypotheses:

1: Projected-SLR delivers better responsiveness and fairness than schedulers which do not use execution time estimates, even when the estimate inaccuracy is significant. P-SLR is competitive with scheduling policies that do make use of execution time estimates.

2: Projected-SLR delivers competitive responsiveness and fairness metrics independent of communication to computation ratios.

4.1 Simulation Details

The evaluation will be run using the simulation framework developed and validated in [6] which implements the models and extensions described above. The platform used is the one shown in Figure 1, with each cluster having 400 cores, and Cluster 1,3,4 being of architecture *Kind1*, whereas Cluster 2 is of architecture *Kind2*. These values have been chosen to reflect those we observed in industry. Because there are elements of randomness in the allocation and in the workloads used, numerous simulation trials will be run for each experiment.

Responsiveness will be measured using the median value of the worst-case SLRs observed in each trial. The worst-case SLR is used because of the desire for high responsiveness to be achieved for all users. Using the median value instead of the mean will prevent any truly pathological cases from biasing the results.

Fairness will be measured using the median of the Gini Coefficients [15] calculated for the SLRs in each trial. The Gini Coefficient (GC) is a measure of the inequality of resources allocated to a given population. In this instance, it is the allocation of responsiveness to jobs by the scheduler. The GC takes a value between 0 (completely fair) and 1 (completely unfair).

Statistical significance will be tested using a repeated measures t-test because the workloads are the same, meaning the job SLRs can be directly compared. The threshold for statistical significance is set at 5%.

4.2 Scheduling Policies

Several common policies will be used as a basis for comparison with P-SLR. The random ordering policy simply chooses a random task to run. While it does not guarantee to be starvation-free, the probability of a task starving forever tends towards zero as time passes. The FIFO Task policy runs tasks in the order that they become ready, which is starvation-free. FIFO Job is an alternative to FIFO Task that runs tasks in the order that their jobs arrived on the HPC . This avoids a problem with FIFO Task where tasks that have just become ready are added to the tail of the queue, which means that jobs with many levels of dependencies can end up waiting the length of the queue multiple times.

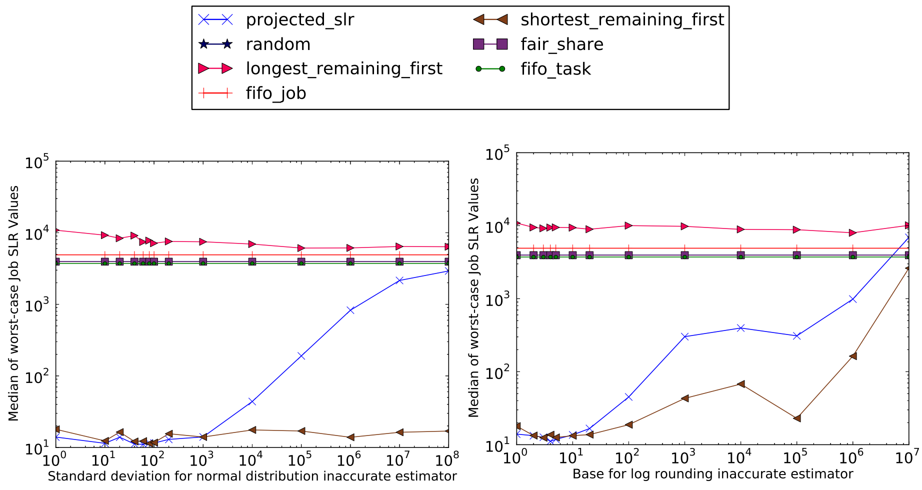
The Fair Share ordering policy [16] is based on the user who submitted a job. Each user is a member of a group, which has a certain share of the resources of the HPC. The share of these groups is organised in a tree. The priority of tasks is not static, but depends on the instantaneous number of resources already being consumed by the user and their parent groups divided by their allocated share.

Simulation of an overload situation is necessary so that some jobs will have to wait, and the ability of the schedulers to keep responsiveness and fairness high can be compared. The overload rate can be defined as a percentage of the arrival rate of jobs compared to the maximum processing rate achievable, and a figure of 120% is used.

5 Results

5.1 Inaccurate Execution Times

For all the results with inaccurate execution times, the Random, FIFO Task, FIFO Job and FairShare policies are not affected by the inaccurate estimates, because they do not make use of these estimates.



(a) Normally distributed inaccuracies (b) Log rounding inaccuracies

Fig. 3. Responsiveness

Responsiveness. With normally distributed inaccuracies (Figure 3a), the P-SLR policy dominates by having the lowest worst-case SLR values until the standard deviation is 1000% of the value of the exact time. It is reasonable to assume that virtually all real-world estimates will have ranges less than 1000%.

The difference between P-SLR and SRTF in this range is not statistically significant, which shows the strength of the P-SLR policy as it adds the guarantee of non-starvation. The divergence after 1000% is due to this guarantee because SRTF is letting the largest tasks starve. The largest tasks have SLRs which are least sensitive to waiting time, keeping the worst-case SLR fairly low.

Once the estimation error gets sufficiently large, the estimates become effectively random. Therefore, the worst-case SLR of the P-SLR orderer rises to similar levels as the schedulers that do not make use of execution time estimates.

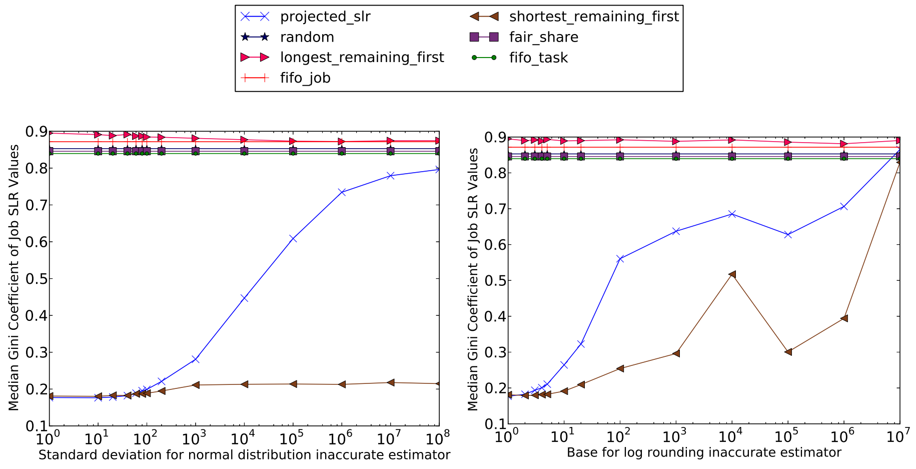
Similar results are apparent where estimates are log-rounded (Figure 3b). Where execution times are rounded to the nearest power of 10 or below, P-SLR dominates the worst-case SLR values, although it is not statistically distinguishable from SRTF. Still, it is to be expected that users could give a good indication of their job taking closer to 1, 10, 100, etc., minutes.

As the estimates get yet more coarse above a base of 10, SRTF provides better worst-case responsiveness than P-SLR. This is to be expected, because inaccurate estimates move the behaviour of schedulers closer to Class 1 behaviour. As P-SLR with accurate estimates exhibits Class 3 behaviour, any perturbation to this will make it tend towards Class 1 behaviour. Whereas for SRTF, because it shows Class 2 behaviour, perturbations will initially make its behaviour more like Class 3, although eventually it too will exhibit Class 1.

The LRTF orderer, as expected, shows poorer worst-case responsiveness than any of the policies that do not consider execution time. This is because it makes the smallest tasks starve, and these tasks are the ones whose SLR is most sensitive to waiting time. LRTF is useful, though, because it gives an upper bound on how poor responsiveness can get because it shows the most extreme Class 1 style behaviour.

These results show that up to a threshold value of 10^3, the P-SLR and SRTF policies have statistically insignificant differences in responsiveness. Responsiveness for P-SLR approaches that of the schedulers that do not include execution time estimates when the error for either normal standard deviation percentage and log rounding is around 10^7. These values are far above the maximum levels of inaccuracy of around 100% found by [10]. This would suggest that in reality, the P-SLR scheduler could be considered most favourable for practical scheduling, because it gives a guarantee of non-starvation, unlike SRTF, and leads to an improvement in responsiveness performance over that of schedulers that do not consider execution time estimates.

Fairness. As with the results for responsiveness, the fairness results for normally distributed error (Figure 4a) are dominated by P-SLR at the lowest values, although they are also statistically indistinguishable from SRTF up to a threshold of 100%. This is to be expected, as P-SLR is designed to show Class 3 behaviour, which emphasises fairness. Above this threshold, P-SLR exhibits progressively



(a) Normally distributed inaccuracies (b) Log rounding inaccuracies

Fig. 4. Fairness

more Class 1-like behaviour, as poor estimates for small tasks cause their responsiveness to fall. SRTF causes the largest jobs to starve, but because their SLRs are less sensitive to waiting time, the SLR distribution remains closer to Class 3.

The normally-distributed inaccurate estimator was not able to introduce sufficient error below a standard deviation percentage of 10^8 to cause significant impact on the fairness of the SRTF policy. If the estimation errors are normally distributed, therefore, SRTF may provide better fairness than P-SLR when the standard deviation of the errors is above 100%.

With the log rounding estimator (Figure 4b), other than the case where there was no inaccuracy, the SRTF orderer was statistically significantly more fair, according to the Gini Coefficients, than for P-SLR. As before, this is due to the SRTF causing the largest jobs to starve, but this not having a large effect on those jobs' SLR values. P-SLR immediately starts to exhibit Class 3 behaviour in the presence of inaccurate estimates, whereas SRTF moves from Class 2, then to Class 3, before eventually showing Class 1 at a rounding power of 10^7 .

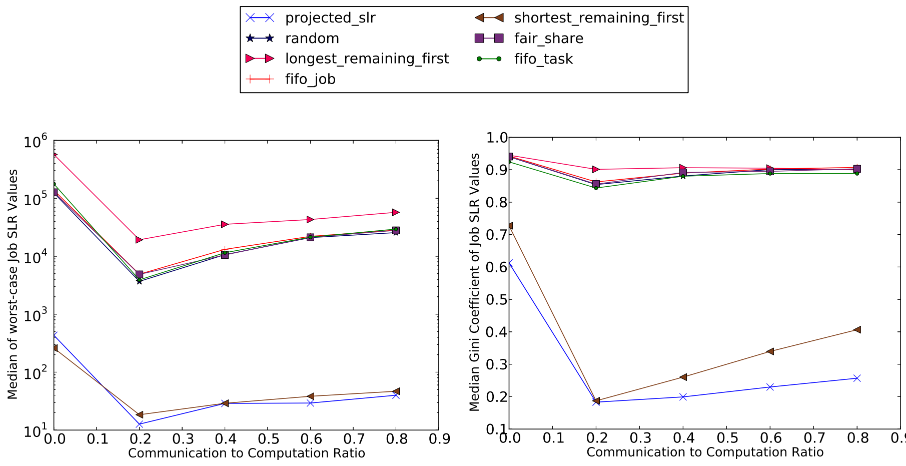
The LRTF policy shows the worst-case unfairness, as it is the most extreme example of Class 1 behaviour. The bound on how unfair it makes things improve as estimates get worse, because it is not as able to achieve the worst case.

The fairness results show that for small inaccuracies in execution time estimates, P-SLR and SRTF show similar results. However, for larger inaccuracies, SRTF gives fairer results as it shows more of a Class 3 behaviour profile, although this is due to the largest jobs being starved of resources.

Hypothesis 1 stated that P-SLR would deliver better responsiveness and fairness than schedulers that do not use execution times, even when the estimate accuracy is significant. This has been shown to be the case, with better responsiveness and fairness when the standard deviation inaccuracy percentage is less than 10^7 and when the log rounding base is less than 10^8, all extremely high levels of inaccuracy. P-SLR has been shown to be competitive with SRTF in responsiveness up to a threshold inaccuracy of 10 times the value of the original estimate. In fairness, P-SLR is competitive at small inaccuracies, but SRTF dominates above this, refuting a part of the hypothesis. It is then a tradeoff for a grid owner to decide whether, if estimates of execution time have large inaccuracies, absolute fairness (SRTF) or an absence of starvation (P-SLR) is more important.

5.2 Networking Delays

Responsiveness. A pronounced feature (Figure 5a) is that there is an improvement in worst-case responsiveness when network costs become present at a CCR of 0.2. This is because any network costs will increase the length of the critical path, which means CPU resources are no longer the single bottleneck.



(a) Responsiveness (b) Fairness

Fig. 5. Network Delays

Throughout the range of network delays examined, P-SLR and SRTF showed similar levels of responsiveness. SRTF was slightly better when there were no network delays, but P-SLR was slightly better when there were delays present. However, P-SLR and SRTF were not statistically significantly different.

The LRTF policy again shows the worst case bound of responsiveness because it tends to starve the smallest tasks.

Fairness. The results in Figure 5b also show greater fairness in the presence of network delays, because of the improvements in overall responsiveness. However, in this case, P-SLR is stasticically significantly more fair than SRTF, except where the CCR takes a value of 0.2. As CCR is increased, the unfairness increases more slowly for P-SLR than for SRTF. This is because although their worst case values are similar (Figure 5a), P-SLR shows more Class 3 behaviour, giving a better balance of SLR values overall.

The LRTF scheduler is the least fair at low values of CCR, but converges to a similar level of fairness as those schedulers that do not consider execution times at higher network costs. All the schedulers other than SRTF are very unfair across the space of network delays when compared to P-SLR.

Hypothesis two considered whether P-SLR delivers competitive responsiveness and fairness across the range of CCRs. P-SLR dominated all schedulers other than SRTF in responsiveness, although it was statistically indistinguishable from SRTF. In fairness, it dominated all other schedulers, except for SRTF where CCR was 0.2. As network costs increased, the rate of decrease in fairness was lower for P-SLR than for SRTF. This confirms this hypothesis, showing that across the space of network costs, P-SLR provides equal or better responsiveness and fairness than the best alternative scheduler, SRTF, but does so while in addition providing a guarantee that no job will ever starve.

6 Conclusion

This paper revisited the work of [6], and investigated the robustness of the P-SLR scheduling policy in the presence of networking delays and inaccurate execution times.

The responsiveness and fairness performance of the P-SLR scheduler was found to be robust to network delays. P-SLR provides equal or better responsiveness (measured by worst-case SLR) and fairness (measured by the Gini Coefficient of SLRs) in the presence of network delays than the best alternative scheduler, SRTF, but does so while in addition providing a guarantee that no job will ever starve.

The responsiveness performance of P-SLR was found to be robust below a certain threshold of execution time inaccuracy. This threshold was 10 times the original execution time of the task. Above this threshold, SRTF was able to provide better responsiveness. P-SLR was not able to give the best fairness compared to SRTF once any significant estimation inaccuracies were present, because SRTF is better at keeping SLRs low for small tasks whose SLRs are more sensitive to longer waiting times. However, P-SLR still dominated all other alternative policies, showing that where estimates of execution time are available, it can make good use of them, even where the inaccuracies are large.

Acknowledgements. We would like to thank the EPSRC (grant number EP/F501374/1) for funding this research through the UK's Large-Scale Complex IT Systems (LSCITS) programme.

References

1. Albodour, R., James, A., Yaacob, N.: High level QoS-driven model for grid applications in a simulated environment. Future Generation Computer Systems 28(7), 1133–1144 (2012)

2. Maheswaran, M., Braun, T.D., Siegel, H.J.: Heterogeneous distributed computing. In: Encyclopedia of Electrical and Electronics Engineering, pp. 679–690. John Wiley (1999)
3. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)
4. Zhao, H., Sakellariou, R.: Scheduling multiple dags onto heterogeneous systems. In: 20th International Parallel and Distributed Processing Symposium, IPDPS 2006, p. 159 (2006)
5. Hirales-Carbajal, A., Tchernykh, A., Yahyapour, R., González-García, J., Röblitz, T., Ramírez-Alcaraz, J.: Multiple workflow scheduling strategies with user run time estimates on a grid. Journal of Grid Computing 10(2), 325–346 (2012)
6. Burkimsher, A., Bate, I., Indrusiak, L.S.: A survey of scheduling metrics and an improved ordering policy for list schedulers operating on workloads with dependencies and a wide variation in execution times. Future Generation Computer Systems (in press, published online December 27, 2012), doi: `http://dx.doi.org/10.1016/j.future.2012.12.005`
7. Chiang, S. H., Vernon, M.K.: Characteristics of a large shared memory production workload. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 159–187. Springer, Heidelberg (2001)
8. Feitelson, D.G., Nitzberg, B.: Job characteristics of a production parallel scientific workload on the nasa ames ipsc/860. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 337–360. Springer, Heidelberg (1995)
9. Sonmez, O., Yigitbasi, N., Iosup, A., Epema, D.: Trace-based evaluation of job runtime and queue wait time predictions in grids. In: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC 2009, pp. 111–120. ACM, New York (2009)
10. Bailey Lee, C., Schwartzman, Y., Hardy, J., Snavely, A.: Are user runtime estimates inherently inaccurate? In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 253–263. Springer, Heidelberg (2005)
11. Schoneveld, A., de Ronde, J.F., Sloot, P.M.A.: On the complexity of task allocation. Complex 3(2), 52–60 (1997)
12. Navaridas, J., Miguel-Alonso, J., Ridruejo, F.J., Denzel, W.: Reducing complexity in tree-like computer interconnection networks. Parallel Computing 36(2-3), 71–85 (2010)
13. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction To Algorithms. MIT Press (2001)
14. Kwok, Y.K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. Journal of Parallel and Distributed Computing 59(3), 381–422 (1999)
15. Litchfield, J.A.: Inequality methods and tools. In: School of Economics (1999)
16. Platform Computing Corporation: Fairshare scheduling (2008), `http://www.cisl.ucar.edu/docs/LSF/7.0.3/admin/fairshare.html`

FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems

Gonzalo Martín[1], Maria-Cristina Marinescu[2], David E. Singh[1], and Jesús Carretero[1]

[1] Universidad Carlos III de Madrid, Leganés, 28911, Spain
gmcruz@arcos.inf.uc3m.es
[2] Barcelona Supercomputing Center, Barcelona 08034, Spain

Abstract. This paper introduces FLEX-MPI, a novel runtime approach for the dynamic load balancing of MPI-based SPMD applications running on heterogeneous platforms in the presence of dynamic external loads. To effectively balance the workload, FLEX-MPI monitors the actual performance of applications via hardware counters and the MPI profiling interface—with a negligible overhead and minimal code modifications. Our results show that by using this approach the execution time of an application may be significantly reduced.

Keywords: Dynamic load balancing, distributed computing, heterogeneous systems, hardware counters.

1 Introduction

The work described in this paper focuses on the efficient distribution of program workloads on heterogeneous platforms composed of processors with the same instruction set architecture (ISA) but with different performance. This work targets parallel applications based on the SPMD (*Single Program Multiple Data*) paradigm. A large proportion of these applications are iterative and alternate phases of computation and communication; linear system solvers such as Jacobi and Conjugate Gradient from NPB [1] are good representatives of this class of applications and are used as benchmarks in our evaluation. We also evaluated our approach on EpiGraph [2], a significantly more complex HPC application.

We introduce FLEX-MPI, an MPI extension which monitors the performance of an application and uses this information to make decisions with respect to the distribution of the workload and the data between processes. We focus on an adaptive strategy for balancing the workload of applications that run on non-dedicated systems, in which several applications run concurrently and share the computing resources, e.g. CPU, memory, and cache. Sharing resources means that applications have external loads which degrade their performance.

We consider both burst and long-term external loads. Burst loads correspond to short-duration external loads which do not significantly affect the application performance. Long-term external loads reduce the application's CPU usage affecting its performance. FLEX-MPI is able to discriminate between these two kinds of loads and flexibly apply different load balance policies depending on their magnitude. One of the advantages of this approach is that it does not require prior knowledge about the underlying architecture. We use hardware counters and the MPI profiling interface to directly measure performance metrics at runtime. The main contributions of this work are:

– A **precise**, **flexible** dynamic load balancing technique based on monitoring the actual performance of the applications via hardware counters and the MPI profiling interface.
– A **powerful**, **decentralized** approach that works for homogeneous and heterogeneous systems which can be either dedicated or non-dedicated.
– A **low overhead**, **generic** method for integrating dynamic load balancing into existing MPI-based SPMD applications.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces FLEX-MPI. Before concluding in Section 5, we present an extensive performance evaluation in Section 4.

2 Related Work

Dynamic load balancing for heterogeneous systems is a topic of great interest since current distributed computing systems—mainly grid and cloud, but also cluster—are becoming predominantly heterogeneous [3]. An efficient approach, originally designed for homogeneous systems, consists in adapting the parallel code by software techniques which balance the load depending on the computational power of each computing unit. But to adapt parallel code to run on a heterogeneous system requires prior knowledge about the characteristics of the architecture [4], which is not always feasible.

The basic approach for dynamic load balancing in heterogeneous systems is based on a theoretical model of the system. Belikov *et al.* [5] present an architecture-aware cost modeling technique based on theoretical CPU speed, cache, RAM, and latency. However, theoretical values do not always match the performance achieved when running a real application and do not consider the external load introduced by other processes running on the same processor.

Several projects propose approaches based on collecting performance metrics at runtime. Galindo *et al.* [6] present a model based on the relative computing power, a metric which is obtained by measuring the execution time invested by a processor in performing a given computation. The computation is measured as the number of rows of a dense matrix, an inaccurate model when it comes to sparse data structures. Their approach only considers executions on dedicated systems. ALBIC [7] is a system based on [6] which measures the system load by collecting performance metrics at runtime. However, this technique is intrusive since to collect this data and feed it to the monitoring system they add a specific module in the Linux kernel. A similar approach is Dyn-MPI [8], a dynamic MPI implementation which targets parallel applications running on non-dedicated architectures. Dyn-MPI requires a daemon running in each computing node to extract performance metrics. It is highly code intrusive since many of the calls, including standard MPI functions, must be instrumented.

Bohn *et al.* [9] measure the performance of compute nodes by extracting information from files of the Linux OS and benchmarking both the processor and the memory, operations which are usually expensive. HeteroMPI [10] is an MPI extension which was specifically designed for programming on heterogeneous systems. It can measure processor performance by using a benchmarking function whose code must be provided by the programmer. HeteroMPI requires a significant intrusive instrumenting, even for simple parallel programs. AdaptiveMPI [11] is an adaptive implementation of MPI built on top of the CHARM++ runtime environment which supports dynamic load balancing through processor virtualization. It only offers full compatibility with the MPI-1.1 features and MPI standard programs need to be significantly modified.

Hardware counters have been demonstrated to be an effective way of measuring computer performance [12]. FLEX-MPI introduces a novel dynamic load balancing algorithm which flexibly adapts to external load in heterogeneous non-dedicated systems. Our approach is based on collecting precise system performance metrics at runtime via hardware counters and can be integrated in existing MPI-based SPMD applications with minimal code modifications.

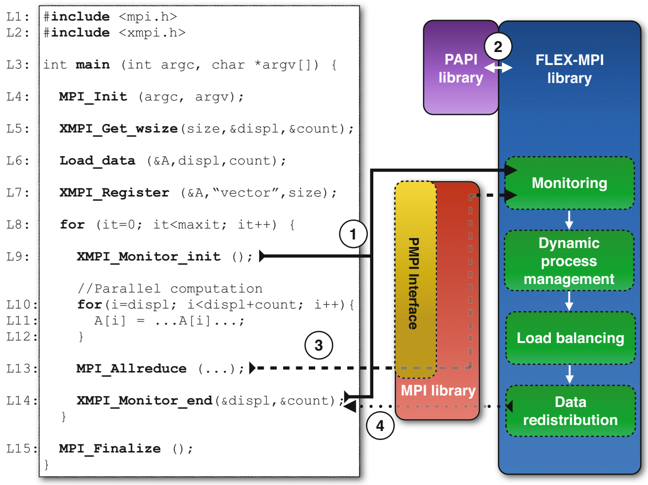
3 FLEX-MPI

FLEX-MPI is an MPI extension which integrates three functionalities: *monitoring*, *load balancing* (LB), and *data redistribution*. We implemented FLEX-MPI as a library on top of the MPICH-2 implementation. This makes it fully compatible with the MPI-2 features and allows it to easily link with any existing MPI-based application. FLEX-MPI's API is described in detail in [13].

3.1 Monitoring

The purpose of the monitoring functionality is to collect performance metrics for each process of the parallel application during its execution. The applications we target are iterative and alternate computation and communication phases. We monitor computation by means of hardware counters (via PAPI [14]) and communication by using the MPI profiling interface (PMPI), which allows to profile the communications without modifying the source code of the application.

FLEX-MPI targets SPMD applications using one-dimensional domain decomposition with distributed data, a parallelization method used by a large number of scientific parallel applications. In these applications the portion of the domain assigned to a process is usually expressed as a combination of a count—which represents the number of elements, rows, or columns assigned to the process—and a displacement. Fig. 1 illustrates an example of a SPMD application using the FLEX-MPI library, in which the data structure managed by the application



```
L1:  #include <mpi.h>
L2:  #include <xmpi.h>

L3:  int main (int argc, char *argv[]) {

L4:    MPI_Init (argc, argv);

L5:    XMPI_Get_wsize(size,&displ,&count);

L6:    Load_data (&A,displ,count);

L7:    XMPI_Register (&A,"vector",size);

L8:    for (it=0; it<maxit; it++) {

L9:      XMPI_Monitor_init ();

         //Parallel computation
L10:     for(i=displ; i<displ+count; i++){
L11:       A[i] = ...A[i]...;
L12:     }

L13:     MPI_Allreduce (...);

L14:     XMPI_Monitor_end(&displ,&count);
       }

L15:   MPI_Finalize ();
     }
```

Fig. 1. Structure and runtime calls of a parallel code linked with the FLEX-MPI library

(vector A) is distributed between the processes. Each process operates in parallel (L10-12) on a different subset of the data structure. The parallel code is instrumented with a set functions to get the initial partition of the domain assigned to every process (L5), register the data structure managed by the application (L7), enable monitoring (L9), and rebalance the load as needed (L14). The monitoring functionality dynamically collects performance metrics provided by PAPI (arrow labeled 2) and the MPI library through the PMPI interface (arrow labeled 3). When a program iteration finishes (at line L14) these metrics are fed to the LB functionality (arrow labeled 1), which computes the new distribution of the workload. In our work, we consider each of the computing cores of modern multiprocessors as an independent processing element (PE). After a new workload distribution has been decided, the data redistribution functionality moves the data to the processes that will need it (arrow labeled 4). A more detailed description of the instrumentation process can be seen in [13].

Our implementation uses low level PAPI interfaces to track the number of floating point operations $FLOP$, the real time $Treal$ (i.e. the wall-clock time), and the CPU time $Tcpu$ (i.e. the time during which the processor is running in user mode). These metrics are collected for each process of the parallel application, and they are preserved during context switching. The $FLOP$ is needed to effectively track and measure the performance at the granularity of each processing element, while the real time and CPU time allow us to identify if there exists external application load. In our model, we assume floating-point based applications which exhibit a linear correlation between the $FLOP$ and the workload size, which is reasonable for many parallel applications (e.g. linear system solvers). An initial calibration is required because in heterogeneous systems the events counted by hardware counters are processor specific. This calibration is carried out by performing a microbenchmark with a negligible overhead before starting the computation of the application.

3.2 Load Balancing

The load balancing functionality receives as input the per-process values for the performance metrics measured via monitoring. When load imbalance is detected, the algorithm decides the new distribution of workload based on the per-process performance metrics. Although monitoring can be performed at every iteration we trigger load balancing only every *sampling interval*—consisting of a fixed number of iterations—to reach a trade-off between the overhead of this operation and the performance gain as result of it.

To decide how much workload to re-assign to each process, the load balancing algorithm first computes the $MFLOPS$ that each process i executed during the previous sampling interval. $MFLOPS_i$ is defined in Equation 1 as the ratio between the number of floating point operations $FLOP_i$ and the real execution time $Treal_i$ during a given sampling interval. The fraction of the workload assigned to process i is computed in Equation 2 depending on the *relative computing power* (RCP_i) of a process i, which is computed as the $MFLOPS_i$ divided by the total $MFLOPS$ for all of the p processes. RCP is used to estimate workload distribution on parallel applications, since it provides a normalized value of the computational power of a process relative to the computational power of the whole system [4,7].

$$MFLOPS_i = \frac{FLOP_i}{Treal_i} \qquad (1) \qquad\qquad RCP_i = \frac{MFLOPS_i}{\sum_{i=0}^{p} MFLOPS_i} \qquad (2)$$

Algorithm 1 shows the pseudocode for the load balancing algorithm, which is evaluated at each sampling interval n. The first step (`line 1`) detects which of the processing elements involved in executing the application are dedicated and which are not. When the difference between the CPU time and the real time of a processing element is small we can safely assume that it executes only one process. When the real time is significantly higher than the CPU time then the processing element is being shared between multiple processes - either of the same, or of a different application. The real time is always a little higher than the CPU time because of OS interrupts; we use a threshold parameter TH_1 to account for this overhead and mark the difference between dedicated and non-dedicated processing elements. We consider that values of the real time that surpass the CPU time by 5% are reasonable for setting the tolerance threshold TH_1. Each process uses a boolean variable `dedicated` to record whether it uses the processing element in exclusive mode or not. By applying a reduce operation (`line 6`) over all processes we know whether there exists any non-dedicated processing element. The reduction result—stored in the variable `global_dedicated`—is false if at least one processing element is non-dedicated.

We evaluate whether we should redistribute the workload if either (1) the processing elements have been used in exclusive mode during the current sampling interval but the application is unbalanced or (2) long-term external load is detected on any of the processing elements. It is possible that the applications that share the resources with our application execute during short, isolated bursts which do not affect the overall performance of our application.

Algorithm 1. Pseudocode for the load balancing algorithm

```
 1: if $((Treal_i - Tcpu_i)/Treal_i) < TH_1$ then
 2:     $dedicated \leftarrow$ true
 3: else
 4:     $dedicated \leftarrow$ false
 5: end if
 6: $global\_dedicated \leftarrow Allreduce(dedicated, AND)$
 7: $buf[n] \leftarrow global\_dedicated$
 8: $external\_load \leftarrow evaluate\_external\_load(buf, k)$
 9: if $(global\_dedicated == true)$ or $(external\_load == long\_term)$ then
10:     $\{FLOP, Treal\} = Allgather(FLOP_i, Treal_i)$
11:     $MFLOPS = compute\_MFLOPS(FLOP, Treal)$
12:     if $(max(Treal) - min(Treal)) > (TH_2 * max(Treal))$ then
13:         $RCP = compute\_RCP(MFLOPS)$
14:         $Data\_redistribution(RCP)$
15:     end if
16: end if
```

In contrast, long-term external loads consume a lot of computer resources during a continuous period of time and significantly degrade the overall performance of our application. In our algorithm each process stores the value of variable `global_dedicated` at each sampling interval (n) (`line 7`). When a processing element has been running in non-dedicated mode during k consecutive sampling intervals it is considered that long-term external load is present on that processing element and the workload should be considered to be redistributed. The function `evaluate_external_load` returns *long_term* when any of the processing elements have been running in non-dedicated mode during the past k sampling intervals (`line 8`). Section 4.2 discusses practical values of k.

If either all of the processing elements are dedicated during the current sampling interval, or long-term external load has been detected (`line 9`), then the algorithm analyzes the load balance of the application. Otherwise, when a bursty external load is detected, the algorithm tolerates it without performing load balancing for $(k - 1)$ consecutive sampling intervals. In the k^{th} sampling interval one of two things will happen: (1) either there will be another burst, in which case it leads to the conclusion that rather than a series of bursts, a long-term load is present, or (2) the processing elements will run in dedicated mode, in which case it will also be a candidate for load balancing evaluation. When the application is evaluated for load balancing, the algorithm gathers and distributes the $FLOP$ and real time numbers of each process (`line 10`) to all the other processes. It then applies Equation 1 to compute $MFLOPS_i$ locally by each process i (`line 11`). If the difference between the maximum and minimum values of $Treal$ is larger than the threshold value TH_2 * *max(Treal)* (`line 12`) then the application is more unbalanced than what it can tolerate. In our experiments, we empirically set TH_2 to 15%. As a result, LB triggers the redistribution of workload based on Equation 2 and the RCP of each process (`line 13`), then uses the new workload distribution to perform the data remapping by invoking the data distribution functionality (`line 14`).

The algorithm implemented focuses on balancing computation workloads and requires precise computation time measurements which do not take into account the time spent on performing communication operations. Otherwise, the algorithm would lead to inaccurate workload distributions. For instance, an imbalanced parallel application where the fastest process spends most of the time waiting idle for other processes involved in communication operations will have a low $FLOP$ count and large $Treal$. By profiling MPI communications we can compute separately the time spent by each process in performing computation and communication, enabling a precise load balancing policy.

3.3 Data Redistribution

In SPMD applications the data is usually distributed—rather than replicated—between processes, which requires redistribution to move the data between processes each time a load balance operation is carried out. FLEX-MPI includes a data redistribution functionality which handles both one-dimensional (e.g. vectors) and two-dimensional (e.g. matrices) data structures, which may be either dense or sparse.

The user has to register each of the data structures which will need to be redistributed as result of load balance operations. The registering function (`XMPI_Register`) receives as input the pointer to the data structure and the size of the data structure. Depending on the domain decomposition type, the sizes of the data structures can be provided either as the number of elements, rows, or columns of the structures. FLEX-MPI can manage several data structures whenever they have been registered using the same type of domain decomposition.

Once the load balancing functionality has computed the RCP of each PE and the new workload distribution has been mapped to a data partition, the data redistribution functionality (1) computes the range of data associated to the new workload partition of every process, and (2) moves the data from the previous to the new owners. `XMPI_Monitor_end` returns—on behalf of the data redistribution functionality—the new count and displacement for the new data mapping used by each process. MPI standard messages are used to efficiently move data between MPI processes.

4 Performance Evaluation

We evaluate our approach using three iterative SPMD applications—Jacobi, Conjugate Gradient, and EpiGraph. Jacobi is an iterative method for solving a system of linear equations which operates on a symmetric dense matrix. Conjugate Gradient (CG) is a linear system solver suited for large sparse matrices. EpiGraph [2] is a distributed simulator for infectious diseases which iteratively operates on a sparse matrix which reflects a social interconnection graph.

For the experiments using Jacobi we generated random square matrices with different sizes: 5,000, 10,000 and 15,000 rows. For CG we used matrices from the University of Florida sparse matrix collection [15]. The matrices we selected are *nd24k* (size: 72,000, *number of nonzero elements* (*nnz*) 28,715,634), *ldoor* (size: 952,203, *nnz* 42,493,817), and *audikw_1* (size: 943,695, *nnz* 77,651,847). This subset is representative of data structures which exhibit regular and irregular data distribution patterns. We run Jacobi and CG for 10,000 iterations each. For the experiments using EpiGraph we simulated a population of 1,000,000 people (matrix size: 1,000,000, *nnz* 38,473,353) for a simulated time span of 20 days (2,880 iterations). The *sampling interval* is problem dependent and we experimentally set it to 100 iterations in our experiments.

Table 1 describes our target platform, a heterogeneous cluster consisting of 10 compute nodes of four different classes. All the compute nodes run under Linux Ubuntu Server 10.10 with 2.6.35-32 kernel and MPICH-2 (v.1.4.1p1), and are interconnected by a Gigabit Ethernet network.

4.1 Heterogeneous Dedicated System

We first evaluate FLEX-MPI by executing Jacobi, CG, and EpiGraph exclusively on heterogeneous configurations running 4, 8, 16, 32, and 64 processes. Table 2 describes the heterogeneous configurations of the cluster.

Table 3 shows the execution times for Jacobi and CG while Table 4 shows the execution times for EpiGraph. In our experiments we show the overall execution time (of the application and FLEX-MPI), including the computation and communication times of the application as well as the overhead of the monitoring, load balancing, and data redistribution of FLEX-MPI. The reference scenario (which execution time is T_{par}) employs an equal-size block distribution of the

Table 1. Heterogeneous cluster with number of nodes (N), sockets per node (S), and processing elements (PE) per socket for each class

Class	N	S	PE	Processor	Frequency	RAM
A	4	1	4	Intel Xeon E5405	2.00 GHz	4 GB
B	2	2	6	Intel Xeon E5645	2.40 GHz	24 GB
C	2	2	6	AMD Opteron 6168	800 MHz	64 GB
D	2	4	6	Intel Xeon E7-4807	1.87 GHz	128 GB

Table 2. Heterogeneous configurations, where $n(p)$ stands for the number of nodes (n) and the number of processes (p) running per node

Config.	Class A	Class B	Class C	Class D
HTC1-4	1 (1)	1 (1)	1 (1)	1 (1)
HTC2-8	1 (2)	1 (2)	1 (2)	1 (2)
HTC3-16	1 (4)	1 (4)	1 (4)	1 (4)
HTC4-32	2 (4)	1 (8)	1 (8)	1 (8)
HTC5-64	4 (4)	2 (8)	2 (8)	2 (8)

data without load balancing. Results show a significant improvement of up to 44% when executing the application with dynamic load balancing. Note that for 64 processes, the communication/computation ratio increases due to low workload per process. This produces performance degradation for the applications, and reduces the efficiency of the load balancing.

Fig. 2 illustrates a typical execution of Jacobi and EpiGraph when using FLEX-MPI. Jacobi is a regular application in which the amount of work done in each iteration is the same. This leads to very small variations over time in the execution time per iteration. In contrast, EpiGraph is an irregular application which exhibits a highly variable workload per iteration. When executing on a heterogeneous dedicated system, Jacobi requires a single data redistribution operation to balance the workload. It is triggered during the first sampling interval, in which the workload imbalance is larger than the imbalance tolerated by the algorithm. From that moment on the application is balanced and no further data redistribution operations are necessary. However, even on dedicated systems, irregular applications such as EpiGraph require several data redistribution operations to balance the workload.

Table 3. Heterogeneous dedicated system with: execution time of the application (T_{par}), execution time of the dynamic load balanced application - with FLEX-MPI (T_{FLX}), percentage of the time saved when executing with FLEX-MPI (T_{sav})

	Jacobi				CG			
Config.	Matrix	T_{par}(sec)	T_{FLX}(sec)	T_{sav}(%)	Matrix	T_{par}(sec)	T_{FLX}(sec)	T_{sav}(%)
HTC1-4	5,000	805	517	35.77	*nd24k*	1247	998	19.96
	10,000	3259	2071	36.45	*ldoor*	2634	2285	13.24
	15,000	7324	4683	36.05	*audikw_1*	4860	3537	27.22
HTC2-8	5,000	414	269	35.02	*nd24k*	682	538	21.11
	10,000	1665	1070	35.73	*ldoor*	1751	1676	4.28
	15,000	3707	2396	35.36	*audikw_1*	3562	2222	37.61
HTC3-16	5,000	208	151	27.40	*nd24k*	381	302	20.73
	10,000	843	698	17.20	*ldoor*	1387	1327	4.32
	15,000	1894	1384	26.92	*audikw_1*	2336	1789	23.41
HTC4-32	5,000	116	95	18.10	*nd24k*	220	188	14.54
	10,000	421	332	21.14	*ldoor*	1253	1234	1.51
	15,000	978	706	27.81	*audikw_1*	1844	1104	40.13
HTC5-64	5,000	108	100	7.40	*nd24k*	147	146	0.68
	10,000	580	446	23.10	*ldoor*	841	815	3.09
	15,000	880	756	16.40	*audikw_1*	1124	911	18.95

Table 4. Results of EpiGraph on heterogeneous dedicated system

	EpiGraph			
Config.	Matrix	T_{par}(sec)	T_{FLX}(sec)	T_{sav}(%)
HTC1-4	1,000,000	356	270	24.16
HTC2-8	1,000,000	222	156	29.73
HTC3-16	1,000,000	202	113	44.06
HTC4-32	1,000,000	161	102	36.65
HTC5-64	1,000,000	165	112	32.12

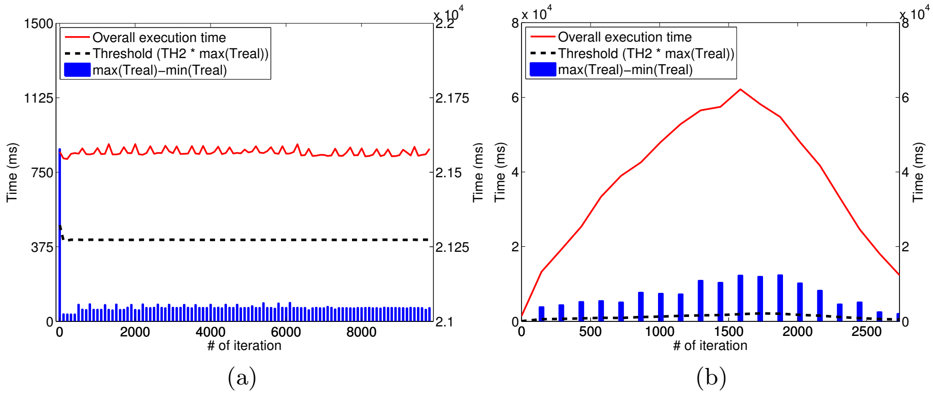


Fig. 2. Jacobi (a) and EpiGraph (b) on heterogeneous dedicated system. Right Y axis is the *Overall execution time* per iteration. Left Y axis shows both the difference between the maximum and minimum *Treal* (for all of the running processes for each sampling interval), and the *Threshold* value tolerated by the algorithm.

4.2 Heterogeneous Non-dedicated System

The following experiment evaluates how well the load balancing algorithm performs when external applications with workload that vary over time are sharing the underlying architecture for execution. We run Jacobi, CG, and EpiGraph for a heterogeneous configuration with 1 node Class A and 1 node Class B, each running 4 processes per node. We artificially introduce an external load which simulates an irregular computing pattern. This load consists of two processes which are simultaneously executed on the Class A node together with the application. The external load consists of a burst of short computing intervals followed by a single long computing interval which lasts until the end of the execution.

Table 5 shows the execution times for the benchmarks on the heterogeneous non-dedicated configuration we described above. We evaluated different values of k and their impact on the execution time. T_{par} stands for the execution time (in seconds) of the application when it runs without adapting to changes in performance due to the dynamic external load; $T_{k=n}$ stands for these execution times when the application does adapt to the external load, for different values of k. The execution time is reduced by up to 39.31% when the applications adapts to the external load. Results confirm our intuition to show that the most

Table 5. Jacobi, CG, and EpiGraph on heterogeneous non-dedicated system

Problem	Matrix	T_{par}(sec)	$T_{k=1}$(sec)	$T_{k=3}$(sec)	$T_{k=5}$(sec)
Jacobi	10,000	1652	1162	1162	1148
CG	*nd24k*	1076	688	665	653
EpiGraph	1,000,000	272	195	179	169

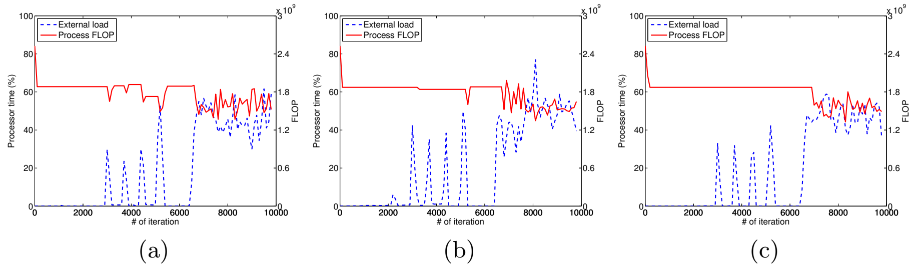


Fig. 3. Adaptive execution of Jacobi on a heterogeneous non-dedicated system for different values of k: (a) $k = 1$, (b) $k = 3$, and (c) $k = 5$. The external load (left Y axis) corresponds to the percentage of real time of the processing element consumed by the external load, while the process $FLOP$ (right Y axis) corresponds with the number of $FLOP$ performed by the process.

promising approach is to tolerate short external loads as to avoid the cost of re-balancing too eagerly.

Fig. 3 shows what happens on processing element $P0$ when we run Jacobi using FLEX-MPI and we introduce a dynamic external load on a subset of the processing elements. The workload redistribution triggered by the load balancing algorithm leads to a different number of $FLOP$ performed by the process (in red in the figure). The amount of data which needs to be redistributed depends on the magnitude of the external load (in blue) and the value of k. We can observe that for $k = 1$ the application adapts immediately to changes in the performance of the processing element, performing load balance for every external load burst. With $k = 3$ the first three smaller bursts are discarded, while larger values of k lead to discarding all the bursts but considering the long-term load.

5 Conclusions

We presented FLEX-MPI, an MPI extension for supporting dynamic load balancing of SPMD applications running on heterogeneous platforms in the presence of dynamic external workload. The extension we provide does not require prior knowledge about the underlying architecture, does not require dedicated resources, and it is based on precise runtime monitoring with negligible overhead. Our results show that by using FLEX-MPI the execution time of an application may be significantly reduced.

There are two main directions for future work that are of particular interest to us. The first extension we plan on developing is to improve FLEX-MPI by considering other types of hardware events, which are used to monitor other performance metrics. This is particularly useful for parallel applications which do not exhibit a linear correlation between the FLOP and the workload size, or applications based on integer operations. The second direction for future work is to improve FLEX-MPI by integrating the dynamic process management features of MPI-2 such that processes could be started or turned off on demand, based on a cost model and the performance goals of the application.

Acknowledgments. This work has been partially supported by the Spanish Ministry of Science under the grant IPT-430000-2010-14.

References

1. Bailey, D., et al.: The NAS parallel benchmarks summary and preliminary results. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, pp. 158–165. IEEE (1991)
2. Martín, G., Marinescu, M., Singh, D., Carretero, J.: Leveraging social networks for understanding the evolution of epidemics. BMC Syst. Biol. 5(suppl. 3) (2011)
3. Xu, C., Lau, F.: Load balancing in parallel computers: theory and practice. Kluwer Academic Publishers (1997)
4. Beltran, M., Guzman, A., Bosque, J.: Dealing with heterogeneity in load balancing algorithms. In: ISPDC 2006, pp. 123–132. IEEE (2006)
5. Belikov, E., Loidl, H., Michaelson, G., Trinder, P.: Architecture-aware cost modelling for parallel performance portability. In: Kolloquium Programmiersprachen und Grundlagen der Programmierung 5
6. Galindo, I., Almeida, F., Badía-Contelles, J.M.: Dynamic load balancing on dedicated heterogeneous systems. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 64–74. Springer, Heidelberg (2008)
7. Martínez, J., Almeida, F., Garzón, E., Acosta, A., Blanco, V.: Adaptive load balancing of iterative computation on heterogeneous nondedicated systems. The Journal of Supercomputing 58(3), 385–393 (2011)
8. Weatherly, D., Lowenthal, D., Nakazawa, M., Lowenthal, F.: Dyn-MPI: Supporting mpi on medium-scale, non-dedicated clusters. Journal of Parallel and Distributed Computing 66(6), 822–838 (2006)
9. Bohn, C., Lamont, G.: Load balancing for heterogeneous clusters of PCs. Future Generation Computer Systems 18(3), 389–400 (2002)
10. Lastovetsky, A., Reddy, R.: HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. Journal of Parallel and Distributed Computing 66(2), 197–220 (2006)
11. Huang, C., Lawlor, O., Kale, L.: Adaptive MPI. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004)
12. Dongarra, J., Malony, A., Moore, S., Mucci, P., Shende, S.: Performance instrumentation and measurement for terascale systems. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J., Zomaya, A.Y. (eds.) ICCS 2003, Part IV. LNCS, vol. 2660, pp. 53–62. Springer, Heidelberg (2003)
13. Martín, G., Marinescu, M., Singh, D., Carretero, J.: FLEX-MPI - Technical Report. Technical report, Universidad Carlos III de Madrid (2012), `http://www.arcos.inf.uc3m.es/~desingh/publications.html`
14. Mucci, P., Browne, S., Deane, C., Ho, G.: PAPI: A portable interface to hardware performance counters. In: Proceedings of the Department of Defense HPCMP Users Group Conference, 7–10 (1999)
15. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. 38(1), 1:1–1:25 (2011)

Enhancing Concurrency in Distributed Transactional Memory through Commutativity

Junwhan Kim, Roberto Palmieri, and Binoy Ravindran

ECE Department, Virginia Tech, Blacksburg, VA, 24061
{junwhan,robertop,binoy}@vt.edu

Abstract. Distributed software transactional memory is an emerging, alternative concurrency control model for distributed systems promising to alleviate the difficulties of lock-based distributed synchronization. We consider the multi-versioning (MV) model to avoid unnecessary aborts. MV schemes inherently guarantee commits of read-only transactions, but limit the concurrency of write transactions. In this paper we propose CRF (Commutative Requests First), a new scheduler tailored for enhancing concurrency of write transactions. CRF relies on the notion of commutative transactions, namely conflicting transactions that leave the state of the shared data-set consistent even if validated and committed concurrently. CRF is responsible to detect conflicts among commutative and non-commutative write transactions and then schedules them according to the execution state. We assess the goodness of the approach by an extensive evaluation of a fully implementation of CRF. The tests reveal that CRF improves throughput over a state-of-the-art DTM solution.

1 Introduction

Lock-based concurrency control suffers from programmability, scalability, and composability challenges [13]. Transactional memory (TM) promises to alleviate these difficulties sparing the programmers from the pitfalls of conventional manual lock-based synchronization, drastically simplifying the development of parallel and concurrent applications. As TM code is composed of read/write operations on shared objects, it is organized as *memory transactions*, which optimistically execute while logging changes on accessed objects. When two transactions conflict, a contention manager is responsible to resolve the conflict by aborting one of them, yielding (the illusion of) atomicity. Aborted transactions are typically re-started, after rolling-back the changes. The contention manager can be supported by the *transactional scheduler*, that is responsible to determine an ordering among concurrent transactions so that conflicts are either avoided or minimized, thereby reducing abort rate and improving performance.

Originally proposed to simplify concurrent programming in centralized environments, TM systems are being growingly employed in distributed settings (Distributed TM or DTM), motivated as an alternative to the more challenging distributed, lock-based, concurrency control. Thanks to the simple distributed programming model provided by DTM, the programmers can focus on the implementation of the application logic putting the charge of distributed synchronization into the hands of DTM [14,15].

With a single copy for each object, i.e., *single-version* STM (SV-STM), when a read/write conflict occurs between two transactions, the contention manager resolves the conflict by aborting one and allowing the other to commit, thereby maintaining the consistency of the (single) object version. SV-STM is simple, but suffers from large number of aborts [17]. In contrast, with multiple versions for each object, i.e., *multi-versioning* STM (MV-STM), unnecessary aborts of transactions, that could have been committed without violating consistency, are avoided [16]. Unless a conflict between operations to access a shared object occurs, MV-STM allows the corresponding transactions to read the object's old versions, enhancing concurrency.

MV-STM has been extensively studied for multiprocessors [17,9] and also for distributed systems [26]. MV-STM uses *snapshot isolation* (SI), which is weaker than serializability [20]. A transaction executing under SI operates on a snapshot taken at the start of the transaction. The transaction successfully commits if the objects updated by the transaction have not been changed externally since the snapshot was taken, guaranteeing that all read transactions will see a consistent snapshot. Many works [20,21,15] used SI for improving performance in centralized and distributed TM environments. Even though SI allows more concurrency among transactions respect to with serializability, a write-write transaction's conflict under SI causes the transaction to abort. In write-intensive workloads, this conflict cannot be avoided because the concurrency of write transactions may violate SI.

In this paper, we address the problem of permitting multiple conflicting transactions to commit concurrently, in order to enhance concurrency of write transactions without violating SI in multi-version DTM for high performance. We propose a transactional scheduler that enables concurrency of write transactions, called Commutative Requests First (CRF). In order to do that, CRF exploits the notion of commutative operations. Two operations are named *commutative* if applying them sequentially in either order, they leave the objects accessed in the same state and both return the same values. A very intuitive example of commutativity is when two operations, $call1(X)$ and $call2(X)$, accessing both to the same object X but different fields of X (**see Section 1(b)**).Thus, CRF checks whether write operations are commutative and lets them to validate and commit simultaneously. Unlike past STM works, that exploit high concurrency based on the commutativity property [10], CRF maintains a scheduling queue to identify commutative and non-commutative transactions, and could decide to allow all commutative transactions to commit first than the others, maximizing their concurrency. However, despite the significant performance obtained by adopting the idea of commutativity transactions of CRF, there could be applications that do not admit such kind of commutativity. CRF addresses this issue by permitting the developer to explicitly specify non-commutative operations.

We implemented a full-working prototype of CRF in the Scala DTM framework, called HyFlow [24], and conducted extensive experimental studies using micro benchmarks (LinkedList and SkipList [18], as well as a real application benchmark (TPC-C [5]) typically used for assessing the performance of DTM. Our studies reveal that transactional throughput is improved by up to 5× over a state-of-the-art DTM solution(DecentSTM [2]).

The rest of the paper is organized as follows. We outline the preliminaries and the system model in Section 2. We describe the CRF scheduler in Section 3. Implementation and experimental studies are reported in Section 4. We discuss the related work in Section 5 and Section 6 concludes the paper.

2 Preliminaries and System Model

We consider a distributed system which consists of a set of nodes $N = \{n_1, n_2, \cdots\}$ that communicate with each other by message-passing links over a network.

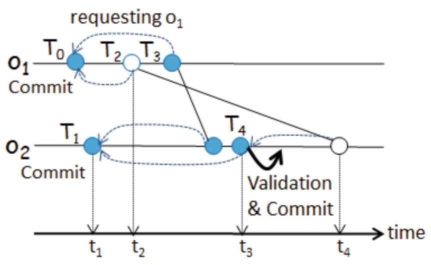
Distributed Transactions. A set of *distributed transactions* $T = \{T_1, T_2, \cdots\}$ is assumed that share objects $O = \{o_1, o_2, \ldots\}$, which are distributed in the network. An execution of a transaction is a sequence of timed operations, reads and writes. An execution ends by either a commit (success) or an abort (failure). Each transaction has a unique identifier, and it is invoked by a node.

We consider data flow DTM model [11] where transactions are immobile and objects move to the node invoking transactions. Each node has a *TM proxy* that provides interfaces to the local application and to proxies at other nodes. When a transaction T_i at node n_i requests object o_j, the TM proxy of n_i first checks whether o_j is in its local cache. If the object is not present, the proxy invokes a distributed cache-coherence protocol (e.g., [6,11]) to fetch o_j from the network. o_j may have multiple versions. The initial value of o_j is denoted by o_j^0. Let the version set of o_j be $\{o_j^0, o_j^1, \cdots\}$. Node n_k, holding the version set, checks whether the requested object version is in use by a local transaction T_k when it receives the request for o_j from n_i. If so, n_k's TM proxy invokes a contention manager to manage the conflict between T_i and T_k for the object version of o_j.

Atomicity, Consistency, and Isolation. We use the *Transactional Forwarding Algorithm* (TFA) [23] to provide *early validation* of remote objects, guarantee a consistent view of shared objects among distributed transactions, and ensure atomicity for object operations in the presence of asynchronous clocks. With early validation we refer to the fact that a transaction has already successfully validated its accessed objects before committing. A validation in distributed systems includes global registration of object ownership.

3 Commutative Requests First in MV-TFA

Multi-Version TFA. In this section we present multi-version MV-TFA, our extension of TFA supporting SI. The basic idea is to record an event whenever requesting and acquiring an object. Let n_i denote a node invoking a transaction T_i. We define two types of events: (1) *Request*$(\mathrm{Req}(\mathrm{n_i}, \mathrm{o_j}))$ representing the request of object o_j from node n_i; (2) *Acquisition*$(\mathrm{Acq}(\mathrm{n_i}, \mathrm{o_j}))$ indicating when node n_i acquires object o_j. Figure 1(a) shows an example execution scenario of MV-TFA. We use the same style in the figure as that of [22]. The solid circles indicate write operations and the empty circles represent read operations. Transactions' evolution is represented on horizontal lines with the circles. The horizontal line corresponding to the status of each object describes the time domain. The dotted line indicates which node requests an object from where.



(a) Example of MV-TFA

Commutativity

insert (x)/ ⇔ insert (y)/ , x ≠ y

remove(x)/ ⇔remove(y)/ , x ≠ y

insert (x)/ ⇔remove(y)/ , x ≠ y

add(x)/*false* ⇔remove(x)/*false* ⇔ contains (x)/

(b) Specification of a Set

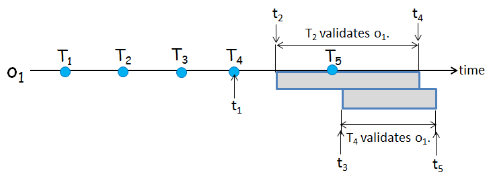
Fig. 1.

Assume that transactions T_0 and T_1 invoked on nodes n_0 and n_1 commit after writing o_1^0 and o_2^0, respectively. Let transactions T_2, on node n_2, and T_3, on node n_3, request objects o_1 and o_2 from nodes n_0 and n_1, respectively. Node n_1 holds the list of versions of o_2. After that, T_3 requests o_1 from n_0 and subsequently T_4 requests o_2 from n_1. Thus, n_1 records the events $Acq(n_3, o_2^0)$ and $Acq(n_4, o_2^0)$. Then T_4 updates o_2 creating a new version o_2^1. When T_4 validates o_2^1 to commit, $Acq(n_4, o_2^0)$ is removed from the events log of n_3, and T_3 has forced to abort because in the n_3's log there is another request $(Acq(n_3, o_2^0))$ on the same object o_2. The presence of this entry in the log means that T_3 has not yet completed, so T_4 definitively commits before that T_3 validates o_2, invalidating the object o_2^0 accessed by T_3. As a consequence of T_4 commitment, node n_4, which invokes T_4, receives the versions o_2^0 and o_2^1 of object o_2. Now, after the commit of T_4, T_2 requests o_2 with the value $\mid t_4$ - $t_2 \mid$ from n_4. It replies with the version o_2^0 instead of the newly o_2^1 because o_2^0 has been updated at time t_1 to T_2, because $\mid t_4 - t_3 \mid < \mid t_4$ - $t_2 \mid < \mid t_4 - t_1 \mid$. Using this mechanism, T_2 can access to a consistent snapshot that is not affected by a write operation by T_4, instead of being aborted due to T_4 's write. This is how MV-TFA ensures SI.

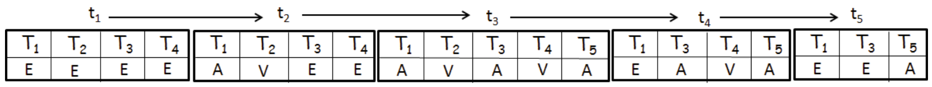
CRF Scheduler Design. MV-TFA shows how to enhance performance in case of workload characterized by mostly read transactions, exploiting multi-versions. In this subsection we focus on how to schedule write transactions concurrently minimizing the abort rate and increasing the parallelism. When a transaction T_1 at node n_1 needs object o_1 for an operation, it sends a request to the o_1's object owner. If the operation is read, a version of o_1 is sent to n_1. If the operation is write, we consider two possible cases in terms of o_1. *(A)* The first case happens when other transactions may have requested o_1 but no transaction has validated o_1. In this case, a version of o_1 is sent to n_1 and T_1's request moves into the scheduling queue of the o_1's owner. *(B)* The second case is when another transaction T_2 is validating o_1. In this case, unless T_2 and T_1 commute, T_1 will abort and T_1's request also moves to the scheduling queue. If T_2 and T_1 commutes, o_1 is sent to n_1 and T_1's request moves to the scheduling queue. The o_1's owner maintains the scheduling queue to execute commutative transactions concurrently. Accordingly, non-commutative transactions is executed serially.

To better assess CRF, we use it to implement the specification of a *Set* provided by [10]. We recall that a *Set* is a collection of items without duplications in which the following operations are provided: *add*(x), *remove*(x) and *contains*(x) where x is the item of the *Set* accessed. Figure 1(b) summarizes *Set* operations' commutativity according to [10]'s definition. In the specification illustrated in Figure 1(b), operations *insert*(x), *insert*(y), and *insert*(z) commutes if $x \neq y \neq z$. Multiple write transactions may be invoked concurrently on the *Set*. CRF identifies commutative and non-commutative transactions and gives to the commutative transactions a chance to validate concurrently an object first. However, if we consider the specification of the *Set*, in which the are no commutative operations declared, and we encapsulate the *Set* into an object (o_1) and we consider the above operations as transactions, then typical concurrency control does not permit to validate and commit concurrently more than one transaction performing an update on the object (namely updating the *Set*). Conversely, by Figure 1(b) is clear that multiple update transactions can be validated concurrently whether they access to different items in the set. The scheduling queue holds requests for those operations. If multiple transactions have requested the same version of o_1, CRF allows the commutative transactions to concurrently validate o_1. Meanwhile, many commutative transactions may validate o_1. This could bring non-commutative transactions to "starve" on o_1. Thus, CRF alternates between periods (called *epochs*), in which it privileges the validation of a group of commutative transactions, with others in which it prefer to validate the non-commutative ones. In this way, CRF handles conflicts between commutative and non-commutative transactions. Although epochs contain commutative transactions, these transactions do not commute with the transactions of the next epoch in the chronological sequence. The terminology "commutative" and "non-commutative" epoch distinguishes between these two epochs. Thus, in commutative epoch, commutative transactions validate o_1 and then in the next (i.e., non-commutative) epoch, non-commutative transactions, excluded in the previous commutative epoch, can validate o_1. If a transaction starts validating o_1, its commutative transactions are also allowed to validate o_1 but its non-commutative transactions abort. The non-commutative transactions will resume after the commutative transactions commit.

CRF checks for whether different operations commute at the level of semantics. Even when commutative operations concurrently update the object, the object preserves a consistent state, ensuring SI. There are two purposes for processing commutative requests first. MV-TFA ensures concurrency of read transactions, and CRF is responsible to detect conflicts among commutative and non-commutative write transactions, reducing the number of conflicts. This leads to higher concurrency. Second, CRF alleviates contention when many write transactions are invoked. Even though a conflict between two write transactions occurs, all subsequent commutative transactions are scheduled first. Non-commutative transactions restart simultaneously after the commutative transactions complete, so CRF avoids further conflicts, decreasing contention.



(a) Requests of Five Transactions and Validation of Two Transactions for Object o_1.



$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_5$

T1	T2	T3	T4
E	E	E	E

T1	T2	T3	T4
A	V	E	E

T1	T2	T3	T4	T5
A	V	A	V	A

T1	T3	T4	T5
E	A	V	A

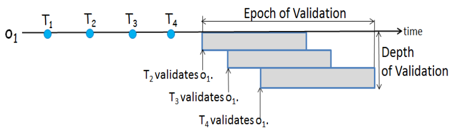
T1	T3	T5
E	E	A

(b) Scheduling Queue Located in o_1 Object Owner. The scheduling queue consists of two rows: Enqueued Transactions and State of the Transactions. V (Validation), A (Abort), and E (Execution)

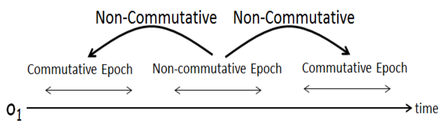
Fig. 2. A Scenario of CRF

Illustrative Example. Figure 2 shows a scenario of CRF. The write transactions T_1=*insert*(x), T_2=*remove*(x), T_3=*insert*(y) and T_4=*remove*(y) request concurrently o_1 from its owner. The transactions obtain the version of o_1. The state of the scheduling queue at t_1, illustrated in Figure 2(b), shows that the transactions are all executing. At t_2, T_2 starts validating o_1. Consequently, T_1 aborts because T_1 and T_2 do not commute. Conversely, T_3 and T_4 can still execute because they are commutative with T_2. Then T_5=*remove*(x) requests o_1 during the validation of T_2 and immediately aborts because T_5 and T_2 do not commute. At t_3, T_4 starts validating o_1 and T_3 aborts because T_3 and T_4 do not commute. Thus, T_2 and T_4 concurrently validate o_1. When T_2 ends validation (i.e., commits) at t_4, the version updated by T_2 is sent to the non-commutative transaction T_1, and T_1 starts executing. Even though T_5 is a non-commutative transaction of T_2, only T_1 starts to avoid a conflict between non-commutative transactions. Finally, the version updated by T_4 at t_5 is sent to T_3. T_1 and T_3 may validate o_1 concurrently because they commute.

Figure 3(a) shows that the validation of commutative transactions may not be completely overlapping, so the period of validation may be stretched. This may lead to the deferred execution of non-commutative transactions. To prevent this, we define a new parameter, called *depth of validation*, namely the number of transactions involved in the validation. Figure 3(a) indicates 3 for that depth,



(a) Epoch and Depth of Validation.



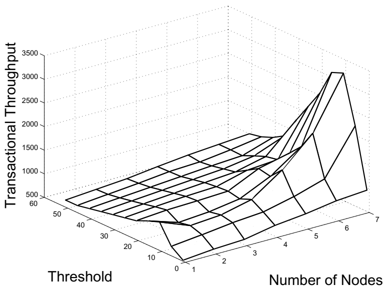
(b) Epochs of Validation

Fig. 3. Epoch-based CRF

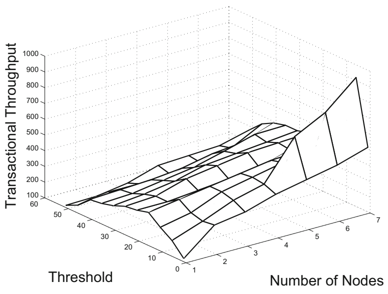
meaning that the commits of three transactions mark the end of the epoch. Non-commutative transactions will start after the epoch. Figure 3(b) illustrates the relationship of epochs. In each epoch, commutative transactions concurrently participate in validation. At the end of the epoch, their non-commutative transactions held in a scheduling queue, restart. Non-commutative transactions will validate in the next epoch.

4 Implementation and Experimental Evaluation

Implementation. We implemented CRF on MV-TFA using Scala's actor model for Java Virtual Machine. The actor model prohibits sharing memory by encapsulating mutable state inside light-weight sequential constructs called actors.



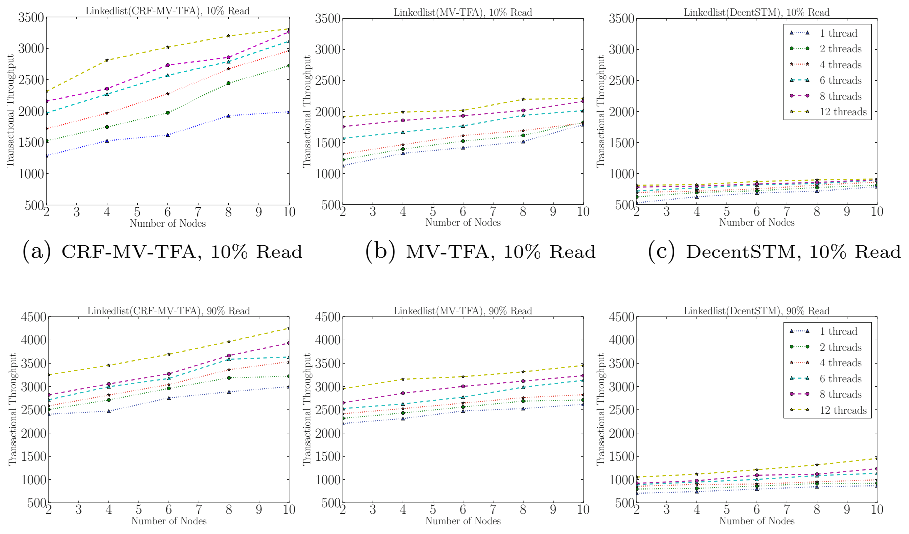
(a) LinkedList



(b) TPC-C

Fig. 4. Throughput Varying Thresholds

Commutativity of Benchmarks. We assess the performance of CRF using LinkedList and SkipList as micro-benchmarks and a TPC-C [5] as real-application benchmark. Regarding the commutativity in micro-benchmarks, the *Set* (see Section 3) can be implemented with LinkedList and SkipList [10], so we rely on the definition of commutativity in Figure 1(b). Regarding TPC-C, the write transactions consist of update, insert, and/or delete operations accessing a database of nine tables maintained in memory. Each row in the tables has a unique key. Multiple operations commute if they access to a row (or object) with the same key and modify different columns. We rely on explicit annotations



(a) CRF-MV-TFA, 10% Read (b) MV-TFA, 10% Read (c) DecentSTM, 10% Read

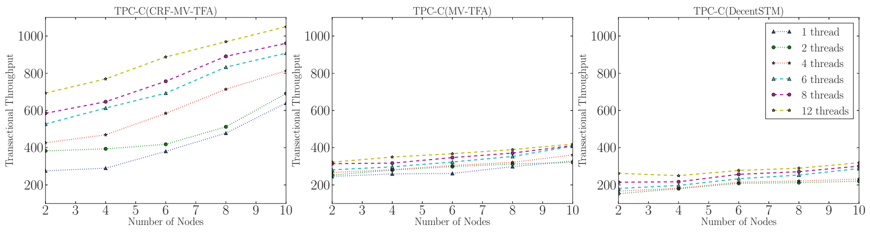
(d) CRF-MV-TFA, 90% Read (e) MV-TFA, 90% Read (f) DecentSTM, 90% Read

Fig. 5. Throughput of CRF, MV-TFA, and DecentSTM Using LinkedList

provided by the programmer, indicating the fields accessed by each transaction profile. We configured the benchmark with a limited number of warehouses (#4) in order to generate high conflicts. We recall that, in data flow model, objects are not bound on fixed nodes but move, increasing likelihood of conflicts.

Experimental Setup. Our test-bed consists of 10 nodes connected via a switched 1 Gigabit network connection. Each node is comprised of 12 Intel Xeon 1.9GHz processor cores. We use the Ubuntu Linux 10.04 server OS. We measured the *transactional throughput* (number of committed transactions per second). To manage garbage collection, versions that are no longer accessible, need to be marked. Unlike multiprocessors, determining old versions for live transactions in distributed systems incurs communication overheads. Thus, we consider a threshold-based garbage collector [4], which checks the number of versions and disposes the oldest if the number of versions exceeds a pre-defined threshold. We consider threshold 4 for measuring the basic event model's throughput, because the observed that the speed-up is relatively less increased after the threshold.

Finding a Depth. The large number of concurrent validations may lead to a significant scheduling overhead due to delayed non-commutative transactions. For the balance of commutative and non- requesting transactions, we consider a threshold-based control, switching the next epoch when either a depth or a number of non-commutative transactions enqueued meets a predefined threshold, called $MaxD$. Figure 4 shows throughput moving the $MaxD$ from 1 to 50. By the plot is clear that CRF's throughput is not improved after $MaxD$=10 for LinkedList and $MaxD$=5 for TPC-C due to the increasing number of non-commutative transactions aborted. With the previous values of $MaxD$, CRF reaches its maximum throughput, so we used those for the experiments.



(a) CRF-MV-TFA (b) MV-TFA (c) DecentSTM

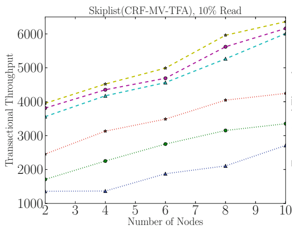
Fig. 6. Throughput of CRF, MV-TFA, and DecentSTM Using TPC-C

Evaluation. Figures 5,7 show the throughput of CRF, MV-TFA and DecentSTM using LinkedList(Figure 5) and SkipList(Figure 7) benchmarks. The legend has to be considered for all the plots and shows the colors differentiate for number of running threads. Each micro-benchmark has been evaluated using two workloads representative of read intensive (10% writes and 90% reads) and write intensive (90% writes and 10% reads) scenarios. The tests have been performed varying the number of nodes and the number of threads per node. Each thread submits requests to the distributed system.

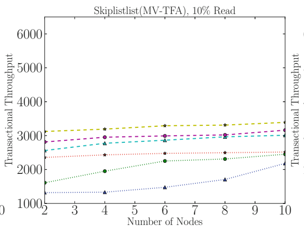
Summarizing, we span scenarios from 2 up to 120 concurrent threads in the system. This allows to exhaustive assess the behavior of CRF. The comparison between CRF and MV-TFA shows how much CRF enhances the concurrency of write transactions. For the LinkedList and SkipList, the new value to add or delete is randomly selected using a uniform distribution. According to the increasing number of threads and nodes, CRF performs better due to the detection of a large number of commutative operations. Even though the throughput of CRF is slightly better than MV-TFA in scenario characterized by most read-only transactions (due to the limited number of commutative write operations), the maximum gain of CRF against competitors is reached in write-intensive workload where CRF exploits the ability to validate and commit concurrently conflicting transactions. In additional the plot reveals that, in write dominated workload, CRF scales better than MV-TFA and DecentSTM. In fact, in contrast with CRF, their performance stall when increasing the number of concurrent threads in the system. This is also confirmed by the plots in Figure 7(a) and 7(b) where CRF outperforms MV-TFA by as much as 2×.

As a competitor, DecentSTM [2] is based on a snapshot isolation algorithm, which requires searching the history of objects to find a valid snapshot. This algorithm incurs a significant overhead. Thus, we observe that the transactional throughput of DecentSTM is not improved as long as requesting nodes increase.

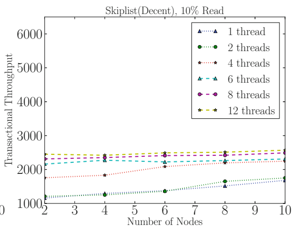
Our evaluations reveal that CRF improves throughput over MV-TFA and DecentSTM by as much as (average) 2× and 3× under 10% read transactions, respectively. Further, our evaluations show that MV-TFA outperforms DecentSTM in throughput as much as 2×. Figure 6 shows the throughput of CRF, MV-TFA, and DecentSTM using TPC-C benchmark. We used the amount of read and write transactions that the specification of TPC-C recommends. TPC-C benchmark accesses large tables to read and write values. Due to the non-negligible transaction execution time, scheduling commutative operations highly impacts the overall performance. In fact, the conflicting transactions generated by the benchmark are well managed by CRF and this results observing that CRF performs better than DecentSTM as much as 5× over 10 nodes.



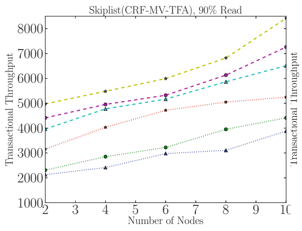
(a) CRF-MV-TFA, 10% Read



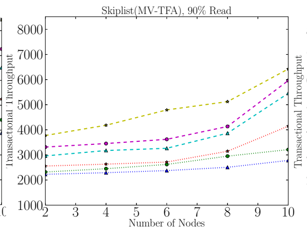
(b) MV-TFA, 10% Read



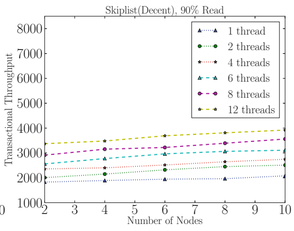
(c) DecentSTM, 10% Read



(d) CRF-MV-TFA, 90% Read



(e) MV-TFA, 90% Read



(f) DecentSTM, 90% Read

Fig. 7. Throughput of CRF, MV-TFA, and DecentSTM Using SkipList

5 Related Work

Transactional scheduling has been explored in a number of multiprocessor STM efforts [8,1,25,7]. In [8], is described an approach that dynamically schedules transactions based on their predicted read/write access sets. In [1], the authors discuss the Steal-On-Abort transaction scheduler, which queues an aborted transaction behind the non-aborted transaction, and thereby prevents the two transactions from conflicting again. In [25] is presented Adaptive Transaction Scheduler (ATS), that adaptively controls the number of concurrent transactions based on the contention intensity: when the intensity is below a threshold, the transaction begins normally; otherwise, the transaction stalls and does not begin until dispatched by the scheduler. CAR-STM scheduling approach [7] uses per-core transaction queues and serializes conflicting transactions by aborting one and queueing it on the other's queue, preventing future conflicts. In [3] has been proposed the Proactive Transactional Scheduler (PTS). This scheme detects hot spots of contention that can degrade performance, and proactively schedules affected transactions around the hot spots.

Steal-On-Abort, CAR-STM, and BIMODAL enqueue aborted transactions to minimize future conflicts in SV-STM. In contrast, CRF only enqueues non-commutative transactions that conflict with commutative transactions. The purpose of enqueuing is to prevent contending transactions from requesting all objects again. Thus, CRF also minimizes conflicts, but the overhead of CRF's scheduling is lower than the others because the number of enqueued transactions is smaller. ATS and PTS determine contention intensity and use it for contention management. Unlike these schedulers which are designed for multiprocessors, CRF maintains contention monitoring only between commutative and non-commutative write transactions, alleviating some of the overhead of contention management. In terms of commutativity, in [12] has been used a similar approach of CRF in order to run in parallel independent parts of the code.

It is important to note that, MV-STM has been extensively studied for multiprocessors and for distributed systems. In [19] is presented a dependency-aware transactional memory (DATM) for multiprocessors, where transaction execution is interleaved, and show substantially more concurrency than two-phase locking.

6 Conclusions

We presented a commutativity-based transactional scheduler for multi-version DTM, called CRF. CRF focuses on how to enhance concurrency of write transactions in multiversioning schemes ensuring SI, where write transactions are exposed to a high probability of conflicts. Our key idea is to detect a conflict between commutative and non-commutative write transactions and allow the first ones to commit concurrently before the others. CRF's design shows how commutativity-based scheduling impacts throughput in DTM. Our experimental evaluation shows that CRF enhances throughput over a state-of-the-art DTM solution, by 3 and 5$\times$ using micro-benchmarks and real-application.

Acknowledgements. This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

References

1. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 4–18. Springer, Heidelberg (2009)
2. Bieniusa, A., Fuhrmann, T.: Consistency in hindsight: A fully decentralized STM algorithm. In: 2010 IEEE IPDPS, pp. 1–12 (April 2010)
3. Blake, G., Dreslinski, R.G., Mudge, T.: Proactive transaction scheduling for contention management. In: Microarchitecture, pp. 156–167 (December 2009)
4. Büttcher, S., Clarke, C.L.A.: Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In: CIKM 2005, pp. 317–318. ACM (2005)
5. TPC Council, tpc-c benchmark, revision 5.11 (February 2010)

6. Demmer, M.J., Herlihy, M.P.: The arrow distributed directory protocol. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 119–133. Springer, Heidelberg (1998)
7. Dolev, S., Hendler, D., Suissa, A.: CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In: PODC (2008)
8. Dragojević, A., Guerraoui, R., et al.: Preventing versus curing: avoiding conflicts in transactional memories. In: PODC 2009, pp. 7–16 (2009)
9. Fernandes, S.M., Cachopo, J.: Lock-free and scalable multi-version software transactional memory. In: PPoPP 2011, pp. 179–188. ACM (2011)
10. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: PPoPP 2008, pp. 207–216. ACM (2008)
11. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing 20(3), 195–208 (2007)
12. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: PLDI (2007)
13. James, R.: Larus and Ravi Rajwar. Transactional Memory. M. and Claypool (2006)
14. Peluso, S., Ruivo, P., Romano, P., Quaglia, F., Rodrigues, L.: When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In: ICDCS 2012 (2012)
15. Peluso, S., Romano, P., Quaglia, F.: SCORe: A scalable one-copy serializable partial replication protocol. In: Narasimhan, P., Triantafillou, P. (eds.) Middleware 2012. LNCS, vol. 7662, pp. 456–475. Springer, Heidelberg (2012)
16. Perelman, K.: On avoiding spare aborts in transactional memory. In: SPAA 2009 (2009)
17. Perelman, K., Fan.: On maintaining multiple versions in STM. In: PODC 2010 (2010)
18. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. (1990)
19. Ramadan, H.E., et al.: Dependence-aware transactional memory for increased concurrency. In: MICRO, pp. 246–257 (2008)
20. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: ACM SIGPLAN TRANSACT 2006 (2006)
21. Riegel, T., Fetzer, C., Felber, P.: Time-based transactional memory with scalable time bases. In: SPAA (2007)
22. Riegel, T., Fetzer, C., Sturzrehm, H., Felber, P.: From causal to z-linearizable transactional memory. In: PODC (2007)
23. Saad, M., Binoy Supporting, R.: STM in distributed systems: Mechanisms and a Java framework. In: ACM SIGPLAN Workshop on Transactional Computing (2011)
24. Turcu, A., Ravindran, B.: Hyflow2: A high performance distributed transactional memory framework in scala, `http://hyflow.org`
25. Yoo, R.M., Lee, H.-H.S.: Adaptive transaction scheduling for transactional memory systems. In: SPAA, pp. 169–178 (2008)
26. Zhang, B., Ravindran, B.: Brief announcement: on enhancing concurrency in distributed transactional memory. In: PODC (2010)

Topic 4: High-Performance Architectures and Compilers
(Introduction)

Denis Barthou, Wolfgang Karl, Ramón Doallo, Evelyn Duesterwald, and Sami Yehia

Topic Committee

The topic "High Performance Architectures and Compilers" deals with architecture design and compilation for high performance systems. The areas of interest range from microprocessors to large-scale parallel machines (including multi-core, possibly heterogeneous, architectures); from general-purpose platforms to specialized hardware (e.g., graphic coprocessors, low-power embedded systems); and from hardware design to compiler technology.

On the compilation side, topics of interest include programmer productivity issues, concurrent and/or sequential language aspects, program analysis, program transformation, automatic discovery and/or management of parallelism at all levels, and the interaction between the compiler and the rest of the system. On the architecture side, the scope spans system architectures, processor micro-architecture, memory hierarchy, and multi-threading, and the impact of emerging trends.

The papers submitted to this topic were thoroughly reviewed and discussed. For each of the papers we obtained four reviews. We would like to thank all reviewers who helped in this process. Finally, four papers were accepted which are summarized below.

The paper "Adaptive Granularity Control in Task Parallel Programs using Multiversioning" by Peter Thoman, Herbert Jordan, and Thomas Fahringer introduces a method to adapt dynamically the granularity of fine-grained parallel programs. The method has two stages: first a set of versions of the input program are generated by a compiler, basically this compilation relies on the unrolling techniques to generate versions of the code with different granularity, then, the optimal granularity is adapted dynamically at run time using a simple algorithm based in heuristics. The evaluation mostly uses benchmarks from BOTS and compares the approach with Cilk, Intel ICC and GCC with OpenMP. The experimental results show that the method is effective to increase the efficiency of recursive parallel programs.

The paper "Adaptive Snoop Granularity in Hardware Transactional Memory" by Ehsan Atoofian presents an approach to reduce the coherency traffic in a Hardware transactional memory (HTM) system. The idea relies on remembering regions of conflicts in a snoop granularity table and filtering out subsequent snoops if it is known that the region is not conflicting with others. The evaluation shows that this approach is effective and also reduces the energy consumption of the bus.

The paper "Towards Efficient Dynamic LLC Home Bank Mapping with NoC-Level Support" by Mario Lodde, José Flich, and Manuel E. Acacio proposes a new approach for optimizing the use of shared last level caches in tiled CMPs. The principle is to dynamically determine a home bank, taking into account the topology of the CMP and the occupation of each LLC. A migration mechanism is included in order to better locate shared blocks. In the context of tiled CMPs, this addresses an important issue concerning scalability and performance.

Finally, the paper "Online Dynamic Dependence Analysis for Speculative Polyhedral Parallelization" by Alexandra Jimborean, Philippe Clauss, Juan Manuel Martinez, and Aravind Sukumaran-Rajam presents a runtime dependence analysis for speculative parallelization, based on VMAD, a framework for program analysis and instrumentation. The dependence analysis used is a combination of range and GCD test, using dependence distance vector as abstraction.

Adaptive Granularity Control in Task Parallel Programs Using Multiversioning

Peter Thoman, Herbert Jordan, and Thomas Fahringer

University of Innsbruck, Institute of Computer Science
Technikerstrasse 21a, 6020 Innsbruck, Austria
`{petert,herbert,tf}@dps.uibk.ac.at`

Abstract. Task parallelism is a programming technique that has been shown to be applicable in a wide variety of problem domains. A central parameter that needs to be controlled to ensure efficient execution of task-parallel programs is the granularity of tasks. When they are too coarse-grained, scalability and load balance suffer, while very fine-grained tasks introduce execution overheads.

We present a combined compiler and runtime approach that enables automatic granularity control. Starting from recursive, task parallel programs, our compiler generates multiple versions of each task, increasing granularity by task unrolling and subsequent removal of superfluous synchronization primitives. A runtime system then selects among these task versions of varying granularity by tracking task demand.

Benchmarking on a set of task parallel programs using a work-stealing scheduler demonstrates that our approach is generally effective. For fine-grained tasks, we can achieve reductions in execution time exceeding a factor of 6, compared to state-of-the-art implementations.

Keywords: Compiler, Runtime System, Parallel Computing, Task Parallelism, Multiversioning, Recursion.

1 Introduction

Task-based parallelism is one of the most fundamental parallel abstractions in common use today [1]. While relatively easy to implement and use, achieving good efficiency and scalability with task parallelism can be challenging. A central feature of every task-based parallel program that significantly affects both efficiency and scalability is *task granularity* [8]. The *granularity* of tasks is defined by the length of the execution time of a single task between interactions with the runtime system, such as spawning new tasks.

Very fine-grained, short-running tasks lead to a loss in efficiency compared to sequential execution due to the runtime overhead associated with generating and launching a task, as well as synchronizing its completion with other tasks in the system. On the other hand, coarse-grained, long-running tasks minimize overhead, but are hard to schedule effectively and may therefore fail to scale well on large parallel systems. Previous work in this area has focused mostly on runtime systems or user-controlled cutoffs to manage granularity (see Section 5). Conversely, we propose an approach that combines a multiversioning compiler with a runtime system which adaptively selects from the generated versions. Our goal is to maximize efficiency by increasing task granularity – and thus decreasing overheads – without negatively affecting load balance or scalability.

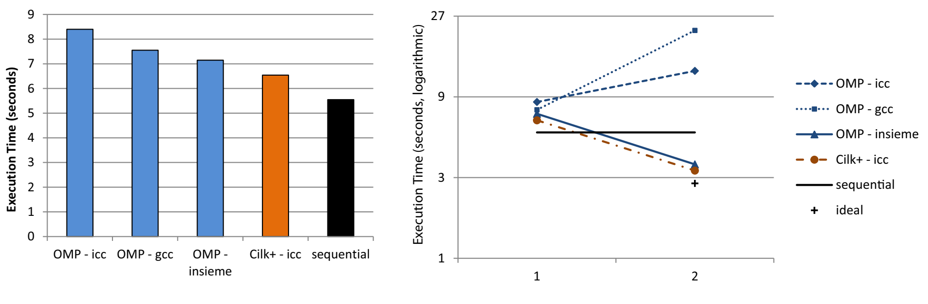
We implemented our method for OpenMP [17] tasks within the Insieme compiler and runtime system [11], but the idea is equally applicable to any other task parallel language. Our concrete contributions are the following:

- A compile-time multiversioning transformation that generates a set of task implementations of increasing granularity by recursive *task unrolling* and subsequent elimination of superfluous synchronization primitives. This transformation is applicable to both simple recursion and N-ary mutual recursion.
- A runtime heuristic for the dynamic adaptation of granularity based on the concept of *task demand*, which automatically choses the code version to execute at each task spawning point.
- Evaluation and analysis of the performance of our method on a number of well-known task parallel benchmarks. We compare with other OpenMP implementations, our own implementation without the multiversioning optimization and Cilk [2] versions which represent the state of the art in fine-grained task parallelism.

The remainder of this paper is structured as follows. In Section 2 we provide some initial results that motivated our work. We then describe our method in detail in Section 3 and evaluate its performance in Section 4, followed by an overview of related work in Section 5. Finally, Section 6 concludes the paper.

2 Motivation

Figure 1(a) shows single-threaded execution times measured for the Barcelona OpenMP Tasks Suite (BOTS) [7] N-Queens benchmark with $N = 13$. For details on the hardware, compiler versions and programs used refer to Section 4.



(a) Single-threaded (b) Scaling from 1 to 2 threads

Fig. 1. Initial Experiments, N-Queens $N = 13$

The lowest execution time amongst the OpenMP versions is achieved by our compiler and runtime system (Insieme), however, this time is still 28% higher than purely sequential execution. Even the Cilk version, while more efficient than any OpenMP implementation, is 19% slower than the sequential version. Our multiversioning method is designed to address this inefficiency. Throughout this paper, when we refer to *inefficient* execution, we mean execution which takes longer than executing purely sequential code (assuming perfect scaling).

Note that the OpenMP runtime systems of ICC [12] and GCC [19] perform special case handling when only a single worker thread is used. This is visible in Figure 1(b), which shows their performance degrading when switching from one to two threads. Further experiments in Section 4 confirm this behavior, with scaling starting after some initial performance degradation when activating multi-threaded execution. The OpenMP version compiled with Insieme and the Cilk version do not suffer from this issue, however they still induce a relative overhead of about 20% compared to ideal linear scaling from the sequential version. We identified the following potential causes for this inefficiency:

1. Task generation overhead. This includes generating a task structure, populating it with values and enqueuing it.
2. Synchronization primitive overhead (e.g. `taskwait`). At the very least, this involves keeping track of all the subtasks launched by each task, and signaling when they are complete.
3. Task library calls. The runtime methods required for tasking are generally implemented in a separate library, and the overhead for their invocation is incurred even if they perform no actual work.
4. Non-inlineable, indirect program function calls. Since the program function implementing a given task needs to be called by the tasking library, a pointer to it is usually passed to the library function. Even if the runtime library decides to directly execute the call, this prevents the benefits – improved instruction scheduling and a reduction in overhead – associated with inlining.

Issues 1 and 2 can be mitigated by a pure runtime approach, e.g. the runtime library can dynamically decide whether to generate a full task structure or directly call the task function. This method is usually referred to as lazy task creation [14]. However, the basic overhead of library function calls (issue 3) and the fact that indirectly called functions in the original program can not be inlined (issue 4) can not be changed at runtime and need to be handled at compile time. This limitation of pure runtime systems motivates our compiler-aided multiversioning approach.

All four potential causes for inefficient execution identified above are directly related to and influenced by the granularity of tasks. The more often individual tasks are generated and synchronized, the higher the impact of the associated overheads on execution time. However, simply increasing the granularity of all tasks is not a solution: such an approach will lead to load imbalance, increasing the probability of workers idling. Therefore, our goal is the generation of different implementations for each task.

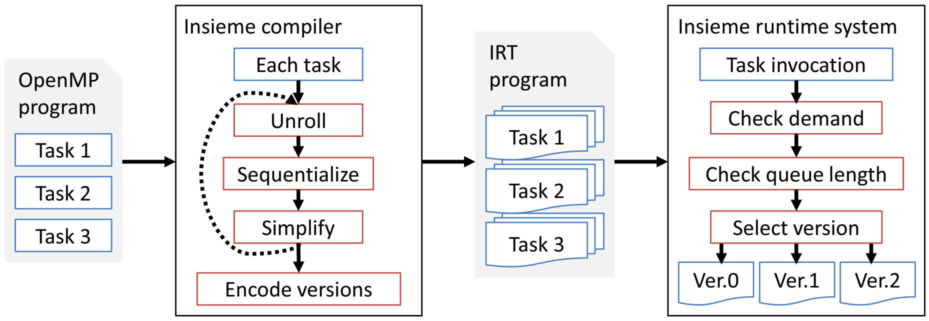


Fig. 2. Overview of our Method

3 Method

Figure 2 provides an overview of our proposed method. Starting from an OpenMP program with parallel tasks, our compiler generates an IRT (Insieme Runtime) program in which multiple different implementation versions of each task are encoded. During execution of the program, whenever a specific task is invoked, the Insieme runtime system selects and launches a version of this task.

3.1 Compile-Time Multiversioning

During compilation our goal is to generate multiple versions of each parallel task, with varying granularity. As depicted in Figure 2 this involves a three step process, which may be applied multiple times to further increase the task size. The individual steps are as follows:

1. **Task unrolling**. Replaces each task invocation site with a direct call to the task function, which is subsequently inlined. This can be thought of as a context and parallelism-aware recursive function inlining step. The name *task unrolling* is adapted from Rugina's usage of *recursion unrolling* [18].
2. **Sequentialization**. This step focuses on identifying which synchronization primitives – if any – were rendered superfluous by the partial elimination of parallel task invocations due to task unrolling, and removing them. It is described in more detail below.
3. **Simplification**. The unrolling and sequentialization may have generated code that can be simplified by basic operations such as arithmetic simplification, constant propagation or dead code elimination. Thus, these are performed before any further processing.

The number of generated versions depends on the granularity of the initial tasks and the largest granularity desired. The versions are generated and encoded into the target program in the following order.

1. **Original**. The original version from the input program.
2. N **times unrolled versions**. Starting from $N = 1$. In these versions, only partial sequentialization is performed. Outer task spawning points are

removed, but the innermost spawning location is kept. This process is illustrated in detail in a code example in Figure 4, described below.

3. **Fully sequentialized version**. In this version all task spawning points are removed and replaced with plain function calls.

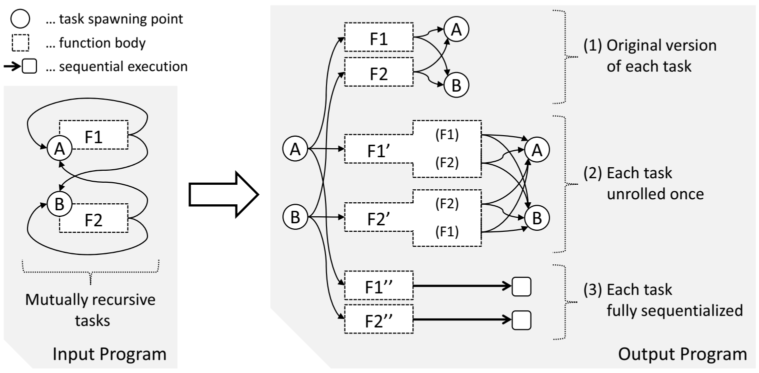


Fig. 3. Version Generation and Control Flow

Figure 3 illustrates the result of generating 3 versions for a mutually recursive task set consisting of two functions $F1$ and $F2$. The original program thus has two task spawning locations, A (which spawns $F1$) and B (spawning $F2$). To improve the clarity of the illustration, these task spawning points have been replicated in the figure, however they are still all referring to the same task.

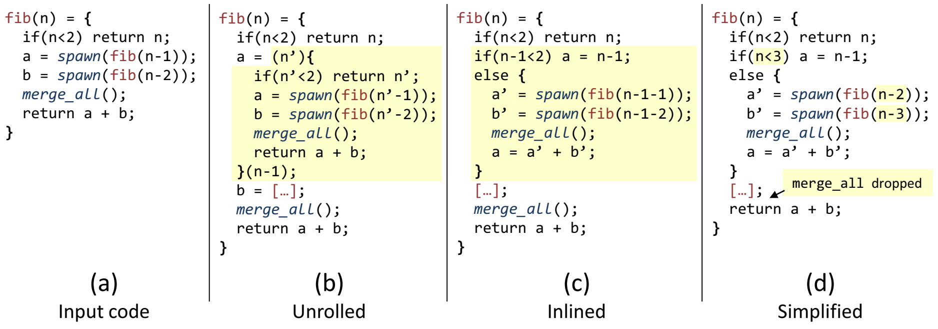
Version (1) is identical to the original program, except that at each spawning point there is now a choice between 3 distinct implementations of each function. In version (2), consisting of $F1'$ and $F2'$, each recursive task invocation was unrolled once, forming tasks of increased granularity. Clearly, if this version is used, more work is performed between individual task invocations and interactions with the runtime library. Finally, version (3), comprising $F1''$ and $F2''$, is fully sequentialized. Once this version is invoked, no further parallel tasks will be spawned on this branch of the recursive descent.

Code Example. Figure 4 illustrates the effect of the steps taken during compilation to generate a task version that has been unrolled once. A pseudo-code formulation is used for reasons of clarity and size. It is C-like, but without the need for explicit type specification, and with two additional keywords: `spawn` implies the generation of a new parallel task (corresponding to `#pragma omp task untied`), while `merge_all` waits for the completion of all launched subtasks (equivalent to `#pragma omp taskwait`).

In (a), the original input code is shown. Moving on to (b), first-level task invocations are removed and replaced with in-place calls of the associated functions. Context-sensitive inlining of these calls results in (c). Finally, redundant applications of the `merge_all` operation are removed and arithmetic simplification is applied. The final generated code for this version is listed in (*d*). This process can be repeated N times to generate increasingly larger task sizes.

After all the versions are generated, each version needs to be modified to enable runtime selection. Figure 5 contains the final code for the original version with task selection (*a*), the unrolled version as discussed previously (*b*) and a fully sequentialized version (*c*). The `pick` keyword implies a possible choice between semantically equivalent versions, which is deferred to the runtime system.

Partial Sequentialization. In most parallel programs there will be some superfluous synchronization statements after task unrolling. Since the execution has been partially sequentialized, instructions that wait for the completion of a task that was unrolled are no longer necessary and should be removed. The transformation eliminating unnecessary synchronization acts as follows on a task version T, effectively removing all `merge_all` operations for which there is no possibility of any task being spawned between them and a previous `merge_all`:



```
fib(n) = {
  if(n<2) return n;
  a = spawn(fib(n-1));
  b = spawn(fib(n-2));
  merge_all();
  return a + b;
}
```

```
fib(n) = {
  if(n<2) return n;
  a = (n'){
    if(n'<2) return n';
    a = spawn(fib(n'-1));
    b = spawn(fib(n'-2));
    merge_all();
    return a + b;
  }(n-1);
  b = […];
  merge_all();
  return a + b;
}
```

```
fib(n) = {
  if(n<2) return n;
  if(n-1<2) a = n-1;
  else {
    a' = spawn(fib(n-1-1));
    b' = spawn(fib(n-1-2));
    merge_all();
    a = a' + b';
  }
  […];
  merge_all();
  return a + b;
}
```

```
fib(n) = {
  if(n<2) return n;
  if(n<3) a = n-1;
  else {
    a' = spawn(fib(n-2));
    b' = spawn(fib(n-3));
    merge_all();
    a = a' + b';
  }
  […];
  return a + b;
}
```

Fig. 4. Example task transformation - Fibonacci - Version generation

1. Determine the set S of all `merge_all` invocations in T.
2. For each `merge_all` $M \in S$:
   (a) Compute the set of all execution paths F from the entry point of T to M.
   (b) Reverse the paths in F.
   (c) If no path in F encounters a `spawn` before reaching a `merge_all`, remove M from T.

3.2 Runtime Version Selection

The previous section outlined how multiple versions with different granularities and trade-offs are generated in the compiler. This provides the runtime system with an opportunity of making a version choice every time a task is spawned. Making the wrong choice can result in not gaining the desired increase in efficiency, or, at worst, greatly diminishing parallelism – e.g. in case a fully sequentialized version is chosen too early. We considered the following design goals and observations when developing our version selection method:

```
fib(n) = {
  if(n<2) return n;
  a = spawn( pick(
    fib(n-1),
    fib_u1(n-1),
    fib_seq(n-1) ) );
  b = spawn( pick(
    fib(n-2),
    fib_u1(n-2),
    fib_seq(n-2) ) );
  merge_all();
  return a + b;
}
```

(a)
Original

```
fib_u1(n) = {
  if(n<2) return n;
  if(n<3) a = n-1;
  else {
    a’ = spawn( pick(
      fib(n-2),
      fib_u1(n-2),
      fib_seq(n-2) ) );
    b’ = spawn(pick(…));
    merge_all();
    a = a’ + b’;
  }
  […];
  return a + b;
}
```

(b)
Unrolled Once

```
fib_seq(n) = {
  if(n<2) return n;
  if(n<3) a = n-1;
  else {
    a’ = fib_seq(n-2);
    b’ = fib_seq(n-3);
    a = a’ + b’;
  }
  […];
  return a + b;
}
```

(c)
Fully Sequentialized

Fig. 5. Example task transformation - Fibonacci - Generated versions

– At the start of the program, the original (most fine-grained) version of the tasks should be used, since the parallelism available in the system is not yet fully leveraged and load-balancing is a priority.
– The impact of conservative behavior – i.e. using more fine-grained tasks – causes more gradual performance degradation than using tasks that are too coarse grained, potentially leading to some worker threads idling.
– The decision procedure needs to be simple and not introduce large overheads on its own, otherwise it could negate any benefits from multiversioning.
– The decision making process should be distributed – no new synchronization points between worker threads should be introduced to facilitate version selection.

Taking these points into account led to the development of a distributed version selection heuristic based on two parameters that are tracked for each individual worker thread. The first is *task demand*, which keeps track of other worker's unfulfilled attempts to steal tasks from the local worker. The second parameter is the *queue length* of each worker, or how many tasks it currently has available to be executed or stolen.

Task demand is tracked in a surprisingly simple, but effective, manner. The demand is stored as an integer which starts at a positive value equal to the maximum task queue length. Whenever a task is generated by a worker thread, it reduces its own task demand by 1. When a worker attempts to steal from another which has no tasks available, that target worker's demand value is reset to the starting value.

Our version selection heuristic is described in Figure 6. In conjunction with the demand tracking outlined above, it has the following desirable properties:

– Evaluating the selection function only takes a few dozen cycles, assuming that all the required values are cached.
– The way in which task demand is completely reset if any stealing operation fails, but is only reduced gradually during normal execution, mirrors the earlier observation about the negative performance impact of wrong granularity selection. It makes the expensive case of idle workers unlikely by reacting very strongly to failed stealing attempts.

`queue_length`	current queue length
`task_demand`	current task demand
`num_versions`	number of versions generated for current task
`MAX_QUEUE`	maximum queue length (fixed)

output: 0	$\Leftrightarrow$ original task
$N = 1 \dots$ `num_versions` $- 2$	$\Leftrightarrow$ unrolled N times
`num_versions` $- 1$	$\Leftrightarrow$ fully sequentialized

```
version = num_versions − ⌈(task_demand/MAX_QUEUE) ∗ num_versions⌉
if version >= num_versions − 1 then
    if queue_length == MAX_QUEUE then
        return num_versions − 1
    end if
    return num_versions − 2
end if
return version
```

Fig. 6. Version Selection Algorithm

– Selecting the fully sequentialized version is a step that should only be taken after careful consideration, since it will prevent any further parallelism from being generated on this branch of the recursive descent. Therefore, the heuristic only takes this step if there has been no demand for additional tasks over a large number of spawn points *and* the queue is full.

The choice of the `MAX_QUEUE` parameter has an impact on the effectiveness of this approach. Experimental evaluation has shown that generally, a longer queue is beneficial on systems with a larger number of cores. For the evaluation in Section 4, `MAX_QUEUE` was set to 32.

4 Evaluation

In this section we will evaluate the performance impact of our optimization on multiple benchmark programs. Subsection 4.1 details our measurement methodology and the experimental setup used. We will perform an in-depth evaluation of one program in Subsection 4.2, and then proceed with an overview of the results of a number of other codes in order to provide a balanced overall impression.

4.1 Experimental Setup

For our experiments we used an Intel-based parallel system, incorporating 4 Xeon E7-4870 processors, each comprising 10 physical cores (20 hardware threads) and 3 levels of cache. Table 1 summarizes the configuration of this system.

Table 1. Hardware and software platform for experimental evaluation

Sockets/	Cache			Software				
Cores	L1d/i	L2	L3	OS	Kernel	GCC	ICC	Insieme
4/40	32K/32K	256K	30M	CentOS 6.3	2.6.32	4.6.3	12.1	g4614502

When running experiments using a subset of cores, all involved threads were bound to individual physical cores such that the resources of one chip are fully utilized before involving an additional processor. All experimental runs were repeated five times, and the median runtime is reported.

While the most important comparison for our evaluation is between our compiler with and without our multiversioning method, we also included the results obtained by other platforms to provide a reference for comparison. Table 1 includes the exact version number of the compilers used in these comparisons. ICC was used as the backend compiler for the Insieme source to source infrastructure, and its built-in Cilk Plus support was employed to compile Cilk programs. The optimization flag "-O3" was enabled for all calls to GCC and ICC.

4.2 A Detailed Evaluation

The first program we will evaluate is the N-Queens benchmark included in BOTS [7]. Each task in N-Queens spawns 0 to N child tasks, and the depth of its task invocation trees varies from 1 to N, while not following any simple pattern. The size of individual tasks is relatively small.

Figure 7 shows the performance of N-Queens using a variety of compilers and implementations. Four OpenMP versions are shown: GCC, ICC and Insieme with and without task optimization. Additionally, we included the results of a Cilk version and a fully sequential version without any parallel language primitives. The execution time is presented in a log-log plot to improve readability. The

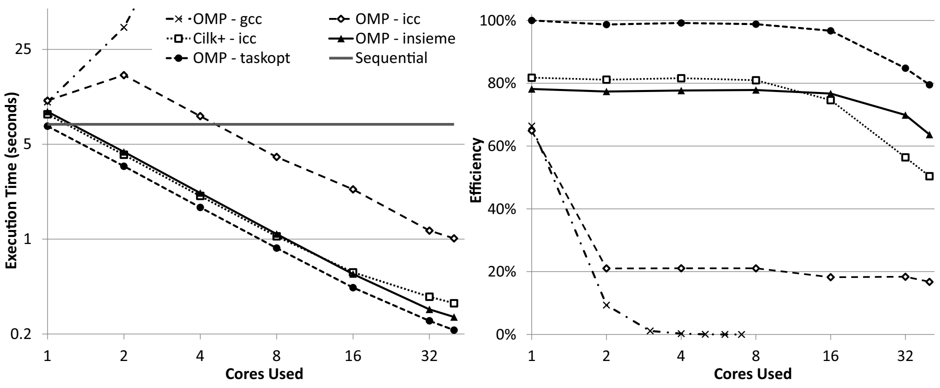


Fig. 7. N-Queens benchmark results, $N = 13$

efficiency plot compares the execution times of the parallel versions against ideal scaling from the sequential version.

In terms of OpenMP results, it is clear that the task granularity in this benchmark is too small to be handled effectively by GCC's GOMP implementation. ICC shows the same behavior that was already partially observed in Section 2 – execution time increases when going from a single-threaded to a multi-threaded setup. However, starting from two threads performance scales relatively well up to 40. Since both of these OpenMP implementations seem ill-equipped to handle very fine-grained tasking well, we also included a Cilk version, which has previously been shown to provide better scaling for fine-grained tasks [15]. Indeed, this implementation performs better in the single-threaded case and scales more smoothly to multiple cores than the GCC and ICC OpenMP versions.

Using Insieme to compile the OpenMP input program results in performance that is comparable to Cilk for up to 16 cores, and scales slightly better beyond this amount. However, a comparison with the fully sequential version indicates that even the Insieme OpenMP version and the Cilk version lose around 20% of performance to overheads incurred due to parallelization. When our task optimization is activated, this overhead is effectively avoided. Even more importantly, this significant reduction in overhead is achieved without negatively affecting the scalability of the program. Performance compared to our implementation without task optimization is improved by 22% to 28% across all measured core counts, with a 25% increase at the full 40 cores.

Compared to the fully sequential version, our approach achieves an efficiency above 99% up to 8 cores, 97% at 16 cores, 85% with 32 cores and 80% at 40 cores. Using the full system (40 cores), our implementation with task optimization improves N-Queens performance by 56% compared to the best competing implementation (Cilk).

4.3 Further Benchmarks

Table 2 summarizes our benchmark results. It includes measurements for the N-Queens benchmark presented above, as well as a number of additional programs.

Sort. Is the *sort* benchmark included in BOTS.

Strassen. Also from BOTS, matrix multiplication using the Strassen algorithm.

Fib. The BOTS *fibonacci* benchmark, remarkable for its very small task size.

Stencil. A task based 2D stencil computation using the cache-oblivious algorithm presented by Frigo and Strumpen [10]. We included this benchmark to represent an important category of cache-oblivious divide-and-conquer algorithms.

Floorplan. The BOTS *floorplan* benchmark. For this application, the binary generated by ICC 12.1 repeatably caused a segmentation fault within ICC's OpenMP library, regardless of the number of threads used. Therefore we are unable to present ICC results for this benchmark.

FFT. A parallel fast fourier transform included in BOTS.

QAP. A branch and bound solver for quadratic assignment problems.

Table 2. Benchmark Results

cores	1	2	5	10	20	40	cores	1	2	5	10	20	40
Queens, $N = 13$ - seq: 7.42							**Fib,** $N = 48$ - seq: 31.09						
gcc	10.23	36.29	148.28	308.16	545.22	725.98	gcc	1960.35	17093.63	>15000	>15000	>15000	>15000
icc	10.49	16.04	6.45	3.81	1.60	0.91	icc	1379.84	2705.65	1135.29	569.15	286.41	157.70
ins	8.69	4.35	1.74	0.87	0.46	0.27	ins	742.40	456.95	247.91	196.59	169.50	155.29
opt	6.79	3.41	1.48	0.69	0.36	0.21	opt	27.06	13.77	6.37	3.30	1.93	1.03
imp	27.92%	27.52%	17.78%	26.64%	25.35%	24.91%	imp	26.43×	32.17×	37.90×	58.15×	86.69×	150.36×
Sort, $N = 2^{27}$ - seq: 21.51							**Strassen,** $N = 8192$ - seq: 158.15						
gcc	21.98	11.80	7.20	17.17	29.43	42.29	gcc	159.74	92.45	39.20	22.10	15.36	19.94
icc	23.87	12.36	5.04	2.80	1.85	1.56	icc	164.43	89.94	39.12	21.81	15.69	19.27
ins	22.94	12.00	4.90	2.71	1.93	1.53	ins	168.84	85.97	37.51	21.98	12.94	8.72
opt	20.81	11.18	4.61	2.52	1.72	1.41	opt	154.27	79.80	35.46	19.81	12.03	8.11
imp	5.61%	5.47%	6.43%	7.47%	7.88%	8.11%	imp	3.54%	7.72%	5.77%	10.08%	7.55%	7.52%
Stencil, $N = 2048$ - seq: 18.90							**Floorplan,** `input.20` - seq: 17.86						
gcc	46.82	62.09	138.51	398.05	576.83	840.61	gcc	27.36	31.04	133.30	352.94	514.51	759.20
icc	30.17	24.65	15.63	14.64	13.84	12.04	icc	*	*	*	*	*	*
ins	32.49	18.48	9.27	6.31	7.50	9.67	ins	23.53	12.48	5.05	2.53	1.72	1.58
opt	24.96	13.84	6.66	4.26	5.15	7.54	opt	17.20	9.51	4.12	2.09	1.43	1.24
imp	20.87%	33.49%	39.17%	47.97%	45.50%	28.29%	imp	36.76%	31.25%	22.62%	21.06%	20.52%	27.68%
FFT, $N = 2^{29}$ - seq: 184.78							**QAP,** `chr18a` - seq: 237.28						
gcc	222.27	132.66	95.88	276.81	420.00	482.07	gcc	488.97	931.43	7471.11	>15000	>15000	>15000
icc	189.73	112.13	55.95	37.44	22.64	16.03	icc	785.36	2539.80	823.00	319.87	179.58	114.93
ins	187.36	104.85	51.39	36.46	21.01	16.96	ins	578.57	294.13	112.80	78.65	70.97	60.71
opt	183.97	100.02	49.66	35.08	19.07	12.03	opt	231.62	110.76	40.24	21.88	15.18	9.90
imp	1.84%	4.84%	3.48%	3.93%	10.16%	33.21%	imp	2.11×	2.66×	2.80×	3.59×	4.68×	6.13×

For every benchmark, the table contains five rows. The results achieved using the GCC and ICC OpenMP implementations are listed in the "gcc" and "icc" rows, respectively. The "ins" row contains the results of our Insieme compiler and runtime without the task multiversioning optimization presented in this paper, while it is enabled for the measurements listed in the "opt" row. Finally, the values in the "imp" row represent the relative improvement achieved using adaptive granularity control, compared to the best result among the other three versions. The columns labeled 1 to 40 correspond to the number of cores used for the computation. All times are given in seconds, and the improvement is provided in percent, except in the case of the Fibonacci and QAP benchmarks where improvement factors are listed instead of very large percentages.

As a frame of reference, the purely sequential time for each benchmark compiled with ICC is provided in each header ("seq"). Note that this time falls between the Insieme time without optimization and the optimized version in most cases, except in the stencil test. Here, the restructuring performed by our compiler prevents some of the low-level sequential optimizations performed by ICC. However, our optimized version executed with one thread is still closer to the sequential performance than any other implementation.

A general trend visible throughout all the benchmark results is the relationship between default task granularity, scaling in GCC and the degree of improvement possible using adaptive task multiversioning and selection. The fibonacci and QAP benchmarks have the most fine grained tasks, and consequently the worst scaling in GCC and the largest improvement with our optimization. On the other end of the spectrum, the FFT, strassen and sort benchmarks feature built-in cutoff values that inherently control task granularity by preventing very small tasks from being generated, resulting in more modest, but still significant, performance improvements with multiversioning. Floorplan, stencil and N-queens fall in between these extremes.

One interesting behavioral pattern which merits some explanation occurs in FFT. Our multiversioning implementation does not result in any significant improvement up to 10 cores, however at 40 cores the measured improvement is 33%. This is due to the FFT benchmark consisting of two separate phases: coefficient calculation and FFT computation. These phases exhibit distinct scaling behaviour, and one of them is affected more significantly by adaptive granularity optimization than the other. Thus, with a larger number of cores, the phase with bad scaling starts to take up a larger portion of the execution time, and the effect of multiversioning on overall performance increases.

5 Related Work

Much previous work on parallel tasks has focused on runtime systems [3] or scheduling policies [16]. As described in section 2, pure runtime modifications are incapable of dealing with all the causes for inefficiency that our combined compiler and runtime approach covers. Moreover, our proposed multiversioning scheme is orthogonal to scheduling decisions and can be combined with any scheduling policy.

A common approach towards dealing with task granularity issues is having the user provide thresholds or cut-off values [8]. In our work, task granularity is controlled entirely by the compiler and runtime system, without requiring manual programmer support. Duran et al. [6] describe an adaptive cut-off method which does not require manual adjustment, but their pure runtime approach does not offer the performance benefit of full sequentialization in the compiler.

Inlining of recursive functions has been previously performed in sequential program transformation [9], even with the express purpose of improving performance in divide and conquer programs by reducing overheads [18]. However, these works do not deal with parallelism, while our approach focuses primarily on minimizing the overhead incurred by parallel task creation and synchronization.

Some recent publications have used compiler multiversioning in a parallel setting [4][13], but they focused exclusively on loop-based data parallelism. Conversely, our multiversioning approach is designed for task-parallel, recursive programs.

Very recently, Deshpande and Edwards used recursion unrolling to improve opportunities for parallelism in Haskell programs [5]. Unlike our method, they do not use multiversioning or version selection at runtime, and their compiler transformations are designed for the Haskell functional language while we process input programs written in C with OpenMP.

6 Conclusion

We have presented a fully automatic, adaptive approach to parallel task granularity control which goes beyond what can be achieved by improving either just a runtime system or focusing only on compilation. By combining a compiler which performs task multiversioning with a runtime system that adaptively selects from these versions, we were able to minimize parallel runtime overhead even for very fine grained tasks. Our method uses a novel combination of compiler transformations to build an optimized set of semantically equivalent task versions which differ in granularity. The availability of this set of implementations in the compiled program in turn enables our runtime heuristic to adjust the amount of tasks generated, while incurring even less overhead than a traditional lazy task creation system with cut-offs.

Evaluating our proposed method across a set of eight benchmarks has shown that our optimization is widely applicable, and that the magnitude of these improvements is related to the task granularity of the input program. For programs with relatively coarse-grained tasks, execution times are reduced by 5% - 10%, while we can achieve improvements of a factor of 6 or more compared to the best competing implementations in fine-grained test cases. Benchmark results also demonstrate that our runtime selection heuristic successfully ensures that scalability (up to 40 cores) is not negatively affected by adaptive task granularity adjustment. Crucially, our adaptive granularity control scheme improves performance in all tested benchmarks and for any given number of cores.

Acknowledgments. This research was partially funded by the Austrian Research Promotion Agency under contract nr. 834307 (AutoCore) and the chist-era project GEMSCLAIM. We would also like to thank the reviewers for their insightful comments.

References

[1] Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley. Tech. rep. EECS Department, University of California (2006)

[2] Blumofe, R.D., et al.: Cilk: an efficient multithreaded runtime system. In: Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPOPP 1995, Santa Barbara, California, United States, pp. 207–216 (1995)

[3] Broquedis, F., Gautier, T., Danjean, V.: libKOMP, an efficient openMP runtime system for both fork-join and data flow paradigms. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 102–115. Springer, Heidelberg (2012)

[4] Chen, X., Long, S.: Adaptive Multi-versioning for OpenMP Parallelization via Machine Learning. In: Proc. 15th Int. Conf. on Parallel and Distributed Systems, ICPADS 2009, pp. 907–912 (2009)

[5] Deshpande, N.A., Edwards, S.A.: Statically Unrolling Recursion to Improve Opportunities for Parallelism. Tech. rep. Department of Computer Science, Columbia University (2012)

[6] Duran, A., et al.: An adaptive cut-off for task parallelism. In: Proc. 2008 ACM/IEEE Conf. on Supercomputing, SC 2008, Austin, Texas, pp. 36:1–36:11 (2008)

[7] Duran, A., et al.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: Proc. 2009 Int. Conf. on Parallel Processing, ICPP 2009, pp. 124–131 (2009)

[8] Turner, D.N. (ed.), et al.: On the Granularity of Divide-and-Conquer Parallelism. In: Glasgow Workshop on Functional Programming, pp. 8–10. Springer (1995)
[9] Fitzpatrick, S., et al.: Unfolding Recursive Function Definitions Using the Paradoxical Combinator (1996)
[10] Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: Proc. 19th Int. Conf. on Supercomputing, ICS 2005, Cambridge, Massachusetts, pp. 361–366 (2005)
[11] Insieme Compiler and Runtime Infrastructure. Distributed and Parallel Systems Group, University of Innsbruck, `http://insieme-compiler.org`
[12] Intel. Intel C and C++ Compilers (2012), `http://software.intel.com/en-us/c-compilers/`
[13] Jordan, H., et al.: A Multi-Objective Auto-Tuning Framework for Parallel Codes. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, Article No. 10. IEEE Computer Society Press, Los Alamitos (2012)
[14] Mohr, E., et al.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. IEEE Transactions on Parallel and Distributed Systems 2 (1991)
[15] Olivier, S., Prins, J.F.: Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. International Journal of Parallel Programming 38(5-6), 341–360 (2010)
[16] Olivier, S.L., et al.: OpenMP task scheduling strategies for multicore NUMA systems. Int. J. High Perform. Comput. Appl. 26(2), 110–124 (2012)
[17] OpenMP Architecture Review Board. OpenMP Specification. Version 3.1 (2011), `http://www.openmp.org/mp-documents`
[18] Rugina, R., Rinard, M.: Recursion Unrolling for Divide and Conquer Programs. In: Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J.F., Pugh, B., Tseng, C.-W. (eds.) LCPC 2000. LNCS, vol. 2017, pp. 34–48. Springer, Heidelberg (2001)
[19] Stallman, R.: Using and Porting the GNU Compiler Collection. In: M.I.T. Artificial Intelligence Laboratory (2001)

Towards Efficient Dynamic LLC Home Bank Mapping with NoC-Level Support

Mario Lodde[1], José Flich[1], and Manuel E. Acacio[2]

[1] Universitat Politècnica de València, Spain
[2] Universidad de Murcia, Spain

Abstract. In tiled Chip Multiprocessors (CMPs) the banks of the built-in last level cache (LLC) are usually distributed among the tiles and logically shared. A static mapping of cache blocks to the LLC banks leads to poor efficiency since a block can be mapped to a bank far away from the tiles which actually access it. Partially dynamic policies have been proposed, which however rely on the static mapping of blocks to a set of banks (D-NUCA) or rely on the OS to dynamically load pages to statically mapped addresses (first-touch).

We propose a new dynamic approach where the LLC home bank is determined at runtime in hardware, with the memory controller in charge to perform the block mapping when fetched from main memory. To speed up the home bank lookup process, we use simple and lightweight NoC optimizations. When compared with alternative solutions (S-NUCA, D-NUCA, first touch, private LLCs) results with PARSEC and SPLASH-2 applications indicate improvement in locality of LLC blocks in the same tile (56.2% from 5.8%) and more than 33% reduction in load and store miss latencies. This leads to an average reduction of 24% in application's execution time compared to static mapping.

1 Introduction

Chip multiprocessor systems (CMPs) usually employ a shared memory programming model, thus requiring a cache coherence protocol to keep data consistency along the cache hierarchy. The on-chip cache is organized hierarchically, with small low-latency caches at the highest level and larger caches with higher access times at the lower levels. This provides high on-chip storage capacity without the high access latency a single, large cache would have. Without losing generality, in this work we assume the tiled CMP system shown in Figure 1 with a two-level cache. Each tile includes a core, separate L1 caches for instructions and data, a bank of L2 cache and a switch to connect the tiles through a 2D mesh.

L1 caches are private to the core in the tile. For the L2 cache, different policies can be implemented, but the common choices are two. The first is to use each L2 cache bank as a private cache to the tile, extending its private cache capacity. This is the best option if the working set of the application fits in the L2 cache bank, since all cached data can be accessed without sending requests over the NoC. If the working set does not fit, this policy generates many L2 cache line replacements, and therefore, off-chip requests. Furthermore, shared blocks are replicated in different L2 cache banks. The second option is to consider the L2 banks as a shared but distributed L2 cache. Data replication is avoided and

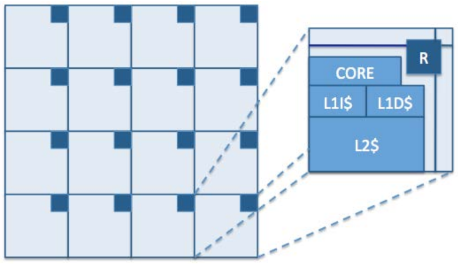


Fig. 1. Tiled CMP system

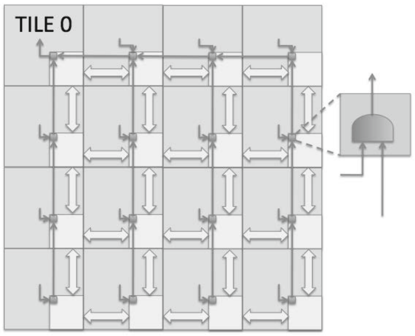


Fig. 2. Gather control network for tile 0

cache resources are used efficiently, but the latency of retrieving a cached data in case of L1 miss will be higher and variable depending on the location of the L1 cache and the L2 bank. Thus, the mapping of blocks to L2 banks is a crucial design parameter for this approach. In this paper we follow this policy.

The L2 bank that hosts a block is called the *home* bank. There are two main design options when deciding which L2 bank is the *home* for a block. On the one hand, block mapping can be done statically (S-NUCA): the address space is divided in subsets and all the blocks of a subset are statically mapped to a bank. This policy is very simple to implement but can be inefficient as blocks may be mapped to banks which are far away from L1 requestors. The second option is to perform the mapping dynamically (D-NUCA) [11], where each subset of blocks is mapped to a group of banks, or bank set, and blocks can migrate within a bank set to move as close as possible to the requestor's tile. This policy has lower miss latencies but is more complex to implement. Furthermore, the process of finding a block within a bank set leads to a tradeoff between access time and NoC traffic since all the banks of a bank set must be accessed, leading to either high latency (sequential search) or more traffic (parallel search).

In this work we propose Runtime Home Mapping (RHM), a new dynamic approach where the LLC home bank is determined at runtime in hardware by the memory controller. While in D-NUCA the mapping is partially static, in RHM a block can be mapped to any L2 cache bank, enabling future optimization strategies such as virtualization, where the *home* banks can be bounded to the region of the chip running a given application, and increased efficiency of thread migration, where migrated threads can attract data to their closest L2 cache banks. Since the *home* bank is not known a priori, a search must be performed each time an L1 miss occurs. Conversely to previous approaches to home location, based on the use of limited-size tables, and therefore prone to costly overflows [1], we combine three different NoC mechanisms to optimize the search phase:

– An efficient *home* search method where a broadcast message is triggered to query the home. Then, a lightweight and simple dedicated control network is used to collect acknowledgments generated in the discovering process.
– Parallel access of L2 tags. We highly couple tag array of L2s into the current NoC router design. At the same time the broadcast message enters the router, the tag array is accessed.

- A router mechanism aimed to reduce the broadcast messaging. On an L2 tag hit, the broadcast message is removed from the router before leaving it, thus reducing traffic by chopping broadcast branches.

In addition, we propose a migration strategy similar to the one of D-NUCA caches but without the bank set constraint: a block can migrate to any bank. Results show RHM is effective in placing the block near the core using it, reducing the average number of hops per request by 60% on average compared to static mapping, which leads to reductions of more than 30% in terms of cache miss latency and 27% in execution time. NoC and LLC energy consumption is also reduced by 40% and 23% on average, respectively.

The rest of the paper is organized as follows: in Section 2 we show RHM and its NoC-level support. In Section 3 we present the router with support for the parallel L2 tag access. In Section 4 we show the evaluation results. In Section 5 we describe related work. In Section 6 we describe future work and conclusions.

2 Runtime Home Mapping with NoC-Level Support

RHM aims to map blocks to L2 banks at runtime, in order to allocate them as close as possible to the requesting cores, possibly in the L2 bank of the local tile. The mapping is performed by the memory controller each time it receives a request. Thus, if a block is removed from the chip, it can be mapped to a different L2 bank the next time depending on the requestor and L2 cache availability.

In case of an L1 miss, a request is sent to the local L2 bank in the same tile. On a miss, a broadcast is sent to all other L2 banks. When a bank receives this broadcast request, it checks its tag array. In case of a hit, it sends the data back to the requestor. In case of a miss, an acknowledgement (ACK) is sent to the L2 which issued the broadcast. If all the L2 banks send an ACK to that L2 bank, it means the block is not cached on chip, so the bank sends a request to the memory controller (MC), which in turn fetches the block from main memory. The MC keeps track of the utilization of each L2 bank, so while waiting for the data it decides which L2 bank will be its *home* depending on the requestor location, utilization statistics and the mapping policy (explained in Section 2.2).

Once a bank is chosen as the *home* for a particular block, the MC notifies the bank so it can start replacing a cache line, if needed, and allocating a line to the incoming block while the MC is still waiting for the block. When the block is received at the MC, it is sent to the chosen *home* bank, which in turn will send the block to the requestor L1 cache.

The L2 home search policy just described above has high network resources demand: every time a request misses in the local L2 bank, a broadcast is issued. Also, all other banks must answer to the broadcast with an ACK or with the data. The first can be attenuated by implementing a tree-based broadcast mechanism within the NoC: a broadcast is sent injecting a single message, that replicates at switches to reach every L2 bank. This reduces NoC traffic and eliminates the serialization of multiple copies of the same request (one per destination). ACKs however still represent a big problem: they are indeed sent roughly at the same time and will probably serialize in the network and, most important, all of them must reach the same L2 cache bank (i.e., the one that initiated the broadcast).

2.1 Gather Control Network

To solve the problems introduced by ACK messages our proposal uses a simple and fast dedicated control network.[1] This network, called Gather Control Network (GCN), can be logically seen as 16 one-bit wide subnetworks, one per tile. Each subnetwork is a tree of AND gates, connecting the destination tile (the root) with all other tiles (located at the leaves of the AND tree).

Figure 2 shows the subnetwork with root in tile 0. A one-bit subnetwork (darker arrows) is added to the regular NoC (bidirectional arrows). If a request misses in the L1 and L2 caches of tile 0 (L1-0 and L2-0 from now on), L2-0 broadcasts the request to all other L2 banks through the regular NoC. When an L2 bank receives this request, it triggers the output signal of the GCN for tile 0. Once all L2 banks have triggered their output signals, the output of the AND tree will notify the L2-0 and thus acts as a global ACK. Sending ACKs through the GCN highly reduces NoC traffic and power consumption and the ACKs transmission latency. Indeed, the GCN does not require routing, flow control nor arbitration at each hop and eliminates message serialization at destination.

The GCN we assume in this work has the same logic behavior of the AND trees but is implemented with sequential logic, which reduces wiring requirements. An extensive discussion on the sequential GCN is published in [4].

2.2 Mapping Algorithm

Each time the MC receives a request, a mapping algorithm chooses the *home* bank for the requested block, in parallel with the access to main memory, depending on the requestor's tile and current L2 banks utilization. The MC takes statistics about cache utilization, which are stored in a table (*alloc* table) with $N \times M$ entries, where N is the number of L2 cache banks and M the number of L2 sets. Each entry contains the number of allocations performed in set m of L2 bank n. If the associativity of L2 sets is Z, the table has to store at minimum $N \times M \times \log_2 Z$ bits. However, the table will double the bits in each entry. For a 4×4 tile system with 16-way 256KB bank sets, the minimum memory requirements for this single table is 2KB ($m = 16$, $M = 256$, $Z = 16$). With the increased size, the table will grow to 4KB (to allow 256 allocations per set). The following pseudocode describes the simple algorithm we implemented:

```
int function allocate(int r, address a) {
 banklist n; bank b; set s; bank h;

 s = get_set(a);
 if(alloc[r,s]<num_ways) {alloc[r, s]++; return r;}

 for(int h = 1; h <= MaxHops; h++){
   n = BanksReachable(r, h);
   for (int i = 0; i < size(n); i++){
     b = SelectBankClockWise(n, i);
     if (alloc[b,s]<num_ways) {alloc[b,s]++; return b;}
   }
```

[1] This control network is similar to the one proposed in previous works [2],[3] to collect ACKs generated by Hammer and Directory coherence protocols.

```
 }

 for(int h = 1; h <= MaxHops; h++){
   n = BanksReachable(r, h);
   for (int i = 0; i < size(n); i++){
     b = SelectBankClockWise(n, i);
     if (alloc[r,s] - alloc[b,s] > UtilThr) {alloc[b,s]++; return b;}
   }
 }

 alloc[r, s]++; return r;
}
```

If there is room in the set in the local L2 bank (*r* tile), then the *home* is the local tile of the requestor. Otherwise, the algorithm scans the neighbor banks in distance order (first *for* loop) to avoid triggering a replacement if the current line can be allocated to a distant bank. This search is performed until the threshold $MaxHops$ is reached, which can be equal to the physical threshold forced by the system size (number of hops from the requestor to the furthest tile) or lower.

If all the L2 banks are full (*alloc* higher than *num_ways*), the algorithm tries to balance the number of allocations (thus, replacements) in all banks (second *for* loop). A threshold (*UtilThr*) is used. If the difference between the number of allocations in the local tile's bank and a neighbor bank is higher than the threshold, then the neighbor bank is selected as the *home* bank.

If all the banks are balanced, then the block is mapped to the requestor's tile. Notice that this does not imply that RHM defaults to private L2 caches. With private caches all the data accessed by a core must be present in the L2 bank of the same tile, while in RHM this does not apply. For instance, a shared block will be replicated in all L2 caches if they are private, while in RHM it will be present only in the *home* tile. The proposed policy defaults to private caches only if all L2 banks are full, each core is requesting private blocks and all banks are uniformly used. In this case, very unlikely in a parallel application, all blocks are allocated in the requestor tile, which indeed is the best choice since it minimizes the data access latency.

2.3 Block Migration

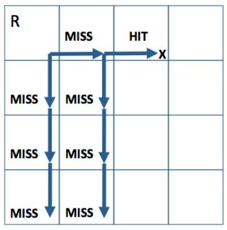
If the initial *home* allocation performed by the MC results sub-optimal, block migration can be enabled to further reduce the number of hops between an L1 cache and the L2 bank. Notice that a sort of migration mechanism is implicit in RHM, since each time a block is replaced from an L2 bank and then requested again it may be mapped to another L2 bank, but this may not always be effective.

We propose a migration scheme similar to the one used in D-NUCA but without the constraint of being limited within a bank set: in RHM a block is allowed to migrate to any L2 bank. However, since the migration process introduces an overhead in terms of traffic and energy, it should be performed only if it actually leads to a benefit in terms of miss latency reduction.

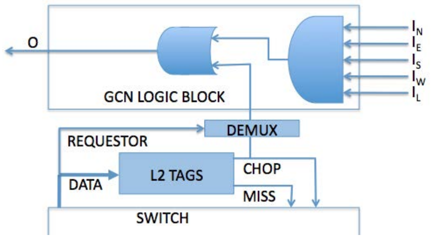
Solutions in D-NUCA reduce unnecessary migrations and avoid the ping-pong effect by using a saturating counter for each direction to which a block can move. A counter is updated each time a request comes from a node located in the counter's direction; when the counter saturates, the migration process towards that direction is triggered. In our case a block may migrate in any direction, so four counters are needed, one per direction. Each time a request hits in an L2 block, the counters associated to the requested cache line are updated adding the distance in hops from the requestor. When a counter is incremented, the one in the opposite direction is decremented. When a counter saturates, it starts the migration process: the block migrates to the L2 bank located in the same tile of the L1 which sent the request that triggered the migration. Counters are reset after a migration and when a request is received from the local core.

3 Parallel Tag Access

Built-in NoC broadcast support and the GCN highly reduce the NoC traffic generated during the *home* search phase. However, traffic can still be reduced by eliminating useless broadcast branches. In Figure 3.a a request misses in the local L2 bank and is broadcast to all other banks. Since the data is found in L2-2, there is no need to propagate the broadcast through east and south directions. We propose a mechanism to allow parallel access to the L2 tags while a broadcast message is crossing the NoC router pipeline. This way, in case of an L2 hit, the broadcast branch can be cropped before the message reaches the crossbar stage of the switch. Figure 4.a shows a basic 4-stage router modified to enable Parallel Tag Access (PTA). As the flit exits the input buffer to enter the routing stage, it is also sent to the L2 tag array to perform the lookup. In case of a hit, the $CHOP$ signal dismisses the flit (the flit is converted to a bubble). In case of a miss, $MISS$ signal allows the flit to cross and replicate. Those signals are also used to generate a global ACK signal in the GCN module of the tile where the broadcast is cropped, as shown if Figure 3.b.



(a) Example of broadcast branch cropping.



(b) Modification of the GCN to support broadcast branch cropping

Fig. 3. Parallel Tag Access: motivation and implementation

Theoretically, the L2 cache must be able to trigger one of the signals within 2 cycles, while the flit is crossing the routing and the VA/SA stages. To limit power consumption and allow a fast cache lookup, a sequential access to the L2 bank must be implemented: the tag array is accessed first and then, in case of hit, the data array is accessed. If the tag access latency is still higher than two cycles, the L2 bank can be partitioned in sub banks until the size of the tag array allows a 2-cycles lookup. Alternatively, the L2 bank may require more cycles than the flit to cross the router (case of a shorter pipeline design as shown in Figure 4.b). In this case, the flit gets blocked at the VA/SA stage until the L2 tag access is performed, then the flit either advances through the crossbar (in case of a miss) or it is dismissed (in case of a hit). Notice that only broadcast request messages, which are single-flit messages, have to wait for the L2 bank to access the tag array. Also, with the obtained high locality of data in the local L2 bank (seen in the evaluation), the effect of this delay is negligible.

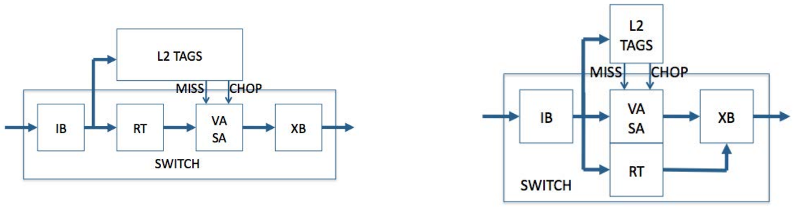


Fig. 4. 4-stages (left) and 3-stages (right) switches modified to allow parallel tag access

We implemented the basic 4-stage switch using the 45nm technology Nangate [5] library with Synopsys DC. The modifications needed to couple the switch and the L2 tags in order to allow PTA resulted in a negligible area overhead.

4 Evaluation

We evaluate RHM and compare it with other proposed NUCA configurations. In the baseline (S-NUCA) blocks are statically mapped to L2 banks using the less significant bits of the block address. In D-NUCA, blocks are statically mapped to a bank-set depending on their addresses. The matrix of L2 banks is divided in four bank-sets, one per column of tiles. Blocks are inserted in the L2 bank located in the same row of the requestor and then can migrate within the bank-set, one hop each time a migration is triggered. A third configuration uses private LLCs. Finally, we consider an S-NUCA configuration in which the blocks are mapped to the L2 banks using a first touch policy [6]. These configurations are compared to RHM, with and without block migration.

Table 1. Network and cache parameters

Routing	XY	Coherence protocol	Directory (MESI)
Flow control	credits	L1 cache size	16 + 16 kB (I + D)
Flit size	8 byte	L1 tag access latency	1 cycle
Switch model	4-stage pipelined	L1 data access latency	2 cycles
Switching	virtual cut-through	L2 bank size	256 kB
Buffer size:	9 flit deep	L2 tag access latency	1 cycle
Virtual channels:	4	L2 data access latency	4 cycles
GCN delay	2 cycles	Cache block size	64 B

The cache coherence protocol for each configuration, the NoC with broadcast support and the GCN have been implemented and simulated using our flit-level cycle-accurate network and cache hierarchy simulator. We embedded it in

Graphite[7], capturing the memory accesses of Graphite's simulated cores and using our tool for cache hierarchy and NoC timing. Different applications of the SPLASH-2 and PARSEC benchmark suites have been run on the 16-core system considered throughout this paper. Network and cache parameters are shown in Table 1. Cache latencies have been obtained using Cacti [8]. One memory controller is placed at the top left corner of the chip. For the sake of fairness, we have used also the broadcast support and the GCN in the D-NUCA approach.

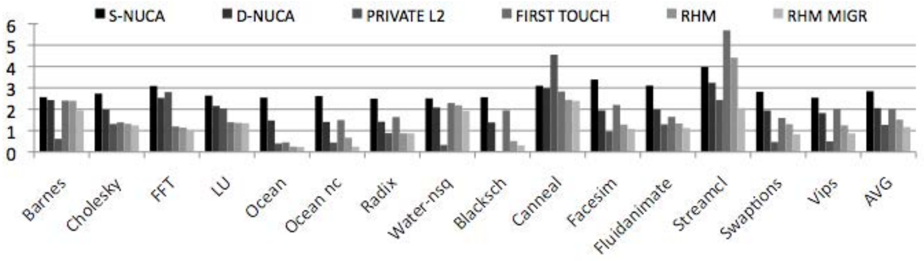


Fig. 5. Avg hop distance between L1 requestors and the tile where the data is found

Figure 5 shows the average hop distance from the requestor to the *home* tile. For S-NUCA, the block is found on average at a distance of 2.85 hops. This distance is roughly the same for most applications as blocks are uniformly distributed among the L2 banks. With other configurations, however, since blocks are dynamically mapped and/or moved from a bank to another, the distance is quite variable depending on the application. For Barnes, dynamic techniques are not so effective, and the average value is always higher than 2 hops, while for other applications, e.g. Ocean, those techniques achieve a large reduction in the average number of hops. On average, RHM locates the data closer to the requestor than the other configurations, and this distance is further reduced if block migration is enabled. Indeed RHM MIGR achieves a locality close to that of PRIVATE L2.

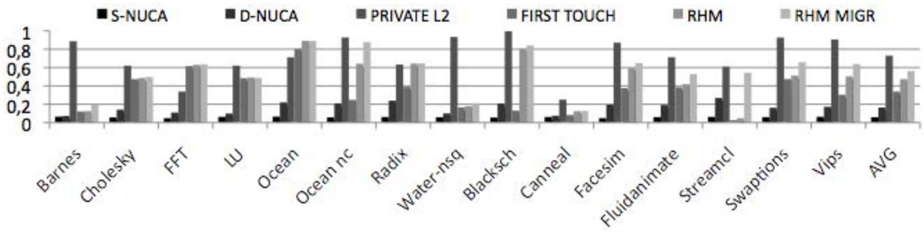


Fig. 6. Percentage of hits in the L2 bank located in the tile's requestor

Figure 6 shows the percentage of requests which hit in the L2 bank located in the same tile of the requestor over the total number of L2 hits. Again, results when using S-NUCA do not depend on the application due to the uniform mapping of the blocks, and this percentage is quite low (6% on average). This percentage increased to 16% for D-NUCA, but is still much lower when compared to First Touch (33%), RHM (49%), RHM with block migration (56%) and Private L2 (72%). Thus, the most effective dynamic method is RHM.

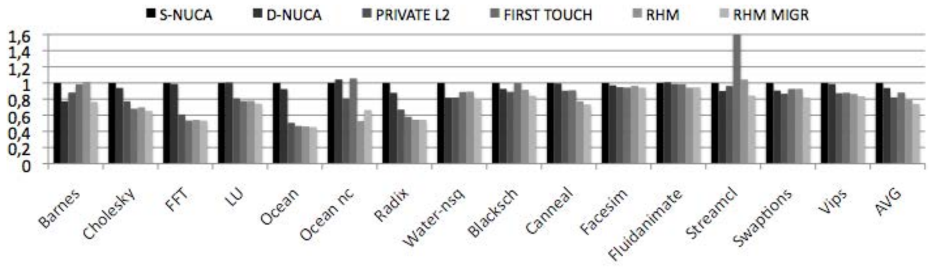
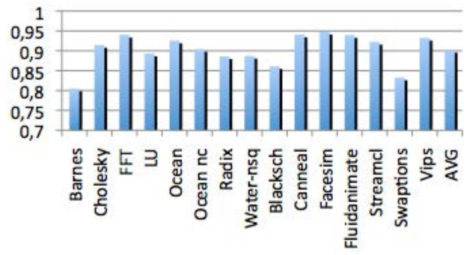


Fig. 7. Normalized execution time (L2 256KB L1 16KB)

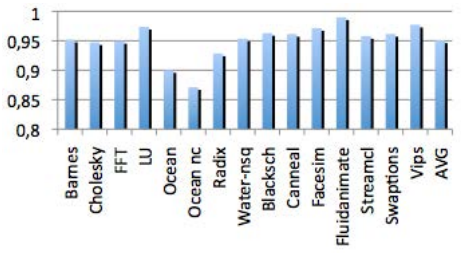
Figures 7 shows the normalized execution time with the six different configurations (PTA is not enabled in this evaluation). We can observe how execution time is largely reduced with average factors ranging between 12% and 24% when using First Touch and RHM (with and without migration enabled). RHM achieves lower execution time due to its achieved higher locality in L2. Also, the migration policy helped in further reducing execution time. Contrary to this, D-NUCA is not able to achieve large reductions when compared to S-NUCA. The use of private caches achieves large execution time reductions but its effectiveness depends on the size of the working set of every application.

Let's now enable the PTA and see how it impacts performance. Figure 8.a shows the normalized reduction in number of broadcast messages received with PTA. On average, PTA helps in reducing the number of received messages by 10%, saving link and router traversals and L2 tag accesses. The average number of messages saved per broadcast is 3.41 (without chopping, the number of messages per broadcast is 15). PTA improves the performance of RHM in two ways: first, as broadcast branches are cropped, the block search phase is faster; second, at the destination tile messages are delivered directly to the L2 bank without having to cross the 4 pipeline stages of the switch. These two effects combined lead to a further average reduction of execution time of 5% (12% for Ocean-nc).

Figure 9 shows the average load and store miss latency, respectively, for the evaluated configurations, normalized to the S-NUCA approach. RHM reduces these latencies more than 35% on average and up to 80% (FFT store miss latency) or even 90% (Radix load miss latency). Again, the effectiveness of RHM in reducing the miss latency depends on the memory access pattern of each



(a) Broadcast messages



(b) Execution time

Fig. 8. Normalized reduction in broadcasts and execution time when using PTA

application. Streamcluster shows a high percentage of blocks which are first accessed by a tile and then by different tiles during different phases of the application. In this case, a first touch policy has the negative effect of overloading the tile where blocks are mapped, and the migration mechanism can effectively move the blocks to the correct tiles.

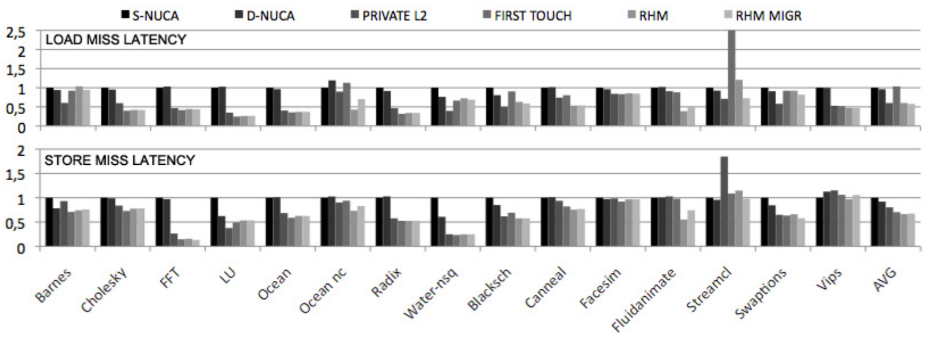


Fig. 9. Normalized load and store miss latency

To conclude performance analysis, it should be observed, that RHM performs better than FirstTouch. Although FirstTouch is a simple mechanism not requiring any hardware assistance, it should be noted RHM allows finer-grained assignments (blocks vs pages) and also more effective thread migration as blocks can be effectively migrated along with threads.

4.1 Energy

Figure 10 shows the normalized dynamic and total energy consumed by the NoC with the six configurations. Resource access (input buffer read/write, routing, switch allocation, crossbar traversal and link traversal) have been accounted and fed into Orion 2.0 [9]. If the request misses in the local L2 bank, RHM consumes more energy than the other schemes, due to the broadcasts. However, the high percentage of hits in the local L2 leads to less network activity compared to an S-NUCA. This, combined with the reduced execution time, leads to average energy reductions of 32%. Energy consumption is further reduced by 55% on average when migration is enabled (RHM MIGR).

Figure 11 shows the normalized energy consumed by the L2 cache. We used Cacti [8] to obtain the dynamic energy and the leakage per bank. Due to the broadcast access, RHM consumes more dynamic energy than other proposals (50% more energy on average), but the leakage component, reduced by the lower execution time, dominates over the dynamic energy for the configuration we choose. On average, energy consumption with RHM is reduced by 29% without block migration and 31% when block migration is enabled.

The area overhead and the power consumption of the LLC utilization table at the memory controller are a minimal fraction of the overall chip area and power requirement, due to its very small size compared to the on-chip cache and to the limited number of accesses compared to L1 and L2 accesses (the table is only accessed in case of L2 miss).

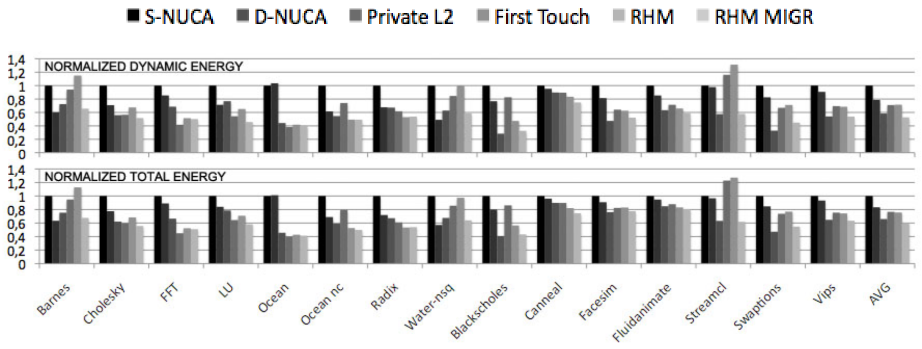


Fig. 10. NoC's energy consumption

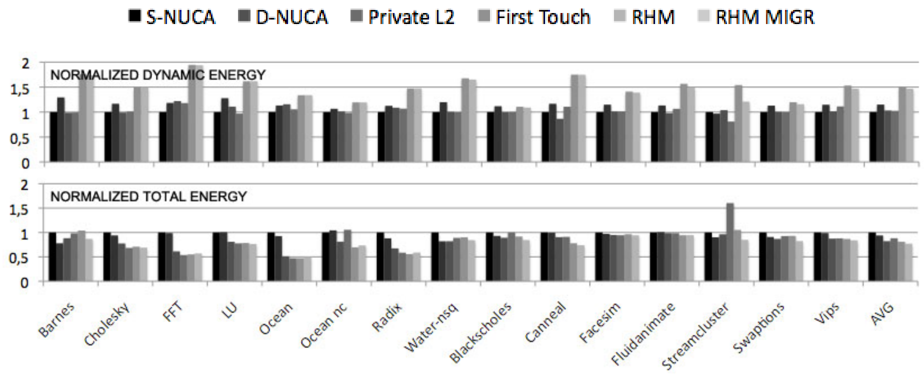


Fig. 11. LLC's energy consumption

5 Related Work

To overcome the wire-delay problem [10], the LLC in CMP systems is usually banked; this configuration is commonly called a Non-Uniform Cache Access architecture (NUCA), initially proposed by Kim *et al.* for a single core system [11] and then extended to many cores and CMPs [12], [13], and in turn offers many options when implementing the mapping of the blocks on each bank, the home bank search policy [15] [16] and the potential migration [11] or replication [14] of blocks. Both private and shared LLCs have their advantages and drawbacks, so hybrid configurations have been proposed to exploit the benefits of both design choices, such as ESP-NUCA [17] and CloudCache [18]. CMP-NuRAPID [19] decouples tags and data to allow data placement and replication in any LLC bank. Reactive-NUCA [14] also allows block replication. CMP-NuRAPID however requires an additional bus, while Reactive-NUCA is optimized for use on a 2D torus topology, and therefore it could perform poorly in a 2D mesh-based system. OS-based techniques to achieve a better mapping of the cache blocks to the LLC banks have been proposed by Cho et al.[6], Ros et al. [20], Das *et al.* [22] to achieve dynamic mapping through OS-level page allocation. Cuesta et al.

[21] deactivate the coherence protocol for blocks which are detected as private by the OS. Compile-time and data-based techniques have also been proposed in [23] and [24]. OS- and compiler-based techniques however rely on static mapping at hardware-level and can't support block migration or replication. Finally, Hammoud *et at.* [1] propose to implement blocks placement strategies at the memory controller(s) to prevent placing a block at an exceedingly pressured local set. To locate cache blocks at the LLC a CTCT [25] policy is assumed, which introduces 3-way communications in some cases, thus increasing the latency of L1 misses. RHM, differently from previous proposals, allows efficient block search between L2 banks in the whole chip. The optimizations/support at the NoC level allow for a aggressive data placement policy requiring only a small table at the memory controller, and avoiding the 3-way communication of some of the previous solutions, or the static assumption of private caches or OS-level solutions.

6 Conclusions and Future Work

In this work we have proposed Runtime Home Mapping (RHM) of the cache blocks to the LLC banks performed at the memory controller with NoC level support. Different improvements and designs at the NoC level enable fast and efficient location of data. The aim is to allocate L2 home blocks as close as possible from requestors. Results indicate a large span of improvement both in execution time and in reduced miss latencies. The current work can be extended in many directions, potentially leading to further improvements. Indeed, in this paper we applied baseline methods for different critical design choices of the method. As a first thing, we have plans to evaluate how the search phase behaves with different broadcast implementations and different network topologies. Second direction is the definition of smart mapping strategies located in the memory controller to address power management and fault tolerance at the LLC level. We also would like to evaluate the performance of dynamic home mapping combined with virtualization where the memory controller cooperates with the OS hypervisor to optimize the partitioning of chip resources to applications. Another future work direction is the replication of shared data blocks in more than one L2 bank. Finally, we plan to extend RHM to use it in a system with more than one memory controller and to reduce scalability issues.

Acknowledgements. This work has been supported by the VIRTICAL project (grant agreement n 288574) which is funded by the European Commission within the Research Programme FP7.

References

1. Hammoud, M., et al.: A dynamic pressure-aware associative placement strategy for large scale chip multiprocessors. IEEE Computer Architecture Letters 9(1), 29–32 (2010)
2. Lodde, M., et al.: Heterogeneous noc design for efficient broadcast-based coherence protocol support. In: NOCS 2012 (2012)
3. Lodde, M., et al.: Heterogeneous network design for effective support of invalidation-based coherency protocols. In: INA-OCMC 2012 (2012)

4. Lodde, M., et al.: Built-in fast gather control network for efficient support of coherence protocols. In: IET-CDT INA-OCMC 2012 Special Issue (2012)
5. The nangate open cell library, 45nm freepdk, `https://www.si2.org/openeda.si2.org/projects/nangatelib/`
6. Cho, S., et al.: Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In: MICRO 2006 (2006)
7. Miller, J.E., et al.: Graphite: A distributed parallel simulator for multicores. In: HPCA 2010 (2010)
8. Cacti 5 technical report, `http://www.hpl.hp.com/techreports/2008/hpl-2008-20.html`
9. Kahng, A., et al.: Orion 2.0: A power-area simulator for interconnection networks. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 20(1) (2012)
10. Matzke, D.: Will physical scalability sabotage performance gains? Computer 30(9), 37–39 (1997)
11. Kim, C., et al.: An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: ASPLOS 2002 (2002)
12. Huh, J., et al.: A nuca substrate for flexible cmp cache sharing. In: ICS 2005 (2005)
13. Beckmann, B., et al.: Managing wire-delay in large chip-multiprocessors caches. In: MICRO 2003 (2003)
14. Hardavellas, N., et al.: Reactive nuca: near-optimal block placement and replication in distributed caches. In: ISCA 2009 (2009)
15. Lira, J., et al.: Hk-nuca: Boosting data searches in dynamic non-uniform cache architectures for chip multiprocessors. In: IPDPS 2011 (2011)
16. Ricci, R., et al.: Leveraging bloom filters for smart search within nuca caches. In: WCED 2006 (2006)
17. Merino, J., et al.: Esp-nuca: A low cost adaptive non-uniform cache architecture. In: HPCA 2010 (2010)
18. Lee, H., et al.: Cloudcache: Expanding and shrinking private caches. In: HPCA 2011 (2011)
19. Christi, Z., et al.: Optimizing replication, communication and capacity allocation in cmps. In: ISCA 2005 (2005)
20. Ros, A., et al.: Distance-Aware Round-Robin Mapping for Large NUCA Caches. In: HiPC 2009 (2009)
21. Cuesta, B., et al.: Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In: ISCA 2011 (2011)
22. Das, A., et al.: Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores. In: DATE 2012 (2012)
23. Li, Y., et al.: Compiler-assisted data distribution for chip multiprocessors. In: PACT 2010 (2010)
24. Zhang, Y., et al.: A data layout optimization framework for nuca-based multicores. In: MICRO 2011 (2011)
25. Hammoud, M., Cho, S., Melhem, R.: ACM: An efficient approach for managing shared caches in chip multiprocessors. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 355–372. Springer, Heidelberg (2009)

Online Dynamic Dependence Analysis for Speculative Polyhedral Parallelization

Alexandra Jimborean[1], Philippe Clauss[2], Juan Manuel Martinez[2], and Aravind Sukumaran-Rajam[2]

[1] UPMARC, University of Uppsala, Sweden
alexandra.jimborean@it.uu.se
[2] Team CAMUS, INRIA, ICube lab., University of Strasbourg, France
{philippe.clauss,juan-manuel.martinez-caamano, aravind.sukumaran-rajam}@inria.fr

Abstract. We present a dynamic dependence analyzer whose goal is to compute dependences from instrumented execution samples of loop nests. The resulting information serves as a prediction of the execution behavior during the remaining iterations and can be used to select and apply a speculatively optimizing and parallelizing polyhedral transformation of the target sequential loop nest. Thus, a parallel lock-free version can be generated which should not induce any rollback if the prediction is correct. The dependence analyzer computes distance vectors and linear functions interpolating the memory addresses accessed by each memory instruction, and the values of some scalars. Phases showing a changing memory behavior are detected thanks to a dynamic adjustment of the instrumentation frequency.

The dependence analyzer takes part of a whole framework dedicated to speculative parallelization of loop nests which has been implemented with extensions of the LLVM compiler and an x86-64 runtime system.

Keywords: Dynamic online dependence analysis, polyhedral transformations, speculative, parallelization, optimization, runtime.

1 Introduction

Speculative parallelization is a classic strategy for automatically parallelizing codes that cannot be handled at compile-time due to the use of dynamic data and control structures. However, since this parallelization scheme requires on-the-fly semantics verification, it is in general difficult to perform advanced transformations for optimization and parallelism extraction. Most speculative systems dedicated to loop nest parallelization launch slices of the original sequential outermost loop in parallel threads, without handling any other code transformations. Thus, verification consists merely in monitoring concurrent updates of the same memory locations using a centralized data structure – which is an important performance bottleneck – and in validating the ones occurring at the earliest iteration according to the original loop indices. However, as soon as the outermost loop carries a dependence, this parallelization strategy fails in numerous rollbacks. Also, it does not consider the current execution context or other factors that impact performance, such as data locality. Hence, more advanced parallelizing and optimizing transformations are required, comparable to the ones applied at compile time when possible. But a new verification strategy is required, ensuring that not only memory writes, but also reads, have to be performed in a semantically correct order. This requires the computation of dependences between memory accesses and the verification of their constancy during the speculatively parallel execution of loop nests.

In this paper, we present a dynamic dependence analyzer of loop nests which incurs a minimal time overhead, such that its results can be used online, in the attempt of speculatively *optimizing* and *parallelizing* the code. It is based on a code instrumentation system specifically dedicated to loop nests, which is applied on small execution samples. This loop sampling mechanism relies on a multiversioning scheme, in which instrumented and non-instrumented versions of each target loop are generated at compile time. Additionally, it embeds a switching mechanism allowing to alternate the execution of instrumented and non-instrumented loop bodies. The instrumented bodies contain instructions devoted to collect the addresses that are accessed by the memory instructions and the values that are assigned to some specific scalars called *basic scalars*. From the collected information, the dependence analyzer computes distance vectors and linear functions interpolating the memory addresses and the values assigned to the basic scalars in order to allow their privatization when parallelizing.

The dependence analyzer is supported by a runtime system which alternates the execution of different versions of the target loop nest. Thus, phases showing a changing memory behavior are detected by launching instrumented versions at some execution points. The frequency in which they are launched can be either fixed or adjusted according to the constancy of the memory behavior. Since the dependence analysis is performed based on instrumenting execution samples, it serves as a prediction for the remaining iterations of the loop nest. Based on its results, the runtime system selects and applies a speculatively optimizing and parallelizing polyhedral transformation, generating a lock-free parallel code, which does not induce any rollback, if the prediction is correct. When parallelizing speculatively, the associated verification system consists in verifying the constancy of the linearly interpolating functions, instead of monitoring concurrent memory accesses, which is the classical approach in speculative systems. Hence, verification is completely distributed among the threads, and does not require any centralized data structure.

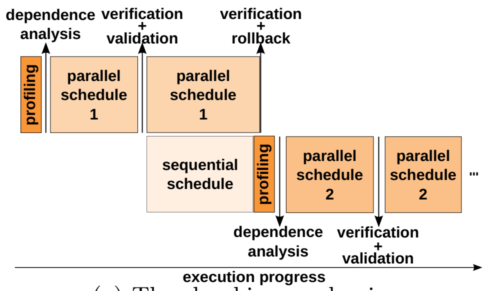
We show on a set of benchmarks that our analyzer is successful in identifying dependences across the iterations of a loop nest with a negligible runtime overhead. This property makes it insensitive to any variations of the input data or to program phases, since it can be applied repeatedly during one execution of the application, preceding the runtime optimizations.

2 Description of the Framework

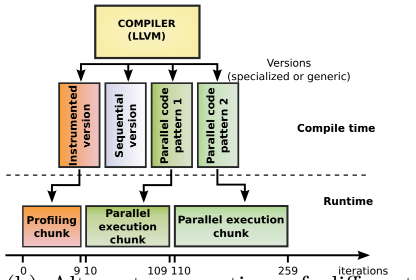
This proposal focuses on the advanced dynamic dependence analysis we propose as part of the TLS framework [4], called VMAD, designed to apply polyhedral transformations at runtime, such as tiling, skewing, interchange, etc., by speculating on the linearity of the loop bounds, of the memory accesses and of the values taken by specific variables, the basic scalars. Speculations are guided by online profiling phases. The instrumentation and analysis processes are thoroughly described in our previous work [6]. The current proposal extends the dependence analyzer described previously [6], with the computation of the exact distance vectors, which are employed in validating more complex loop transformations, rather than straightforward parallelization.

A key aspect is instrumentation by sampling, in which the execution of instrumented and non-instrumented loop versions are alternated. The instrumented version is executed for a small number of consecutive iterations of each loop in the nest to collect sufficient information for performing the dynamic dependence analysis. Next, a non-instrumented version is launched, to limit the overhead. Instrumentation is re-launched with a varying frequency, in the view of detecting new phases characterized by a different pattern of the memory accesses. When there is a change of phases, the runtime system triggers a new instrumentation. This guarantees that the minimal amount of instrumentation is performed during the execution of the loop nest, but sufficient to characterize each new phase. Please note that in case the system detects frequent changes of phases (i.e. an instable behavior of the nest), it aborts the instrumentation and the attempt to speculatively parallelize the nest, since the speculations would most probably be invalidated. Hence, the system is able to self-control its overhead.

Using the chunking mechanism presented in Fig. 1(a), we slice the iteration space of the outermost loop into successive chunks, as detailed in our previous work [4]. Each chunk represents a subset of consecutive iterations of the outermost loop and can embed a different loop version (either instrumented, original or optimized). Note that chunking is performed at the level of the outermost loop only, nevertheless, during the execution of the profiling chunk, the instrumented and non-instrumented versions of the *innerloops* alternate, as presented in [6], to incur a minimal overhead. Following the results of the profiling, the dependence analysis validates a suitable polyhedral transformation for each loop phase. In this paper we focus on the process of computing the cross-iteration data dependences and validating polyhedral transformations, addressing the reader to our previous work [4] for more details regarding the TLS framework which applies the results of the dependence analyzer. During the speculative execution, the predictions are verified, initiating a rollback upon a misspeculation and resuming the execution with a sequential chunk. Misspeculations indicate a change of phase and they trigger a new instrumentation and analysis phase, after the faulty iterations are reexecuted sequentially. If validation succeeds, a new parallel chunk is launched. The process is depicted in Fig. 1(a). The implementation of VMAD consists of two parts: a *static* part, implemented in the LLVM compiler [10], designed to prepare the loops for instrumentation and parallelization,



(a) The chunking mechanism



(b) Alternate execution of different versions during one loop nest's run

Fig. 1. Multiversioning

and a *dynamic* part, in the form of an x86-64 runtime system whose role is to build interpolating functions, to perform dynamic dependence analysis and transformation selection and to guide the execution.

Static Component. Our modified LLVM compiler generates customized versions of each loop nest of interest: original, instrumented and several parallel code patterns, together with a mechanism for switching between the versions. The patterns represent parameterized code versions, instantiated at runtime based on the results of the dependence analysis. To complete the loop's execution and adapt to the current phase, we automatically link at runtime the different versions of the original code. Each version is launched in a chunk to execute a subpart of the loop which is followed by the others, as in relay races. The support for chunking the outermost loop and linking distinct versions is illustrated in Fig. 1(b). The instrumented, original and two parallel code patterns are built at compile time. At runtime, one or another version is automatically selected to be executed for a number of iterations.

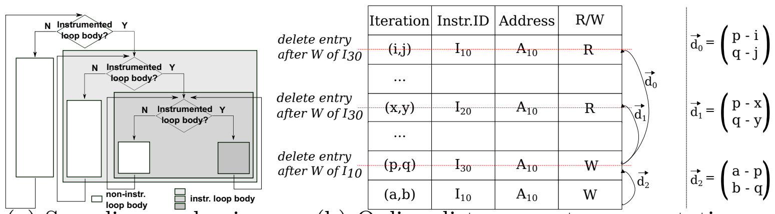
Dynamic Component. The runtime system collaborates tightly with the static component. During the instrumentation phase, it retrieves the accessed memory locations, the values assigned to the basic scalars, and computes interpolating linear functions of the enclosing loop indices. Instrumentation is performed on samples to limit the time overhead and is followed by the dependence analysis which evaluates whether a polyhedral transformation can be efficiently applied. If successful, this information can be useful in speculatively executing an optimized and parallelized version of the loop.

3 Dynamic Dependence Computation

A dedicated pragma allows the user to mark interesting loop nests in the source code. We have implemented dedicated extensions to the LLVM compiler that generate automatically, for our instrumentation purposes, two different versions of each target loop nest: instrumented and non-instrumented.

Instrumentation. The instrumented version associates to each memory instruction additional code which collects the target memory address and writes it in a buffer that will be read by the runtime system. Similarly, other instrumenting instructions are associated to monitor some specific scalars, called *basic scalars.* They have the interesting property of being at the origin of the computations of all other scalars used in the loop bodies, as for instance the target address computations. The basic scalars are identified at compile-time, being defined in the loop bodies as ϕ-nodes, since the intermediate representation of the LLVM compiler is in static single assignment form (SSA). They also carry dependences, since their values in a given iteration depend on the values they have been assigned in previous iterations. Hence, the opportunity for applying loop transformations and parallelizations depends on the possibility of privatizing them by predicting their values at each iteration. For this purpose, in the parallel code patterns, the basic scalars are initialized using the predicting linear functions (depending only on the indices of the transformed loops, and not on their value in the previous iterations). Straightforward examples of such basic scalars are the loop indices of for-loops, which are incremented at each iteration.

Since any kind of loops – for, while, do-while – are targeted, the instrumented and non-instrumented versions generated at compile-time contain one new iterator per loop, initialized with zero and incremented with a step of one. These iterators are injected by the compiler and used in the computation of the interpolating linear functions, as detailed in [5,6].


N
Instrumented loop body?
Y
non-instr. loop body
instr. loop body
delete entry after W of I30
delete entry after W of I30
delete entry after W of I10

Iteration	Instr.ID	Address	R/W
(i,j)	I_{10}	A_{10}	R
...			
(x,y)	I_{20}	A_{10}	R
...			
(p,q)	I_{30}	A_{10}	W
(a,b)	I_{10}	A_{10}	W

$$\vec{d_0} = \begin{pmatrix} p - i \\ q - j \end{pmatrix} \quad \vec{d_1} = \begin{pmatrix} p - x \\ q - y \end{pmatrix} \quad \vec{d_2} = \begin{pmatrix} a - p \\ b - q \end{pmatrix}$$

(a) Sampling mechanism for a 3-depth loop nest (b) Online distance vector computation

Fig. 2. Instrumentation by sampling and computation of distance vectors

Dedicated Sampling. Executions of the instrumented versions of the target loop nests are obviously more time-consuming than the original versions. However, these versions are run for small slices of the outermost loops of the target nests using the chunking mechanism presented in the previous section. Additionally, we implemented a dedicated sampling system allowing to instrumented only slices of each loop composing the nest. Thus, instrumentation activation does not depend only on the current loop, but also on the parent loops, making instrumented and non-instrumented bodies alternate, as illustrated in figure 2(a). Sizes of the instrumented slices can be either fixed or adjusted at runtime.

Polyhedral Transformations. The dynamic dependence analyzer is designed to compute distance vectors, and then verify if these distance vectors characterize completely the memory behavior observed during the run of the instrumented version. This latter verification is achieved using an address value range analysis and a GCD test, as explained below. If the computed distance vectors conveniently characterize the target code, then they are used to select an optimizing parallelizing transformation of the loop nest, which results in the generation of a lock-free multithreaded version that should not induce any rollback if the memory behavior remains stable.Transformations that may be applied are polyhedral transformations [2] that change the order in which the iterations are scanned, such that at least one parallel loop is exhibited and the iteration schedule is optimized to address important issues, like data locality. Such a transformation is defined by a unimodular matrix T, applied to the space of the loop indices. T is valid *w.r.t.* any dependence distance vector d if the transformed vector $T \cdot d = d'$ is lexicographically positive, *i.e.*, if its first non-null component is positive. This component gives the depth of the loop which carries the associated dependence, therefore this loop cannot be parallelized. The outermost parallel loop is then the outermost loop which does not carry any dependence, considering all the transformed distance vectors.

Distance Vectors Computation. The runtime system reads the values communicated through the buffer and, when possible, builds linear interpolating functions for each memory instruction or basic scalar assignment, whose variables are the loop indices. It also computes linear functions to interpolate the loop bounds of the inner loops. Simultaneously, the collected memory addresses are used to compute online dependence distance vectors. The addresses are stored in a table whose entries also contain the access type (Read or Write), the loop index values at which the memory access occurred and the memory instruction identifier. Each time a new entry is created, a table look-up finds the previous accesses at the same address, computes the corresponding distance vectors and removes the entries that are becoming useless. The implemented algorithm is shown in the first part of table 1 and illustrated by figure 2(b).

Since only a sample of the execution tracks the memory accesses, the so-computed distance vectors may not entirely characterize the dependences that may occur during the whole execution of the target loop nest. If the instrumentation is performed on a loop slice of size S, a dependence whose distance is greater than S can obviously occur. We handle this issue by considering each couple of memory instructions, where at least one is a write, and for which no distance vector has been computed. Their associated interpolating linear functions are then used to verify if any dependence may occur between these instructions. First, a value range analysis is performed. For each linear function, their maximum and minimum reached values are computed using the interpolated loop bounds. If the respective ranges of touched addresses overlap, then a dependence may occur. In this case, a second analysis is performed through the GCD test, concluding if there may be a solution when considering the integer equation where both functions are equal. The algorithm is shown in the second part of table 1. These latter dependence tests are obviously less time-consuming than exact solving of integer equations, which would induce an overhead unacceptable for a dynamic parallelization system. Moreover, an empty solution of these tests guarantees that the computed distance vectors entirely characterize the dependences, which allows one to validate the correctness of polyhedral transformations with a high probability, for a significant part of the execution.

Table 1. Dependence analysis algorithms

Distance vector computation algorithm.

```
create an entry in the table
if the current access is a write
    look for all reads at the same address, until finding a write
        for each found read, compute a distance vector which characterizes
          an anti-dependence
        if a write has been found, compute a distance vector which characterizes
          an output dependence
    remove all these entries excepting the current one
if the current access is a read
    look for a previous write at the same address
        if a write has been found, compute a distance vector which characterizes
          a flow dependence
```

Value range and GCD tests application algorithm.

```
build the couples of memory instructions not characterized by distance vectors,
  where at least one instruction is a write
for each couple
    compute their respective ranges of touched addresses by computing
      the extreme values reached by their associated linear functions
    if their ranges overlap
        perform the GCD test on the corresponding integer equation
        if the test fails (empty solution), the couple does not carry any dependence
        else, the couple may carry a dependence
    else, the couple does not carry any dependence
```

4 Experiments

Experiments were conducted on the Polybench benchmark suite [12]. For each program, we selected the most time-consuming loop nest. Although these codes can be analyzed statically to detect dependences, we stressed our system to extract dependences at runtime, in order to show its accuracy and ability in deducing speculative parallelizations. To test the ability of our system in detecting dependence phases, we modified the Polybench codes by introducing if-statements in the innermost loop body in order to alternate between three successive phases, each being characterized by slightly modified memory accesses that may introduce dependences. For instance, for a nest whose outermost loop ranges from 0 to N, we inserted if-statements that induce different memory behaviors for each subset of $N/3$ iterations. We modified the memory references for two of the three subsets by adding some integer constants to the original array references.

Our measurements are presented in table 2, where the kernel loop nest of each program is analyzed. The left part of the table shows the measurements performed on the original programs of the Polybench suite, while the right part shows the measurements performed on the modified programs exhibiting phases with different dependences. The second column shows the size of the instrumented chunks. For each program, we execute successively three instrumented runs with different instrumented chunk sizes (3, 10, 20) in order to compare the accuracy of the analyses, relatively to the number of instrumented iterations, as well as their respective overheads. The third column shows the percentage of time-overhead induced by instrumenting execution samples and determining dependences, computed as: *(instrumentationTime - originalTime)/originalTime.* The original codes were compiled using Clang-LLVM 3.0 with flag O3, on an AMD Opteron 6172, 2.1 Ghz, running Linux 3.2.0-27-generic x86_64.

For the experiments reported in this paper, the number of instrumented chunks launched by the runtime system depends on the number of iterations of the outermost loop, following the strategy: the first instrumented chunk is followed by a non-instrumented chunk of 100 iterations. Then again an instrumented chunk is launched. If the result of the dependency is equal to the result obtained from the previous instrumentation, then a non-instrumented chunk of $100 \times 2 = 200$ iterations is launched. Thus, the size of the non-instrumented chunk is doubled continuously as long as the dependences remain the same. If the dependences change, then the size of the non-instrumented chunk is reset to 100, as a new phase is detected. Please note that this strategy is devoted solely to the goal of performing online dependence analysis. In a complete speculatively parallelizing system, it is the speculation verification performed by the speculatively parallel code which would detect new phases and provoke the launching of an instrumented chunk. Additional experiments indicate that adjusting the frequency of the instrumentation based on phases detected at runtime by the TLS system, reduces the overhead of the analyzer to 15% at most, since it is performed only once for each phase, and the number of instrumented iterations is small, relative to the total number of iterations. However, this is out of the scope of the current article.

The fourth and fifth columns show the number of instrumented memory instructions and base scalar assignments, respectively; the sixth column shows the computed distance vectors for each phase and their types; finally, the seventh column suggests speculative parallelizations that could be applied considering the distance vectors and the results of the GCD tests.

The right part of the table illustrates the results on the modified programs exhibiting dependence phases. A new column shows the number of detected phases, and only one row per phase is presented for some instrumented chunk sizes, due to space constraints. For the same reason, although we successfully analyzed the dependences of all Polybench codes, it is impossible to show the results obtained for all of them. Only some representative ones are selected.

Table 2. Online dynamic dependence analysis on benchmark programs

Program	Instr. Chunk Sizes	DDA overhead	# Memory Accesses	# Basic Scalars	Dep. Dist. Vectors **output - flow - anti**	Speculative Par. Opportunities	DDA overhead	# Memory Accesses	# Basic Scalars	# Phases	Dep. Dist. Vectors **output - flow - anti**	Speculative Par. Opportunities
correlation/phase 1	**3**	2.96%	5	3	$\begin{pmatrix}0\ 0\ 1 : f\end{pmatrix}$	outermost loop	3.25%	9	4	2	$\begin{pmatrix}0\ 0\ 1 : f\end{pmatrix}$	outermost loop
phase 2											$\begin{pmatrix}0\ 0\ 1 : o - f - a\\ 1\ 0\ 0 : o - a\end{pmatrix}$	2^{nd} loop
phases 1,2	**10**	3.62%	5	3	same as above	same as above	5.63%	9	4	2	same as above	same as above
phases 1,2	**20**	67.74%	5	3	same as above	same as above	206.16%	9	4	2	same as above	same as above
cholesky/phase 1	**3**	224.55%	5	4	$\begin{pmatrix}1 & -1 & 1 : a\\ 1 & 0 & 1 : a\\ 1 & 1 & 1 : a\\ . & . & . \dots\end{pmatrix}$	$\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}$	226.56%	7	4	3	$\begin{pmatrix}1 & -1 & 1 : a\\ 1 & 0 & 1 : a\\ 1 & 1 & 1 : a\\ . & . & . \dots\end{pmatrix}$	$\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}$
phase 2											$\begin{pmatrix}1 & 7 & 1 : a\\ 8 & 1 & 1 : a\\ . & . & . \dots\end{pmatrix}$	$\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}$
phase 3											$\begin{pmatrix}5 & -3 & 1 : a\\ 4 & -3 & 1 : a\\ . & . & . \dots\end{pmatrix}$	$\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}$
phases 1,2,3	**10**	239.67%	5	4	$\begin{pmatrix}1 & -1 & 1 : a\\ 1 & 0 & 1 : a\\ . & . & . \dots\end{pmatrix}$	$\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}$	248.48%	7	4	3	larger set than above	$\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}$
phases 1,2,3	**20**	2408.03%	5	4	$\begin{pmatrix}1 & -1 & 1 : a\\ 1 & 0 & 1 : a\\ . & . & . \dots\end{pmatrix}$	$\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}$	4566.81%	7	4	3	larger set than above	$\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}$
adi/phase 1	**3**	3.19%	9	2	$\begin{pmatrix}0\ 1 : f - a\end{pmatrix}$	outermost loop	1.47%	19	3	3	$\begin{pmatrix}0\ 1 : f - a\end{pmatrix}$	outermost loop
phase 2											$\begin{pmatrix}0\ 1 : f - a\\ 1\ 0 : a\\ 1\ 1 : a\end{pmatrix}$	$\begin{pmatrix}1&1\\0&1\end{pmatrix}$
phase 3											$\begin{pmatrix}0 & 1 : f - a\\ 1 & 0 : f\\ 1 & -1 : f\end{pmatrix}$	$\begin{pmatrix}2&1\\1&0\end{pmatrix}$
phases 1,2,3	**10**	4.7%	9	2	same as above	same as above	7.13%	19	3	3	same as above	same as above
phases 1,2,3	**20**	67.88%	9	2	same as above	same as above	131.52%	19	3	3	same as above	same as above
bigc/phase 1	**3**	31.08%	8	2	$\begin{pmatrix}1\ 0 : o - a\\ 0\ 1 : f\end{pmatrix}$	$\begin{pmatrix}1&1\\0&1\end{pmatrix}$	189.03%	10	2	3	$\begin{pmatrix}1\ 0 : o - a\\ 0\ 1 : f\end{pmatrix}$	$\begin{pmatrix}1&1\\0&1\end{pmatrix}$
phase 2											$\begin{pmatrix}1 & 0 : o - a\\ 0 & 1 : o - f\\ 1 & -1 : a\\ . & . \dots\end{pmatrix}$	none
phase 3											$\begin{pmatrix}1 & 0 : o - a\\ 0 & 1 : o - f\\ 1 & -1 : a\\ . & . \dots\end{pmatrix}$	none
phases 1,2,3	**10**	36.34%	8	2	same as above	same as above	206.93%	10	2	3	same as above	same as above
phases 1,2,3	**20**	178.26%	8	2	same as above	same as above	600.48%	10	2	3	same as above	same as above
gemm/phase 1	**3**	-14.12%	4	3	$\begin{pmatrix}0\ 0\ 1 : f\end{pmatrix}$	outermost loop	-13.71%	10	3	2	$\begin{pmatrix}0\ 0\ 1 : f\end{pmatrix}$	outermost loop
phase 2											$\begin{pmatrix}0\ 0\ 1 : f\\ 1\ 0\ 0 : a\end{pmatrix}$	2^{nd} loop
phases 1,2	**10**	-13.85%	4	3	same as above	same as above	-13.48%	10	3	2	same as above	same as above
phases 1,2	**20**	-3.73%	4	3	same as above	same as above	16.1%	10	3	2	same as above	same as above
3mm/phase 1	**3**	-22.49%	3	3	$\begin{pmatrix}0\ 0\ 1 : f\end{pmatrix}$	outermost loop	-10.58%	4	3	2	$\begin{pmatrix}0\ 0\ 1 : f\end{pmatrix}$	outermost loop
phase 2											$\begin{pmatrix}0\ 1\ 0 : a\end{pmatrix}$	outermost loop
phases 1,2	**10**	8.38%	3	3	same as above	same as above	-10.38%	4	3	2	same as above	same as above
phases 1,2	**20**	19.41%	3	3	same as above	same as above	30.09%	4	3	2	same as above	same as above

Results in the table obviously show that larger instrumented chunks yield larger time overheads. In particular, one can observe a change of scale when considering chunks of size 20. However, since the accuracy of the dependency analysis is not significantly better with such large chunks, it argues to limit chunk sizes to at most 10. We observed that a high accuracy is often achieved with the smallest size of the instrumented chunk, 3. This is also due to the regularity of the handled benchmarks whose dependences are constant. In general, 10 iterations are sufficient to obtain a good accuracy, with a relatively low overhead. For the smallest instrumented chunk size, the overheads vary from -14% to 226%. Speed-ups can be explained by beneficial side-effects of chunking, or different optimizations triggered on our code versions compared to the ones generated by Clang on the original codes. When the time-overhead is the highest, it still remains acceptable: less than 3.5× with sizes 3 and 10, since parallelization should provide speed-ups that would substantially hide this overhead. With size 20, the overhead can become dangerously high in some cases, showing that instrumentation by sampling must remain under a relatively small threshold.

Our experiments also indicate that some codes require more advanced transformations than just parallelizing the original loops. It often occurs that every loop carries dependences. A standard TLS system would continuously rollback in such cases. On the other hand, a loop transformation, as the ones suggested by the transformation matrices in the table, would yield a semantically equivalent nest with at least one dependence-free loop that can be parallelized. This emphasizes the need of relatively accurate dependence analysis for TLS systems.

5 Related Work

Dependence analysis is an essential aspect in systems designed to perform automatic optimizations. However, previous research works focused on performing dynamic dependence analysis dedicated to an offline usage, thus, the overhead incurred by such profilers varies from 3×[13] to 70×[9]. Traditional TLS systems either rely on the results of such profilers or perform an optimistic, simple, straightforward parallelization of the outermost loop, for which no advanced dependence analysis is required. In contrast, we developed an ultra-fast dynamic dependence analyzer that can be used online and applied repeatedly during one execution, to adapt to different phases. This has been made possible thanks to a specific instrumentation system dedicated to loop nests, and able to switch between instrumented and non-instrumented code following a counter of the number of instrumented iterations. A similar approach to reduce the cost of instrumented code is presented in [1]. Our instrumentation system has two major differences. It is dedicated to loop nests and thus includes specific sampling management to coordinate loop levels as a whole. It also includes a different sampling mechanism provided by the chunking system to instrument only a small slice of the outermost loop. In the following, we review various state of the art techniques, however they are expected to be used offline, due to their large overhead.

Kim et al. [8] describe the fragility of static analysis, pleading for speculative parallelization, by speculating on some memory or control dependences.

Statically, a PDG (program dependence graph) is built. All dependences occurring less frequently than a certain threshold are speculatively removed from the PDG and the code is parallelized. Nevertheless, the dependence analysis is very simple and cannot be employed in validating aggressive code transformations, other than straightforward parallelization. The work of Praun et al. [14] identifies potential candidates for speculative parallelization by analyzing the density of runtime dependences in critical sections, *w.r.t.* the total number of executed instructions. Similarly to our proposal, the model can adapt to different program phases, and detect the ones suitable for speculative parallelization. No information regarding the profiler's overhead is presented, thus we conclude that the results of the analysis are used offline. The recent work of Vanka et al. [13] proposes a form of dependence analysis based on set operations using software signatures. In contrast to other works relying on sampling to achieve better performance, they group dependent operations into sets, and operate on relationships between sets, instead of considering pair-wise dependences. Additionally, they only profile queries relevant to the optimization being performed, rather than all possible queries. Thus, the profiler is highly accurate and well performing in comparison to previous works, introducing a 2.97× slowdown in average. Similarly, Oancea and Mycroft [11] propose a dynamic analysis for building dependence patterns. They map dependent iterations on the same thread, such that no dependence violations occur. Mapping is based on congruences of sets, computed from the dependence pattern. Still, the system incurs a considerable overhead.

Ketterlin and Clauss propose a system called Parwiz [7] that empirically builds a data dependence graph, after instrumenting samples or complete executions of an application with several representative inputs. Their goal is to identify potentially parallel regions of sequential programs and provide hints to the programmer. Similar tools analyze the data dependences across one execution and suggest parallelization strategies. Embla [3] performs an offline dynamic analysis and reports all occurring dependences, exhibiting parallelization opportunities. SD^3 [9] performs dynamic dependence analysis to provide suggestions to the developer on which modifications are desirable, such that the code becomes suitable for parallelization. SD^3 shows a 70× slowdown on average. Alchemist [15] is designed to identify dependences across loop iterations, loop boundaries and methods. It can be used offline by speculative systems, as it provides a very precise dependence analysis, analyzing complex data. Nevertheless, it induces a large overhead and it is not aimed for a runtime usage.

6 Conclusion

The increasing usage complexity of multi-core architectures require to shift advanced parallelization techniques from static to dynamic. Related to this purpose, we presented a dynamic dependence analyzer for loop nests dedicated to capture cross-iteration dependences, with a minimal time-overhead. Thus, it can be integrated in a TLS system for an online usage and even invoked repeatedly to characterize each new phase. The analyzer relies on instrumentation by sampling and computes distance vectors, which can be employed in validating polyhedral transformations for each phase of the nest.

References

1. Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.
2. U. Banerjee. *Loop Transformations for Restructuring Compilers - The Foundations.* Kluwer Academic Publishers, 1993.
3. K-F Faxén, K. Popov, S. Jansson, and L. Albertsson. Embla - data dependence profiling for parallel programming. In *Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '08, pages 780–785, Washington, DC, USA, 2008. IEEE Computer Society.
4. A. Jimborean, Ph. Clauss, B. Pradelle, L. Mastrangelo, and V. Loechner. Adapting the polyhedral model as a framework for efficient speculative parallelization. In *PPoPP '12*, 2012.
5. A. Jimborean, M. Herrmann, V. Loechner, and Ph. Clauss. VMAD: A virtual machine for advanced dynamic analysis of programs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 125–126, Washington, DC, USA, 2011. IEEE Computer Society.
6. A. Jimborean, L. Mastrangelo, V. Loechner, and Ph. Clauss. VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework. In Michael OBoyle, editor, *Compiler Construction*, volume 7210 of *Lecture Notes in Computer Science*, pages 220–239. Springer Berlin Heidelberg, 2012.
7. A. Ketterlin and Ph. Clauss. Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization. In *MICRO-45, The 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Canada, 2012.
8. H. Kim, N.P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August. Automatic speculative doall for clusters. In *CGO '12*. ACM, 2012.
9. M. Kim, H. Kim, and C-K Luk. SD3: a scalable approach to dynamic data-dependence profiling. In *MICRO '43*. IEEE Computer Society, 2010.
10. LLVM compiler infrastructure. `http://llvm.org`.
11. C. E. Oancea and A. Mycroft. Set-congruence dynamic analysis for thread-level speculation (TLS). In *LCPC '08*. Springer-Verlag, 2008.
12. Polybenchs. http://www-rocq.inria.fr/pouchet/software/polybenchs.
13. R. Vanka and J. Tuck. Efficient and accurate data dependence profiling using software signatures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 186–195, NY, USA, 2012.
14. C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 185–196, New York, NY, USA, 2008. ACM.
15. X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO '09*. IEEE Computer Society, 2009.

VGTS: Variable Granularity Transactional Snoop

Ehsan Atoofian

Electrical Engineering Department
Lakehead University
Thunder Bay, Canada
atoofian@lakeheadu.ca

Abstract. Transactional Memory (TM) is an appealing abstraction for increasing productivity of programmers and making parallel programming accessible to a wide community of non-experts. In TM systems, conflict detection is an essential element in maintaining correctness of transactions. Hardware Transactional Memories (HTMs) rely on cache coherence protocols to detect and resolve conflicts. In HTMs, when a transactional write misses in a cache, it broadcasts a snoop request asking remote caches for sharing information. While this method detects conflicts at the earliest possible time it is not efficient in term of power. We found that a significant fraction of transactional snoops in cache coherence protocols are unnecessary and waste power of interconnect and caches. Furthermore, many transactional snoops occur in coarse regions of memory. In this work, we introduce Variable Granularity Transactional Snoop (VGTS) which dynamically changes snoop granularity for transactions. VGTS monitors transactions and dynamically matches snoop granularity to the transactions' address patterns. Our simulation results reveal that VGTS is effective and reduces power of interconnect up to 44% and eliminates unnecessary cache snoops up to 43%.

Keywords: Hardware Transactional Memory, Synchronization, Cache Coherence Protocol, Power.

1 Introduction

Chip Multiprocessors (CMPs) are becoming mainstream in server, desktop, and even embedded systems to overcome power wall and other constrains in single-core processors. While CMPs reduce power and cost of cooling systems, they bring some unprecedented challenges. Software developers can no longer rely on next year's processor to hide the cost of new features in sequentially written software packages and gain speedup. In order to maintain pace of performance improvement in software applications, programmers need to develop parallel programs. The traditional method for parallel programming is lock. However, this approach entails difficult trade-offs: performance vs. complexity. Parallel programming with coarse-grain locking is simple but results in poor performance. On the other side, fine-grain locking yields better performance but is error-prone and difficult to understand and maintain.

Transactional Memory (TM) [1] is a promising programming model which addresses the problems of lock-based programming by providing the potential for performance of fine-grain locking with simplicity of coarse-grain locking. A transaction is a sequence of instructions that should execute atomically. In a TM program, a programmer just marks sections of the program as transactions and the underlying system guarantees atomicity. As such, in TM systems, programmers specify what should be done atomically, leaving the system determining how this is achieved. This relieves programmers from burden of worrying about synchronization bugs such as deadlock and convoying.

TM systems can be broadly classified into three categories: 1) Hardware Transactional Memory (HTM) [2], 2) Software Transactional Memory (STM) [14], or 3) Hybrid Transactional Memory (HyTM) [7]. STMs entail significant overhead and there is growing interest in transactional support in hardware to improve performance. HTM research has caught the attention of industry and some companies proposed extension for their architectures to support TM, such as Intel's Transactional Synchronization Extensions (TSX) [6], and the AMD Advanced Synchronization Facility (ASF) [15].

TM systems utilize resources in CMPs by execution multiple transactions concurrently. To maintain consistency of concurrent transactions, only conflict-free transactions commit successfully. A conflict happens when two or more number of transactions accesses a memory location and at least one of them writes into the memory location. In the event of conflict, only one transaction can commit and the rest should abort and restart or stall in the hope that the conflict is resolved later [2].

HTMs exploit cache coherence protocols to detect and resolve conflicts [2]. When a transaction writes to a memory address, it broadcasts a snoop request to invalidate cache blocks of other transactions that have accessed the same memory address. If a transaction finishes without having had any of its entries invalidated by remote caches, then the transaction commits by writing back its dirty entries in the cache (or write buffer) to the memory in lazy policy [17] or discarding its log buffer in eager policy [2]. On the other side, if another thread invalidates a transactional entry in the cache, the transaction aborts and restarts.

We find that cache coherence protocols dissipate a significant fraction of their power on unnecessary snoops when running transactional applications. Many transactions access disjoint memory regions and so a significant portion of transactional snoops result in cache misses. Therefore, the vast majority of the transactional snoops are unnecessary as they fail to find sharers in remote caches. In this work, we propose Variable Granularity Transactional Snoop (VGTS) to reduce unnecessary snoops generated by transactions. When a transaction executes, VGTS monitors transactional reads and writes and determines granularity of snoop requests generated by the transaction. Later, if the same transaction executes and broadcasts a snoop request, in addition to the requested address, we may ask remote cores to snoop a region of addresses. The responses received from remote cores are stored in a local filter and used for future snoops. In future, if a snoop request is found in the local filter, then the request is not broadcasted as none of the remote cores have the request. As such, VGTS avoids needless snoop broadcasts and reduces power of interconnection network and caches.

The rest of the paper is organized as follows. Section 2 provides background on how cache coherence protocols work in HTMs and motivates the need for adjusting snoop granularity per transaction. Section 3 discusses the implementation details and presents the hardware design of VGTS. Section 4 describes the evaluation methodology and presents quantitative results. Section 5 discusses related work and Section 6 concludes the paper.

2 Background

In this work, we use an HTM system similar to LogTM [2]. LogTM is an implementation of HTM which requires moderate augmentation of existing hardware and uses software support to restore state of a processor in the event of abort. LogTM implements eager version management by creating a per-thread transaction log in private caches, which holds the old values of all transactional writes. To detect transactional conflicts, LogTM uses a modified version of MOESI cache coherence protocol [10].

In LogTM, each cache block has two additional state bits: read and write bits. These bits are set if a transaction reads/writes a cache block. LogTM detects conflicts using MOESI and read/write bits. If a transactional read misses in a local cache, then the cache broadcasts a snoop request asking remote caches for the missed address. If the missed address exists in a cache of a remote node, then the remote node responds to the snoop request. If read bit of the remote cache block is set, then the two transactions can proceed without conflict. However, if the write bit of the remote cache block is set, then the two transactions conflict and at least one of them should abort. Similarly, if a transactional write results in a cache miss, then a snoop request is broadcasted on the interconnection network. If the address exists in a remote cache and the corresponding read or write bit is set, then a conflict happens and one of the two transactions should abort. To resolve the conflict, the transaction which initiates snoop aborts and the other transaction proceeds. In LogTM, a software handler walks through the transaction log and restores register and memory values in the event of abort.

LogTM broadcasts a snoop request to all cores on a cache miss. If all of the cores are caching the missed address, the broadcast would not be wasteful. However, in some applications, a significant part of snoop requests misses in remote caches. Figure 1 shows the percentage of transactional coherence requests that are redundant in Stamp v0.9.10 benchmark suite (please refer to Section 4.1 for methodology and configuration). A redundant coherence request is one that does not exist in any of the remote caches, thus unnecessarily wastes processor resources. We found that more than 75% of coherence requests are redundant across all benchmarks and thus unnecessarily consume network power, while also resulting in redundant snoop-induced cache look-ups.

3 Implementation

VGTS dynamically adjusts snoop granularity for transactions. When a transaction executes, VGTS monitors addresses generated by the transaction. If the transaction accesses consecutive memory locations then VGTS sets the snoop granularity to the size of the region accessed by the transaction. If the transaction accesses multiple disjoint regions, then VGTS sets the snoop granularity to the size of the smallest region to avoid wasting network bandwidth. To provide better insight into snoop granularity found by VGTS, we discuss part of a code region taken from Labyrinth. Figure 2.a shows part of the Labyrinth program. This benchmark finds shortest paths between pairs of starting and ending points for a given maze. gridPtr is a pointer to the shortest paths found in this benchmark. Figure 2.b shows address of cache blocks generated by the program in 2.a. Note that the granularity of snoops in cache coherence protocols is cache block. So, VGTS analyses addresses in the granularity of cache blocks not bytes. The sequence of addresses in 2.b forms three disjoint regions. The size of the smallest region is two cache blocks (15, 16). So, VGTS sets granularity of the transaction in 2.a. to two.

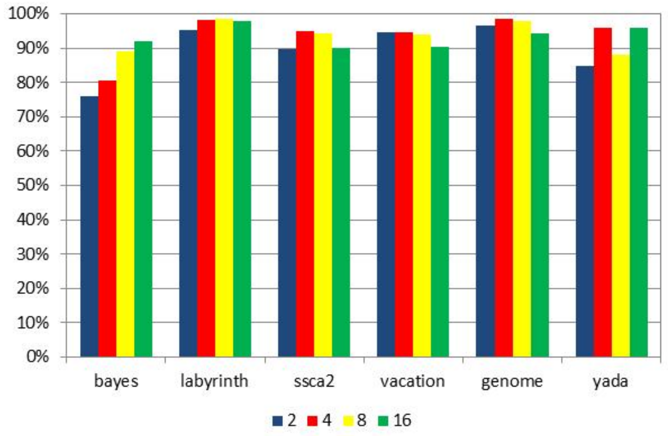


Fig. 1. Redundant snoops in Stamp v0.9.10 benchmark suite. In each benchmark, the number of threads changes between two and 16.

VGTS is a speculative approach and relies on the most recent execution of a transaction to decide on snoop granularity. VGTS is effective only if snoop granularity of a transaction does not change over time. To investigate pattern of snoop granularities in transactions, we measure locality of snoop granularity in Stamp benchmarks. Locality of snoop granularity shows how often snoop granularity of a transaction is the same in two consecutive executions. Figure 3 shows locality of snoop granularity in Stamp benchmarks [7]. For each benchmark, the number of threads varies from two to 16. The locality is measured by counting the number of times snoop granularity is the same in two consecutive executions of transactions and dividing by the total number of transactional commits. The locality changes from 63% in Labyrinth to 99.9% in Ssca2. This Figure proves that snoop granularity of transactions is predictable.

```
TM_BEGIN();
        ...
        TMGRID_ADDPATH(gridPtr, ...);
        ...
TM_END();
```

a)

address
8, 9, 8, 10,11
15, 16
21, 22, 22,
23, 24
...

b)

Fig. 2. a) Part of the Labyrinth program. b) Memory addresses (in granularity of cache block) pointed by gridPtr in Labyrinth.

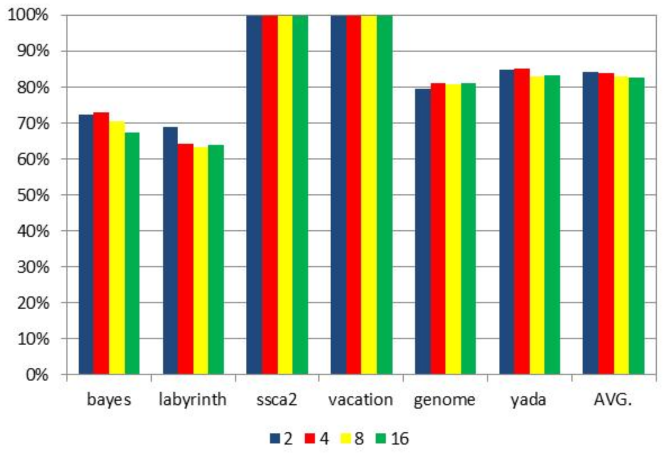


Fig. 3. Locality of snoop granularity in Stamp v0.9.10 benchmarks. For each benchmark, the number of threads changes from two to 16.

VGTS relies on Snoop Granularity Tables (SGTs) to keep record of snoop granularities of transactions (Figure 4). Each core has its own SGT. Each entry in SGT comprises two fields: snoop granularity and valid bit. When a transaction commits, VGTS writes snoop granularity of the transaction in to SGT. When a transaction starts, VGTS indexes SGT using starting address of the transaction. If a valid entry exists, then VGTS uses the corresponding snoop granularity for the transaction.

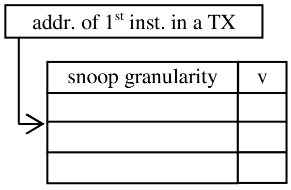


Fig. 4. Snoop Granularity Table (SGT)

VGTS exploits filters to reduce redundant coherence requests. The filter sits alongside the last-level private cache and maintains local region-level sharing information. Figure 5 shows the structure of the filter. The filter is a simple table with entries comprising the starting block address of a region, size of the region, and a valid bit. If a transactional instruction experiences a cache miss for the first time, then the cache controller broadcasts a snoop request asking other nodes to provide sharing information for the region determined by SGT. If the region is not shared by any other nodes, then the cache controller allocates an entry in the filter and sets block address, size, and valid bit. Prior to issuing a snoop request, each node looks up the local filter and if a matching entry exists, then the cache controller knows that forwarding the request is redundant. The filter entries are evicted either as a result of limited space or when the cache controller receives a snoop request for a matching region.

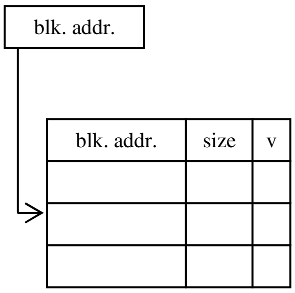


Fig. 5. Structure of a filter

4 Evaluation

In this section we evaluate VGTS. We describe our simulation environment in section 4.1. Then, in section 4.2, we present results for VGTS.

4.1 Experimental Setup

For all our evaluations, we perform full-system simulation using Gem5 [8].We model a CMP based on Alpha 21264 architecture. Each core has a private instruction cache and data cache. Table 1 shows configuration of the processor. We used Orion 2.0 [9] to estimate power consumption in bus. We also model the extra region links and the power consumed by these links. To estimate the overhead of filters, we used CACTI [16]. The technology node that we assumed is 65 nm.

We use Stamp v0.9.10 benchmark suite [7] to evaluate VGTS. We evaluated VGTS against baseline scheme. To find out the appropriate filter size, we explored a design with infinite table entries (Oracle) and four designs with 16, 32, 64, and128 entries, respectively.

We use an 8-entry SGT in our evaluations. We found that aliasing in a tag-less SGT with 8-entry is close to zero and increasing the size of SGT beyond 8 does not improve VGTS further.

Table 1. Configuration of Processors Modeled by Gem5 Full System Simulator

Benchmarks	Input Parameters
Processors	2-16 cores Alpha ISA, 2GHz
L_1 I&D Caches	64kB, 2-way associative, 64-byte line size, 1 cycle latency
Interconnect	Shared bus running at 1GHz
Orion Parameters	65 nm, V_{dd}: 1-V
L_2 Cache	Shared 2MB, 8-way associative, 64-byte line size, 10 cycles latency
Main Memory	2048MB, 100 cycles latency

4.2 Results

Figures 6 shows the normalized bus power (smaller is better) in VGTS with filter sizes being infinite (Oracle), 16, 32, 64, and 128 entries. The number of threads in benchmarks is eight. All values are normalized to bus power in the baseline scheme. By filtering redundant snoops, VGTS is able to reduce bus power. The figure shows that on average, VGTS reduces bus power by 14%, 16%, 17%, and 22% for filter sizes of 16, 32, 64, and 128, respectively. It can also be seen that even with an infinite filter table size, the bus power can be reduced by 24%, on average. This shows that VGTS with 128 entries filters is reasonably close to what an Oracle filter implementation would achieve.

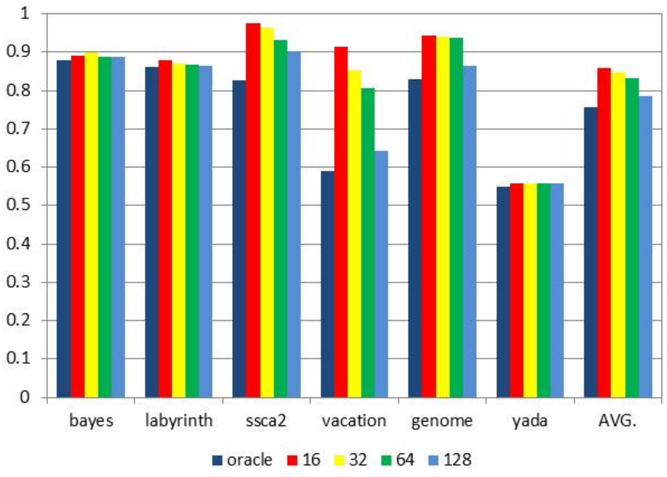


Fig. 6. Bus power reduction in VGTS relative to the baseline scheme when the number of threads is 8. For each benchmark, we use filters with infinite, 16, 32, 64, and 128 entries.

Figure 7 shows energy of snoop induced tag lookups in VGTS relative to the baseline scheme (smaller is better) when the number of threads is eight. The energy overhead of filters is included in Figure 7. In a CMP, both the processor and the bus access the L1 cache. With only one tag array, if both processor and bus need to access tag array simultaneously, one side has to stall which will degrade the overall performance. To address this problem we use two mirror tag arrays [10], one for the processor side and one for the bus side. Figure 7 shows energy reduction in the tag array on the bus side. On average, VGTS reduces energy of snoop induced tag lookups from 14% to 21% when the filter size varies between 16 and 128. An oracle filter eliminates snoop induced tag lookups by 25% which is close to a filter with 128 entries.

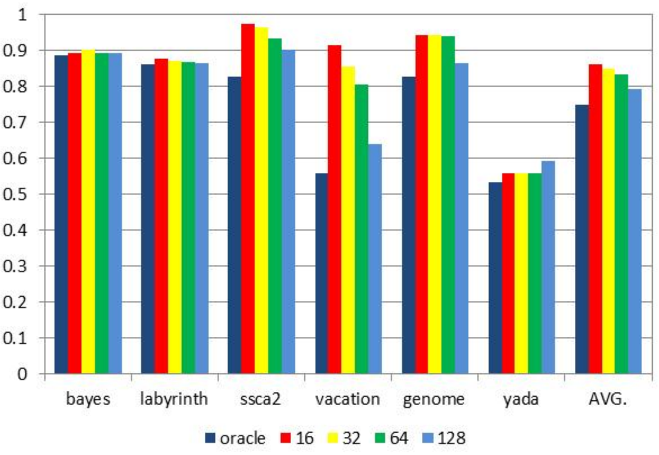


Fig. 7. Energy of snoop induced tag lookups avoided by VGTS when the number of threads is 8. For each benchmark, we use filters with infinite, 16, 32, 64, and 128 entries.

To study the scalability of our filtering proposal, we performed the above experiments for 2, 4, 8, and 16 threads. All parameters, as shown in Table 1, remain the same except the number of cores. The number of cores increases and is equal to the number of threads. Figure 8 shows the normalized bus power consumption (smaller is better) in the system with filter size equal to 64. In most of the benchmarks, power of bus changes negligibly with the number of threads. This shows that VGTS is able to maintain its power efficiency across different number of cores.

To provide better insight, we compare VGTS with RegionScout [11]. RegionScout [11] exploits coarse-grain data sharing information to reduce power of bus and snoop-induced tag lookups. In RegionScout, memory is statically divided into a number of regions. The hardware keeps record of non-shared regions and avoids broadcasting unnecessary snoops. The main limitation of RegionScout is that region size is determined statically. On the other side, VGTS adjusts snoop granularity dynamically and based on applications' behavior. Figure 9 shows power of bus and number of snoop-induced tag lookups in RegionScout. We change region size from 2KB to 16KB [11]. The number of threads is 8 and filter size is 128-entry in both RegionScout and VGTS. While RegionScout improves power of bus (Figure 9.a.) it increases snoop-induced tag lookups significantly (Figure 9.b.). In Labyrinth, RegionScout increases snoop-induced tag lookups up to 461 times. The main reason that RegionScout falls behind VGTS is that RegionScout decides on snoop granularity once and uses it across all applications. However, VGTS dynamically changes snoop granularity based on pattern of addresses generated by transactions and is able to reduce both power of bus and snoop-induced tag lookups.

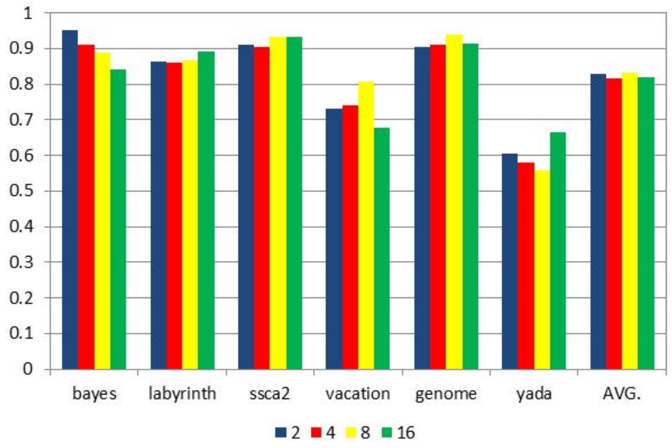
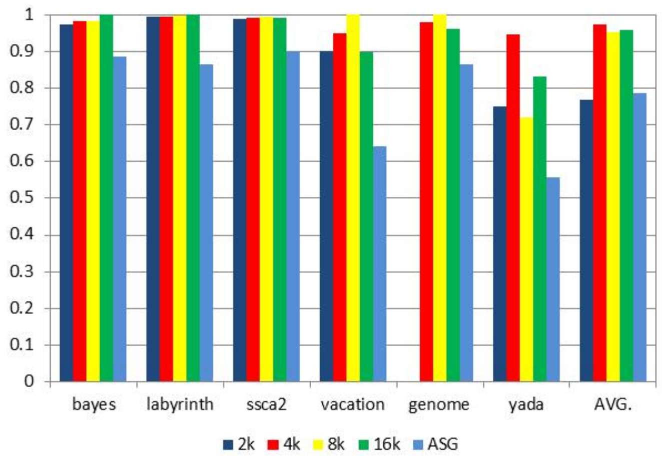


Fig. 8. Power of bus in VGTS relative to the baseline scheme. For each benchmark, the number of threads varies between two and 16.

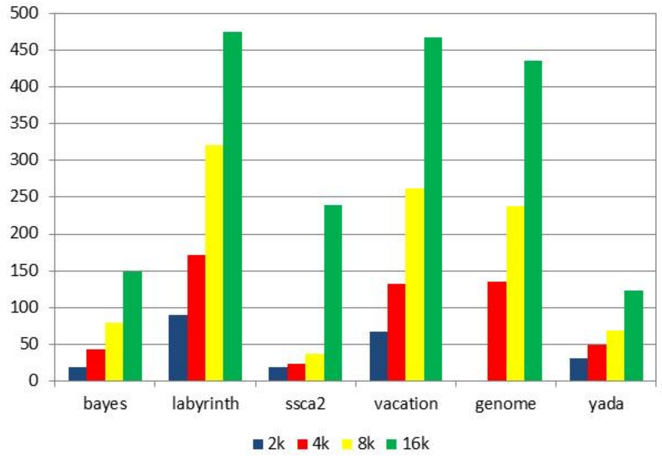
In the interest of space, we do not show a Figure for runtime in VGTS. VGTS reduces runtime of Stamp applications by 3.1% on average with maximum slowdown of 0.4%.

5 Related Work

Ferri et al. [19] proposed an energy efficient transactional memory design for embedded processors. They used a fully-associative Transactional Cache (TC) to hold speculative values generated in transactional sections. In the event of overflow in TC, the system enters an inefficient serial mode to allow the transactions complete. To reduce power consumption of memory hierarchy, they embedded a new mechanism in the memory hierarchy. When the mechanism is triggered, the contents of the TC are flushed into the traditional cache hierarchy allowing TC to power down. In a subsequent work, Ferri et al. [18] investigated alternative cache architecture for transactional values. This architecture aims at reducing the likelihood of cache overflow with the additional goal of reducing overall energy consumption. L1 cache is the primary storage space for holding both transactional data and non-transactional data. In addition, a small victim cache holds transactional data evicted from the L1 cache due to conflict misses. They show that the victim cache reduces pressure from sharing and improves energy-delay product.



a)



b)

Fig. 9. a) Bus power in RegionScout and VGTS relative to the baseline scheme. b) Snoop induced tag lookups in RegionScout relative to the baseline scheme. In both RegionScout and VGTS, number of threads is 8 and size of filters is 128-entry.

Gaona-Ramirez et al. [4] compared performance and energy of two well-known HTM systems: LogTM-SE [3] and TCC [17]. These two HTMs employ opposite policies for data versioning and conflict management. LogTM-SE uses eager policy and TCC use lazy policy for data versioning and conflict management. Gaona-Ramirez et al. showed that although on average TCC beats LogTM-SE, there are considerable deviations in performance depending on the particular characteristics of each application. Contention in LogTM-SE results in a large number of either stalled or aborted transactions depending on their write sets interactions. This behavior increases network traffic due to the persistent stall process. On the other side, TCC guarantees that at least one transaction will be able to commit in the presence of contention.

VGTS is different from all the above work since it focuses on cache coherence protocol to improve power consumption in HTMs.

A number of prior mechanisms relied on the tendency of coherence requests to reduce power in shared memory multiprocessors. Moshovos et al. proposed Jetty [12] to reduce power consumption in L2 cache. They exploited a filter between L2 cache and bus to skip snoops that would miss in L2 caches. VGTS is different since it focuses on regions of consecutive addresses to optimize power of interconnect and caches.

Ballapuram et al. [13] exploited the semantics of variables in a program to optimize snoops in a shared memory environment. Stacks are local to threads and they are not visible to the outside world. Conventional cache coherence protocols do not differentiate private and shared data. Ballapuram et al. proposed two techniques to eliminate snoop probes for stacks: Selective Snoop Probe (SSP) and Essential Snoop Probe (ESP). SSP is implemented in hardware and ESP is implemented in compiler. The advantage of using the SSP is that previously compiled binaries can benefit from this technique without recompilation. However, as there is no information provided by the software, the energy savings achieved is limited. On the other hand, the ESP technique lets the compiler takes full advantage of programs semantics to achieve higher energy savings. SSP and ESP can be combined with VGTS to reduce snoop power associate with stack regions.

Lotfi-Kamran et al. [5] investigated power of coherence directories when running commercial server and scientific workloads. They found that a significant fraction of directory power is dissipated on unnecessary lookups. They proposed TurboTag, a filtering mechanism to skip unnecessary directory lookups. VGTS is different since a requesting node can determine in advance that a request would miss in all the other nodes. With TurboTag, every node still broadcasts snoop requests to remote nodes. Advance knowledge of global region misses allows optimization of bus power that is impossible with TurboTag.

6 Conclusion

In this paper, we proposed and evaluated VGTS which is a novel snoop filtering mechanism for HTMs. We observed that many transactional snoop requests not only do not hit in any remote caches but also do not find any other blocks in a much larger surrounding region. VGTS is a speculative approach and monitors transactional reads and writes to decide on snoop granularity. VGTS exploits a set of filters to keep track of non-shared regions. These filters are transparent to the programmers and do not impose any limits on content of caches. We evaluated the design of VGTS in a full-system simulator and found that VGTS reduces power of bus and avoids many unnecessary cache snoops.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: The Proceedings of the Twentieth Annual International Symposium on Computer Architecture (1993)

2. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based transactional memory. In: The Proceedings of HPCA, pp. 254–265 (2006)
3. Yen, L., Bobba, J., Marty, M.R., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: LogTM-SE: Decoupling hardware transactional memory from caches. In: HPCA-13, pp. 261–272 (February 2007)
4. Gaona-Ramírez, E., Gil, J.R.T., Fernández, J., Acacio, M.E.: Characterizing Energy Consumption in Hardware Transactional Memory Systems. In: The Proceedings of SBAC-PAD 2010, pp. 9–16 (2010)
5. Lotfi-Kamran, P., Ferdman, M., Crisan, D., Falsafi, B.: TurboTag: lookup filtering to reduce coherence directory power. In: The Proceedings of ISLPED, pp. 377–382 (2010)
6. Intel Corp. Intel Architecture Instruction Set Extensions Programming Reference, 319433-012a edition (February 2012)
7. Minh, C.C., Trautmann, M., Chung, J.W., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In: The Proceedings of ISCA (June 2007)
8. Binkert, N., Dreslinski, R., Hsu, L., Lim, K., Saidi, A., Reinhardt, S.: The M5 simulator: Modeling networked systems. IEEE Micro 26(4), 52–60 (2006)
9. Kahng, A.B., Li, B., Peh, L., Samadi, K.: Orion 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In: The Proceedings of DATE (2009)
10. Culler, D.E., Singh, J., Gupta, A.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufman Publishers, San Francisco (1999)
11. Moshovos, A.: RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In: The Proceedings of ISCA, Washington, DC, USA (2005)
12. Moshovos, A., Memik, G., Falsafi, B., Choudhary, A.: JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In: The Proceedings of HPCA (2001)
13. Ballapuram, C.S., Sharif, A., Lee, H.S.: Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors. In: The Proceedings of ASPLOS (2008)
14. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: The Proceedings of the 20th International Symposium on Distributed Computing, pp. 194–208 (September 2006)
15. Chung, J., Yen, L., Diestelhorst, S., Pohlack, M., Hohmuth, M., Grossman, D., Christie, D.: ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory. In: Proceedings of MICRO, Atlanta, Ga (December 2010)
16. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: CACTI 6.0: A Tool to Model Large Caches, Technical report, Hewlett Packard (2009)
17. Casper, H.C.J., Carlstrom, B.D., McDonald, A., Minh, C.C., Baek, W., Kozyrakis, C., Olukotun, K.: A scalable, non-blocking approach to transactional memory. In: HPCA-13, pp. 97–108 (February 2007)
18. Ferri, C., Wood, S., Moreshet, T., Bahar, R.I., Herlihy, M.: Energy and Throughput Efficient Transactional Memory for Embedded Multicore Systems. In: The Proceedings of Hi-PEAC 2010, Pisa, Italy, January 25-27 (2010)
19. Ferri, C., Viescas, A., Moreshet, T., Bahar, R.I., Herlihy, M.: Energy Implications of Transactional Memory for Embedded Architectures. In: Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods, EPHAM 2008 (April 2008)

Topic 5: Parallel and Distributed Data Management
(Introduction)

Maria S. Perez-Hernandez, André Brinkmann, Stergios Anastasiadis, Sandro Fiore, Adrien Lébre, and Kostas Magoutis

Topic Committee

Nowadays we are facing an exponential growth of new data that is overwhelming the capabilities of companies, institutions and the society in general to manage and use it in a proper way. Ever-increasing investments in Big Data, cutting edge technologies and the latest advances in both application development and underlying storage systems can help dealing with data of such magnitude. Especially parallel and distributed approaches will enable new data management solutions that operate effectively at large scale.

Topic 5 sought papers covering all the aspects of parallel and distributed data management, from the analysis layer (knowledge discovery, data and data stream mining, information retrieval) to the infrastructure layer (storage systems, data-intensive clouds and grids, peer-to-peer systems, multi-core architectures) as well as parallel and distributed solutions (parallel and distributed databases, transactions and query processing, mobile data applications, large data sets, security and privacy in data management).

Each submission in Topic 5 was reviewed by at least four reviewers. Finally two papers have been selected. Both papers propose approaches that try to increase the performance of state-of-the-art solutions. The paper entitled "Multi-level Clustering on Metric Spaces using a multi-GPU platform" by R.J. Barrientos, C. Tenllado, M. Prieto Matias and P. Zezula describes multi-GPU metric space techniques capable to perform similarity search in large datasets. The paper entitled "The Contention-Friendy Tree" by V. Gramoli and M. Raynal proposes a lock-based concurrent binary tree using a methodology for writing concurrent data structures, which limits the high contention of nowadays multicore environments.

We would like to sincerely thank all the authors for their submissions, the Euro-Par 2013 Organizing Committee for their valuable help and the reviewers for their excellent review work. All of them have contributed to make this topic and Euro-Par an excellent forum to discuss parallel and distributed approaches to big data challenges.

Multi-level Clustering on Metric Spaces Using a Multi-GPU Platform[⋆]

Ricardo J. Barrientos[1], José I. Gómez[1], Christian Tenllado[1], Manuel Prieto Matias[1], and Pavel Zezula[2,⋆⋆]

[1] Architecture Department of Computers and Automatic, ArTeCS Group, Complutense University of Madrid, Madrid, España
ribarrie@ucm.es

[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic
zezula@fi.muni.cz

Abstract. The field of similarity search on metric spaces has been widely studied in the last years, mainly because it has proven suitable for a number of application domains such as multimedia retrieval and computational biology, just to name a few. To achieve efficient query execution throughput, it is essential to exploit the intrinsic parallelism in respective search algorithms. Many strategies have been proposed in the literature to parallelize these algorithms either on shared or distributed memory multiprocessor systems. More recently, GPUs have been proposed to evaluate similarity queries for small indexes that fit completely in GPU's memory. However, most of the real databases in production are much larger. In this paper, we propose multi-GPU metric space techniques that are capable to perform similarity search in datasets large enough not to fit in memory of GPUs. Specifically, we implemented a hybrid algorithm which makes use of CPU-cores and GPUs in a pipeline. We also present a hierarchical multi-level index named *List of Superclusters* (*LSC*), with suitable properties for memory transfer in a GPU.

Keywords: Similarity Search, Metric Spaces, GPU, Range queries.

1 Introduction

Similarity search has been widely studied in recent years and it is becoming more and more relevant due to its applicability in many important areas [6]. It is often undertaken by using metric-space techniques on large databases whose objects are represented as high-dimensional vectors. A distance function exists and operates on those vectors to determine how similar the objects are to a given query object. A range search with radius r for a query q, represented as (q, r), is the operation that obtains from the database the set of objects whose distance to the query object q is not larger than the radius r.

Efficient range searches, which are usually dominated by expensive distance evaluations, are crucial for the success of many applications. In fact, the range

[⋆] This research was funded by the Spanish government's research contracts TIN2008-005089, TIN2012-32180 and the Ingenio 2010 Consolider ESP00C-07-20811.

[⋆⋆] Supported by the grant GACR P103/12/G084.

search operation can be considered as a basic search kernel since it is a commonly used component of more complex search operations, such as the *nearest neighbors search*. In the current technological context, one of the most promising alternatives for the acceleration of this operation is the exploitation of its intrinsic parallelism on Graphics Processing Units (GPUs). Range searches exhibit different levels of parallelism: we can process in parallel many queries, many distances from a given query or even exploit the parallelism in the distance operation itself. This feature matches well with the architecture of the GPU and Multi-GPU systems. However, these architectures have complex memory hierarchies and it has been empirically shown that their efficient exploitation is one of the key elements for the acceleration of many applications.

Previous related work, which focuses on search systems devised to solve large streams of queries, has shown that conventional parallel implementations for clusters and multi-core systems, that exploit coarse-grained inter-query parallelism, are able to improve query throughput by employing index data structures constructed off-line upon the database objects [8]. These index structures are used to perform an efficient filtering on the database and reduce the search space. However, their use introduces a complex and irregular memory access pattern in the search algorithm, making it very inefficient for the GPU memory system. The cost of the additional data transfers introduced by using the index can hide the benefits of keeping the database objects smartly indexed.

In this paper we propose the development of two efficient pipeline strategies for coordinating the CPU and the GPU, which is able to hide most of the CPU-GPU data transfer latency. We also propose a new hierarchical index (LSC), which fits very well into these pipelined strategies: the CPU discards elements at the top level of hierarchy and the GPU completes the work using the low level of the index hierarchy. In addition, we have analyzed the impact of the index structure that we used, and the size and nature of the database.

The remaining of the paper is as follows. Section 2 gives some background on similarity search and metric-space databases, and summarizes some previous related work. In Section 3 we describe our proposals to deal with large databases on a multi-GPU platform. Section 4 present the experimental results of our analysis, and finally Section 5 summarizes the main conclusions of this work.

2 Similarity Search Background and Related Work

Searching similar objects from a database to a given query object is a problem that has been widely studied in recent years. The solutions are based on the use of a data structure that acts as an index to speed up the processing of queries. Similarity can be modeled as a metric space as stated by the following definitions:

Metric Space [13]: A *metric space* (X, d) is composed of an universe of valid objects $\mathbb{X}$ and a *distance function* $d \ : \ \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$ defined among them. The distance function determines the similarity between two given objects and holds several properties such as strict positiveness ($d(x, y) > 0$ and if $d(x, y) = 0$ then $x = y$), symmetry ($d(x, y) = d(y, x)$), and the triangle inequality ($d(x, z) \leq$

$d(x, y)+d(y, z)$). The finite subset $\mathbb{U} \subset \mathbb{X}$ with size $n = |\mathbb{U}|$, is called the database and represents the collection of objects of the search space. There are two main queries of interest, kNN and *range* queries.

Range Query [6]: The goal is to retrieve all the objects $u \in \mathbb{U}$ within a radius r of the query q (i.e. $(q, r)_d = \{u \in \mathbb{U}/d(q, u) \leq r\}$).

The k Nearest Neighbors (kNN): The goal is to retrieve the set $kNN(q) \subseteq \mathbb{U}$ such that $|kNN(q)| = k$ and $\forall u \in kNN(q), v \in \mathbb{U} - kNN(q), d(q, u) \leq d(q, v)$.

The solution of range queries are used as basis to solve kNN queries, and because of this, the present paper is focused on solving range queries. To avoid as many distance computations as possible, many indexing approaches have been proposed. We have focused on the *List of Clusters* (*LC*) [5] index, since (1) it is one of the most popular non-tree structures that are able to prune the search space efficiently and (2) it holds its index on dense matrices which are very convenient data structures for mapping algorithms onto GPUs. We are not affirming that this index is the most suitable for GPU, but its properties make it a good candidates to become it. Besides, finding the best metric index for GPU is not a target of this paper; we mainly want to show the high performance achieved using a metric index on GPU compared to sequential and traditional multi-core approaches.

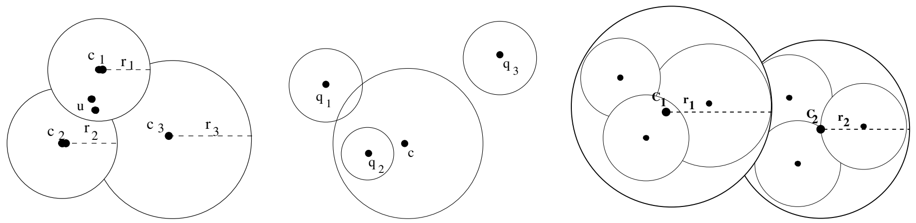
In [11,2] the authors propose solutions for similarity search using a GPU card. All these papers take the initial assumption that the whole index fits on GPU memory, with capacity of a few GiB. In this paper we propose solutions to deal with large databases, which is usually the real case, where the databases fit just partially on the GPU memory.

In the following subsections we explain the construction of the *LC* index and is described how range queries are solved using it.

2.1 List of Clusters (LC)

This index [4,5] can be implemented dividing the space in two different ways: taking a fixed radius for each partition or using a fixed size. In this paper, to ensure good load balance in a parallel platform, we consider partitions with a fixed size of K elements, thus the radius r_c of a cluster with center c is the maximum distance between c and its K-nearest neighbor.

The *LC* data structure is formed from a set of centers (objects). The construction procedure (illustrated in Figure 1(a)) is roughly as follows. We (randomly) chose an object $c_1 \in \mathbb{U}$ which becomes the first center. This center determines a cluster (c_1, r_1, I_1) were I_1 is the set $k\text{NN}_{\mathbb{U}}(c_1, K)$ of K-nearest neighbors of c_1 in $\mathbb{U}$ and r_1 is the distance between the center c_1 and its K-nearest neighbor in $\mathbb{U}$ (r_1 is called *covering radius*). Next, we choose a second center c_2 from the set $E_1 = \mathbb{U} - (I_1 \cup \{c_1\})$. This second center C_2 determines a new cluster (c_2, r_2, I_2) where I_2 is the set $k\text{NN}_{E_1}(c_2, K)$ of K-nearest neighbors of c_2 in E_1 and r_2 is the distance between the center C_2 and its K-nearest neighbor in E_1. Let $E_0 = \mathbb{U}$, the process continues in the same way choosing each center c_n ($n > 2$) from the set $E_{n-1} = E_{n-2} - (I_{n-1} \cup \{c_{n-1}\})$, till E_{n-1} is empty.



(a) Illustration of the *LC* with three center: c_1, c_2 and c_3.

(b) Cases of searching.

(c) Illustration of the *LSC* with three superclusters.

Fig. 1. List of Cluster (LC)

Note that, a cluster created first during construction has preference over the following ones when their corresponding covering radius overlap. All the elements that lie inside the cluster corresponding to the first center c_1 are stored in it, despite that they may also lie inside the subsequent clusters (Figure 1(a)). This fact is reflected in the search procedure. Figure 1(b) illustrates all the situations that may arise between a range query (q, r) and a given cluster.

During the processing of a range query (q, r), the idea is that if the first cluster is (c_1, r_1, I_1), we evaluate $d(q, c_1)$ and add c_1 to the result set if $d(q, c_1) \leq r$. Then, we scan exhaustively the objects in I_1 only if the range query (q, r) intersects the cluster with center c_1 and radius r_1, i.e. only if $d(q, c_1) \leq r_1+r$ (q_1 in Figure 1(b)). Next, we continue with the remaining set of clusters following the construction order. However, if a range query (q, r) is totally contained in a cluster (c_i, r_i, I_i), i.e. if $d(q, c_i) \leq r_i - r$, we do not need to traverse the remaining clusters, since the construction process of the *LC* ensures that all the elements that are inside the query (q, r) have been inserted in I_i or in a previous clusters in the building order (q_2 in Figure 1(b)). In [5], authors analyzed different heuristics for selecting the centers, and showed experimentally that the best strategy is to choose the next center as the element that maximizes the sum of distances to previous centers. This is the heuristic used in our work.

2.2 List of Superclusters (LSC)

We propose a hierarchical multi-level *LC*, named *List of Superclusters* (*LSC*) that takes into account the organization of the GPU memory.

The construction of the *LSC* has two steps. First, based on the construction procedure of the *LC* with fixed size of K elements, we get N clusters of size K. Each i-th cluster is composed by its center C_i, covering radius r_i and the K nearest elements to C_i ($k\text{NN}_{\mathbb{U}}(C_i, K)$). These N clusters are named *superclusters* and integrate the first level of the hierarchy. In the second step, we create a *LC* index into each supercluster with their own elements following the construction procedure of the *LC* (Section 2.1).

To process a range query (q, r) we have to calculate the distances $d(C_i, q), i \in [1, N]$ between the centers of the superclusters and the query. We add the center

C_i to the result set if $d(C_i, q) \leq r$. Using triangle inequality we try to discard each supercluster (i.e. if $d(C_i, q) \leq r_i + r$), and for each non-discarded supercluster, we apply the searching procedure of the *LC* over its elements.

If we think in the clusters as unit of transfers to device memory in the GPU, the *LSC* makes better use of the bandwidth, because each non-discarded supercluster is a set of clusters that must be processed. The Figure 1(c) shows an example of a *LSC* index.

3 Strategies to Process Similarity Queries

In this section we describe our proposed methods to process range queries on a multi-GPU platform. All the following strategies are designed assuming that the database does not fit in device memory, i.e. just a subset of the clusters can be loaded at a time. Thus, for every batch of queries, the whole database may be (potentially) loaded to the GPU to perform the search.

In all the following strategies the kernels are launched with one CUDA Block per query. Each CUDA Block processes a different query, which has several advantages, such as, to be able to synchronize the threads that solve the same query, to exploit coarse-grained parallelism solving a batch of queries in parallel, or to exploit fine-grained parallelism solving a query with a set of threads.

3.1 1-Stage Strategy

In [2], assuming that the whole dataset fits into device memory, a multi-GPU strategy was proposed, using the *LC* index, and called *1-Stage*. We used the *1-Stage* strategy as baseline, but in this case we load in device memory just a percentage of the clusters at a time. The aim of this strategy is to solve each query in just one *kernel* in the GPU, avoiding to launch consecutive kernels and copying data to communicate them.

We used one CPU-thread per GPU, each one controls a different GPU. The centers, covering radius and their respective clusters are distributed among the GPUs (in a circular manner), and because of this, each query must be processed by all the GPUs.

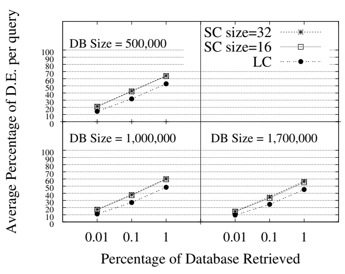
The discard of clusters and searching on them is performed inside the kernel, composed by two steps. (1) Each thread performs a distance evaluations between a different center and the query (corresponding to the current CUDA Block), and stores in shared memory a variable indicating if the cluster is discarded. (2) According to the variables in shared memory, all the non-discarded clusters are distributed (in a circular manner) among the threads, and each thread calculates the distance between an element and the query in the same kernel.

Due to the memory restrictions of space in the GPU, we load N centers and Q queries in device memory, and we process them iteratively. In the first iteration we process a batch of Q queries with N clusters, in the second iteration we load the next N cluster and process the same Q queries, and so on, until all clusters were loaded. The same process is repeated with all the batches of queries.

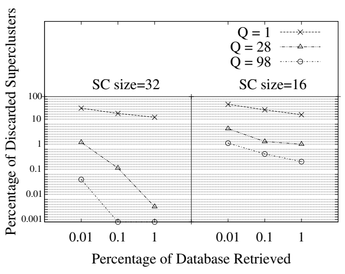
3.2 List of Superclusters on GPU

We created the *LSC* index with the method described in Section 2.2. We set the parameters of the *LSC* to create versions with N and $N/2$ clusters per supercluster, where N is the maximum number of cluster allowed in device memory. After loading a complete supercluster, a kernel is launched with Q CUDA Blocks (Q is the quantity of queries of the current query batch) to search into it.The kernel is composed of three steps as follows. (1) The first D threads cooperate to get the distance d between the center of the supercluster and the query, where D is the dimension of the elements. (2) As we described in Section 2.2, we use the distance d and the triangle inequality property to try to discard the supercluster. (3) If the supercluster is not discarded, then we search in the *LC* index inside the supercluster, with the method used by 1-Stage strategy (Section 3.1).

The Figure 2 shows results in sequential computation of the *LSC* against the *LC*, using the datasets described in Section 4. The Figure 2(a) compares the average of distance evaluations between the *LSC* and *LC*, where the *LC* always takes advantage. To try to know the efficiency for discarding of the *LSC*, the Figure 2(b) exposes the average of the percentage of discarded superclusters, processing query batches of different sizes (represented by Q in the graph). In this figure a supercluster is considerd discarded just if its covering radius does not intersect any of the Q queries of the current batch query. Therefore, the larger the Q, the less the probability of discarding a cluster. We observe that trying to discard superclusters taking account query batches of size 98 or higher, the *LSC* is able to discard less than 2% of the superclusters. The Figure 2(a) represents values with Q=1, and we can observe in the Figure 2(b) that the *LSC* reaches 69% of discard for Q=1 with the database of 500,000 elements, but even with this, the *LC* performs less distance evaluations. This is because the low discard of elements in the non-discarded superclusters by the *LSC*, close to 17%.



(a) Average of the distance evaluations (D.E.) per query, between *LSC* of 16 and 32 clusters per supercluster and the *LC* index.



(b) Discarding of supercluster in the *LSC* with 16 and 32 clusters per supercluster, using the database of 500,000 elements.

Fig. 2. Results in sequential computation of *LSC* and *LC*

Despite the total number of distance evaluations increases, in Section 4 we show that our implementation of *LSC* in GPU outperforms the *LC* one. This counterintuitive behavior is largely explained due to the higher transfer efficiency of the *LSC*. The minimum unit of discarding in the *LC* is a cluster and in the *LSC* is a supercluster, which also are the minimum unit of transfer. Therefore, the layout of the data to be transfered from CPU to GPU gets much more irregular when using *LC*, and the available bandwidth is poorly exploited.

3.3 Building a CPU-GPU Pipeline

To minimize the number of transfers to GPU and in order to increase the degree of parallelism, we developed a hybrid pipeline between CPU and GPU, where the CPU helps to discard some elements to avoid them to be transfered to the GPU. We used P CPU-threads, where P is the quantity of CPU-cores of the machine, and from those P the first G threads ($G < P$) manage a different GPU.

Considering that N is the allowed quantity of clusters in device memory, and Q is the quantity of the current batch query, the steps of the pipeline are as follows. (1) In CPU, we try to discard N clusters of the *LC* just using the center and covering radius of the clusters. For this we distribute (circularly) the clusters among the threads, and each thread discards its cluster if its covering radius does not intersect with any of the Q queries. (2) We load in GPU just the non-discarded clusters according to the previous step, and we process the queries with them. (3) While the second step (with the first G threads) is in execution, the first step (with the rest of the threads) is in execution too, but attempting to discard the next N clusters.

We implemented this pipeline for both *LC* and *LSC* indexes. In the case of the *LC* the threads that run on CPU cores try to discard clusters, and in the *LSC* they try to discard superclusters. As result, with this pipeline we get to load less quantity of clusters (or superclusters) in GPU.

3.4 Exploiting CUDA Asynchronous Copies

`cudaMemcpyAsync` allows to perform transfers to (and from) device memory while a kernel is in execution. This is possible by using CUDA *streams*, where each CUDA stream can contain a sequence of instructions. Copies and kernels from different streams can be executed at the same time.

Starting from the base non-pipelined implementation, we exploit the asynchronous copies for both *LC* and *LSC* indexes. If N is the quantity of clusters allowed in device memory, then we create two CUDA streams, and each stream is composed of the following instructions. (1) To copy $N/2$ clusters to device memory, and in the case of the *LSC* to copy one supercluster of $N/2$ clusters. (2) launch a kernel to process the queries with the loaded clusters (or supercluster). We create just two CUDA streams and not more, because this quantity makes a good balance in running time between copies and kernels, which effectively builds a two stage transfer - kernel pipeline.

We always copy the clusters of the LC or supeclusters of the LSC with just one `cudaMemcpyAsync` because the elements of a cluster or a supercluster are contiguous in the database; this is key to efficiently exploit the huge bandwidth between CPU and GPU, since short transfers cannot hide the initial latency.

3.5 Multi-pipeline Strategy

Our final proposal combined the two previous strategies in one *multi-pipeline strategy.* We use the LSC index (Section 3.2), and we create P CPU threads, one per CPU-core (Section 3.3), leaving G threads in charge of G GPUs ($G < P$). Each GPU create two CUDA streams to build a pipeline between copies and kernels (Section 3.4). The Figure 3 shows a scheme of this strategy, which is composed by three steps separated by OpenMP barriers, the steps are as follows. (1) Discard of superclusters with threads running on CPU-cores. (2) To copy the ID of the non-discarded superclusteres to be read by the threads in charge of GPUs. (3) Each GPU create two CUDA streams, and each stream copy to device memory one supercluster per `cudaMemcpyAsync`, and after a supercluster is loaded in device memory, immediately is launched a kernel to search on it. The steps 1 and 3 are executed on the same time, because the pipeline method described in Section 3.3.

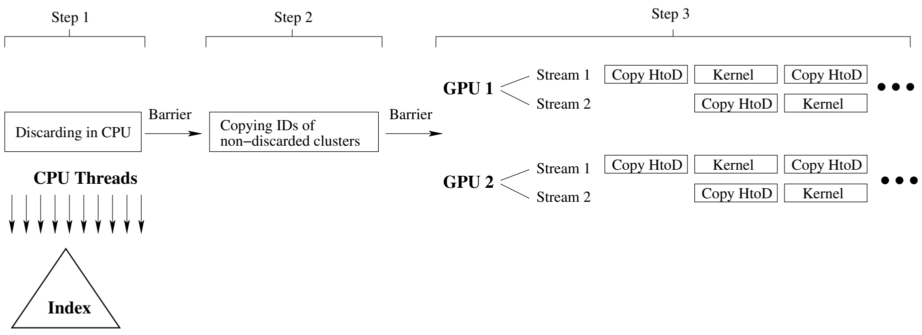


Fig. 3. Scheme of the multi-pipeline strategy

4 Experimental Results

All our GPU experiments were carried out on two NVIDIA Tesla M2070, and each one is shipped with 14 multiprocessors, 32 cores per multiprocessor, 48KB of shared memory and 5GB of device memory. The host CPU is a 2xIntel Quad-Xeon processor of 2.66GHz with 16 GB of RAM.

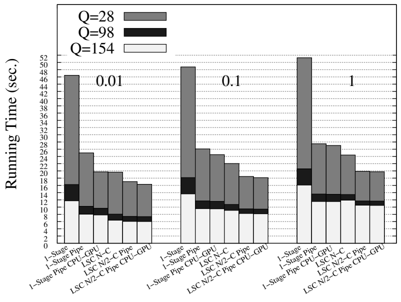
We have used as reference database the *CoPhIR* (Content-based Photo Image Retrieval) dataset [3]. This consists of metadata extracted from the Flickr photo sharing system. It is a collection of 106 million images containing for each image five MPEG-7 visual descriptors, specifically Scalable Colour, Colour Structure, Colour Layout, Edge Histogram, and Homogeneous Texture. For the purposes of this paper, we just used the *Colour Structure* MPEG-7 image feature, which represents a 64 dimensional vector for each image. We used the *Euclidean distance* as a distance measure. As in previous papers [7,9], the radii used were those that retrieve on average the 0.01%, 0.1% and 1% of the elements of the database per query.

To our knowledge, there is not a public and real query log for similarity search in images. But recently, a public website was presented in [10]. It applies the MUFIN [12] search engine for images of CoPhIR dataset and is used by many users all round the world. From this website, we got our query log, which represents the processed queries by several days. We used 30,000 queries that are represented by its *Colour Structure* MPEG-7 image feature of dimension 64. We have made this query log public [1].

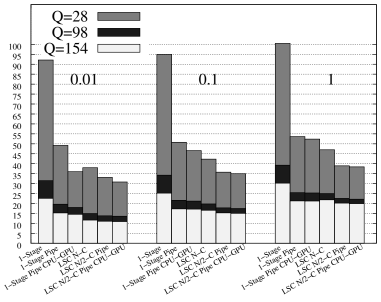
The Figures 4(a), 4(b) and 4(c) presents the running time of all the strategies described in Section 3. The first column stands for the 1-Stage strategy (Section 3.1), which is implemented using the *LC* index. After loading N clusters a kernel is launched to search on them (N is the number of clusters allowed in device memory). The second column (*1-Stage Pipe*) stands for the 1-Stage strategy, but using two CUDA streams (Section 3.4), therefore after loading $N/2$ clusters in device memory we launch a kernel to search on them. The third column (*1-Stage Pipe CPU-GPU*) is similar to the second one, but implementing the pipeline CPU-GPU (Section 3.3), where the threads that run on CPU-cores try to discard clusters of the *LC* in parallel with the GPUs processing of the previous batch query. The fourth column (*LSC N-C*) stands for the *LSC* index (Section 3.2), with N clusters per supercluster, and after loading a supercluster a kernel is launched. The fifth column (*LSC N/2-C Pipe*) stands for the *LSC* index with $N/2$ clusters per supercluster, and using two CUDA streams (Section 3.4), therefore after loading a supercluster by a stream, a kernel is launched by using the same stream to search on it. The last column (*LSC N/2-C Pipe CPU-GPU*) stands for the *Multi-pipeline Strategy* described in Section 3.5.

In all our experiments we always set the cluster size equal to 256, because it has been empirically proved a good parameter. Therefore each supercluster of the *LSC* index is composed by clusters of size 256. We set the clusters allowed in device memory in N=32, and we just copy the results from GPU when a batch query is completely processed. In all the strategies, we copy a cluster of the *LC* (or supercluster in case of *LSC*) with one `cudaMemcpy`, or one `cudaMemcpyAsync` in the columns labeled with *Pipe*. All the strategies that implement the asynchronous copies pipeline (Section 3.4), use page-locked (pinned) memory to transfer data. This memory allows copies to device memory in parallel with kernel processing, and also decrease the time of the copies. Therefore, to be fair we use pinned memory for the transfers in all the strategies.

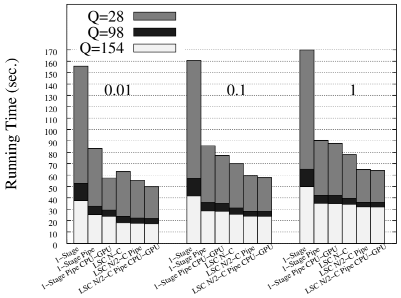
Each bar in the figures represents the running time of the corresponding strategy. For example, in the first bar of Figure 4(a), the running time of the 1-Stage strategy, processing the queries in batches of Q=28 is 46.4 seconds, with Q=98 is 16.2 seconds, and with Q=154 is 11.7 seconds. We process the queries in batches of 28, 98 and 154, because these numbers are multiples of 14, which is the



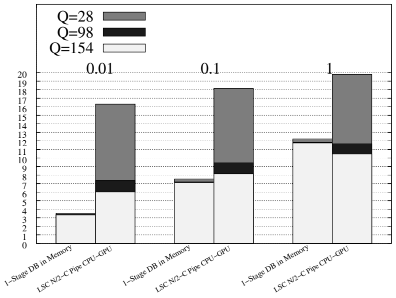
(a) DB Size = 500,000



(b) DB Size = 1,000,000



(c) DB Size = 1,700,000



(d) 1-Stage loading all the data in device memory, using the database of 500,000 elements

Fig. 4. Running time of the *LC* and *LSC* indexes combined with the pipelines described in Section 3

number of multiprocessors in our GPUs, and taking account that we are processing each query with a different CUDA Block, a multiple of 14 improves the load balance of CUDA Blocks across multiprocessors. Note that, in the worst case (if no discard is performed at the CPU level), the complete DB must be transferred from the CPU to the GPU for every batch query: 195 times for Q=154, 307 times for Q=98 and 1072 times for Q=28. It is imperative to efficiently hide this latency to attain good results.

Our baseline implementation, labeled as *1-Stage* strategy and described in section 3.1, achieves the worst performance in all the databases for all Q. The *1-Stage Pipe* strategy outperforms the previous one, because it reduces latency of the copies to device memory by using the pipeline described in Section 3.4, implemented with CUDA streams. The *1-Stage Pipe CPU-GPU* strategy outperforms the previous two, because the reduction in the quantity of clusters copied to device memory. This reduction is made by the threads running on CPU cores that calculate the distances between the centers and the batch query, avoiding to copy the discarded clusters.

The next three bars show the results for the proposed index, *List of superclusters*. The *LSC N-C* corresponds with the strategy explained in section 3.3 applied to the *LSC* index. As shown previously, this index finally performed more distance evaluations than the simpler *LC*; however, its final execution time is most of the times because the bandwidth is more efficiently managed: in each copy to device memory is transferred contiguous data of size N clusters to device memory (N is the number of clusters allowed in device memory). Next ,the *LSC N/2-C Pipe* strategy achieves better performance than the previous ones, because the implementation of the pipeline using CUDA streams, and the unit of transfers is a supercluster of $N/2$ cluster. This strategy is copying data while a kernel is in execution, hiding transfers latency. Finally, the strategy labeled as *LSC N/2-C Pipe CPU-GPU* (named as *multi-pipeline strategy* in Section 3.5) achieves the best performance. This strategy is similar to the previous one, but the threads running on CPU cores try to discard superclusters while the GPUs are processing the previous batch query.

The advantages of using the CPU-GPU pipeline (Section 3.3) in the *LSC* is more evident with Q=28, because the larger Q, the less is the discard of clusters (Figure 2(b)). This seems to indicate a certain degree of locality in the query log, which is lost when the batch is made too large. However, the much larger number of transfers due to a reduce Q does mitigate the benefits of this locality.

To complete our study, we consider the case where the whole DB actually fits in the GPUs main memory (i.e. the whole DB must just be copied once at the beginning of the process). Our smallest DB may be distributed amongst the two available GPUs, so we could compare our best implementation with this unrealistic scenario.Figure 4(d) shows the results.

Not surprisingly, the *all-fit* implementation outperforms our proposal when searching with small radius. Also as expected, in this ideal version, there is almost no penalty when reducing the Q: it does not entail further transfers, so there is not huge penalty form that side. However, it is very noticeable that, for the larger search radius (1% of the DB retrieved) our implementation actually outperforms the *all-fit* version for Q=154. With such a large radius, the discard efficiency is quite low. Thus, kernel execution times are always able to completely hide transfers penalties. Then, we only pay the latency of transferring one *supercluster* per batch of queries. The higher the Q, the lesser the number of batches and, for this example, we are able to be competitive with the *all-fit* version. For the smallest Q value, the number of not-hidden transfers increase too much (and the kernel work decreases), largely degrading performance.

5 Conclusions

In this paper we have presented a hierarchical multi-level structure, built on the *LC* index named *List of Superclusters* (*LSC*), to solve similarity queries in metric spaces. The *LSC*, which is composed by *superclusters*, has been designed to perform well on GPUs. A supercluster is made by a center, a covering radius and elements, but with the elements of each supercluster is created a *LC* index.

Grouping clusters in superclusters allows for a fast discard at CPU level and, using it as the minimal CPU-GPU transfer unit, ensures that the bandwidth is always efficiently exploited.

To study the index efficiency, we have implemented a pipelined hybrid CPU-GPU version of both the *LC* and *LSC* indexes. The CPUs perform a first round of discards for a batch query Q_i while the GPUs are finishing the processing of the previous batch, Q_{i-1}. Moreover, the CPU-GPU transfers and the GPU kernels execution is further pipelined using *streams* and asynchronous copies. The transfer latency is almost completely hidden that way; indeed, even if the complete list of clusters is copied for each batch query (except those clusters discarded by the CPU), the total exposed latency may be even lower than the experienced when transferring the complete DB just once.

Our study with a real query log for similarity search in images, shows that there exits a locality amongst queries: i.e. the sets of clusters accessed by two consecutive queries have a non null intersection. This motivates further exploration to reduce transfers by carefully scheduling queries.

References

1. Query log, `http://kataix.umag.cl/~ribarrie/Programs.html`
2. Barrientos, R., Gómez, J., Tenllado, C., Prieto, M., Marin, M.: Range query processing in a multi-gpu environment. In: 10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2012), pp. 419–426 (2012)
3. Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., Rabitti, F.: Cophir: a test collection for content-based image retrieval. CoRR abs/0905.4627 (2009), `http://cophir.isti.cnr.it`
4. Chavéz, E., Navarro, G.: An effective clustering algorithm to index high dimensional metric spaces. In: The 7th International Symposium on String Processing and Information Retrieval (SPIRE 2000), pp. 75–86. IEEE CS Press (2000)
5. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters 26(9), 1363–1376 (2005)
6. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (2001)
7. Costa, V.G., Barrientos, R.J., Marín, M., Bonacic, C.: Scheduling metric-space queries processing on multi-core processors. In: Danelutto, M., Bourgeois, J., Gross, T. (eds.) PDP, pp. 187–194. IEEE Computer Society (2010)
8. Marin, M., Ferrarotti, F., Gil-Costa, V.: Distributing a metric-space search index onto processors. In: 39th International Conference on Parallel Processing, ICPP 2010, pp. 433–442. IEEE Computer Society, San Diego (2010)
9. Navarro, G., Uribe-Paredes, R.: Fully dynamic metric access methods based on hyperplane partitioning. Information Systems 36(4), 734–747 (2011)
10. Novak, D., Batko, M., Zezula, P.: Generic similarity search engine demonstrated by an image retrieval application. In: 32nd ACM SIGIR Conference on Research and Development in Information Retrieval, p. 840. ACM, Boston (2009)

11. Uribe-Paredes, R., Arias, E., Sánchez, J.L., Cazorla, D., Valero-Lara, P.: Improving the performance for the range search on metric spaces using a multi-GPU platform. In: Liddle, S.W., Schewe, K.-D., Tjoa, A.M., Zhou, X. (eds.) DEXA 2012, Part II. LNCS, vol. 7447, pp. 442–449. Springer, Heidelberg (2012)
12. Zezula, P.: Multi feature indexing network mufin for similarity search applications. In: Bieliková, M., Friedrich, G., Gottlob, G., Katzenbeisser, S., Turán, G. (eds.) SOFSEM 2012. LNCS, vol. 7147, pp. 77–87. Springer, Heidelberg (2012)
13. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. Advances in Database Systems, vol. 32. Springer (2006)

A Contention-Friendly Binary Search Tree

Tyler Crain[1], Vincent Gramoli[2], and Michel Raynal[1,3]

[1] IRISA, Université de Rennes 1, France
[2] NICTA and University of Sydney, Australia
[3] Institut Universitaire de France
tyler.crain@inria.fr, vincent.gramoli@sydney.edu.au,
raynal@irisa.fr

Abstract. This paper proposes a new lock-based concurrent binary tree using a methodology for writing concurrent data structures. This methodology limits the high contention induced by today's multicore environments to come up with efficient alternatives to the most widely used search structures.

Data structures are generally constrained to guarantee a big-oh step complexity even in the presence of concurrency. By contrast our methodology guarantees the big-oh complexity only in the absence of contention and limits the contention when concurrency appears. The key concept lies in dividing update operations within an *eager abstract access* that returns rapidly for efficiency reason and a *lazy structural adaptation* that may be postponed to diminish contention. Our evaluation clearly shows that our lock-based tree is up to 2.2× faster than the most recent lock-based tree algorithm we are aware of.

Keywords: Binary tree, Concurrent data structures, Efficient implementation.

1 Introduction and Related Work

Today's processors tend to embed more and more cores. Concurrent data structures, which implement popular abstractions such as key-value stores [1], are thus becoming a bottleneck building block of a wide variety of concurrent applications. Maintaining the invariants of such structures, like the balance of a tree, induces contention. This is especially visible when using speculative synchronization techniques as it boils down to restarting operations [2]. In this paper we describe how to cope with the contention problem when it affects a non-speculative execution or technique.

As a widely used and studied data structure in the sequential context, binary trees provide logarithmic access time complexity given that they are balanced, meaning that among all downward paths from the root to a leaf, the length of the shortest path is not far apart the length of the longest path. Upon tree update, if the height difference exceeds a given threshold, the structural invariant is broken and a rebalancing is triggered to restructure accordingly. This threshold depends on the considered algorithm: AVL trees [3] do not tolerate the longest length to exceed the shortest by 2 whereas red-black trees [4] tolerate the longest to be twice the shortest, thus restructuring less frequently. Yet in both cases the restructuring is triggered immediately when the threshold is reached to hide the imbalance from further operations. In a concurrent context, slightly weakened balance requirements have been suggested [5], but they still require immediate restructuring as part of update operations to the abstractions.

We introduce the *contention-friendly* tree as a tree that transiently breaks its balance structural invariant without hampering the abstraction consistency in order to reduce contention and speed up concurrent operations that access (or modify) the abstraction. More specifically, we propose a partially internal binary search tree data structure implementing a key-value store, decoupling the operations that modify the abstraction (we call these *abstract operations*) from operations that modify the tree structure itself but not the abstraction (we call these *structural operations*). An abstract operation either searches for, logically deletes, or inserts an element from the abstraction where in certain cases the insertion might also modify the tree structure. Separately, some structural operations rebalance the tree by executing a distributed rotation mechanism as well as physically removing nodes that have been logically deleted.

Context. On the one hand, the decoupling of update and rebalancing dates back from the 70's [6] and was exclusively applied to trees, including B-trees [7], $\{2,3\}$-trees [8], AVL trees [9] and red-black trees [10] (resulting in largely studied chromatic trees [11, 12] whose operations cannot return before reaching a leaf). On the other hand, the decoupling of the removals in logical and physical phases is more recent [13] but was applied to various structures: linked lists [13, 14], hash tables [15], skip lists [16], binary search trees [2, 17] and lazy lists [18]. Our methodology generalizes both kinds of decoupling by distinguishing an abstract update from a structural modification.

The guarantees of some data structures, like list-based stack, are relaxed to tolerate the high concurrency induced by multicores [19]. This idea is quite different from ours. It aims at avoiding performance of some highly contended structures to drop below sequential ones whereas we aim at designing highly-concurrent structures that leverage multi-/many-cores. Finally, the corresponding solution lies in trading off atomicity for quiescent consistency, guaranteeing that the last-in-first-out policy of an access is only with respect to preceding calls when no other accesses execute concurrently. By contrast, our solution guarantees atomicity even in concurrent executions.

We have recently observed the performance benefit of decoupling accesses while preserving atomicity within speculative executions (specifically transactional memory). Our recent speculation-friendly tree splits updates into separate transactions to avoid a conflict with a rotation from rolling back the preceding insertion/removal [2]. While it benefits from the reusability and efficiency of elastic transactions [20], it suffers from the overhead of bookkeeping accesses with software transactional memory. The goal was to bound the asymptotic step complexity of speculative accesses to make it comparable to the complexity of pessimistic lock-based ones. Although this complexity is low in pessimistic executions, our new result shows that the performance of a lock-based binary search tree greatly benefits from this decoupling.

Content of the Paper. We present a contention-friendly methodology which lies essentially in splitting accesses into an eager abstract access and a lazy structural adaptation. We illustrate our methodology with a contention-friendly binary search tree. In particular, we compare its performance against the performance of the most recent practical binary search tree we are aware of [21]. Although both algorithms are lock-based binary search trees, ours speeds up the other by $2.2\times$.

2 Overview

In this section, we give an overview of the Contention-Friendly (CF) methodology by describing how to write contention-friendly data structures as we did to design a lock-free CF skip-list [22, 23]. The following section will describe how this is specifically done for the binary search tree.

The CF methodology aims at modifying the implementation of existing data structures using two simple rules without relaxing their correctness. The correctness criterion ensured here is linearizability [24]. The data structures considered are *search structures* because they organize a set of items, referred to as *elements*, in a way that allows to retrieve the unique position of an element in the structure given its key. The typical abstraction implemented by such structures is a collection of elements that can be specialized into various sub-abstractions like a set (without duplicates) or a map (that maps each element to some value). We consider *insert*, *delete* and *contains* operations that, respectively, inserts a new element associated to a given key, removes the element associated to a given key or leaves the structure unchanged if no such element is present, and returns true if there is element associated to a given key or false if such an element is absent. Both inserts and deletes are considered *updates*, even though they may not modify the structure.

The key rule of the methodology is to decouple each update into an *eager abstract modification* and a *lazy structural adaptation*. The secondary rule is to make the removal of nodes selective and tentatively affect the less loaded nodes of the data structure. These rules induce slight changes to the original data structures that result in a corresponding data structure that we denote using the *contention-friendly* adjective to differentiate them from their original counterpart.

2.1 Eager Abstract Modification

Existing search structures rely on strict invariants to guarantee their big-oh (asymptotic) complexity. Each time the structure gets updated, the invariant is checked and the structure is accordingly adapted as part of the same operation. While the update may affect a small sub-part of the abstraction, its associated restructuring can be a global modification that potentially conflicts with any concurrent update, thus increasing contention.

The CF methodology aims at minimizing such contention by returning eagerly the modifications of the update operation that make the changes to the abstraction visible. By returning eagerly, each individual process can move on to the next operation prior to adapting the structure. It is noteworthy that executing multiple abstract modifications without adapting the structure does no longer guarantee the big-oh step complexity of the accesses, yet, as mentioned in the Introduction, such complexity may not be the predominant factor in contended executions.

A second advantage is that removing the structural adaption from the abstract modification makes the cost of each operation more predictable as operations share similar cost and create similar amount of contention. More importantly the completion of the abstract operation does not depend on the structural adaptation (like they do in existing algorithms), so the structural adaptation can be performed differently, for example, using global information or being performed by separate, unused resources of the system.

2.2 Lazy Structural Adaptation

The purpose of decoupling the structural adaptation from the preceding abstract modification is to enable its postponing (by, for example, dedicating a separate thread to this task, performing adaptations when observed to be necessary), hence the term “lazy” structural adaptation. The main intuition here is that this structural adaptation is intended to ensure the big-oh complexity rather than to ensure correctness of the state of the abstraction. Therefore the linearization point of the update operation belongs to the execution of the abstract modification and not the structural adaptation and postponing the structural adaptation does not change the effectiveness of operations.

This postponing has several advantages whose prominent one is to enable merging of multiple adaptations in one simplified step. Only one adaptation might be necessary for several abstract modifications and minimizing the number of adaptations decreases accordingly the induced contention. Furthermore, several adaptations can compensate each other as the combination of two restructuring can be idempotent. For example, a left rotation executing before a right rotation at the same node may lead back to the initial state and executing the left rotation lazily makes it possible to identify that executing these rotations is useless. Following this, instead of performing rotations as a string of updates as part of a single abstract operation, each rotation is performed separately as a single local operation, using the most up to date balance information.

Although the structural adaptation might be executed in a distributed fashion, by each individual updater thread, one can consider centralizing it at one dedicated thread. Since these data structures are designed for architectures that use many cores, performing the structural adaptation on a dedicated single separate thread leverages hardware resources that might otherwise be left idle.

Selective Removal. In addition to decoupling level adjustments, removals are preformed selectively. A node that is deleted is not removed instantaneously but is marked as deleted. The structural adaptation then selects among these marked nodes those that are suitable for removal, i.e., whose removal would not induce high contention. This selection is important to limit contention. Removing a frequently accessed node requires locking or invalidating a larger portion of the structure. Removing such a node is likely to cause much more contention than removing a less frequently accessed one. In order to prevent this, only nodes that are marked as deleted and have at least one of their children as an empty subtree are removed. In addition, marked deleted nodes that have not been physically removed can then be added back into the abstraction by simply unmarking them during an *insert* operation. This leads to less contention, but also means that certain nodes that are marked as deleted may not be removed. In similar, partially external/internal trees, it has already been observed that only removing such nodes [2], [21] results in a similar sized structure as existing algorithms.

3 The Contention-Friendly Tree

The CF tree is a lock-based concurrent binary search tree implementing classic insert/delete/contains operations. Each of its nodes contains the following fields: a key *k*, pointers *l* and *r* to the left and right child nodes, a *lock* field, a *del* flag indicating if the node has been logically deleted, a *rem* flag indicating if the node has been physically removed (note that the *rem* flag an additional third state (by-left-rot) which evaluates to true) , and the integers *left-h*, *right-h* and *local-h* storing the estimated height of the node and its subtrees used in order to decide when to perform rotations.

This section will now describe the CF tree algorithm by first describing three specific CF modifications that reduce contention during traversal, followed by a description of the CF abstract operations.

3.1 Avoiding Contention during Traversal

Each abstract operation of a tree is expected to traverse $O(\log n)$ nodes when there is no contention. During an update operation, once the traversal is finished a single node is then modified in order to update the abstraction. In the case of delete, this means setting the *del* flag to true, or in the case of insert changing the child pointer of a node to point to a newly allocated node (or unmarking the *del* flag in case the node exists in the tree). Given then, that the traversal is the longest part of the operation, the CF tree algorithm tries to avoid here, as often as possible, producing contention. Traditionally, concurrent data structures often require synchronization during traversal (not even including the updates done after the traversal). For example, performing hand-over-hand locking in a tree helps ensure that the traversal remains on track during a concurrent rotation [21], or, using optimistic strategy (such as transactional memory), validation is done during the traversal, risking the operation to restart in the case of concurrent modifications [18, 25, 26]. The following paragraphs describe the modifications made to the algorithm in order to allow avoiding contention during traversal.

Physical Removal. As previously mentioned, the algorithm attempts to remove only nodes whose removal incurs the least contention. Specifically, removing a node *n* with a subtree at each child requires finding its successor node *s* in one of its subtrees, then replacing *n* with *s*. Therefore precautions must be taken (such as locking all the nodes) in order to ensure any concurrent traversal taking place on the path from *n* to *s* does not violate linearizability. Instead of creating contention by removing such nodes, they are left as logically deleted in the CF tree; to be removed later if one of their subtrees becomes empty, or to be unmarked if a later insert operation on the same node occurs.

In the CF tree, nodes that are logically deleted and have less than two child subtrees are physically removed lazily (cf. Algorithm 1). Since we do not want to use synchronization during traversal these removals are done slightly differently than by just unlinking the node. The operation starts by locking the node *n* to be removed and its parent *p* (line 6). Following this, the appropriate child pointer of *p* is then updated (lines 12-13), effectively removing *n* from the tree. Additionally, before the locks are released, both of *n*'s left and right pointers are modified to point back to *p* and the *rem* flag of *n* is set to true (lines 14-15). These additional modifications allow concurrent abstract operations to keep traversing safely without using synchronization as they will then travel back to *p* before continuing their traversal, much like would be done in a solution that uses backtracking.

Algorithm 1. Remove and rotate operations

```
1:  remove($parent, left\text{-}child)_p$:
2:    if $parent.rem$ then return false
3:    if $left\text{-}child$ then $n \leftarrow parent.\ell$
4:    else $n \leftarrow parent.r$
5:    if $n = \perp$ then return false
6:    lock($parent$); lock($n$);
7:    if $\neg n.del$ then return false // release locks
8:    if $(child \leftarrow n.\ell) \neq \perp$ then
9:      if $n.r \neq \perp$ then
10:       return false // release locks
11:   else $child \leftarrow n.r$
12:   if $left\text{-}child$ then $parent.\ell \leftarrow child$
13:   else $parent.r \leftarrow child$
14:   $n.\ell \leftarrow parent$; $n.r \leftarrow parent$;
15:   $n.rem \leftarrow$ true;
16:   // release locks
17:   update-node-heights();
18:   return true.

19: right-rotate($parent, left\text{-}child)_p$:
20:   if $parent.rem$ then return false
21:   if $left\text{-}child$ then $n \leftarrow parent.\ell$
22:   else $n \leftarrow parent.r$
23:   if $n = \perp$ then return false
24:   $\ell \leftarrow n.\ell$;
25:   if $\ell = \perp$ then return false
26:   lock($parent$); lock($n$); lock($\ell$);
27:   $\ell r \leftarrow l.r$; $r \leftarrow n.r$;
28:   // allocate a node called new
29:   $new.k \leftarrow n.k$; $new.\ell \leftarrow \ell r$;
30:   $new.r \leftarrow r$; $\ell.r \leftarrow new$;
31:   if $left\text{-}child$ then $parent.\ell \leftarrow \ell$
32:   else $parent.r \leftarrow \ell$
33:   $n.rem \leftarrow$ true; // by-left-rot if left rotation
34:   // release locks
35:   update-node-heights();
36:   return true
```

Rotations. Rotations are performed to rebalance the tree so that traversals execute in $O(\log n)$ time once contention decreases. As described in Section 2, the CF tree uses localized rotations in order to minimize conflicts. Methods for performing localized rotation operations in the binary trees have already been examined and proposed in several works such as [5]. The main concept used here is to propagate the balance information from a leaf to the root. When a node has a $\perp$ child pointer then the node must know that this subtree has height 0 (the estimated heights of a node's subtrees are stored in the integers *left-h* and *right-h*). This information is then propagated upwards by sending the height of the child to the parent, where the value is then increased by 1 and stored in the parent's *local-h* integer. Once an imbalance of height more than 1 is discovered, a rotation is performed. Higher up in the tree the balance information might become out of date due to concurrent structural modifications, but, importantly, performing these local rotations will eventually result in a balanced tree [5].

Apart from performing rotations locally as unique operations, the specific CF rotation procedure is done differently in order to avoid using locks and aborts/rollbacks during traversals. Let us consider specifically the typical tree right-rotation operation procedure. Here we have three nodes modified during the rotation: a parent node p, its child n who will be rotated downward to the right, as well as n's left child ℓ who will be rotated upwards, thus becoming the child of p and the parent of n. Consider a concurrent traversal that is preempted on n during the rotation. Before the rotation, ℓ and its left subtree exist below n as nodes in the path of the traversal, while afterwards (given that n is rotated downwards) these are no longer in the traversal path, thus violating correctness if these nodes are in the correct path. In order to avoid this, mechanisms such as hand over hand locking [21] or keeping count of the number of operations currently traversing a node [5] have been suggested, but these solutions require traversals to make

Algorithm 2. Restructuring process

```
background-struct-adaptation$()_p$:
    while true do
        // continuous background restructuring
        restructure-node$(root)_s$.

propagate$(n)_s$:
    if $n.\ell \neq \bot$ then $n.left\text{-}h \leftarrow n.\ell.localh$
    else $n.left\text{-}h \leftarrow 0$
    if $n.r \neq \bot$ then $n.right\text{-}h \leftarrow n.r.localh$
    else $n.right\text{-}h \leftarrow 0$
    $n.localh \leftarrow$ max$(n.left\text{-}h, n.right\text{-}h) + 1$.

restructure-node$(node)_s$:
    if $node = \bot$ then return
    restructure-node$(node.\ell)_s$;
    restructure-node$(node.r)_s$;
    if $node.\ell \neq \bot \wedge node.\ell.del$ then
        remove$(node,$ false$)$
    if $node.r \neq \bot \wedge node.r.del$ then
        remove$(node,$ true$)$
    propagate$(node)$;
    if $|node.left\text{-}h - node.right\text{-}h| > 1$ then
        // Perform appropriate rotations.
```

themselves visible at each node, creating contention. Instead, in the CF tree, the rotation operation is slightly modified, allowing for safe, concurrent, invisible traversals.

The rotation procedure is then performed as follows as shown in Algorithm 1: The parent p, the node to be rotated n, and n's left child ℓ are locked in order to prevent conflicts with concurrent insert and delete operations. Next, instead of modifying n like would be done in a traditional rotation, a new node *new* is allocated to take n's place in the tree. Thus, the key, value, and *del* fields of *new* are set to be the same as n's. The left child of *new* is set to $\ell.r$ and the right child is set to $n.r$ (these are the nodes that would become the children of n after a traditional rotation). Next $\ell.r$ is set to point to *new* and p's child pointer is updated to point to ℓ (effectively removing n from the tree), completing the structural modifications of the rotation. To finish the operation $n.rem$ is set to true (or by-left-rot, in the case of a left-rotation) and the locks are released. There are two important things to notice about this rotation procedure: First, *new* is the exact copy of n and, as a result, the effect of the rotation is the same as a traditional rotation, with *new* taking n's place in the tree. Second, the child pointers of n are not modified, thus all nodes that were reachable from n before the rotation are still reachable from n after the rotation, thus, any current traversal preempted on n will still be able to reach any node that was reachable before the rotation.

3.2 Structural Adaptation

As mentioned earlier, one of the advantages of performing structural adaptation lazily is that it does not need to be executed immediately as part of the abstract operations. In a highly concurrent system this gives us the possibility to use processor cores that might otherwise be idle to perform the structural adaptation, which is exactly what is done in the CF tree. A fixed structural adaption thread is then assigned the task of running the background-struct-adaptation operation which repeatably calls the restructure-node procedure on the root node, as shown in Algorithm 2, taking care of balance and physical removal. restructure-node is simply a recursive depth-first procedure that traverses the entire tree. At each node, first the operation attempts to physically remove its children if they are logically deleted. Following this, it propagates balance values from its children and if an imbalance is found, a rotation is performed.

While here we have a single thread constantly running, other possibilities such as having several structural adaptations threads, or distributing the work amongst application threads can be used. It should be noted that, in a case where there can be multiple threads performing structural adaptation, we would need to be more careful on when and the order in which the locks are obtained (for example they could be obtained in a global order based on their key).

Algorithm 3. Abstract operations

```
contains$(k)_p$:
  $node \leftarrow root$;
  while true do
    $next \leftarrow$ get_next$(node, k)$;
    if $next = \bot$ then break
    $node \leftarrow next$;
  $result \leftarrow$ false;
  if $node.k = k$ then
    if $\neg node.del$ then $result \leftarrow$ true
  return $result$.

insert$(k)_p$:
  $node \leftarrow root$;
  while true do
    $next \leftarrow$ get_next$(node, k)$;
    if $next = \bot$ then
      lock$(node)$;
      if validate$(node, k)$ then break
      unlock$(node)$;
    else $node \leftarrow next$
  $result \leftarrow$ false;
  if $node.k = k$ then
    if $node.del$ then
      $node.del \leftarrow$ false; $result \leftarrow$ true
  else // allocate a node called new
    $new.key \leftarrow k$;
    if $node.k > k$ then $node.r \leftarrow new$
    else $node.\ell \leftarrow new$
    $result \leftarrow$ true;
  unlock$(node)$;
  return $result$.

delete$(k)_p$:
  $node \leftarrow root$
  while true do
    $next \leftarrow$ get_next$(node, k)$;
    if $next = \bot$ then
      lock$(node)$;
      if validate$(node, k)$ then break
      unlock$(node)$;
    else $node \leftarrow next$
  $result \leftarrow$ false;
  if $node.k = k$ then
    if $\neg node.del$ then
      $node.del \leftarrow$ true; $result \leftarrow$ true
  unlock$(node)$;
  return $result$.

get-next$(node, k)_s$:
  $rem \leftarrow node.rem$;
  if $rem =$ by-left-rot then $next \leftarrow node.r$
  else if $rem$ then $next \leftarrow node.\ell$
  else if $node.k > k$ then $next \leftarrow node.r$
  else if $node.k = k$ then $next \leftarrow \bot$
  else $next \leftarrow node.\ell$
  return $next$.

validate$(node, k)_s$:
  if $node.rem$ then return false
  else if $node.k = k$ then return true
  else if $node.k > k$ then $next \leftarrow node.r$
  else $next \leftarrow node.\ell$
  if $next = \bot$ then return true
  return false.
```

3.3 Abstract Operations

The abstract operations are shown in Algorithm 3. Each of the abstract operations begin by starting their traversal from the *root* node. The traversal is then performed, without using locks, from within a while loop where each iteration of the loop calls the get-next procedure, which returns either the next node in the traversal, or $\bot$ in the case that the traversal is finished.

The get-next procedure starts by reading the *rem* flag of *node*. If the flag was set to by-left-rotate then the node was concurrently removed by a left-rotation. As we saw in the previous section, a node that is removed during rotation is the node that would be rotated downwards in a traditional rotation. Specifically, in the case of the left rotation, the removed node's right child is the node rotated upwards, therefore in this case, the get-next operation can safely travel to the right child as it contains at least as many nodes in its path that were in the path of the *node* before the rotation. If the flag was set to true then the node was either removed by a physical removal or a right-rotation, in either case the operation can safely travel to the left child, this is because the remove operation changes both of the removed node's child pointers to point to the parent and the right-rotation is the mirror of the left-rotation. If the rem flag is false then the key of *node* is checked, if it is found to be equal to k then the traversal is finished and $\perp$ is returned. Otherwise the traversal is performed as expected, traversing to the right if the $node.k$ is bigger than k or to the left if smaller.

Given that the insert and delete operations might modify *node*, they lock it for safety once $\perp$ is returned from get-next. Before the node is locked, a concurrent modification to the tree might mean that the traversal is not yet finished (for example the node might have been physically removed before the lock was taken), thus the validate operation is called. If false is returned by validate, then the traversal must continue, otherwise the traversal is finished. Differently, given that it makes no modifications, the contains operation exits the while loop immediately when $\perp$ is returned from get-next.

The validate operation performs three checks on *node* to ensure that the traversal is finished. First it ensures that $rem =$ false, meaning that the node has not been physically removed from the tree. Then it checks if the key of the node is equal to k, in such a case the traversal is finished and true is returned immediately. If the key is different from k then the traversal is finished only if *node* has $\perp$ for the child where a node with key k would exist. In such a case true is returned, otherwise *false* is returned. Once the traversal is complete the rest of the code is straightforward. For the contains operation, true is returned if $node.k = k$ and $node.del =$ false, false is returned otherwise. For the insert operation, if $node.k = k$ then the *del* flag is checked, if it is false, then false is returned; otherwise if the flag is true it is set to false, and true is returned. In the case that $node.k \neq k$, a new node is allocated with key k and is set to be the child of *node*. For the delete operation, if $node.k \neq k$, then false is returned. Otherwise, the *del* flag is checked, if it is true then false is returned, otherwise if the flag is false, it is set to true and true is returned.

Linearization. Given that the insert and delete operations that return false do not modify the tree and that all other operations that modify nodes only do so while owning the node's locks, these failed insert and delete operations can be linearized at any point during the time that they own the lock of the node that was successfully validated. The successful insert (i.e., the one that returns true) operation is linearized either at the instant it changes $node.del$ to false, or when it changes the child pointer of *node* to point to *new*. In either case, k exists in the abstraction immediately after the modification. The successful delete operation is linearized at the instant it changes $node.del$ to true, resulting in k no longer being in the abstraction.

The contains operation is a bit more difficult as it does not use locks. To give an intuition of how it is linearized, first consider a system where neither rotations nor physical removals are performed. In this system, if $node.k = k$ on line 8 is true, then the linearization point is when $node.del$ is read (line 9). Otherwise if $node.k \neq k$, then the linearization point is either on line 50 or 52 of the get-next operation where $\perp$ is read as the next node (meaning at the time of this read k does not exist in the abstraction).

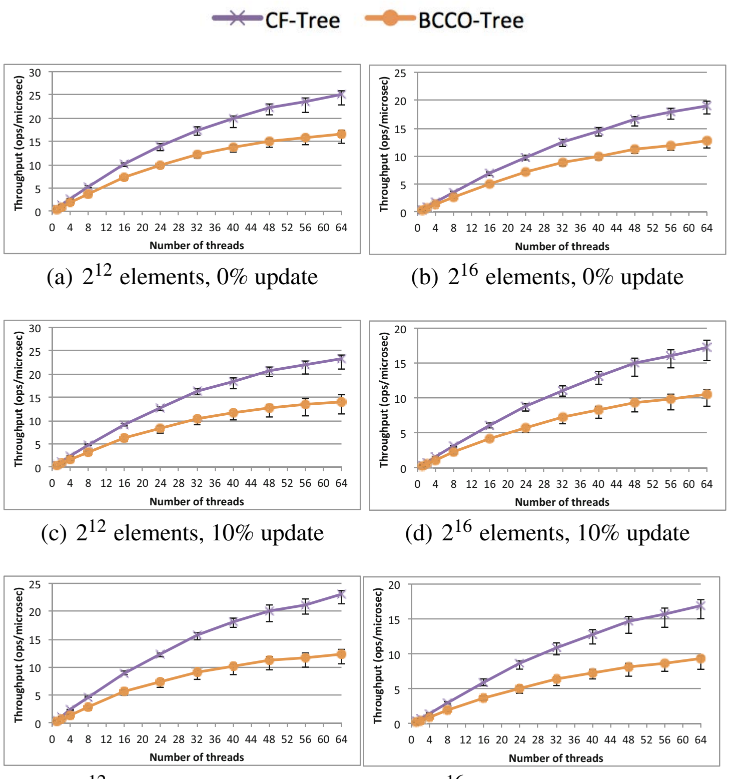
Now, if rotations and physical removals are performed in the system, then a contains operation who has finished its traversal might get preempted on a node that is removed from the tree. First consider the case where $node.k = k$, since neither rotations nor removals will modify the *del* flag of a node, then in this case the linearization point is simply either on line 50 or 52 of the get-next operation where the pointer to *node* was read. Now consider the case where $node.k \neq k$. First notice that when false is not read from node.rem (line 47 of get-next) then the traversal will always continue to another node. This is due to the facts that after a right (resp. left) rotation, the node removed from the tree will always have a non-$\perp$ left (resp. right) child (this is the child rotated upwards by the rotation) and that a node removed by a remove operation will never have a $\perp$ child pointer. Therefore if the traversal finishes on a node that has been removed from the tree, it must have read that node's *rem* flag before the rotation or removal had completed. This read will then be the linearization point of the operation. In this case, for the contains operation to complete, the next node in the traversal must be read as $\perp$ from the child pointer of *node*, meaning that the removal/rotation has not made any structural modifications to this pointer at the time of the read (this is because rotations make no modifications to the child pointers of the node they remove, and removals point the removed node's pointers towards its parent). Thus, given that removals and rotations will lock the node removed meaning no concurrent modifications will take place, effectively the contains operation has observed the state of the abstraction immediately before the removal took place.

4 Evaluation

We compare the performance of the contention-friendly tree (CF-tree) against the most recent lock-based binary search tree we are aware of (BCCO-tree [21]) on an UltraSPARC T2 with 64 hardware threads. For each run, we present the maximum, minimum, and averaged numbers of operations per microsecond over 5 runs of 5 seconds executed successively as part the same JVM for the sake of warmup. We used Java SE 1.6.0 12-ea in server mode and HotSpot JVM 11.2-b01 for both tree algorithms.

Figure 1 compares the performance of the practical BCCO tree against performance of our binary search tree with 2^{12} (left) and 2^{16} elements (right) and on a read-only workload (top) and workloads comprising up to 20% updates (bottom). The variance of our results is quite low as illustrated by the relatively short error bars we have. While both trees scale well with the number of threads, the BCCO tree is slower than its contention-friendly counterpart in all the various settings.

In particular, our CF tree is up to $2.2\times$ faster than its BCCO counterpart. As expected, the performance benefit of our CF tree increases generally with the level of contention. (We observed this phenomenon at higher update ratios but omitted these graphs for the sake of space.) First, the performance improvement increases with the level of concurrency on Figures 1(c), 1(d), 1(e) and 1(f). As each thread updates the memory with



(a) 2^{12} elements, 0% update (b) 2^{16} elements, 0% update

(c) 2^{12} elements, 10% update (d) 2^{16} elements, 10% update

(e) 2^{12} elements, 20% update (f) 2^{16} elements, 20% update

Fig. 1. Performance of our contention-friendly tree and the practical concurrent tree [21]

the same (non-null) probability the contention increases with the number of concurrent threads running. Second, the performance improvement increases with the update ratio. This is not surprising as our tree relaxes the balance invariant during contention peaks whereas the BCCO tree induces more contention to maintain the balance invariant.

5 Concluding Remarks

To conclude, lock-based data structures can greatly benefit from the contention-friendly methodology on multicore architectures. In particular the decoupling of accesses allow the CF tree to scale with a reasonably large number of hardware threads. An interesting future work would be to experimentally assess the performance gain due to applying contention-friendliness to other data structures. Additionally, a contention metric that complements the traditional asymptotic step complexity seems to be necessary to capture the cost of a multicore data structure.

References

1. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: EuroSys., pp. 183–196 (2012)

2. Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: PPoPP, pp. 161–170 (2012)
3. Adelson-Velskii, G., Landis, E.M.: An algorithm for the organization of information. In: Proc. of the USSR Academy of Sciences, vol. 146, pp. 263–266 (1962)
4. Bayer, R.: Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Informatica 1 1(4), 290–306 (1972)
5. Bougé, L., Gabarro, J., Messeguer, X., Schabanel, N.: Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical Report RR1998-18, ENS Lyon (1998)
6. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: FOCS, pp. 8–21 (1978)
7. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. In: Proc. of the ACM SIGFIDET Workshop on Data Description, Access and Control, pp. 107–141 (1970)
8. Huddleston, S., Mehlhorn, K.: A new data structure for representing sorted lists. Acta Inf. 17, 157–184 (1982)
9. Kessels, J.L.W.: On-the-fly optimization of data structures. Commun. ACM 26(11), 895–901 (1983)
10. Nurmi, O., Soisalon-Soininen, E.: Uncoupling updating and rebalancing in chromatic binary search trees. In: PODS, pp. 192–198 (1991)
11. Nurmi, O., Soisalon-Soininen, E.: Chromatic binary search trees. A structure for concurrent rebalancing. Acta Inf. 33(6), 547–557 (1996)
12. Boyar, J., Fagerberg, R., Larsen, K.S.: Amortization results for chromatic search trees, with an application to priority queues. J. Comput. Syst. Sci. 55(3), 504–521 (1997)
13. Mohan, C.: Commit-LSN: a novel and simple method for reducing locking and latching in transaction processing systems. In: VLDB, pp. 406–418 (1990)
14. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001)
15. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA, pp. 73–82 (2002)
16. Fraser, K.: Practical lock freedom. PhD thesis, Cambridge University (September 2003)
17. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: PODC, pp. 131–140 (2010)
18. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, p. 528. Morgan Kaufmann (2008)
19. Shavit, N.: Data structures in the multicore age. Commun. ACM 54(3), 76–84 (2011)
20. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 93–107. Springer, Heidelberg (2009)
21. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: PPoPP (2010)
22. Crain, T., Gramoli, V., Raynal, M.: Brief announcement: A contention-friendly, non-blocking skip list. In: Aguilera, M.K. (ed.) DISC 2012. LNCS, vol. 7611, pp. 423–424. Springer, Heidelberg (2012)
23. Crain, T., Gramoli, V., Raynal, M.: No hot spot non-blocking skip list. In: ICDCS (2013)
24. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 463–492 (1990)
25. Herlihy, M.P., Lev, Y., Luchangco, V., Shavit, N.: A simple optimistic skiplist algorithm. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 124–138. Springer, Heidelberg (2007)
26. Raynal, M.: Concurrent Programming: Algorithms, Principles, and Foundations. Springer (2013)

Topic 6: Grid, Cluster and Cloud Computing (Introduction)

Erwin Laure, Odej Kao, Rosa M. Badia, Laurent Lefevre,
Beniamino Di Martino, Radu Prodan, Matteo Turilli, and Daniel Warneke

Topic Committee

Grid and cloud computing have changed the IT landscape in the way we access and manage IT infrastructures. The use of computing resources has become essential for many applications in various areas. Both technologies provide easy-to-use and on-demand access to large-scale infrastructures. The high number of submissions to "Topic 6: Grid, Cluster and Cloud Computing" reflected the importance of the research area. The papers addressed key challenges regarding design, deployment, operation and use of Grid and cloud infrastructures. Moreover, several innovative algorithms and methods for fundamental capabilities and services that are required in a heterogeneous environment, such as adaptability, scalability, reliability and security, and to support applications as diverse as ubiquitous local services, enterprise-scale virtual organizations, and internet-scale distributed supercomputing were proposed. Finally, many experimental evaluations and use-cases delivered an insight into the deployment in real-world scenarios and showed interesting future application domains. Each submission was reviewed by at least four reviewers and, finally, we were able to select nine high-quality papers. The papers were grouped in four sessions that are briefly summarized in following.

The first 3 papers discuss various aspects of cloud scheduling, from application centric resource provisioning for Amazon EC2 spot instances, online-optimization of workflow activity granularity, to on-demand reserved cloud instances.

Aspects of cloud deployment and MapReduce are being discussed in the second 3 papers, particularly optimization of Pig analytics, job ordering optimization in MapReduce workloads, and content exchange for on-demand VM multi-deployments.

The last papers finally focus on energy- and carbon-efficient VM placement, and improvements in quality of service through market mechanisms and dynamically adjusting parallelism degrees in distributed parallel applications.

We would like to take the opportunity of thanking the authors who submitted a contribution, as well as the Euro-Par Organizing Committee, and the external referees with their useful comments, whose efforts have made this conference and this topic possible.

Scheduling Jobs in the Cloud Using On-Demand and Reserved Instances

Siqi Shen[1], Kefeng Deng[1,2], Alexandru Iosup[1], and Dick Epema[1]

[1] Delft University of Technology, Delft, The Netherlands
{S.Shen,A.Iosup,D.H.J.Epema}@tudelft.nl
[2] National University of Defense Technology, Changsha, China
Dengkefeng@nudt.edu.cn

Abstract. Deploying applications in leased cloud infrastructure is increasingly considered by a variety of business and service integrators. However, the challenge of selecting the leasing strategy — larger or faster instances? on-demand or reserved instances? etc.— and to configure the leasing strategy with appropriate scheduling policies is still daunting for the (potential) cloud user. In this work, we investigate leasing strategies and their policies from a broker's perspective. We propose, *CoH*, a family of Cloud-based, online, Hybrid scheduling policies that minimizes rental cost by making use of both on-demand and reserved instances. We formulate the resource provisioning and job allocation policies as Integer Programming problems. As the policies need to be executed online, we limit the time to explore the optimal solution of the integer program, and compare the obtained solution with various heuristics-based policies; then automatically pick the best one. We show, via simulation and using multiple real-world traces, that the hybrid leasing policy can obtain significantly lower cost than typical heuristics-based policies.

1 Introduction

A growing number of applications are running in the cloud. Academia [1–7] and industry [8] are both increasingly using cloud resources as infrastructure to serve their users, due to the elastic, flexible, and pay-as-you-go features of Infrastructure-as-a-Service (IaaS) clouds. Cloud brokers need to lease resources from IaaS clouds cheaply, yet execute the users' jobs in time. To achieve this, cloud brokers must use scheduling policies that match diverse requirements. *Finding scheduling polices that can schedule diverse workloads with zero waiting time yet cheaply is the focus of this work*

IaaS clouds offer their users various types of machine configurations: different amount of CPU cores, memory, and disk. It is non-trivial for a cloud broker to determine the combination of machine configurations for user demands. This situation is complicated by the current IaaS pricing models: machines configurations are not priced linearly with their performance. For example, an EC2 `large` instance can serve more web requests per core than the `small` instance, but their price per core is the same [9]. Moreover, for the same machine configuration, the clouds offer different billing options on-demand-, reserved-, and spot-instances, which are charged differently. Scheduling enough resources to meet user demands yet keep the cost low while adapting to workload changes remains challenging, despite recent research efforts [9–11].

In this work, we present a Cloud-based, online, Hybrid scheduling policy (*CoH*), which keeps the rental cost of cloud resources low by finding the best combination of machine configurations and billing options. At the core of this policy are its provisioning and allocation strategies. We formulate these strategies as Integer Programming Problems (IPP). As *CoH* needs to be executed online, the time to obtain a decision should be low. We limit the time to solve the IPP, and run simultaneously various heuristics. The *CoH* compares the result of IPP and heuristics, and picks the best one as its scheduling decision. Thus, a novel aspect of *CoH* is its portfolio-based scheduling strategy [12] adapted to IaaS clouds. Further, we devise, *CoH-R*, an extension of *CoH* to also makes use of reserved instances, which can lead to significant cost reduction compare to policies that use on-demand instances only.

The major contributions of this work are three-fold.

1. A novel online scheduling policy, *CoH*, which makes scheduling decision using a portfolio of IPP and heuristics-based approaches (Section 3).
2. A policy extended from *CoH*, *CoH-R*, which also makes use of reserved instances to reduce rental cost (Section 4).
3. An evaluation of our policies for two broad application domains, grid computing and online game hosting, using trace-based simulation (Section 5).

2 System Model

2.1 Workload and Resource Model

The workload model in this work is a set of independent jobs. The resource requirements and the runtime of each job are known when the job arrives in the system. Once started, jobs run to completion, so we do not consider task preemption or migration during execution.

Each job can be described by a tuple (r_i, a_i, d_i), where r_i is the resource requirement of job i, a_i is the arrival time of job i, and d_i is its departure time. We assume that a computer can host one or multiple jobs. This model is similar to the work of Stillwell et al [13]. This kind of jobs is common: a compute node can run multiple MapReduce tasks; an online game hoster may consolidate several game servers on the same machine; etc. The resource requirements of each job, r_i, could be a vector indicating multiple resource requirements (e.g., CPU and Memory), or a scalar value (e.g., CPU only). We focus on the CPU requirement. In practice, r_i can be obtained though profiling [9, 14] or can be provided by the user.

We model the operation and billing model of cloud providers based on the real case of Amazon EC2. We assume that clouds have infinite capacity. Each newly provisioned VM needs serval minutes to be booted [10, 14]. An VM is charged per hour; even a factional consumption of less than one hour is counted as one hour. An VM indexed by j, has capacity denoted by w_j and hourly cost c_j.

2.2 Scheduling Model

In our scheduling model, all machines are provisioned exclusively from clouds. The cloud broker has pre-configured and stored in the cloud all the necessary VM images to run users' jobs. All the incoming jobs are enqueued into a queue. A system-level scheduler, running on a dedicated system, manages all the jobs and a pool of machines, and decides whether to provision new VM from clouds and/or to allocate jobs to VMs.

The scheduler is executed periodically (e.g., every 10 seconds). At each scheduling moment, the scheduler performs five tasks: (1) Predicting future incoming workloads; (2) Provisioning necessary VMs in advance, from clouds; (3) Allocating jobs to VMs, (4) Releasing idle VMs (which don't have job running on them) if its Billing Time Unit (BTU) is close to increase (e.g., 10 second before the leased hour). (5) If the wait time of un-allocated jobs is high, starting the necessary number of VMs. We design in the next section a scheduling policy, *CoH* to perform tasks (3) and (4). We further extend this policy in Section 4 to also use reserved cloud instances. As workload prediction is not the focus of this paper, we assume that there exists a predictor that can achieve perfect prediction of future workload. Relatively good predictor [15] already exists for the type of workload we target in this work.

3 Scheduling Using On-Demand Instances

This section describes *CoH*, a Cloud-based, online, Hybrid scheduling policy using on-demand instances. The strategy of *CoH* is presented in Section 3.1. *CoH* needs to take both provisioning and allocation decisions, that is, to find a combination of VMs, and a mapping between jobs and VMs. We formulate the above problem as an Integer Programming Problem (IPP) in Section 3.2 and then select various heuristics to assist *CoH* in Section 3.3.

3.1 Policy Overview

CoH actively provisions VMs before they are needed, and maps jobs to already provisioned VMs according to the best mapping it can find. *CoH* finds the combination of VMs and the mapping by solving an online scheduling problem through solving (partially) one IPP and by using several heuristics, independently and simultaneously. As an online scheduler, *CoH* needs to take scheduling decisions within limited amounts of time; thus, it limits the time used to solve IPP, and compares the result of the IPP and heuristics. *CoH* acts as a portfolio-based scheduler, in which multiple strategies are considered simultaneously at each scheduling moment. The strategy that has the best objective value (defined in formula (1)) is picked as the scheduling decision. Heuristics are needed because the solution of IPP under limited time may be suboptimal or even infeasible (*CoH* may not find a feasible solution of the IPP in limited time).

Table 1. Overview of notations in Section 3

x_{ij_k}	whether job i is assigned to jth VM of type k, $x_{ik_j} \in \{0,1\}$
y_{j_k}	whether jth VM of type k is to-be-provisioned, $y_{k_j} \in \{0,1\}$
z_{ij}	whether job i is assigned to jth running VM, $z_{ij} \in \{0,1\}$
c_k	hourly cost of VM of type k
c_j	hourly cost of running VM j
w_j	capacity of running VM j
fc_k	full capacity of VM type k
r_i	resource consumption of job i
d_i	departure/end time of job i
s_j	The start time of VM j: the time when it started to boot
ld_j	latest departure time: the time that the final running job finish in VM j
ct	current time
M	number of newly arrived jobs, $\mathbb{M} = \{1, ..., M\}$
N	number of running VMs, $\mathbb{N} = \{1, ..., N\}$
K	number of types of VMs, $\mathbb{K} = \{1, ..., K\}$
$\lceil t \rceil$	math operation, divide time t by 3600 and get its ceil value.

3.2 Formalization of the Scheduling Problem

CoH needs to provision enough number of VMs to support all the incoming jobs, and to allocate all the jobs smartly such that the rental cost is minimized. An VM that has jobs running on it cannot be shut down, so it will still incur cost. Unnecessary cost will be incurred if long-running, low-resource requiring jobs are assigned to expensive VMs. We formulate the scheduling problem as follows. The goal of the scheduling problem, as defined in formula (1), is to minimize the cost while ensuring enough VMs for the incoming jobs. All the notations used in this section are listed in Table 1. An VM *to-be-provisioned* is identified by its identifier j and its type k, while a running VM is identified only by its identifier j.

Minimize

$$\sum_{k=1}^{K}\sum_{j=1}^{M}(y_{j_k} \times \lceil \max_{i \in \mathbb{M}}(d_i \times x_{ij_k}) - ct \rceil \times c_k) + R \tag{1}$$

$$R = \sum_{j=1}^{N}(\lceil (max\{\max_{i \in \mathbb{M}}(d_i \times z_{ij}), ld_j\} - s_j) \rceil \times c_j)$$

subject to

$$\sum_{i=1}^{M} z_{ij} \times r_i \leq w_j \qquad \forall j \in \mathbb{N} \tag{2}$$

$$\sum_{k=1}^{K}\sum_{i=1}^{M} x_{ij_k} \times r_i \leq fc_k \times y_{j_k} \qquad \forall j \in \mathbb{M} \tag{3}$$

$$\sum_{j=1}^{N} z_{ij} + \sum_{k=1}^{K}\sum_{j=1}^{M} x_{ij_k} = 1 \qquad \forall i \in \mathbb{M} \tag{4}$$

$$x_{ij_k} \leq y_{j_k} \qquad \forall i,j \in \mathbb{M}, \forall k \in \mathbb{K} \tag{5}$$

The cost of scheduling consists two parts: the cost of to-be-provisioned VMs and the cost of running VMs (defined by R). Each to-be-provisioned VM is charged between the current time (ct) and the latest departure time of its allocated jobs. The sum of the cost of running VMs, denoted by R, is defined similarly: each running VM is charged between the time it was started (s_j) and the latest departure time of its jobs (jobs that are running on VM and the jobs to-be-allocated to it).

This IPP is subject to a few constrains, which we describe in turn. Constraint (2) ensures that the allocated jobs in each VM cannot exceed the running VMs' capacity. Constraint (3) ensures that the allocated jobs in each *to-be-provisioned* VM can not exceed the VM's capacity. Constraint (4) ensures that each job is only allocated to one VM. Constraint (5) ensures that each job will not be allocated to a VM that will not be provisioned. The decision variables x_{ij_k} and y_{k_j} are binary. If the result of this IPP is that $y_{k_j} = 0, \quad \forall k \in \mathbb{K}, \quad \forall j \in \mathbb{M}$, there will be enough VM capacity left to allocate all the future jobs. Otherwise, more VMs are needed. If $x_{ij_k} = 1$, job i will be allocated to the to-be-provisioned VM with identifier j type k.

3.3 Scheduling Heuristics

We explore for *CoH* a large class of scheduling heuristic algorithms. They work as follows. While there are un-allocated jobs, each algorithm performs a loop consisting of four steps. Firstly, the algorithm sorts all the un-allocated jobs using *job selection* criteria and sorts all the VMs using *VM selection* criteria. Secondly, the algorithm picks the first un-allocated job. Thirdly, it picks the first VM which should have enough capacity left for the job. And then allocate the job to the selected VM. If such an VM does not exist, a new VM is provisioned according to *VM type selection* criteria and the job will be allocated in the next loop.

This general class of heuristic algorithms uses three criteria: *job selection*, *VM selection*, and *VM type selection* criteria. All *job selection* and *VM selection* criteria used in this work are listed in Tables 2 and 3. For *VM type selection*, we use a Cost-Efficient heuristic, which always chooses the VM with the largest *capacity/cost* value.

Most of the selection criteria we use in this work are simple, which allows them to be run online. We describe some of the criteria below. Latest arrival time (LA) sorts the VM according to the latest arrival time of jobs in each VM in decreasing order. Opposite to LT, Earliest arrival time (EA) sorts the VM by earliest arrival time of jobs in increasing order. Similar to LT, Latest departure time (LD) picks the VM which has the job that has the latest departure time; Earliest departure time (ED) does the opposite. The latest average arrival time (LAA) and earliest average arrival time (EAA) sorts VMs according to the average arrival time of their jobs in decreasing and increasing order, respectively. Close to full hour (CFH) makes use of the billing model of EC2; it always puts jobs on VM whose Billing Time Unit (BTU) is closest to be increased, while Far from Full Hour (FFH) is the opposite. In this work, the scheduling heuristic method specified by its job and VM selection criteria is uniquely identified as {*job selection*}-{*VM selection*}. For example, the FCFS-Rnd heuristic uses First-Come-First-Server (FCFS) for job selection, random (Rnd) for VM selection and the cost-efficient criteria for VM type selection.

Table 2. Job selection criteria

Name	Description
FCFS	First-come-first-server
RR	round-robin
LJF	Largest job first
SJF	Smallest job first
LTJF	Longest run-Time job first
STJF	Shortest run-Time job first

Table 3. VM selection criteria

Name	Description
Rnd	Random
LM	Largest capacity VM first
SM	Smallest capacity VM first
LA	Latest arrival time
EA	Earliest arrival time
LD	Latest departure time
ED	Earliest departure time
LAA	Latest average arrival time
EAA	Earliest average arrivial Time
CFH	Close to Full Hour
FFH	Far from full hour

4 Scheduling Using Reserved and On-Demand Instances

Cloud providers allow their users to reserve VM instances, long-term, for reduced cost. For instance, Amazon offers *reserved* instance, which can be rented for 1-3 years for a lower price than their on-demand counter-parts. When using reserved instead of on-demand instance, for the same VM configuration, users can pay a higher upfront cost (UF_i) for a lower hourly cost (C_i). Currently, there are three types of reserved instances supported in EC2: lightly utilized, medium-utilized, and heavily utilized reserved instances. For the lightly utilized and medium-utilized instances, users need to pay an upfront cost and pay for each hour the VM is running. For the heavily utilized instances, users need to pay an upfront cost and pay for each hour during the reserved term even if the VM is not running. The hourly cost of Amazon EC2 instances are listed in Table 5. We present *CoH-R*, an extension of *CoH*, which uses reserved instances to reduce the operational cost. We describe the strategy of *CoH-R* in Section 4.1, then describe the method used to determine the number and types of reserved instances to be used in Section 4.2.

4.1 Policy Overview

Assuming that it is given an arbitrary amount of reserved instances, *CoH-R* makes use of these reserved instances as follows: the heavily utilized instances are always on, while the medium and lightly utilized instances are shut-down when they do not have any jobs running on them before their Billing Time Unit (BTU) is about to increase (m seconds before the BTU increases). Whenever *CoH* plans to start a new VM of type k, *CoH-R* firstly looks at medium-utilized instances of type k, and starts one of them if any is off. If no medium-utilized instance of type k exists, *CoH-R* tries to use a lightly utilized instance of type k. As a last resort, *CoH-R* uses on-demand instance of type k.

Having too few reserved instances will not benefit much from the reduced price; while reserving too much may actually increase operational cost. We do not seek to find the optimal number of reserved instances, because obtaining the optimal solution requires exact workload information of the entire reservation period (e.g., one year). Even if we can know the workload of the upcoming time period, obtaining the optimal solution via solving an IPP that takes the exact workload as input is computationally infeasible.

4.2 Determining the Reservation Plan

CoH-R only requires the workload distribution instead of exact information of the number of VMs needed at each time interval. For simplicity of analysis, we assume VM start-up and shut-down time are instantaneous (In experiment, we set the start-up time as two minutes.). Assuming the number of VMs (resource demand) needed for the current time interval t is D_t, we can obtain the cost needed at each interval $B(D_t)$ as follows. If D_t is lower than the number of heavily-utilized instances (N_3), no other VMs will be needed, as the heavily utilized instances can provide enough computing resources. If D_t is high than N_3, but lower than the total number of heavily and medium-utilized instances ($D_t \leq N_3 + N_2$), *CoH-R* needs to provision $D_t - N_3$ medium-utilized instances. If D_t is higher than the sum of heavily and medium-utilized instances ($D_t > N_3 + N_2$) but lower than the total number of reserved instances, *CoH-R* needs to provision all the heavily and medium-utilized instances and $D_t - (N_3 + N_2)$ lightly utilized instances. Last, if D_t is higher than the sum of all reserved instances, *CoH-R* needs all the reserved instances and $D_t - (N_1 + N_2 + N_3)$ on-demand instances.

CoH-R obtains the number of reserved VMs needed, of each type, via finding the combination of number of reserved instances (N_i) that minimizes $\sum_{t=1}^{T} B(D_t) + \sum_{k \in \mathbb{K}} (UF_k \times N_k)$, where T is the number of intervals (e.g, hour) of a time period (e.g, year or month). Further, if the resource demand of each interval does not affect the other time intervals (in practice, most of the jobs' runtime is short, in the order of tens of minutes), the goal can be reformulated via only using the probability of VMs needed at each time interval as below, $(\sum_{i=0}^{M} Pr(D = i) \times B(i)) \times T + \sum_{k \in \mathbb{K}} (UF_k \times N_k)$, where $Pr(D = i)$ is the probability distribution of demand, and $\mathbb{K}$ is the set of reserved type, and M is the maximal number of VM needed.

We extend the above method which deal with one machine configuration only, to allow it to deal with multiple machine configurations. The goal is to find the number of reserved instances (N_{jk}) of machine configuration j and reserved type k, needed, to minimize the cost defined in formula (6).

$$(\sum_{i=0}^{M} Pr(D=i) \times B(i)) \times T + \sum_{j \in \mathbb{J}} \sum_{k \in \mathbb{K}} (UF_{jk} \times N_{jk}) \tag{6}$$

In formula (6), $\mathbb{J}$ is the set of machine configurations and UF_{jk} is the upfront cost of reserved instance of machine configuration j and reserved type k. The billing function $B(D)$ need to be changed to be the lowest cost to meet the demand D via finding the combination of reserved and on-demand instances to be used. $B(D) = \{Minimize \sum_{j \in \mathbb{J}} \sum_{k \in \mathbb{K}} (n_{jk} \times c_{jk}) + \sum_{j \in \mathbb{J}} (n_j^{od} \times c_j^{o})\}$, where n_{jk} and c_{jk} are the number and the cost of the reserved instance with machine configuration j and reserved type k, respectively. n_j^{od} and c_j^{o} are the number and the cost of the on-demand instances of machine configuration j, respectively. The capacity offered by n_{jk} reserved instances and n_j^{od} on-demand instances should be enough to satisfy demand D.

5 Experimental Results

In this section, we evaluate the performance of our proposed approaches using multiple real-world traces corresponding to two separate but popular domains: grid computing and online game hosting. Firstly, we compare *CoH* against various commonly used heuristics. Then, we evaluate *CoH-R*, which uses reserved instances to further reduce cost, and compare it to *CoH*. Our results indicate that our proposed approaches can lead to significant lower cost than heuristics.

5.1 Experimental Setup

We conduct experiments using three real-world workloads *LCG*, *Grid5000*, and *Dotalicious* which are taken from public workload archives [16–18]. *LCG* and *Grid5000* contain information about the computing activities of two grids while *DotaLicious* contains workload information of a game platform. We use the first year traces *Grid5000* and *Dotalicious*, and the full trace of LCG (13 days) as our input workloads. The common data we find in the above traces are, for each recorded job, its job id, the arrival time, and the departure time. The basic statistics of these workloads are listed in Table 4. Notably, the gaming server have similar runtime (CPU requirement) to Grid5000 jobs in the order of tens of minutes. Game servers are also computationally intensive, a result of having to perform virtual world physical simulation.

As not all the traces contain resource requirements for each job, we generate for each job resource requirements using 3 different methods: Heterogeneous, Constant-100, and Constant-10. For Heterogeneous workload, we generate each jobs's resource requirements as ten times a random number which is between 1 to

10. For Constant-100 method, each job's resource required is 100. For Constant-10 method, each job's resource required is 10. We only consider two instance types: `small` and `large`. We model a `small` EC2 instance's capacity as 100 and a `large` instance's capacity is 410. `Large` instance is more cost efficient than `small` instance. Their cost[1] are summarized in Table 5.

As running all the heuristics online is time consuming, we evaluate the heuristics by running simulation and pick the heuristics that have good performance as alternative method to compete with the solution obtained by solving IPP. We find that none of heuristics can perform always best, across all scenarios, and find that the job selection criteria does not have a significant impact on cost but VM selection criteria does have an important impact on cost. We pick FCFS-SM when the input workload is heterogeneous, and use FCFS-LD and FCFS-CFH when the workload is homogeneous (Constant-10 and Constant-100).

All the experiments are conducted using our own simulator[2] and repeated at least 10 times. We set the acquisition time of an VM to two minutes and the scheduler is executed every 10 seconds. We use IBM CPLEX to solve the formulated IPP when the number of jobs to be scheduled is lower than 50 and set the time limited as two seconds. As our methods have proactively provision VMs for all the jobs, the wait time of each job is zero. We evaluate one metric, the rental cost. The rental cost is the price paid to cloud providers for all the rented computing resource. We focus on cost because it is a major barrier for cloud adoption.

For calculation of the utility of all the methods, we compare the lower-bound for cost against actually paid cost. The lower bound for cost is calculated by assuming that we have an ideal computer that it can vertically scale to the any of the desired capacity. The vertical scaling takes zero time and the VM is charged by its actual usage of resource which scales linearly with its capacity. So the optimal cost can be computed as $[\sum_{i=1}^{N} r_i \times (d_i - a_i)] \div w_k \times c_k, \quad \forall i \in \mathbb{N}$, where N is the total number of jobs, and w_k and c_k are the capacity and the cost of the most cost-efficient VM, respectively.

Table 4. Overview of traces

Trace	#jobs	average runtime [s]	duration	source
Grid5000	200,450	2728	May 2004 - May 2004	Grid workload archive [17]
LCG	188,041	8971	Nov 2005 - Dec 2005	Parallel workload archive [16]
DotaLicious	109,251	2231	Apr 2010 - Apr 2011	Game trace archive [18,19]

Table 5. Overview of cost of EC2 instances: Small and Large

	Small (hourly, upfront) [$]	Large (hourly, upfront) [$]
On demand	(0.065, 0)	(0.26, 0)
Lightly utilized	(0.039, 69)	(0.156, 276)
Medium utilized	(0.024, 160)	(0.096, 640)
Heavily utilized	(0.016, 195)	(0.064, 780)

[1] `http://aws.amazon.com/ec2/pricing/`

[2] `http://www.pds.ewi.tudelft.nl/~siqi/simulator.htm`

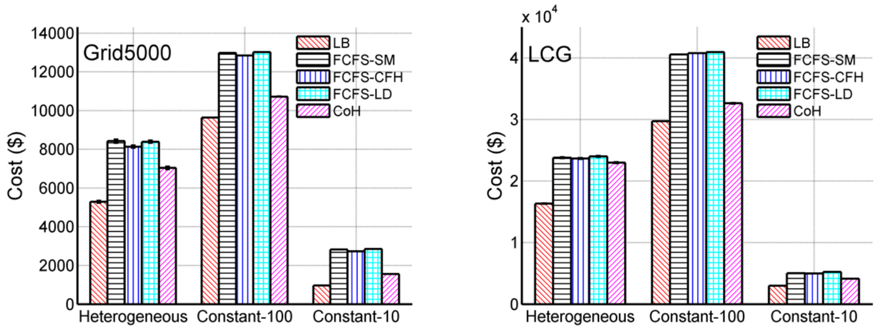


Fig. 1. Cost of various scheduling methods: Grid5000 (Left) and LCG (Right)

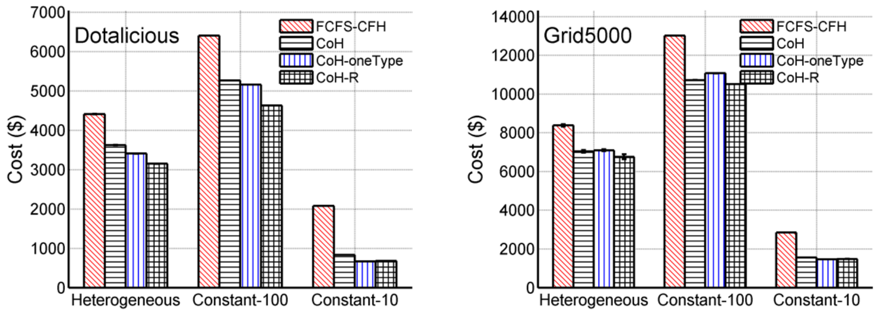


Fig. 2. Effect of using reserved instances: Dotalicious (Left) and Grid5000 (Right)

5.2 Results

We first evaluate *CoH* against various heuristic methods. Figure 1 shows the average experiment results using *Grid5000* and *LCG* datasets, respectively. The error bars are the standard deviation. Figure 1 shows the lower bound for cost (LB), and results for FCFS-SM, FCFS-CFH, FCFS-LD, and *CoH*, from left to right; grouped by type of workloads. We find that *CoH* performs better than any of the heuristics. For the *Grid5000* dataset, *CoH* can obtain about 20% to 40% lower cost than any heuristic. For the *LCG* dataset, *CoH* can obtain 5% to 20% lower cost. This indicates that CoH can find better combinations of VMs, and better mapping between jobs and VMs.

The cost obtained through *CoH* is about 1.1 to 1.6 times higher than LB. The utilization of *CoH*, that is, the average use of leased VMs, ranged from 90% to 63%. We identify three reasons why *CoH* is higher than the lower bound (LB): Firstly, our scheduler is run online, thus not having all the necessary information. Secondly, the billing model of the cloud: a fractional consumption of a VM's capacity is charged as the fully busy VM. Thirdly, the boot-up time of VM is not negligible. One possible way to lower the gap between LB and *CoH* is to allow the jobs to wait for better scheduling opportunity, so that scheduler can pack more jobs in the same VM instead of starting a VM for each short job. This approach would be particularly effective during bursts in the workload.

We evaluate *CoH-R* using the *Dotalicious* and *Grid5000* datasets. The results are shown in Figure 2. We do not evaluate *LCG* dataset because it lasts for only 13 days (less than the minimal reservation period of EC2). We compare in Figure 2: FCFS-CFH, *CoH*, CoH-oneType, and *CoH-R*. CoH-oneType is a variation of *CoH-R* which only uses heavily-utilized instance. For the *Dotalicious* dataset, *CoH-R* and CoH-oneType obtain lower cost than *CoH*. *CoH-R* can obtain lower cost than CoH-oneType, because it takes advantage of the cost reduction and flexibility provided by different reserved types. The result obtained by *CoH-R* using the *Dotalicious* dataset is about 13% to 20% lower than *CoH* and about 30% to 60% lower than FCFS-CFH. For the *Grid5000* dataset, the performance of *CoH-R* obtain about 3% to 5% lower cost than *CoH*, but still about 20% to 50% lower cost than the heuristic. The reason why *CoH-R* only obtains a small improvement on *Grid5000* is because *Grid5000* contains busty workloads with short jobs, and some very long jobs. As *CoH-R* always schedules jobs to VMs as soon as the jobs arrive in the system, this cause some long jobs to run on on-demand instances instead of the cheaper reserved instances. In summary, *CoH-R* can obtain about 20% and up to 60% lower cost than the heuristic. *CoH-R can obtain significantly lower cost than heuristics which use on-demand instances only.*

6 Related Work

A significant body of work has already focused on cloud resource scheduling from a cloud provider's perspective [20–23]. In this context, the common goals are to reduce the storage/electricity cost and to improve platform utilization. In contrast, in this study we schedule resources from a broker's perspective, with the goal to minimize the rental cost.

Previous studies have focused on provisioning and allocation of cloud resources, under various constraints. In contrast to these studies, which we describe in the following, we consider multiple instance types, billing models and heterogeneous workload. Closest to our work, Genaud and Gossa [24] evaluate provisioning heuristics for on-demand resources. Villegas et al. [11] conduct a performance-cost analysis of scheduling policies for IaaS Cloud. Deng [25] et al develop a portfolio scheduler. Oprescu and Kielmann [26] schedule bag-of-tasks on clouds focusing on budgets and runtime. They formulate the provisioning problem as a Bounded Knapsack Problem and allocate jobs to VMs round-robin. Mao et al. [27] propose a linear program for provisioning, and allocate jobs randomly to VMs. Sharma et al. [9] use on-demand instances and use migration but only for homogeneous workloads.

Hong et al [28] use a method to determine number of reserved instances of one reservation type. We show in our experiments that it is necessary to use multiple reserved instance types to reduce cost. Chaisiri [29] propose an algorithm to determine the number and types of reserved VM to be used by solving a stochastic IPP to minimize expected cost. They limit the on-demand instances can be only provisioned in specific provision phase, while we proactively provision VM at any necessary time. Ostermann and Prodan [30], and Song et al. [31] use spot-instance to reduce cost. Their work complement ours.

7 Conclusion

It is challenging to select among machine configurations and billing options offered by clouds to fit user demand while reducing operational cost. In this work, we propose *CoH*, a Cloud-base, online, Hybrid scheduling policy which make uses of multiple machine configurations to plan enough capacity for users with less cost. We formulate the resource provisioning and the job allocation problems as Integer Programming Problems (IPP). To obtain the scheduling decision online, *CoH* limits the time of exploration for a solution and only obtains an intermediate IPP solution. *CoH* makes scheduling decision by picking the best among the solution of IPP and various heuristics; thus, *CoH* operates as a portfolio scheduler. Further, we propose *CoH-R*, a policy that makes use of both on-demand and reserved instances to reduce cost. Via simulation using real-world traces, we show that our approaches can lead to significant lower cost than heuristics while operating online. We plan to investigate the wait-time and rental cost trade-off for bursty workload comprised of many short jobs.

Acknowledgement. The work is supported by CSC-TUD grant, the National Basic Research Program of China (973) under grant No.2011CB302603, NSFC under grant No.60903042, and by the STW/NOW Veni grant 11881.

References

1. Marshall, P., Keahey, K., Freeman, T.: Elastic site: Using clouds to elastically extend site resources. In: CCGrid 2010, pp. 43–52 (2010)
2. Schwiegelshohn, U., Badia, R.M., Bubak, M., et al.: Perspectives on grid computing. In: FGCS 2010, vol. 26(8) (2010)
3. Murphy, M., Kagey, B., Fenn, M., Goasguen, S.: Dynamic provisioning of virtual organization clusters. In: CCGrid 2009, pp. 364–371 (2009)
4. Ben-Yehuda, O.A., Schuster, A., Sharov, A., Silberstein, M., Iosup, A.: Expert: Pareto-efficient task replication on grids and a cloud. In: IPDPS 2012, pp. 167–178 (2012)
5. Fölling, A., Hofmann, M.: Improving scheduling performance using a Q-learning-based leasing policy for clouds. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 337–349. Springer, Heidelberg (2012)
6. de Assuncao, M.D., Costanzo, A.d., Buyya, R.: Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In: HPDC 2009, pp. 141–150 (2009)
7. Warneke, D., Kao, O.: Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. In: TPDS 2011, pp. 985–997 (2011)

8. Webb, J.: How the cloud helps Netflix (May 2011), `http://radar.oreilly.com/2011/05/netflix-cloud.html`
9. Sharma, U., Shenoy, P., Sahu, S., Shaikh, A.: A cost-aware elasticity provisioning system for the cloud. In: ICDCS 2011, pp. 559–570 (2011)
10. Nicolae, B., Cappello, F., Antoniu, G.: Optimizing multi-deployment on clouds by means of self-adaptive prefetching. In: Euro-Par 2011, pp. 503–513 (2011)
11. Villegas, D., Antoniou, A., Sadjadi, S.M., Iosup, A.: An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds. In: CCGrid 2012 (2012)
12. Huberman, B.A.: An Economics Approach to Hard Computational Problems. Science 275, 51–54 (1997)
13. Stillwell, M., Vivien, F., Casanova, H.: Dynamic fractional resource scheduling for hpc workloads. In: IPDPS 2010 (2010)
14. Iosup, A., Ostermann, S., Yigitbasi, N., Prodan, R., Fahringer, T., Epema, D.: Performance analysis of cloud computing services for many-tasks scientific computing. TPDS (2010)
15. Nae, V., Iosup, A., Prodan, R.: Dynamic resource provisioning in massively multiplayer online games. TPDS 22(3) (2011)
16. Feitelson, D.: Parallel Workloads Archive, `http://www.cs.huji.ac.il/labs/parallel/workload/`
17. Iosup, A., Li, H., Jan, M., Anoep, S., Dumitrescu, C., Wolters, L., Epema, D.H.J.: The grid workloads archive. FGCS 2008 24(7), 672–686 (2008)
18. Guo, A.Y., Iosup: The game trace archive. In: NETGAMES (2012)
19. Guo, Y., Shen, S., Visser, O., Iosup, A.: An Analysis of Online Match-Based Games. In: MMVE 2012 (2012)
20. Zhang, T., Du, Z., Chen, Y., Ji, X., Wang, X.: Typical virtual appliances: An optimized mechanism for virtual appliances provisioning and management. Journal of Systems and Software 84(3), 377 (2011)
21. Hadji, M., Zeghlache, D.: Minimum cost maximum flow algorithm for dynamic resource allocation in clouds. In: CLOUD 2012, pp. 876–882 (2012)
22. Ren, S., He, Y., Xu, F.: Provably-efficient job scheduling for energy and fairness in geographically distributed data centers. In: ICDCS 2012, pp. 22–31 (2012)
23. Tordsson, J., Montero, R.S., Moreno-Vozmediano, R., Llorente, I.M.: Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. Future Gener. Comput. Syst. 28(2) (2012)
24. Genaud, S., Gossa, J.: Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In: CLOUD 2011 (2011)
25. Deng, K., Verboon, R., Iosup, A.: A Periodic Portfolio Scheduler for Scientific Computing in the Data Center. In: JSSPP (2013)
26. Oprescu, A., Kielmann, T.: Bag-of-tasks scheduling under budget constraints. In: CloudCom 2010, pp. 351–359 (2010)
27. Mao, M.M., Li, J., Humphrey: Cloud auto-scaling with deadline and budget constraints. In: GRID 2010, pp. 41–48 (2010)
28. Hong, Y.J., Xue, J., Thottethodi: Selective commitment and selective margin: Techniques to minimize cost in an iaas cloud. In: ISPASS 2012, pp. 99–109 (2012)
29. Chaisiri, S., Lee, B.S., Niyato, D.: Optimization of resource provisioning cost in cloud computing. Transactions on Services Computing, 164–177 (2012)
30. Ostermann, S., Prodan, R.: Impact of variable priced cloud resources on scientific workflow scheduling. In: Euro-Par 2012, pp. 350–362 (2012)
31. Song, Y., Zafer, M., Lee, K.W.: Optimal bidding in spot instance market. In: INFOCOM 2012, pp. 190–198 (2012)

On-Line, Non-clairvoyant Optimization of Workflow Activity Granularity on Grids

Rafael Ferreira da Silva[1], Tristan Glatard[1], and Frédéric Desprez[2]

[1] University of Lyon, CNRS, INSERM, CREATIS, Villeurbanne, France
{rafael.silva,glatard}@creatis.insa-lyon.fr
[2] INRIA, University of Lyon, LIP, ENS Lyon, Lyon, France
Frederic.Desprez@inria.fr

Abstract. Controlling the granularity of workflow activities executed on widely distributed computing platforms such as grids is required to reduce the impact of task queuing and data transfer time. Most existing granularity control approaches assume extensive knowledge about the applications and resources (e.g. task duration on each resource), and that both the workload and available resources do not change over time. We propose a granularity control algorithm for platforms where such clairvoyant and offline conditions are not realistic. Our method groups tasks when the fineness degree of the application, which takes into account the ratio of shared data and the queuing/round-trip time ratio, becomes higher than a threshold determined from execution traces. The algorithm also de-groups task groups when new resources arrive. The application's behavior is constantly monitored so that the characteristics useful for the optimization are progressively discovered. Experimental results, obtained with 3 workflow activities deployed on the European Grid Infrastructure, show that (i) the grouping process yields speed-ups of about 2.5 when the amount of available resources is constant and that (ii) the use of de-grouping yields speed-ups of 2 when resources progressively appear.

1 Introduction

Software-as-a-service (SaaS) platforms deployed on production grids, for instance the Virtual Imaging Platform (VIP [1]) and other science gateways [2,3,4], usually have no *a-priori* model of the execution time of their applications because (i) task costs depend on input data with no explicit model, and (ii) characteristics of the available resources, in particular network and RAM, depend on background load. Modeling application execution time in these conditions requires cumbersome experiments which cannot be conducted for every new application in the platform. As a consequence, such SaaS platforms operate in *non-clairvoyant* conditions, where little is known about executions before they actually happen. Such platforms also run in *online* conditions, i.e. users may launch or cancel applications at any time and resources may appear or disappear at any time too. Our ultimate goal is to control the behavior of these non-clairvoyant, online platforms to limit human intervention required for their operation. In other works, we address error recovery [5] and fairness of resource allocation [6] among executions. This paper focuses on task granularity optimization.

The low performance of *lightweight* (a.k.a. *fine-grained*) tasks is a common problem on widely distributed platforms where the communication overhead and queuing time are high, such as grid systems. To address this issue, fine-grained tasks are commonly grouped into *coarse-grained* tasks [7,8,9,10,11], which reduces the cost of data transfers when grouped tasks share input data [7] and saves queuing time when resources are limited [8]. However, task grouping also limits parallelism and therefore should be used sparingly.

We consider such a granularity problem in a SaaS platform executing workflows on a grid. Workflows are compositions of *activities* that consist only of a program description. At runtime, activities receive data and spawn tasks for which the executable name and input data are known, but the computational cost and produced data volume are not. We propose an algorithm to optimize the granularity of workflow activities on non-clairvoyant online grid platforms. Our algorithm progressively discovers the characteristics of the running applications to compute a metric quantifying the fineness degree of a task group. This fineness metric includes measured task queuing times, and median-based estimations of task running times and transfer time of shared input data. Tasks are grouped when the fineness metric goes beyond a threshold learned from platform traces. In addition, a de-grouping mechanism is triggered when parallelism losses are detected, i.e. when the number of queued tasks is lower than the number of running tasks. The method is implemented in VIP, and evaluated with different applications, in production conditions, on the European Grid Infrastructure (EGI[1]). The contributions of this work are the following:

- We propose a new metric to quantify workflow activity fineness in online and non-clairvoyant conditions;
- We design task grouping and de-grouping algorithms that are triggered by the fineness metric in the control loop described in [5];
- We show, on 3 different applications, that the method provides significant speed-up in production conditions, on the European Grid Infrastructure.

To the best of our knowledge, this algorithm is the first example of task granularity control in a non-clairvoyant online context. The next Section gives an overview of the related work, Section 3 details the granularity control process, Section 4 reports experiments and results, and the paper closes with a discussion and conclusions.

2 Related Work

Muthuvelu et al. [9] proposed an algorithm to group bag of tasks based on their granularity size – defined as the processing time of the task on the resource.

[1] `http://www.egi.eu`

Resources are ordered by their decreasing values of capacity (in MIPS) and tasks are grouped up to the resource capacity. This process continues until all tasks are grouped and assigned to resources. Then, Keat et al. [10] and Ang et al. [11] extended the work of Muthuvelu et al. by introducing bandwidth in the scheduling framework to enhance the performance of task scheduling. Resources are sorted in decreasing order of bandwidth, then assigned to grouped tasks downward ordered by processing requirement length. The size of a grouped task is determined from the task cost in millions instructions (MI).

Later, Muthuvelu et al. [12] extended [9] to determine task granularity based on QoS requirements, task file size, estimated task CPU time, and resource constraints. Meanwhile, Liu & Liao [13] proposed an adaptive fine-grained job scheduling algorithm (AFJS) to group lightweight tasks according to processing capacity (in MIPS) and bandwidth (in Mb/s) of the current available resources. Tasks are sorted in decreasing order of MI, then clustered by a greedy algorithm. To accommodate with resource dynamicity, the grouping algorithm integrates monitoring information about the current availability and capability of resources. Afterwards, Soni et al. [14] proposed an algorithm to group lightweight tasks into coarse-grained tasks (GBJS) based on processing capability, bandwidth, and memory-size of the available resources. Tasks are sorted into ascending order of required computational power, then, selected in first come first serve order to be grouped according to the capability of the resources. Zomaya and Chan [15] studied limitations and ideal control parameters of task clustering by using genetic algorithms. Their algorithm performs task selection based on the earliest task start time and task communication costs; it converges to an optimal solution of the number of clusters and tasks per cluster.

Although the reviewed works significantly reduce communication and processing time, neither of them are non-clairvoyant and online at the same time. Recently, Muthuvelu et al. [16,7] proposed an online scheduling algorithm to determine the task granularity of compute-intensive bag-of-tasks applications. The granularity optimization is based on task processing requirements, resource-network utilisation constraint (maximum time a scheduler waits for data transfers), and users QoS requirements (user's budget and application deadline). Submitted tasks are categorised according to their file sizes, estimated CPU times, and estimated output file sizes, and arranged in a tree structure. The scheduler selects a few tasks from these categories to perform resource benchmarking. Tasks are grouped according to seven objective functions of task granularity, and submitted to resources. The process restarts upon task arrival. Although this is an online approach, the solution is still clairvoyant.

3 Task Granularity Control Process

Algorithm 1 describes our task granularity control composed of two processes: (i) the fineness control process groups too fine task groups for which the fineness degree η_f is greater than threshold τ_f, and (ii) the coarseness control process de-groups too coarse task groups for which the coarseness degree η_c is greater

Algorithm 1. Main loop for granularity control

```
1: input: $n$ waiting tasks
2: create $n$ 1-task groups $T_i$
3: while there is an active task group do
4:     wait for timeout or task status change
5:     determine fineness degree $\eta_f$
6:     if $\eta_f > \tau_f$ then
7:         group task groups using Algorithm 2
8:     end if
9:     determine coarseness degree $\eta_c$
10:    if $\eta_c > \tau_c$ then
11:        degroup coarsest task groups
12:    end if
13: end while
```

than threshold τ_c. This section describes how η_f, η_c, τ_f and τ_c are computed, and details the grouping and de-grouping algorithms.

3.1 Fineness Control

Fineness Degree η_f. Let n be the number of waiting tasks in a workflow activity, and m the number of task groups. Tasks related to an activity are assumed independent, but with similar execution times (bag of tasks). This hypothesis is critical for our method. Initially, 1 group is created for each task ($n = m$). T_i is the set of tasks in group i, and n_i is the number of tasks in T_i. Groups are a partition of the set of waiting tasks: $T_i \bigcap_{i \neq j} T_j = \emptyset$ and $\sum_{i=1}^{m} n_i = n$. The activity fineness degree η_f is the maximum of all group fineness degrees f_i:

$$\eta_f = \max_{i \in [1,m]} (f_i). \tag{1}$$

All η_f are in [0,1], and high fineness degrees indicate fine granularities. We use a max operator in this equation to ensure that *any* task group with a too fine granularity will be detected. The fineness degree f_i of group i is defined as:

$$f_i = d_i \cdot r_i, \tag{2}$$

where d_i is the ratio between the transfer time of the input data shared among all tasks in the activity, and the total execution time of the group:

$$d_i = \frac{\tilde{t}_{\_shared}}{\tilde{t}_{\_shared} + n_i(\tilde{t} - \tilde{t}_{\_shared})},$$

where $\tilde{t}_{\_shared}$ is the median transfer time of the input data shared among all tasks in the activity, and $\tilde{t}$ is the sum of its median task phase durations corresponding to application setup, input data transfer, application execution and output data transfer: $\tilde{t} = \tilde{t}_{\_setup} + \tilde{t}_{\_input} + \tilde{t}_{\_exec} + \tilde{t}_{\_output}$. Median values $\tilde{t}_{\_shared}$ and $\tilde{t}$ are computed from values measured on completed tasks. When less than 2 tasks are completed, medians remain undefined and the control process is inactive. This online estimation makes our process non-clairvoyant with respect to the task duration which is progressively estimated as the workflow activity runs. Yet, it assumes that all tasks in an activity have similar durations.

In equation 2, r_i is the ratio between the max of the task current queuing times q_i in the group (measured for each task individually), and the total round-trip time (queuing+execution) of the group:

$$r_i = \frac{\max_{j\in[1,n_i]} q_j}{\max_{j\in[1,n_i]} q_j + \tilde{t}_{\_shared} + n_i(\tilde{t} - \tilde{t}_{\_shared})}$$

Group queuing time is the max of all task queuing times in the group; group execution time is the time to transfer shared input data plus the time to execute all task phases in the group except for the transfers of shared input data. Note that d_i, r_i, and therefore f_i and η_f are in $[0, 1]$. η_f tends to 0 when there is little shared input data among the activity tasks or when the task queuing times are low compared to the execution times; in both cases, grouping tasks is indeed useless. Conversely, η_f tends to 1 when the transfer time of shared input data becomes high, and the queuing time is high compared to the execution time; grouping is needed in this case.

Threshold Value $\boldsymbol{\tau_f}$. The threshold value for η_f separates configurations where the activity's fineness is acceptable ($\eta_f \leq \tau_f$) from configurations where the activity is too fine ($\eta_f > \tau_f$). We determine τ_f from execution traces, inspecting the modes of the distribution of η_f. Values of η_f in the highest mode of the distribution, i.e. which are clearly separated from the others, will be considered too fine.

We use traces collected from VIP [1] between January 2011 and April 2012, made available through the science-gateway workload archive [17]. The data set contains 680,988 tasks (including resubmissions and replications) linked to activities of 2,941 workflows executed by 112 users; task average waiting time is about 36 min. Applications deployed in VIP are described as workflows executed using the MOTEUR workflow engine [18]. Resource provisioning and task scheduling are provided by DIRAC [19] using so-called "pilot jobs". Resources are provisioned online with no advance reservations. Tasks are executed on the biomed virtual organization (VO) of the European Grid Infrastructure (EGI)[2] which has access to some 150 computing sites world-wide and to 120 storage sites providing approximately 4 PB of storage. Fig. 1 (left) shows the distribution of sites per country supporting the biomed VO.

The fineness degree η_f was computed after each event found in the data set. Fig. 1 (right) shows the histogram of these values. The histogram appears bimodal, which indicates that η_f separates platform configurations in two distinct groups. We assume that these groups correspond to "acceptable fineness" (lowest mode) and "too fine granularity" (highest mode), and thus we choose $\tau_f = 0.55$. For $\eta_f \geq 0.55$, task grouping will therefore be triggered.

[2] `http://www.egi.eu`

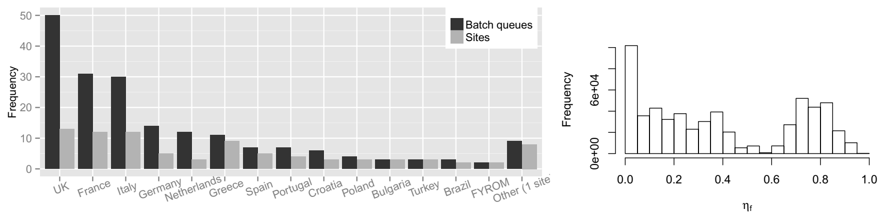


Fig. 1. Distribution of sites and batch queues per country in the biomed VO (January 2013) (*left*) and histogram of fineness incident degree sampled in bins of 0.05 (*right*)

Task Grouping. We assume that running tasks cannot be pre-empted, i.e. only waiting tasks can be grouped. Algorithm 2 describes our task grouping. Groups with $f_i > \tau_f$ are grouped pairwise until $\eta_f \leq \tau_f$ or until the amount of waiting groups Q is smaller or equal to the amount of running groups R. Although η_f ignores scattering (Eq. 1 uses a max), the algorithm considers it by grouping tasks in all groups where $f_i > \tau_f$. Ordering groups by decreasing f_i values tends to equally distribute tasks among groups. The grouping process stops when $Q \leq R$ to avoid parallelism loss. This condition also avoids conflicts with the de-grouping process described in the next sub-section.

Algorithm 2. Task grouping

```
1: input: $f_1$ to $f_m$ //group fineness degrees, sorted in decreasing order
2: input: $Q$, $R$ // number of queued and running task groups
3: for $i = 1$ to $m - 1$ do
4:     $j = i + 1$
5:     while $f_i > \tau_f$ and $Q > R$ and $j \leq m$ do
6:         if $f_j > \tau_f$ then
7:             Group all tasks of $T_j$ into $T_i$
8:             Recalculate $f_i$ using Equation 2
9:             $Q = Q - 1$
10:        end if
11:        $j = j + 1$
12:    end while
13:    $i = j$
14: end for
15: Delete all empty task groups
```

3.2 Coarseness Control

Condition $Q > R$ used in Algorithm 2 ensures that all resources will be exploited *if the number of available resources is stationary.* In case the number of available resources decreases, the fineness control process may further reduce the number of groups. However, if the number of available resources increases, task groups may need to be de-grouped to maximize resource exploitation. This de-grouping is implemented by our coarseness control process.

The coarseness control process monitors the value of η_c defined as:

$$\eta_c = \frac{R}{Q + R}. \tag{3}$$

The threshold value τ_c is set to 0.5 so that $\eta_c > \tau_c \Leftrightarrow Q < R$.

Table 1. Example

Let's consider a workflow composed of one activity with 10 tasks initially split in 10 groups, and assume that task input data are shared among all tasks (i.e. $\tilde{t}\_shared = \tilde{t}\_input$).
Let $\tilde{t} = 10$ and $\tilde{t}\_shared = 7$ (in arbitrary time units) obtained from two completed task groups. At time t, we assume $R = 2$ and $Q = 6$ with the following values for waiting task groups:

i	$\max_{j\in[1,n_i]} q_j$	d_i	r_i	f_i
5	50	0.70	0.83	0.58
6	48	0.70	0.82	0.58
7	45	0.70	0.81	0.57
8	43	0.70	0.81	0.57
9	41	0.70	0.80	0.56
10	40	0.70	0.80	0.56

Eq. 1 gives $\eta_f = 0.58$. As $\eta_f > \tau_f = 0.55$ and $Q > R$, the activity is considered too fine and task grouping is triggered. Groups with $f_i > \tau_f$ are grouped pairwise until $\eta_f \leq \tau_f$ or $Q \leq R$:

i	$\max_{j\in[1,n_i]} q_j$	d_i	r_i	f_i
11 [5,6]	50	0.53	0.79	0.42
12 [7,8]	45	0.53	0.77	0.41
13 [9,10]	41	0.53	0.76	0.40

Groups 5 and 6, 7 and 8, and 9 and 10 are grouped into groups 11, 12, and 13.

Let's consider that at time $t' > t$, group 11 starts running, thus $Q = 2 < R = 3$.
Eq. 3 gives $\eta_c = 0.6$. As $\eta_c > \tau_c = 0.5$, the activity is consider too coarse and task de-grouping is triggered. Then, group 13 is de-grouped to balance η_c.

When an activity is considered too coarse, its groups are ordered by increasing values of η_f and the first groups (i.e. the coarsest ones) are split until $\eta_c < \tau_c$. Note that de-grouping increases the number of queued tasks, therefore tends to reduce η_c. Table 1 illustrates the method on a simple example.

4 Experiments and Results

The experiments presented hereafter evaluate the fineness control process under stationary load, and the interest of controlling coarseness under non-stationary load in a production environment.

4.1 Experiment Conditions

The granularity control process was implemented as a plugin of the MOTEUR workflow manager, receiving notifications about task status changes and task phase durations. The plugin then uses this data to group and de-group tasks according to Algorithm 1, where the timeout value is set to 2 minutes.

The target computing platform for these experiments is the biomed VO where the traces used to determine τ_f were acquired (see Section 3.1). To ensure resource limitation without flooding the production system, experiments are performed only on 3 sites of different countries. Tasks generated by MOTEUR are submitted to the biomed VO of EGI using the DIRAC scheduler.

Three workflow activities, implementing different kinds of medical image simulation, are used in the experiments. `SimuBloch` [20] is a very short activity made of 25 concurrent tasks; task CPU time is of a few seconds; input data size is about 15 MB and output is less than 5 MB; $\tilde{t}_{\_shared}$ is about 90% of the execution time. `FIELD-II` [21] consists of 122 data-intensive concurrent tasks ranging from a few seconds to 15 minutes of CPU time (tasks have the same cost, but their duration is resource-dependent); it transfers 208 MB of input data and outputs about 40 KB of data; $\tilde{t}_{\_shared}$ ranges from 40% to 60% of the execution time. `PET-Sorteo/emission` [22] has 80 tasks of 2 CPU minutes; input data size is about 20 MB and output is about 50 MB; $\tilde{t}_{\_shared}$ ranges from 50% to 80% of the execution time.

Two sets of experiments are conducted, under different load patterns. Experiment 1 evaluates the fineness control process only under stationary load. It consists of separated executions of `SimuBloch`, `FIELD-II`, and `PET-Sorteo/emission`. A workflow activity using our task grouping mechanism (`Fineness`) is compared to a control activity (`No-Granularity`). Resource contention on the 3 execution sites is maintained high and constant so that no de-grouping is required.

Experiment 2 evaluates the interest of using the de-grouping control process under non-stationary load. It uses activity `FIELD-II`. An execution using both fineness and coarseness control (`Fineness-Coarseness`) is compared to an execution without coarseness control (`Fineness`) and to a control execution (`No-Granularity`). Executions are started under resource contention, but the contention is progressively reduced during the experiment. This is done by submitting a heavy workflow before the experiment starts, and canceling it when half of the experiment tasks are completed.

For both experiments, control and tested executions are launched simultaneously to ensure similar grid conditions. As no online task modification is possible in DIRAC, we implemented task grouping by canceling queued tasks and submitting grouped tasks as a new task. For each grouped task resubmitted in the `Fineness` or `Fineness-Coarseness` executions, a task in the `No-Granularity` is resubmitted too to ensure equal race conditions for resource allocation, and that each execution faces the same re-submission overhead. Five repetitions of each experiment are performed, along a time period of 4 weeks to cover different grid conditions. We use MOTEUR 0.9.21, configured to resubmit failed tasks up to 5 times, and with the task replication mechanism described in [5] activated. We use the DIRAC v6r6p2 instance provided by France-Grilles[3]. Results could not be compared to other grouping/de-grouping methods due to the lack of non-clairvoyant, online method available in the literature (see Section 2).

4.2 Results and Discussion

Experiment 1: Fig. 2 shows the makespan of `SimuBloch`, `FIELD-II`, and `PET-Sorteo/emission` executions. `Fineness` yields a significant makespan reduction for all repetitions. Table 2 shows the makespan (M) values and the number of task groups. The task grouping mechanism is not able to group all `SimuBloch` tasks in a single group because 2 tasks must be completed for the process to have enough information about the application (i.e. $\tilde{t}_{\_shared}$ and $\tilde{t}$

[3] `https://dirac.france-grilles.fr`

can be computed). This is a constraint of our non-clairvoyant conditions, where task durations cannot be determined in advance. `FIELD-II` tasks are initially not grouped, but as the queuing time becomes important, tasks are considered too fine and grouped. `PET-Sorteo/emission` is an intermediary case where only a few tasks are grouped. Results show that the task grouping mechanism speeds up `SimuBloch` and `FIELD-II` executions up to a factor of 2.6, and `PET-Sorteo/emission` executions up to a factor of 2.5.

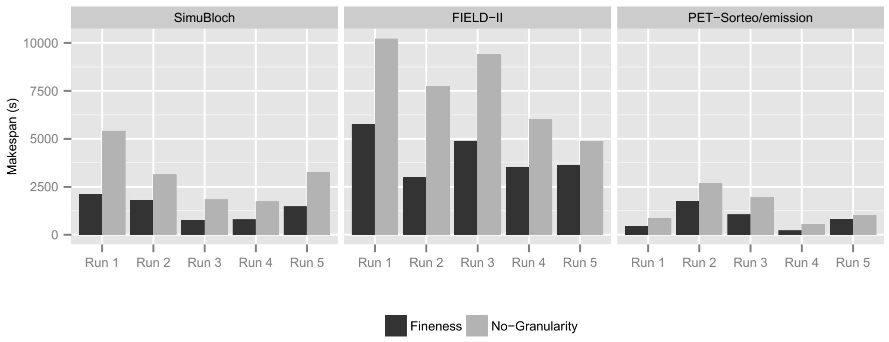


Fig. 2. Experiment 1: makespan for `Fineness` and `No-Granularity` executions for the 3 workflow activities under stationary load

Experiment 2: Fig. 3 shows the makespan (top) and evolution of task groups (bottom). Makespan values are reported in Table 3. In the first three repetitions, resources appear progressively during workflow executions. `Fineness` and `Fineness-Coarseness` speed up executions up to a factor of 1.5 and 2.1. Since `Fineness` does not benefit of newly arrived resources, it has a lower speed up compared to `No-Granularity` due to parallelism loss. In the two last repetitions, the de-grouping process in `Fineness-Coarseness` allows to reach similar performance than `No-Granularity`, while `Fineness` is penalized by its lack of adaptation: a slowdown of 20% is observed compared to `No-Granularity`.

Table 3 also shows the average queuing time values for Experiment 2. The linear correlation coefficient between the makespan and the average queuing time is 0.91, which indicates that the makespan evolution is indeed correlated to the evolution of the queuing time induced by the granularity control process.

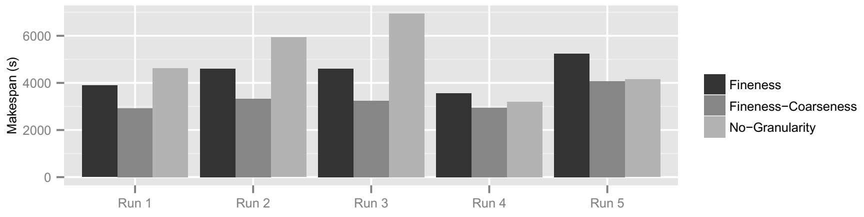
Our task granularity control process works best under high resource contention, when the amount of available resources is stable or decreases over time (Experiment 1). Coarseness control can cope with soft increases in the number of available resources (Experiment 2), but fast variations remain difficult to handle. In the worst-case scenario, tasks are first grouped due to resource limitation, and resources suddenly appear once all task groups are already running. In this case the de-grouping algorithm has no group to handle, and granularity control penalizes the execution. Task pre-emption should be added to the method to address this scenario.

Table 2. Experiment 1: makespan (M) and number of task groups for `SimuBloch`, `FIELD-II` and `PET-Sorteo/emission` executions for the 5 repetitions

		SimuBloch		FIELD-II		PET-Sorteo	
		M (s)	Groups	M (s)	Groups	M (s)	Groups
1	No-Granularity	5421	25	10230	122	873	80
	Fineness	2118	3	5749	80	451	57
2	No-Granularity	3138	25	7734	122	2695	80
	Fineness	1803	3	2982	75	1766	40
3	No-Granularity	1831	25	9407	122	1983	80
	Fineness	780	4	4894	73	1047	53
4	No-Granularity	1737	25	6026	122	552	80
	Fineness	797	6	3507	61	218	64
5	No-Granularity	3257	25	4865	122	1033	80
	Fineness	1468	4	3641	91	831	71

Table 3. Experiment 2: makespan (M) and average queuing time ($\bar{q}$) for `FIELD-II` workflow execution for the 5 repetitions

	Run 1		Run 2		Run 3		Run 4		Run 5	
	M (s)	$\bar{q}$ (s)	M (s)	$\bar{q}$ (s)	M (s)	$\bar{q}$ (s)	M (s)	$\bar{q}$ (s)	M (s)	$\bar{q}$ (s)
No-Granularity	4617	2111	5934	2765	6940	3855	3199	1863	4147	2295
Fineness	3892	2036	4607	2090	4602	2631	3567	1928	5247	2326
Fineness-Coarseness	2927	1708	3335	1829	3247	2091	2952	1586	4073	2197



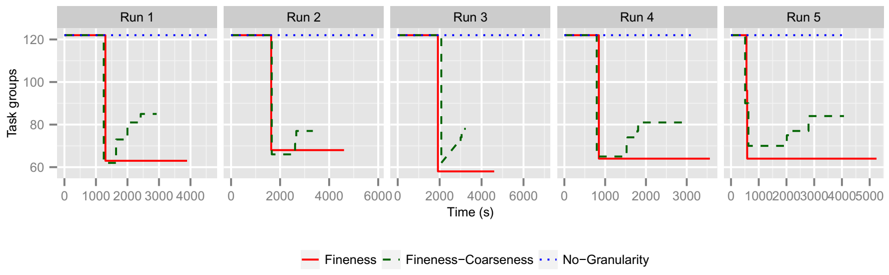


Fig. 3. Experiment 2: makespan (top) and evolution of task groups (bottom) for `FIELD-II` executions under non-stationary load (resources arrive during the experiment)

In addition, our method is dependent on the capability to extract enough accurate information from completed tasks to handle active tasks using median estimates. This may not be the case for activities which execute only a few tasks.

5 Conclusion

We presented a method to address task granularity in distributed workflows in an online and non-clairvoyant environment. We defined a metric η_f for online determination of task fineness based on queue waiting time and estimated data transfer time of shared input data. For high η_f values, tasks are considered too fine and task grouping is triggered. Queued tasks are grouped pairwise as long as the number of queued tasks is greater than the number of running tasks and η_f is considered too fine. We also define a metric η_c for online determination of task coarseness based on the ratio of the number of queued tasks related to the number of running tasks. This metric aims at maximizing resource exploitation by de-grouping tasks groups when the number of available resources increases.

The task granularity control strategy was implemented in the MOTEUR workflow engine and deployed on EGI with the DIRAC resource manager. We tested it on three applications extracted from the Virtual Imaging Platform, a science gateway for medical simulation. Two experiments were conducted, to evaluate the fineness control process only under stationary load and the fineness and coarseness control process under non-stationary load. Results showed that under stationary load, our fineness control process significantly reduces the makespan of all applications. Under non-stationary load, task grouping is penalized by its lack of adaptation, but our de-grouping algorithm corrects it in case variations in the number of available resources are not too fast. In our future work, task preemption will be added to the method to further improve the handling of resource dynamicity. A comparative study against pilot job approaches and clairvoyant methods will also be considered. We will also study the impact of task duration variability on the proposed method.

Acknowledgment. This work is funded by the French National Agency for Research under grant ANR-09-COSI-03 "VIP". The research leading to this publication has also received funding from the EC FP7 Programme under grant agreement 312579 ER-flow = Building an European Research Community through Interoperable Workflows and Data, and the framework LABEX ANR-11-LABX-0063 of Universit de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR). Results obtained in this paper were computed on the biomed virtual organization of the European Grid Infrastructure (http://www.egi.eu). We thank the European Grid Infrastructure and its supporting National Grid Initiatives, in particular France-Grilles, for providing the technical support, computing and storage facilities.

References

1. Glatard, T., et al.: A virtual imaging platform for multi-modality medical image simulation. IEEE Transactions on Medical Imaging 32, 110–118 (2013)
2. Shahand, S., et al.: Front-ends to Biomedical Data Analysis on Grids. In: Proceedings of HealthGrid 2011, Bristol, UK (June 2011)
3. Kacsuk, P.: P-GRADE Portal Family for Grid Infrastructures. Concurrency and Computation: Practice and Experience 23(3), 235–245 (2011)

4. Barbera, R., et al.: Supporting e-science applications on e-infrastructures: Some use cases from latin america. In: Grid Computing, pp. 33–55 (2011)
5. Ferreira da Silva, R., Glatard, T., Desprez, F.: Self-healing of operational workflow incidents on distributed computing infrastructures. In: CCGrid 2012, pp. 318–325 (2012)
6. da Silva, R.F., Glatard, T., Desprez, F.: Workflow fairness control on online and non-clairvoyant distributed computing platforms. In: Wolf, F., Mohr, B., an Ney, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 102–113. Springer, Heidelberg (2013)
7. Muthuvelu, N., et al.: Task granularity policies for deploying bag-of-task applications on global grids. FGCS 29(1), 170–181 (2012)
8. Singh, G., et al.: Workflow task clustering for best effort systems with pegasus. In: Mardi Gras 2008, pp. 9:1–9:8. ACM, New York (2008)
9. Muthuvelu, N., et al.: A dynamic job grouping-based scheduling for deploying applications with fine-grained tasks on global grids. In: Proceedings of the 2005 Australasian Workshop on Grid Computing and E-Research, ACSW Frontiers 2005, vol. 44, pp. 41–48. Australian Computer Society, Inc. (2005)
10. Keat, N.W., et al.: Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing. Malaysian Journal of Computer Science 19 (2006)
11. Ang, T., et al.: A bandwidth-aware job grouping-based scheduling on grid environment. Information Technology Journal 8, 372–377 (2009)
12. Muthuvelu, N., Chai, I., Eswaran, C.: An adaptive and parameterized job grouping algorithm for scheduling grid jobs. In: 10th International Conference on Advanced Communication Technology, ICACT 2008, vol. 2, pp. 975–980 (2008)
13. Liu, Q., Liao, Y.: Grouping-based fine-grained job scheduling in grid computing. In: ETCS 2009, vol. 1, pp. 556–559 (2009)
14. Soni, V.K., et al.: Grouping-based job scheduling model in grid computing. World Academy of Science, Engineering and Technology 41, 781–784 (2010)
15. Zomaya, A.Y., Chan, G.: Efficient clustering for parallel tasks execution in distributed systems. In: 18th IPDPS, pp. 167–174 (2004)
16. Muthuvelu, N., Chai, I., Chikkannan, E., Buyya, R.: On-line task granularity adaptation for dynamic grid applications. In: Hsu, C.-H., Yang, L.T., Park, J.H., Yeo, S.-S. (eds.) ICA3PP 2010, Part I. LNCS, vol. 6081, pp. 266–277. Springer, Heidelberg (2010)
17. Ferreira da Silva, R., Glatard, T.: A Science-Gateway Workload Archive to Study Pilot Jobs, User Activity, Bag of Tasks, Task Sub-Steps, and Workflow Executions. In: CoreGRID, Rhodes, GR (2012)
18. Glatard, T., Montagnat, J., Lingrand, D., Pennec, X.: Flexible and Efficient Workflow Deployment of Data-Intensive Applications on Grids with MOTEUR. IJHPCA 22(3), 347–360 (2008)
19. Tsaregorodtsev, A., et al.: DIRAC3. The New Generation of the LHCb Grid Software. Journal of Physics: Conference Series 219(6), 062029 (2009)
20. Cao, F., Commowick, O., Bannier, E., Ferré, J.-C., Edan, G., Barillot, C.: MRI estimation of t_1 relaxation time using a constrained optimization algorithm. In: Yap, P.-T., Liu, T., Shen, D., Westin, C.-F., Shen, L. (eds.) MBIA 2012. LNCS, vol. 7509, pp. 203–214. Springer, Heidelberg (2012)
21. Jensen, J., Svendsen, N.: Calculation of pressure fields from arbitrarily shaped, apodized and excited ultrasound transducers. IEEE T-UFFC 39(2), 262–267 (1992)
22. Reilhac, A., et al.: PET-SORTEO: Validation and Development of Database of Simulated PET Volumes. IEEE Trans. on Nuclear Science 52, 1321–1328 (2005)

Application-Centric Resource Provisioning for Amazon EC2 Spot Instances

Sunirmal Khatua[1] and Nandini Mukherjee[2]

[1] University of Calcutta, Kolkata, India
skhatuacomp@caluniv.ac.in
[2] Jadavpur University, Kolkata, India
nmukherjee@jdvu.ac.in

Abstract. In late 2009, Amazon introduced spot instances to offer their unused resources at lower cost with reduced reliability. Amazon's spot instances allow customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current spot price. The spot price changes periodically based on supply and demand of spot instances, and customers whose bid exceeds it gain access to the available spot instances. Customers may expect their services at lower cost with spot instances compared to on-demand or reserved. However the reliability is compromised since the instances (IaaS) providing the service (SaaS) may become unavailable at any time without any notice to the customer. In this paper, we study various checkpointing schemes to increase the reliability over spot instances. Also we devise a novel checkpointing scheme on top of application-centric resource provisioning framework that increases the reliability while reducing the cost significantly.

Keywords: resource provisioning, spot instances, checkpointing.

1 Introduction

The era of cloud computing provides high utilization and high flexibility of managing the computing resources. The elasticity and on demand availability features of cloud computing ensure high utilization of resources. Furthermore, resources can be availed from templates that enforce standards so that resources can be used with best management considerations without prior knowledge. Therefore, flexibility of managing the computing resources is also high in a cloud environment. The cloud computing service models incorporate Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS provides raw computing resources with different capacity in the form of Virtual Machines (VM). Cloud Service Providers (CSP), like Google [16], Amazon [15] etc. provide these services and charge prices against these services from the Cloud Service Users (CSU). Among many such providers, Amazon defines the capacity of resources in the form of different instance types [11] based on storage, compute unit and I/O performance. The cost of these instance types depends on the purchasing models [12] defined by Amazon namely on-demand, reserved and spot.

$On - demand$ instances let one pay for compute capacity by the hour with no long-term commitments or upfront payments. However, with on-demand instances one may not have access to the resources immediately due to high demand for a specific instance type in a specific availability zone. On the other hand, *reserved instances* facilitate the client to make a low, one-time, upfront payment for an instance, reserve it and get significant discount on hourly charge over on-demand instances. Reserved instances are always available for the duration for which the clients reserve. In contrast with the above two policies, where rates are fixed, *spot instances* provide the ability for customers to purchase compute capacity with no upfront commitment and at a variable hourly rate with a customer-defined upper bound (bid) on the rate. Spot instances are available only during the time when the spot price is below the customer defined bid.

Thus spot instances make the resources unreliable in nature and inappropriate for long running jobs like image processing, gene sequence analysis etc. At the same time, they offer the opportunity to accomplish such jobs at a much lower cost than on demand or reserved policies. Clearly, checkpointing (saving partially completed jobs to be resumed latter) may be a good option to make a tradeoff between the cost and reliability. Again, the time of taking a checkpoint and the frequency of taking the checkpoints directly affect the cost and reliability. Sufficient research effort is needed to properly set the time and frequency of taking the checkpoints.

The rest of the paper is organized as follows. A brief review of the related works is presented in Section 2. An overview of the application centric resource provisioning framework is given in Section 3. Section 4 deals with the existing checkpointing schemes for spot instances while a proposed checkpointing scheme for the application centric resource provisioning framework is described in Section 5. A simulated result for comparing the proposed checkpointing scheme with existing ones is presented in Section 6. Finally, we conclude with a direction of future work in Sections 7.

2 Related Work

During the last couple of years, a lot of works [1] [8]-[9] concentrate on the cloud management aspect from the economic point of view. Most of them adapt a middleware based (broker) approach to optimize the resource requirement for a given cloud application. In our previous work [1], we provide a novel framework for such a middleware. It identifies the key components of the middleware for auto deploying, auto scaling, providing robustness and availability of heterogeneous cloud applications. A model for optimal cloud resource scheduling based on stochastic integer programming technique is proposed in [8]. A similar technique is also used in [9] to optimize the resource requirement of a cloud application. This work tries to minimize the total provisioning cost by adjusting the tradeoff between the reserved and on-demand resource provisioning plans.

Some research works [2]-[6] also consider Amazon EC2 spot instances [13] for providing economic benefit to cloud service users considering availability and reliability. Various checkpointing techniques have been discussed in [2] to provide reliability with Amazon spot instances at lower cost. In this paper, we study some of these techniques and evaluate their performances. We also investigate the effectiveness of application centric resource provisioning framework [1] for actively monitoring the deployed spot instances for an application and for taking necessary actions as the spot intances become unavailable or the spot price changes. Finally, we propose and evaluate a novel checkpointing scheme for the application centric resource provisioning framework that outperforms all the checkpointing schemes defined in [2].

3 Application-Centric Resource Provisioning Framework

An Application-centric resource provisioning framework along with the unified definition of an application is proposed in [1]. A brief description of functioning of the application centric resource provisioning framework is depicted in Figure 1. The framework consists of two key subsystems namely Provisioning subsystem and Monitoring subsystem.

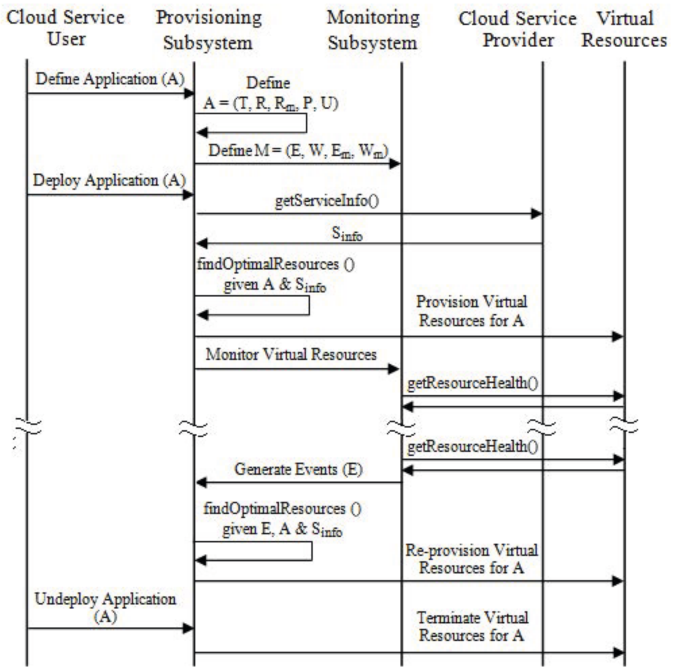

Cloud Service User
Provisioning Subsystem
Monitoring Subsystem
Cloud Service Provider
Virtual Resources
Define Application (A)
Define A = (T, R, R_m, P, U)
Define M = (E, W, E_m, W_m)
Deploy Application (A)
getServiceInfo()
S_info
findOptimalResources () given A & S_info
Provision Virtual Resources for A
Monitor Virtual Resources
getResourceHealth()
getResourceHealth()
Generate Events (E)
findOptimalResources () given E, A & S_info
Re-provision Virtual Resources for A
Undeploy Application (A)
Terminate Virtual Resources for A

Fig. 1. Resource provisioning algorithm

3.1 Provisioning Subsystem

The provisioning subsystem determines optimal provisioning of virtual resources for an application A) satisfying the policies (P) specified for it. The application's required service level is stored in the policy (P). The provisioning subsystem queries various providers to get information about their offered services (S_{info}). S_{info} consists of provider id, service id, QoS id and the associated cost. The provisioning subsystem uses P (desired service level), S_{info} and an optimization algorithm to find the optimal resource requirement for the application while maintaining the desired service level.

3.2 Monitoring Subsystem

The Monitoring subsystem implements a feedback system to inform the provisioning subsystem about the current state of the deployed application. The monitoring subsystem actively monitors the state of the deployed application and generates various events [1] to designate a change in the application state. Once an event is generated, the monitoring subsystem sends the event to the provisioning subsystem. Once an event(E) is received, the provisioning subsystem analyzes the event and uses E, P, S_{info} and an optimization algorithm for reprovisioning the application onto appropriate resources.

4 Checkpointing Schemes for Amazon EC2 Spot Instances

In this paper multiple providers of application centric resource provisioning are not considered. Instead, we consider the spot market of Amazon EC2 public CSP only. The concept can be generalised to any CSP supporting spot model.

As discussed earlier, the variable price of spot instances makes them an important consideration for optimizing resource requirement for an application. However, their volatile nature makes them inherently unreliable and hence the optimization algorithms become more challenging than the other instances.

4.1 Characteristics of Spot Instances

Before dealing with the challenges of optimizing the use of spot instances, let us summarize the characteristics of Amazon EC2 spot instances [13] as listed below:

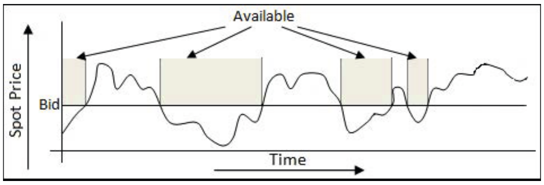


Fig. 2. Resource provisioning algorithm

- Spot instances are available when the user's bid exceeds the current spot price (refer Fig. 2).
- Spot instances are terminated (becomes unavailable) without any notification to the user whenever the current spot price exceeds the user's bid.
- The price per instance-hour for a spot instance is set at the beginning of each instance-hour. Any change to the spot price will not be reflected until the next instance-hour begins.
- Amazon will not charge the last partial hour if the spot instance is terminated due to out-of-bid situation. However Amazon will charge the full hour if the user terminate the instance forcefully.
- Amazon provides the history of spot prices of a spot instance at a specific availability zone for the last 3 months free of cost.

4.2 Existing Checkpointing Schemes for Spot Instances

The characteristics of spot instances make them appealing for long running jobs with divisible workloads [10]. Clearly, taking checkpoints at regular interval increases the utilization of spot instances. Various existing checkpointing schemes can be adopted for saving the completed tasks and resuming the remaining tasks as and when the spot instances become available. The checkpointing schemes proposed in [2] are briefly described below:

1. No Checkpointing (NONE): Checkpoints are not taken and all the partially completed tasks for a job are required to be repeated after every out-of-bid events.
2. Optimal Checkpointing (OPT): Checkpoints are taken just prior to the out-of-bid events. Clearly, it will save the maximum number of tasks out of each available interval for a given instance type and a user's bid.
3. Hourly Checkpointing (HOUR): Checkpoints are taken just prior to the beginning of next instance hour. Since Amazon is not charging any partial hour, this scheme will save as much tasks as the user is paying.
4. Rising edge-driven Checkpointing (EDGE): Checkpoints are taken after every increase (rising edge) of the current spot price.
5. Adaptive Checkpointing (ADAPT): Checkpoints are taken or skipped at regular intervals based on the expected recovery time for skipping (R_{skip}) or taking (R_{take}) a checkpoint. The estimation of R_{skip} and R_{take} is given in the equations 1 and 2. Here r is the task recovery time, t_p is the present time, $f(t)$ is the probability density function of out-of-bid events, t_r is the time needed to complete a job, t_c is the time needed to take a checkpoint and $T(t, t_p)$ is the expected execution time for a job of length t started at time t_p. Checkpoints are taken when R_{skip} is greater than R_{take}.

$$R_{skip}(t, t_p) = \sum_{k=0}^{t_r-1} (k + r + T(t, t_p)) f(k + t_p) \tag{1}$$

$$R_{take}(t, t_p) = \sum_{k=0}^{t_r-1} (k+r)) f(k+t_p) + t_c \sum_{k=t_r}^{\infty} f(k+t_p) + T(t, t_p - t) \sum_{k=0}^{t_c-1} f(k+t_p) \tag{2}$$

$$T(t, t_p) = (t \sum_{k=t}^{\infty} f(k + t_p) + \sum_{k=0}^{t-1} (k + r) f(k + t_p))/(1 - \sum_{k=0}^{t-1} f(k + t_p)) \tag{3}$$

Out of the above five checkpointing schemes, NONE and OPT provide two extreme results without any practical value. They are used to provide comparative study of the other realistic checkpointing schemes.

5 A Novel Checkpointing Scheme over Application-Centric Resource Provisioning Framework

In this section, we propose a novel checkpointing scheme for spot instances on top of application-centric resource provisioning framework. For the purpose, we devise a new event generation scheme that deals with spot instances. The new checkpointing scheme is targeted to achieve performance comparative to OPT checkpointing scheme described above. Before describing the scheme, we introduce a modified event generation scheme for our application-centric resource provisioning framework.

5.1 Event Generation Scheme for Spot Instances

The event generation schemes proposed in [1] is extended to include new events that support spot instances. As discussed in Section 4.1, the availability of spot instances depends on the current spot price and the user defined bid. Also, spot instances become unavailable without prior notification to the clients that makes them inherently unreliable. The reliability can be increased by taking checkpoints (saving completed tasks) during the available periods. However, the time and frequency of taking checkpoints affect the reliability as well as job completion time and cost.

Accordingly, in this paper we propose a new event generation scheme to handle spot instances. Three events are proposed, namely E_{ckpt}, $E_{terminate}$ and E_{launch}. E_{ckpt} is used for taking checkpoint, $E_{terminate}$ is used to terminate a spot instance forcefully and E_{launch} is used to relaunch a previously terminated spot instance. We define two bid values for the purpose - one for the application (A_{bid}) and other for the spot instance (S_{bid}). S_{bid} is sufficiently large and is used in the request for spot instance. Clearly, the value is maintained at such a high level, that Amazon will never terminate the spot instances due to out-of-bid situation. On the other hand, A_{bid} is used by the monitoring subsystem to maintain user's budget.

The monitoring subsystem actively monitors the current spot price and generates the two events, E_{ckpt} and $E_{terminate}$, for the provisioning subsystem. On the basis of these two events, the provisioning subsystem either takes a checkpoint or terminate the corresponding spot instance respectively. However, to increase the performance, the monitoring subsystem will query the current spot price only at specific points of time called decision points. Since the cost of spot instance is not changed during an instance hour and is fixed at the beginning of that instance hour, the decision points should be relative to the beginning of the next instance hour. Accordingly, we define two decision points just prior to each hour boundary as follows:

$$t_{cd} = t_h - t_c - t_w \tag{4}$$

$$t_{td} = t_h - t_w \tag{5}$$

where t_{cd} and t_{td} are the decision points for checkpointing and terminating a spot instance. t_h is an hour boundary, t_c is the time needed to take a checkpoint and t_w is the waiting time to get the current spot price. The monitoring subsystem will generate E_{ckpt} at t_{cd} if the current spot price exceeds A_{bid} and will generate $E_{terminate}$ at t_{td} if the current spot price is still above the A_{bid}. It will generate E_{launch} at the start of each available period of a spot instance with respect to A_{bid}.

5.2 The Application-Centric Checkpointing Scheme

In this section, we propose a checkpointing scheme on top of the application centric resource provisioning framework, called Application Centric Checkpointing(ACC). ACC is based on the event generation scheme discussed in the previous subsection and is described by the sequence diagram shown in Fig. 3.

The following unified definition can be used for an application with divisible workloads to be run on spot:

$$A = (T, R, R_m, P, U, M) \tag{6}$$

where $T = \{t_1\}$
$R = \{r_1, r_2\}$, r_1.provider = ec2, r_1.type = spot instance,
r_1.size = $< instance\_type >$
r_2.provider = ec2, r_2.type = EBS, r_2.size = 1GB
$R_m = \{ r_1 \rightarrow t_1, r_2 \rightarrow t_1 \}$
$P = \{ sla \}$

$$M = (E, W, E_m, W_m) \tag{7}$$

where $E = \{E_{ckpt}, E_{terminate}, E_{launch}\}, threshold\ for\ all\ events =< A_{bid} >$
$E_{launch}.bid =< S_{bid} >$
$W = \{W_{start}, W_{ckpt}, W_{terminate}, W_{launch}\}$
W_{start} = { Launch spot; Mount EBS; Copy job to EBS; Start job },
W_{ckpt} = {Save results to EBS},
$W_{terminate}$ = {Terminate spot} &
W_{launch} = { Launch spot; Mount EBS; Resume tasks },
$E_m = \{E_{ckpt} \rightarrow r_1,\ E_{terminate} \rightarrow r_1, E_{launch} \rightarrow r_1\})$
$W_m = \{W_{ckpt} \rightarrow E_{ckpt}, W_{terminate} \rightarrow E_{terminate}, W_{launch} \rightarrow E_{launch}\}$

The Elastic Block Storage (EBS) [14] is used to save the completed tasks during checkpoint. The parameters $instance\_type$, A_{bid} and S_{bid} can be set either manually by the end user or by some optimization or greedy algorithms.

The provisioning subsystem starts an application (job) by executing W_{start} workflow for that application. The W_{start} workflow launches a spot instance as per the specification of the resource r_1 and an EBS volume as per the specification of the resource r_2. The workflow then mounts the EBS volume to the spot instance, copy the job from the application repository to the EBS and starts the job.

Once the application is deployed, EC2 starts charging for the resources. The monitoring subsystem calculates t_{cd} and t_{td} as per Equ. 4 & 5 for the current hour boundary. At t_{cd} the monitoring subsystem retrieves the current spot price(P). If P exceeds A_{bid}, it generates E_{ckpt} event for the provisioning subsystem. On receiving E_{ckpt} event, the provisioning subsystem executes W_{ckpt} workflow. The W_{ckpt} workflow just saves the results (the completed tasks) to the EBS volume. The monitoring subsystem also retrieves the current spot price(P) at t_{td}. If P still exceeds A_{bid}, it generates $E_{terminate}$ event for the provisioning subsystem. On receiving $E_{terminate}$ event, the provisioning subsystem executes $W_{terminate}$ workflow. The $W_{terminate}$ workflow terminates the

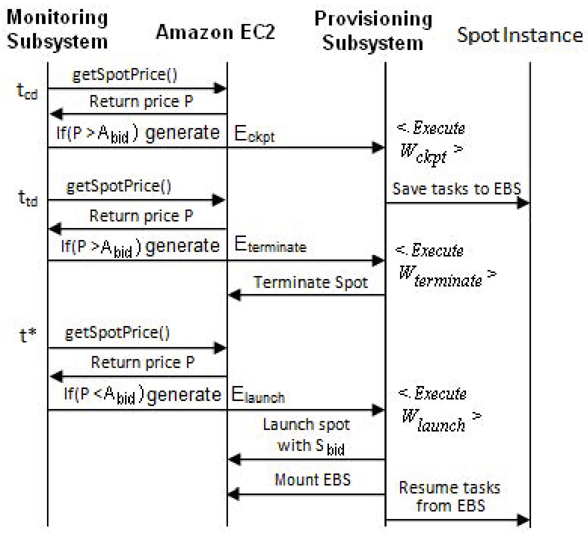


Fig. 3. Application Centric Checkpointing Scheme

spot instance forcefully. The monitoring subsystem repeats the above procedure till P does not exceed A_{bid} at t_{td} for all the subsequent hour boundaries.

If the instance is terminated at some t_{td}, the monitoring subsystem will have to query for the current spot price to determine the next available period at some specific instance of time(t*). However, the frequency of making the query is defined by the end user which may affect the job completion time slightly. At the start of the new available duration, the monitoring subsystem generates E_{launch} event for the provisioning subsystem. On receiving E_{launch} event, the provisioning subsystem executes W_{lauch} workflow. The W_{launch} workflow launches a new spot instance as specified in r_1, mount the existing EBS volume to that instance and resume the remaining tasks of the job.

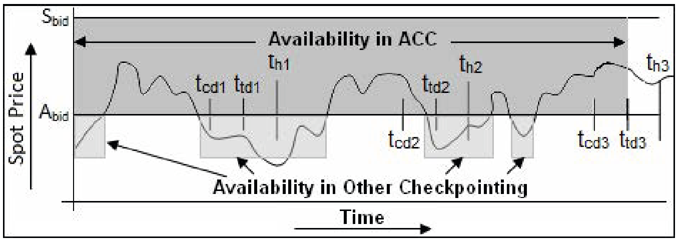


Fig. 4. Decision Points for Event Generation

The novelty of the scheme is illustrated in Fig. 4. ACC will generate neither E_{ckpt} nor $E_{terminate}$ for the hour boundary t_{h1} since the current spot price is bellow A_{bid} at both the decision points. That means, it will neither take a checkpoint nor terminate the spot instance at t_{h1}. It will generate E_{ckpt} but not $E_{terminate}$ for the hour boundary t_{h2} since the current spot price is above A_{bid} at t_{cd2} and below A_{bid} at t_{td2}. That means, it will take a checkpoint but will not terminate the spot instance at t_{h2}. Similarly, for the hour boundary t_{h3}, it will generate both E_{ckpt} and $E_{terminate}$ since the user will have to pay above A_{bid} for the next hour. So, it will take a checkpoint as well as terminate the spot instance at t_{h3}. Clearly availability is increased and more continuous in ACC compared to other checkointing schemes as shown in Fig. 4.

6 Implementation and Evaluation

In this section we analyze and compare our proposed ACC checkpointing scheme with the existing checkpointing schemes. The experiments have been carried out on 64 spot instance types using the same data set, parameters, algorithms and assumptions used in the simulator [26].

We obtain the simulation result for $job\ completion\ time$, $total\ monetary\ cost$ and the $product\ of\ monetary\ cost$ x $completion\ time$ for all the EC2 instance types. To simplify the discussion, we present the result of a linux based extra large (m1.xlarge) instance type in the eu-west-1 region. We concentrate on the performance of our proposed ACC checkpointing scheme compared to the theoritical optimal checkpointing scheme, OPT. We also include NONE, HOUR, EDGE and ADAPT checkpointing schemes in our result for completeness.

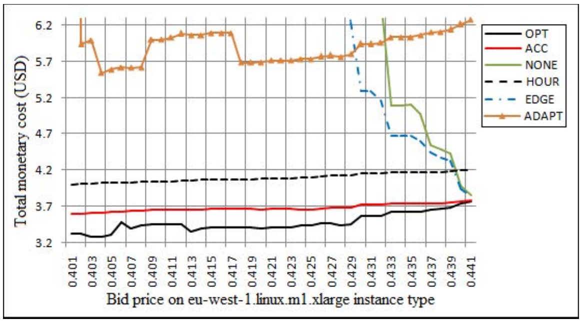


Fig. 5. Total monetary cost of Job completion

Fig 5 shows the comparison of total monetary cost needed to complete a job of length 500 minutes under different user's bid(A_{bid}) from \$0.401 to \$0.441. The result shows that ACC reduces the job completion cost significantly over the other realistic checkpointing schemes. However the cost is increased by 5.94% on average (min 0.33%, max 10.30%) compared to OPT scheme. This is because the OPT scheme guarantees payment of the actual progress of the job as well as executing some fraction of the job free of cost for the partial hours.

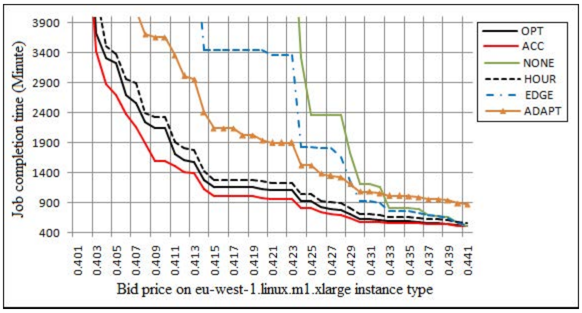


Fig. 6. Job completion time

In Fig. 6 we illustrate the comparison of various checkpointing schemes for the metric $job\ completion\ time$. Here we observe that ACC scheme outperforms all the checkpointing schemes including OPT. This is because ACC allows the job to continue even when the current spot price exceeds A_{bid} in between a t_{td} and the previous hour boundary (refer to Fig. 4). With OPT, the available duration is fragmented as shown in Fig. 2 while ACC allows the spot instance to be continuously available till t_{td3} a shown in Fig. 4 without affecting the job completion cost. That means the interruption to job execution is much less in ACC compared to OPT. In fact the ACC scheme reduces the job completion time by an average value of 10.77% over the OPT scheme.

We plot the comparative study for the $product\ of\ monetary\ cost$ x $completion\ time$ in Fig. 7. Here also we observe that the ACC scheme reduce this metric by an average value of 5.56% over the OPT scheme.

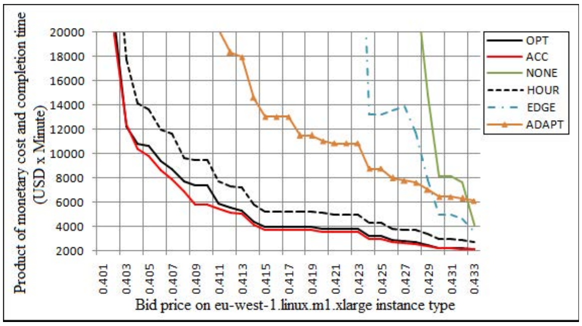


Fig. 7. Product of total cost and completion time

To gain confidence in our result, we have computed the average values of the above mentioned metrics for different bid values on all the 64 instance types. A sample of 15 difference instance types for the metric $product\ of\ monetary\ cost$ x $completion\ time$ is shown in Fig. 8. For these 15 instance types, a gain of 4.03% for ACC over OPT is observed. We also observe that such percentage gain is increased for costly instance types.

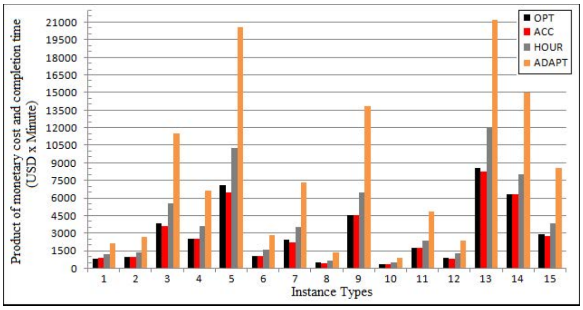


Fig. 8. Product of cost and completion time for different instance types

In the previous research work [2], the authors conclude that OPT is the optimal checkpointing scheme and none of the practical schemes can perform better than OPT. That is true only if we use the same bid values for launching the spot instance and executing the checkpoint. However, our proposed ACC checkpointing scheme perform very close to OPT or even better than OPT (for time and product metrics) by separating these two bid values. Thus ACC outperforms all the existing checkpointing schemes for spot instances. ACC achieves such performance gain by increasing availability at the same cost as shown in Fig. 4.

7 Conclusion and Future Work

Checkpointing plays an important role in reliability of job execution over EC2 spot instances. In this paper, we propose a checkpointing scheme on top of application-centric resource provisioning framework that not only increases the reliability but also reduces the cost significantly over the existing checkpointing schemes. The job completion cost under the proposed scheme is very close to the optimal checkpointing scheme. It performs better than all the practical checkpointing schemes for spot instances. In future, we want to investigate more on finding the optimal bid (A_{bid}) and the corresponding instance type for a given job.

References

1. Khatua, S., Ghosh, A., Mukherjee, N.: Application-centric Cloud Managemengt. In: 9th IEEE/ACS International Conference on Computer Systems and Applications(AICCSA), pp. 9–15 (2011)
2. Yi, S., Andrzejak, A., Kondo, D.: Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. IEEE Transactions on Services Computing 5, 512–524 (2011)
3. Voorsluys, W., Buyya, R.: Reliable Provisioning of Spot Instances for Compute-intensive Applications. In: 26th IEEE AINA, pp. 542–549 (2012)
4. Javadi, B., Thulasiramy, R.K., Buyya, R.: Statistical Modeling of Spot Instance Prices in Public Cloud Environments. In: 4th IEEE UCC, pp. 219–228 (2011)
5. Yi, S., Zafer, M., Kang-Won, L.: Optimal bidding in spot instance market. IEEE INFOCOM, 190–198 (2012)

6. Mazzucco, M., Dumas, M.: Achieving Performance and Availability Guarantees with Spot Instances. In: 13th IEEE HPCC, pp. 296–303 (2011)
7. Padala, P., et al.: Adaptive control of virtualized resources in utility computing enironments. In: Proceedings of EuroSys (2007)
8. Li, Q., Guo, Y.: Optimization of Resource Scheduling in Cloud Computing. In: 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 315–320 (2010)
9. Chaisiri, S., Lee, B., Niyato, D.: Optimization of Resource Provisioning Cost in Cloud Computing. IEEE Transactions on Services Computing (2011)
10. Yang, Y., Casanova, H.: Umr: A multi-round algorithm for scheduling divisible workloads. IPDPS 24 (2003)
11. Amazon EC2 Instance Types, `http://aws.amazon.com/ec2/instance-types/`
12. Amazon EC2 Purchasing Options, `http://aws.amazon.com/ec2/purchasing-options/`
13. Amazon EC2 spot instances, `http://aws.amazon.com/ec2/spot-instances/`
14. Elastic Block Storage, `http://aws.amazon.com/ec2/ebs/`
15. Garfinkel, S.: An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. Tech. Rep. TR-08-07, Harvard University (2007)
16. Google Cloud Offering, `http://cloud.google.com/products/`
17. Barham, P., et al.: Xen and the Art of Virtualization. In: Proceedings of the 19th ACM symposium on Operating Systems Principles (2003)
18. Wolsky, R., et al.: Eucalyptus: A Technical Report on an Elastic Utility Computing Archietcture Linking Your Programs to Useful Systems. Tech. Rep. 2008-10, University of California, Santa Barbara (2008)
19. Zabbix: an enterprise-class open source distributed monitoring solution for networks and applications, `http://www.zabbix.com/`
20. Harmer, T., et al.: An application-centric model for cloud management. In: Proceedings of 6th World Congress on Services, pp. 439–446 (2010)
21. Lim, H.C., et al.: Automated control in cloud computing: challenges and opportunities. In: Proceedings of the 1st workshop on Automated control for datacenters and clouds, Spain (2009)
22. Mills, T.C.: Time Series Techniques for Economists. Cambridge University Press (1990)
23. Buyya, R., et al.: Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. In: 10th IEEE International Conference on High Performance Computing and Communications, pp. 5–13 (2008)
24. Iqbal, W., Dailey, M.N., Carrera, D., Janecek, P.: Adaptive resource provisioning for read intensive multi-tier applications in the cloud. Future Generation of Computer Systems 27, 871–879 (2011)
25. Shao, J., Wang, Q.: A Performance Guarantee Approach for Cloud Applications Based on Monitoring. In: 35th IEEE Annual Computer Software and Applications Conference Workshops, pp. 25–30 (2011)
26. Checkpointing Simulator for spot instances, `http://spotckpt.sourceforge.net`

PonIC: Using Stratosphere to Speed Up Pig Analytics

Vasiliki Kalavri[1], Vladimir Vlassov[1], and Per Brand[2]

[1] KTH Royal Institute of Technology
{kalavri,vladv}@kth.se
[2] Swedish Institute of Computer Science
Stockholm, Sweden
perbrand@sics.se

Abstract. Pig, a high-level dataflow system built on top of Hadoop MapReduce, has greatly facilitated the implementation of data-intensive applications. Pig successfully manages to conceal Hadoop's one input and two-stage inflexible pipeline limitations, by translating scripts into MapReduce jobs. However, these limitations are still present in the backend, often resulting in inefficient execution.

Stratosphere, a data-parallel computing framework consisting of PACT, an extension to the MapReduce programming model and the Nephele execution engine, overcomes several limitations of Hadoop MapReduce. In this paper, we argue that Pig can highly benefit from using Stratosphere as the backend system and gain performance, without any loss of expressiveness.

We have ported Pig on top of Stratosphere and we present a process for translating Pig Latin scripts into PACT programs. Our evaluation shows that Pig Latin scripts can execute on our prototype up to 8 times faster for a certain class of applications.

1 Introduction

Large-scale data management and analysis is currently one of the biggest challenges in the area of distributed systems. Industry, as well as academia, is in urgent need of data analytics systems, capable of scaling up to petabytes of data. Such systems need to efficiently analyze text, web data, log files and scientific data. Most of the recent approaches use massive parallelism and are deployed on large clusters of hundreds or even thousands of commodity hardware.

MapReduce [1], proposed by Google, is the most popular framework for large-data processing; its open-source implementation, Hadoop[1], is nowadays widely used. However, it has several limitations, including the limitation on the number of input datasets (only one input set) and the limitation on a structure of a program that must follow a static fixed pipeline pattern of the form split-map-shuffle-sort-reduce. This pipeline is suitable for simple applications, such as log-file analysis, but severely complicates the implementation of relational queries or

[1] http://hadoop.apache.org/

graph algorithms. These limitations have led researchers to develop more general-purpose systems, inspired by MapReduce [2–6]. One of them is Stratosphere [6], which consists of a programming model, PACT (Parallelization Contracts), and the Nephele execution engine. The system is essentially a generalization of MapReduce and aims to overcome the limitations mentioned above.

Both models, MapReduce and PACT, require significant programming ability and in-depth understanding of the systems' architectures. Applications usually lead to complex branching dataflows which are low-level and inflexible. In order to save development time and make application code easier to maintain, several high-level languages have been proposed for these systems. Currently, high-level platforms on top of Hadoop include JAQL [7], Hive [8] and Pig [9]. Pig Latin, which is the language of the Pig platform [10], offers the simplicity and declarativeness of SQL, while maintaining the functionality of MapReduce. Pig compiles Pig Latin into MapReduce jobs which are executed in Hadoop. Pig hides Hadoop's one-input and two-stage dataflow limitations from the programmer and provides built-in functions for common operations, such as filtering, join and projection. It also directly benefits from Hadoop's scalability and fault-tolerance. However, even if not obvious to the users, the limitations and inflexibility of Hadoop are still present in the Pig system. The translation of relational operators for the static pipeline of Hadoop produces an inefficient execution plan since data have to be materialized and replicated after every MapReduce step.

The goal of Pig was to make MapReduce accessible to non-experts and relieve the programmer from the burden of repeatedly coding standard operations, like joins. Another goal was to make Pig independent of any particular backend execution engine. However, Pig was developed on top of Hadoop, ended up solving specific Hadoop problems and became highly coupled with its execution engine. The Stratosphere data-parallel computing framework offers a superset of MapReduce functionality, while overcoming some of the major weaknesses of the MapReduce programming model. It allows data pipelining between execution stages, enabling the construction of flexible execution strategies and removing the demand for materialization and replication in every stage. Moreover, the PACT programming model of Stratosphere supports multiple inputs.

In this paper, we present PonIC (Pig on Input Contracts), an integration of the of Pig System with Stratosphere. We have analyzed the internal structure of Pig and have designed a suitable integration strategy. In order to evaluate the benefits of the integration, we have developed a prototype implementation. The current prototype supports a subset of the most common Pig operations and it can be easily extended to support the complete set of Pig Latin statements. Thus, we show that it is possible to plug a different execution engine into the Pig system and we identify the parts of Pig that can be reused. With our Pig to PACT translation algorithm and our prototype, we show that Stratosphere has desirable properties that significantly simplify the plan generation. We have developed a set of basic scripts and their native MapReduce and PACT equivalents and we provide a comparison of PonIC with Pig, as well as the corresponding native programs. We observe that Stratosphere's relational operators are much more efficient than their MapReduce equivalents. As a result, PonIC has a great advantage over Pig on Hadoop and often executes faster than native Hadoop MapReduce. The main contributions of this paper are as follows.

– Our integration is entirely transparent to Pig's end-users and existing Pig Latin applications can be executed on PonIC without any modification. The syntax and the semantics are completely unchanged.
– We show that Pig can be harnessed to alternative execution engines and present a way of integration.
– We identify the features of Pig that negatively impact execution time.
– We show that Pig can be integrated with Stratosphere and gain performance.
– We propose a complete translation process of Pig Logical Plans into Stratosphere Physical Plans and we present and evaluate PonIC.

The rest of this paper is structured as follows. In Section 2, we provide the necessary background on the Pig and Stratosphere systems. Section 3 discusses the restrictions that MapReduce poses on Pig's current implementation and presents our Pig-to-Stratosphere translation process. In Section 4, we discuss our prototype implementation in detail. Section 5 contains the evaluation of PonIC against Pig on Hadoop, native Hadoop MapReduce and native PACT Stratosphere. In Section 6, we comment on related work, while we provide our conclusions, open issues and vision for the future in Section 7.

2 Background

In this section, we provide the essential background. We briefly discuss the MapReduce programming model, the Pig system and the Stratosphere system.

2.1 The MapReduce Programming Model

MapReduce is a data-parallel programming model. Its architecture is inspired by functional programming and consists of two second-order functions, Map and Reduce, which form a static pipeline. Data are read from an underlying distributed file system and are transformed into key-value pairs, which are grouped into subsets and processed by user-defined functions in parallel. Data distribution, parallelization and communication are handled by the framework, while the user only has to write the first-order functions wrapped by the Map and Reduce functions. However, this abstraction comes with loss of flexibility. Each job must consist of exactly one Map function followed by one Reduce function and no step can be omitted or executed in a different order. Moreover, if an algorithm requires multiple Map and Reduce steps, these can only be implemented as separate jobs, and data can only be passed from one job to the next through the file system. This limitation can frequently add a significant overhead to the execution time. MapReduce was initially proposed by Google and its open-source implementation, Hadoop and HDFS [11] are nowadays widely used.

2.2 Pig

Pig consists of a declarative scripting language, Pig Latin, and an execution engine that allows the parallel execution of data-flows on top of Hadoop. The Pig System takes a Pig Latin program as input and produces a series of MapReduce jobs to be executed on the Hadoop engine. Compilation happens in several steps. First, the parser transforms a Pig Latin script into a Logical Plan. Each Logical operator is compiled down to one or more Physical Operators. The Physical Plan is then passed to the compiler that transforms it into a DAG of MapReduce operators. MapReduce operators are topologically sorted and connected between them using a store-load combination, producing the MapReduce Plan as output. The generated jobs are finally submitted to Hadoop and monitored by Pig.

2.3 Stratosphere

Stratosphere is a parallel data-processing framework, which consists of a programming model, PACT (Parallelization Contracts), and an execution engine, Nephele, capable of executing dataflow graphs in parallel. Nephele is an execution engine designed to execute DAG-based data flow programs. It manages task scheduling and setting up communication channels between nodes. Moreover, it supports dynamic allocation of resources during execution and fault-tolerance mechanisms. The PACT programming model is a generalization of the MapReduce programming model. It extends the idea of the Map and Reduce second-order functions, introducing the *Input Contracts*. An Input Contract is a secondary function that accepts a first-order user-defined function and one or more data sets as inputs. Input Contracts do not have to form any specific type of pipeline and can be used in any order that respects their input specifications. In the context of the PACT programming model, Map and Reduce are Input Contracts. The following three more Contracts are defined in PACT:

- The *Cross* Input Contract accepts multiple inputs of key value pairs and produces subsets of all possible combinations among them, building a Cartesian product over the input.
- The *Match* Contract operates on two inputs and matches each pair of the first input with one pair of the second input that has the same key value.
- The *CoGroup* Contract creates independent subsets by combining all pairs that share the same key.

3 Plan Compilation

As explained in the previous section, a Pig Latin script is parsed and transformed into a graph of logical operators, each corresponding to one command. This graph, the *Logical Plan*, is then translated into a *Physical Plan*, a graph of physical operators, which defines how the logical operations will be executed. Multiple strategies can be used to map logical operators to physical ones and it's the system's compiler job to choose a strategy, depending on the underlying execution engine's capabilities, dataset characteristics, hints provided by the developer, etc. The translation process in Pig is briefly explained next.

3.1 Plan Compilation in Pig

Pig's compiler translates logical to physical operators, with the additional restriction that each physical operator needs to be expressed in terms of MapReduce steps or parts thereof. The compiler keeps track of the current phase during translation and knows if it is a map or a reduce step. For each operator, it checks if it can be merged into the current phase. If communication is required, the current phase is finalized and a new phase is started in order to compile the operator. If the current phase is a map, a reduce phase will be initiated; otherwise, a new MapReduce job needs to be created and store-load combination is required to chain the jobs. We explain the translation process using an example from a slightly modified query of the PigMix benchmark[2] shown below:

Example Query 1

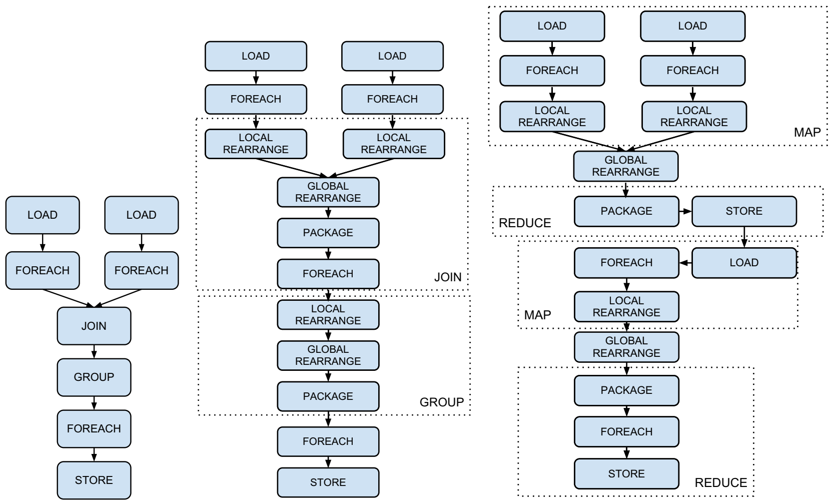
```
A = load 'page_views' as (user, timestamp, revenue);
B = foreach A generate user, revenue;
alpha = load 'users' as (name, phone, address, city);
beta = foreach alpha generate name;
C = join beta by name, B by user;
D = group C by $0;
E = foreach D generate group, SUM(C.revenue);
store E into 'out';
```

The simple Example Query 1 loads two datasets, performs a join on a common attribute to find the set of users who have visited some webpages, groups the resulting dataset and generates the estimated revenue for each user. Figure 1(a) shows the simplified Logical Plan for the above script, whereas Figure 1(b) shows the generated Physical Plan. Note that the join operator is replaced by four new operators and the group operator is translated into three physical operators similarly. The Physical Plan is then translated into a MapReduce Plan, as shown in Figure 1(c). First, a map phase is created and as the Physical Plan is traversed, operators are added to it. When the global rearrange operator is reached, shuffling is required, therefore the map phase is finalized and a reduce phase is initiated. When a new MapReduce job is created, a store-load pair is added in between to set the output of the first as the input of the second.

Our example shows that even for a small script, generated plans can be long and cumbersome. If the generated Logical Plan does not fit well the MapReduce static pipeline, performance might degrade. Adding store-load combinations and materialization of results in between jobs is also a source of inefficiency.

In contrast to MapReduce, using Stratosphere as the backend for Pig significantly simplifies the translation process. Input Contracts can be greatly exploited to generate shorter and more efficient plans, without any extra effort from the programmer. We present our translation algorithm next.

[2] http://cwiki.apache.org/PIG/pigmix.html



(a) Logical Plan (b) Physical Plan (c) MapReduce Plan

Fig. 1. Pig Plans for Example Query 1

3.2 Pig to PACT Plan Translation

Pig Latin offers a large set of commands that are used for input and output, relational operations, advanced operations and the declaration of user-defined functions. We chose the most common and useful ones and we describe here how they are be translated into PACT operators. A more detailed description of the translation process we followed can be found in [12].

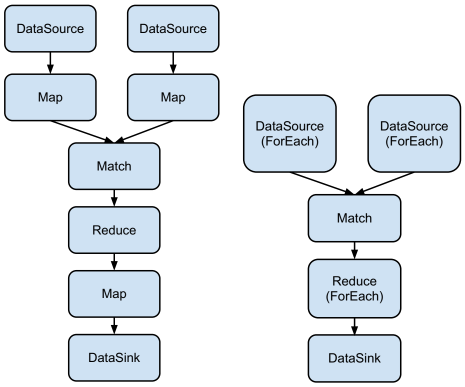
Input/Output. Pig provides the LOAD and the STORE commands for data input and output. These two logical operators can be mapped directly to the *GenericDataSource* and the *GenericDataSink* Input Contracts of Stratosphere. In our implementation, we only support input and output from and to files, so we have based our implementation on the more appropriate Contracts, *FileDataSource* and *FileDataSink*. The generic Contracts can be easily extended to support other kinds of input and output sources.

Relational Operators. PACTs support most of the common relational operations. The FILTER and FOREACH statements correspond to a Map Contract. The GROUP logical operator naturally maps to the *Reduce* Input Contract, while INNER and OUTER JOIN operations can be implemented using the *Match* and *CoGroup* Input Contracts. Pig's ORDER BY operator can sort the input records in ascending or descending order, specifying one or more record fields as the sorting key. Pig realizes the ORDER BY operation by creating two MapReduce jobs. With PACTs, the same functionality can be offered in a much simpler way using the *GenericDataSink* Contract.

Advanced Operators. From the set of the advanced Pig operators, we choose CROSS and UNION. The CROSS operator can be directly mapped to the *Cross* Input Contract, while the *Map* Input Contract can be used to realize UNION. The Map Contract (Stratosphere 0.2) offers a method, which provides the functionality we need to implement UNION.

Our translation consists of two stages. At the first stage the Logical Plan is translated into a plan of PACT operators. This PACT Plan is the equivalent of Pig's Physical Plan. The second stage translates the PACT Plan into actual Input Contracts and submits the PACT Plan to the Nephele execution engine.

The Plan generation for the Example Query 1 is shown in Figure 2(a). There is an one-to-one mapping of logical operators to PACT operators and consequently Input Contracts, which makes the graph and the translation process much simpler. The resulting graph can be further optimized, by merging filter and foreach operators into the preceding Contracts, as shown in Figure 2(b).



(a) Initial Plan (b) Optimized Plan

Fig. 2. PACT Plans for Example Query 1

3.3 Discussion

Even though we have considered only a subset of Pig operators, it is important to stress that the completeness of our proposal is guaranteed. The PACT programming model is a generalization of the MapReduce programming model. Since every Pig Latin program and Logical Plan can be translated into a MapReduce Plan, it can therefore also be translated into a PACT Plan.

Using Stratosphere and Input Contracts as the backend results into a more straightforward translation process. The one-to-one Pig-to-PACT mapping requires less communication, due to less shuffling. Data is pipelined between Input Contracts, eliminating the need for frequent materialization. Also, the execution plan benefits from optimizations of the Logical Plan by Pig's Logical Plan optimizer and of the PACT Plan by Stratosphere's optimizer[3].

[3] `http://stratosphere.eu/wiki/doku.php/wiki:pactcompiler`

4 Implementation

PonIC has been implemented as an extension to the Pig system and reuses Pig functionality where possible. Pig classes or wrappers are used in order to make them compatible with the new features. The source code is publicly available[4].

We have identified the parts of the Pig software stack that are not tightly coupled to the Hadoop execution engine, namely the parser and the Logical Plan layer. The underlying layers have been replaced with our compilation layer that tranforms the Logical Plan into a Stratosphere execution plan.

Pig's Logical Plan is traversed in a depth-first fashion. The traversal starts from the plan's roots and a *visit()* method is responsible for recognizing the operator type and creating the appropriate PACT operator, according to the mappings of Table 1. It is also responsible for setting the correct parameters, such as data types, operator alias, result types, as well as connecting the newly created operator to its predecessors. This way, a graph of PACT operators is gradually constructed. When the PACT Plan has been created, it is submitted to Nephele for execution. Table 1 summarizes the Pig to PACT translation mappings for the subset of the Pig operators considered in this study.

Table 1. Pig to PACT operators mapping (for the chosen subset of Pig operators)

Pig Operator	Input Contract
LOAD	FileDataSource
STORE	FileDataSink
GROUP	Reduce
INNER JOIN	Match
OUTER JOIN / COGROUP	CoGroup
UNION	Map
FILTER / FOREACH	Map
ORDER	FileDataSink

The most significant extensions made to the Pig codebase are:

– An additional execution mode to allow starting Pig in Stratosphere execution mode with the command `pig -x strato`.
– An extension of Pig's `HExecutionEngine` class as an engine for Stratosphere.
– A re-implementation os the relational and expression operators to support the new APIs.
– A `LogToPactTranslationVisitor` class, based on Pig's `LogToPhyTranslationVisitor` class, as the first-level compiler.
– A package of PACT operators, based on Pig's physical operators.
– A `PactCompiler` class, as the second-level compiler.
– Stratosphere-specific load and store functions.
– A `contractsLayer` and a stubsLayer packages, which contain wrapper classes of Stratosphere's Input Contracts and Stub classes.

[4] `http://github.com/PonIC/PonIC`

5 Evaluation

We conducted our experiments on an OpenStack cluster, using 10 ubuntu Virtual Machines (VMs), each having 4 VCPUs, 90GB of disk space and 8GB of RAM. We deployed Hadoop version 1.0.0, Pig version 0.10.0 and Stratosphere version 0.2. Hadoop's NameNode and JobTracker, as well as Stratosphere's JobManager run on a dedicated VM, while the remaining 9 VMs serve as slave nodes. Default parameters were used for HDFS block size and replication factor.

We used the PigMix data generator to create a `page_views` dataset of 10 million rows (approximately 15GB) and the corresponding `users` table. We developed five scripts for evaluation, namely a Load/Store operation, a Filter script which filters out 50% of the input, a Group operation, a Join of the `page_views` and the `users` dataset and a Mixed query, corresponding to the Example Query 1, containing a combination of Load, Group, Join and Store operators. Each test was executed 5 times and the results presented here have a standard deviation of less than 1% in all cases. The test applications were developed in Pig Latin (executed both on Pig and PonIC), native MapReduce and PACT.

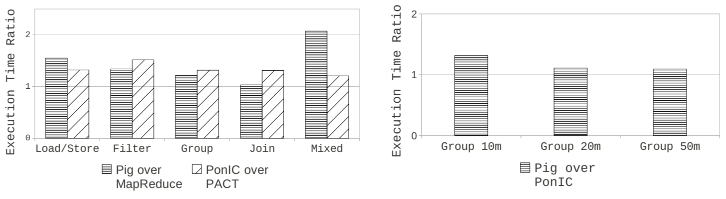
5.1 Implementation Overhead

Whenever using high-level languages, there is an overhead users have to pay in exchange for the abstraction offered. This overhead is one of the factors defining the value of the abstraction. Figure 3(a) shows the performance overhead for the Pig system over the corresponding native Hadoop MapReduce implementations and for PonIC over PACT. For Pig, this overhead includes setup, compiling, data conversion and plan optimization time. The results for Pig confirm already published results [9]; Pig is around 1.2 to 2 times slower than a native MapReduce application. Figure 3(a) also shows that PonIC's overhead is significantly lower and smaller than 1.6 in all cases. We believe that the smaller overhead is mainly due to the more efficient translation process. Since PonIC only supports a subset of Pig's features, the overhead could increase in a complete implementation. However, as we described in Section 3.3, in the worst case, an operator could be translated into PACT, using only the Map and Reduce Contracts. Such a naive translation would result into an overhead comparable to Pig's overhead.

In order to have a better idea on how the overhead changes depending on the dataset size, we ran the Group query for three different sizes of the `page_views` dataset. The results in Figure 3(b) show that the overhead caused by setup and compilation time has a heavier influence on smaller datasets.

5.2 Comparison with Pig and Hadoop MapReduce

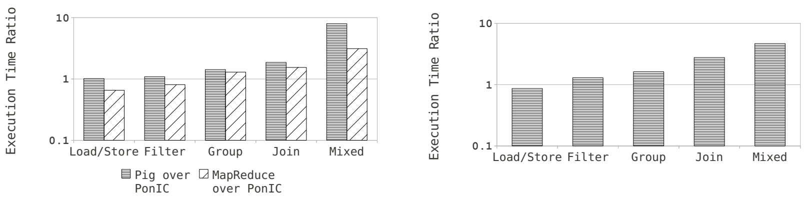
Figure 4(a) shows the execution time ratio of Pig and native Hadoop MapReduce over PonIC. Y axis is in logarithmic scale. PonIC matches Pig's execution time for the Load/Store and the Filter queries, while it is significantly faster in the rest of the cases. When compared to native MapReduce, PonIC is also faster, except from the Load/Store and Filter operations, for which setup and data conversion



(a) PonIC vs. Pig/MapReduce (b) Varying Data Sizes

Fig. 3. Evaluation Results: Overhead

times are dominant. In the case of Mixed query, PonIC is 8 times faster than Pig. The MapReduce Plan that Pig creates for this query contains two MapReduce jobs in order to implement the join and the group operations, involving a materialization step in between them. On the other hand, PonIC can execute faster, exploiting Stratosphere's data pipelining between Input Contracts. The main reason why PonIC is generally faster than Pig is demonstrated in Figure 4(b), which is a comparison between the execution time of native MapReduce and PACT implementations. It shows that, in all the cases except Load/Store, Stratosphere is faster than native MapReduce.



(a) PonIC vs. Pig/MapReduce (b) Native MapReduce vs. PACT

Fig. 4. Evaluation Results: Execution Time Comparison

6 Related Work

Among the supported high-level languages for MapReduce, Hive is probably the most popular and has been used in work similar to ours. Hive has been integrated with the ASTERIX system [5]. ASTERIX provides a data-agnostic algebra layer, which allows Hive to run on top of the Hyracks runtime. Hive execution plans are translated to ASTERIX algebra plans and better performance is achieved without any changes in the HiveQL queries. To our knowledge, no published evaluation measurements exist to support this claim.

The Shark system [13] allows HiveQL queries to execute on top of Spark [4], in an analogous way to ours with Pig and Stratosphere. However, Shark's goal is to provide a unified system where both SQL queries and iterative analytics applications can co-exist and execute efficiently. Our work and the Shark project share some discoveries regarding the limitations of the MapReduce-based execution engines, which result in inefficient execution, namely the expensive data materialization and inflexibility of static pipelines over general DAGs.

There has been recent work in integrating JAQL with the Stratosphere system [14], which led to the creation of Meteor [15]. Meteor is a high-level language inspired by JAQL and lies on top of a relational algebra layer, Sopremo. Meteor programs are translated into Sopremo operators, which are then compiled into Input Contracts, in a way similar to our work. However, Meteor, like JAQL, only supports the JSON data model and no performance measurements are yet available, as far as we know. With our work, we benefit both Pig and Stratosphere users. Pig developers can gain improved performance without changing their applications, while Stratosphere users can now exploit the expressiveness of the Pig Latin language to develop applications faster and execute them on the Nephele execution engine, with only minimal compilation overhead.

7 Conclusions and Future Work

Existing programming models for Big Data analytics, such as MapReduce and PACT, have been a great contribution and are widely used. However, in order to fully exploit the possibilities provided by the increasing amounts of data in business and scientific applications, data analysis should become accessible to non-experts, who are used to work with higher-level languages. Therefore, improving the performance of systems like Pig is of great importance.

In this paper, we examined the feasibility of integrating Pig with Stratosphere. We show that Pig can highly benefit from using Stratosphere as the backend system and gain performance, without any loss of expressiveness. We concluded that, even though Pig is tightly coupled to the Hadoop execution engine, integration is possible by replacing the stack below the Logical Plan layer. The translation algorithm and prototype integration of Pig with Stratosphere allows execution of Pig Latin scripts in the Stratosphere execution engine, without modifying the scripts, while offering improved performance.

Several issues remain unexplored and are interesting for further investigation. We certainly believe that creating a system that fully supports Pig Latin and generates Stratosphere jobs is not the limit of this research. Several optimizations can now be added to Pig because of the underlying Nephele execution engine. For example, Pig Latin could be extended to include keywords corresponding to Output Contracts or PACT's compiler hints. Since Stratosphere now offers its own high-level language, Meteor, it would also be very interesting to compare its expressiveness, usability and performance against Pig.

Acknowledgements. This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043. The authors would also like to thank the Stratosphere team for their help throughout this work.

References

1. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI 2004: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation. USENIX Association (2004)
2. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev (2007)
3. Warneke, D., Kao, O.: Nephele: efficient parallel data processing in the cloud. In: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers. ACM, New York (2009)
4. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, HotCloud 2010 (2010)
5. Alsubaiee, S., Behm, A., Grover, R., Vernica, R., Borkar, V., Carey, M.J., Li, C.: Asterix: scalable warehouse-style web data integration. In: Proceedings of the Ninth International Workshop on Information Integration on the Web, IIWeb 2012. ACM (2012)
6. Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., Warneke, D.: Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In: Proceedings of the 1st ACM symposium on Cloud computing, SOCC 2010, pp. 119–130. ACM, New York (2010)
7. Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M.Y., Kanne, C.C., Özcan, F., Shekita, E.J.: Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. PVLDB 4, 1272–1283 (2011)
8. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow. 2(2), 1626–1629 (2009)
9. Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a high-level dataflow system on top of map-reduce: the pig experience. Proc. VLDB Endow. 2(2), 1414–1425 (2009)
10. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 1099–1110. ACM, New York (2008)
11. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). IEEE Computer Society, Washington, DC (2010)
12. Kalavri, V.: Integrating pig and stratosphere. Master's thesis, KTH, School of Information and Communication Technology, ICT (2012)
13. Engle, C., Lupher, A., Xin, R., Zaharia, M., Franklin, M.J., Shenker, S., Stoica, I.: Shark: fast data analysis using coarse-grained distributed memory. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, New York (2012)
14. Lawerentz, C., Nagel, C., Berezowski, J., Guether, M., Ringwald, M., Kaufmann, M., Vu, N.T., Lobach, S., Pieper, S., Bodner, T., Wurtz, C.: Project jaql on the cloud. Final report, TU Berlin (2011)
15. Heise, A., Rheinlaender, A., Leich, M., Leser, U., Naumann, F.: Meteor/sopremo: An extensible query language and operator model. In: Proceedings of the International Workshop on End-to-end Management of Big Data (BigData) in conjunction with VLDB 2012, Istanbul, Turkey (2012)

MROrder: Flexible Job Ordering Optimization for Online MapReduce Workloads

Shanjiang Tang, Bu-Sung Lee, and Bingsheng He

School of Computer Engineering, Nanyang Technological University
{stang5,ebslee,bshe}@ntu.edu.sg

Abstract. MapReduce has become a widely used computing model for large-scale data processing in clusters and data centers. A MapReduce workload generally contains multiple jobs. Due to the general execution constraints that map tasks are executed before reduce tasks, different job execution orders in a MapReduce workload can have significantly different performance and system utilization. This paper proposes a prototype system called *MROrder* to dynamically optimize the job order for online MapReduce workloads. Moreover, MROrder is designed to be flexible for different optimization metrics, e.g., *makespan* and *total completion time*. The experimental results show that MROrder is able to improve the system performance by up to 31% for makespan and 176% for total completion time.

1 Introduction

MapReduce [1] is a popular computing paradigm for large-scale data intensive processing. A map-reduce job computation generally contains two phases: 1) a map phase, consisting of many map tasks, and 2) a reduce phase, consisting of many reduce tasks. Apache Hadoop, an open source framework of MapReduce, has been widely deployed on large clusters consisting of thousands of machines by companies such as Facebook, Amazon, and Yahoo. Generally, MapReduce and Hadoop are used to support batch processing for multiple large jobs (i.e., MapReduce workloads). Despite many research efforts have been devoted to improve the performance of a single MapReduce job (e.g., [1,2]), there is relatively little attention that has been paid to the system performance of MapReduce workloads. Therefore, this paper attempts to improve the system performance of MapReduce workloads.

The job execution order in a MapReduce workload is important for the system performance. To show the importance of job ordering, Figure 1 gives an example illustrating that the performance can differ by nearly 100% for two varied job submission orders for a batch of jobs. However, the job ordering optimization for MapReduce workloads is challenging, due to the following facts: (i). There is a strong data dependency between the map tasks and reduce tasks of a job, i.e., reduce tasks can only perform after the map tasks, (ii). map tasks have to be allocated with map slots and reduce tasks have to be allocated with reduce slots, (iii). Both map slots and reduce slots are limited computing resources, configured by hadoop administrator in advance [3].

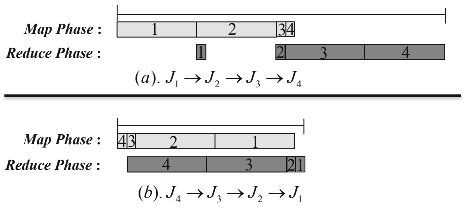

Map Phase :
Reduce Phase :
(a). $J_1 \to J_2 \to J_3 \to J_4$
(b). $J_4 \to J_3 \to J_2 \to J_1$

Fig. 1. Performance comparison for a batch of jobs under different job submission orders

The job ordering optimization for MapReduce workloads is important as well as challenging, due to the following facts: (i). There is a strong data dependency between the map tasks and reduce tasks of a job, i.e., reduce tasks can only perform after the map tasks, (ii). map tasks have to be allocated with map slots and reduce tasks have to be allocated with reduce slots, (iii). Both map slots and reduce slots are limited computing resources, configured by hadoop administrator in advance [3].

In this paper, we propose a prototype system *MROrder*[1] that can perform job ordering automatically for arriving jobs queued in Hadoop FIFO buffer. There are two core components for *MROrder*, namely, *policy module* and *ordering engine*. The policy module decides when and how to perform job ordering. The ordering engine, consisting of two approaches (i.e., simulation-based ordering approach and algorithm-based ordering approach), gives the job ordering. MROrder is designed to be flexible for different performance metrics, such as *makespan* and *total completion time*.

We evaluate our MROrder system using both synthetic workloads. Both makespan and total completion time are considered. Experimental results show that there is about 11–31% performance improvement based on MROrder system, depending on the characteristic of testbed workloads. Moreover, for synthetic Facebook workloads which contain lots of small-size jobs, the MROrder can improve the performance of the total completion time up to 176%.

2 Related Work

The batch job ordering optimization has been extensively researched in HPC literature [4]. In those studies, parallel tasks can be classified into three types: *rigid* (the number of processors to execute the task is fixed a priori), *moldable* (the number of processors to execute the task is not fixed but determined before the execution) and *malleable* (the number of processors for a task may change during the execution) [5]. In contrast, the *malleable task* is the most popular and widely studied. Its has been proved to be $\mathcal{NP}$-hard for *makespan* optimization [4], and a number of approximation and heuristic algorithms (e.g., [5,10]) were proposed. Meanwhile, there are some bi-criteria optimization algorithms proposed for optimizing *makespan* and *total completion time* simultaneously, such as [6].

[1] MROrder is open source and available at `http://sourceforge.net/projects/mrorder/`

The previous optimization works in HPC are implicitly targeted at the single-stage parallelism. In contrast, MapReduce is an interleaved parallel and sequential computation model [7]. It is close to the two-stage hybrid flow shop (HFS) [8]. Specifically, when each job contains only one map task and one reduce task, the MapReduce job ordering problem turns to be a two-stage HFS. The *makespan* optimization for two-stage HFS is strongly $\mathcal{NP}$-hard when at least one stage contains multiple processors [11]. There has been a large body of approximation and heuristic algorithms (e.g., [12,13]) for it. Besides, for HFS, there are also works (e.g., [15,14]) targeted at the bi-criteria optimization of both *makespan* and *total completion time*.

However, a MapReduce job runs multiple map/reduce tasks concurrently in each phase, which is different from the traditional HFS that allows only at most one task to be processed at a time. The MapReduce is more similar to the two-stage Hybrid flow shop with multiprocessor tasks (HFSMT) [16,17], which allows a task at each stage to be processed on multiple processors simultaneously. However, there is a strict requirement for HFSMT that a task at each stage can only be scheduled to execute only when there are enough idle processors for the task [17]. In contrast, the number of running map/reduce tasks for a MapReduce job is dynamically scaling up and down at runtime by allocating the tasks with available map/reduce slots.

In summary, this paper has taken into account all of these similarities to HFS as well as differences for MapReduce jobs. The most related work to us for MapReduce are [3,18]. Moseley et al. [3] presented a 12-approximation algorithm for the offline workloads of minimizing the *total flow time*, which is the sum of the time between the arrival and the completion of each job. Verma et al. [18] proposed two algorithms for *makespan* optimization of offline jobs. One is a greedy algorithm based on Johnson's Rule. The other one is a heuristic algorithm called BalancedPool. They evaluated their strength experimentally. In contrast, our work considers both *makespan* and *total completion time* optimization for *online* recurring MapReduce workloads, where jobs arrive over time and perform recurring computations in different time windows. Particularly, previous study showed that 75% of queries from Microsoft are recurring workloads [24]. MROrder is designed to optimize the performance for such scenarios.

3 Definition and Performance Metrics

Variable Definition. In an Hadoop cluster, let $\mathcal{M}$ denote the map phase and $\mathcal{R}$ denote the reduce phase. Let its slot configuration be $\mathcal{S} = \{\mathcal{S}^{\mathcal{M}}, \mathcal{S}^{\mathcal{R}}\}$, where $\mathcal{S}^{\mathcal{M}}$ denotes the set of map slots and $\mathcal{S}^{\mathcal{R}}$ denotes the set of reduce slots. Therefore, the *map task capacity* is $|\mathcal{S}^{\mathcal{M}}|$ and the *reduce task capacity* is $|\mathcal{S}^{\mathcal{R}}|$. For a batch of online jobs $J = \{J_1, J_2, ..., J_n\}$, let t_i^a denote the arriving time. Let $t_i^{\mathcal{M}}$ be the average processing time of a map task and $t_i^{\mathcal{R}}$ be the average processing time of a reduce task for each job J_i. Moreover, the set of its map tasks is denoted as $J_i^{\mathcal{M}}$ and the set of its reduce tasks is denoted as $J_i^{\mathcal{R}}$. Then the number of map tasks for J_i is $|J_i^{\mathcal{M}}|$ and the number of reduce tasks is $|J_i^{\mathcal{R}}|$.

Performance Metrics. There are several classical performance metrics for job ordering optimization, e.g., *makespan*, *total completion time* and *total flow time*. Let c_i denote the *completion time* of the job J_i. The *makespan* for the whole jobs J is defined as the *maximum* completion time of any job, i.e., $C_{max} = \max\{c_i\}$. The *total completion time (TCT)* for the whole jobs J is defined as $C_{tct} = \sum c_i$.

Problem Definition. Our goal is to minimize *makespan* and *total completion time*. That is, how to order the *online* arriving jobs automatically such that the makespan (or the total completion time) for all the jobs is minimized?

4 MROrder System

This section describes the design and implementation of the *MROrder* system.

4.1 System Overview

Figure 2 presents the overall design architecture for *MROrder* system. It gives the job order for arriving jobs, which can be submitted by a user or from other softwares such as Pig, Hive, Mahout, Oozie, etc. Particularly, the jobs are submitted in an *ad hoc* manner from users. We do not have assumption for the arrival order as well as arrival rate of jobs. There is a *JobDispatchingQueue* for queueing arriving jobs before submitting them to the MapReduce cluster. The *MROrder* job ordering manager handles the job ordering for arriving jobs queued in *JobDispatchingQueue* automatically. For each MapReduce job, the *MROrder* system needs to know the following information, i.e., the number of map (or reduce) tasks, the average time for each map (or reduce) task, and its arriving time. There are two key components for *MROrder* job ordering manager, namely, *Policy Module* and *Ordering Engine*. The *policy module* determines when and how to perform job ordering for MapReduce jobs. Once a policy command is issued, the *ordering engine* then deals with the job ordering work automatically. The specific description for each component is detailed in the following sections.

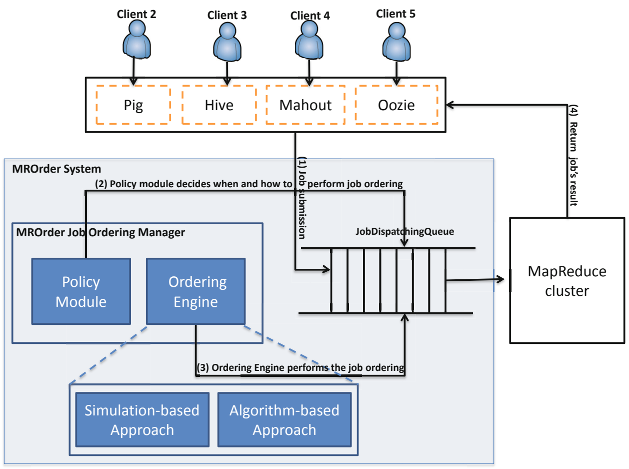


Fig. 2. The overall architecture for *MROrder* system

4.2 Policy Module

The policy module is invoked when there are arriving jobs queued in the *JobDispatchingQueue*, pending to be dispatched to the MapReduce cluster. It determines a good job ordering strategy to optimize target performance metrics (e.g., *makespan* or *total completion time*). The strategy is a combination of the choice of job ordering approach, the policy for the number of jobs for ordering and time policy (when to perform job ordering). It chooses the job ordering approaches (e.g., simulation-based approach, algorithm-based approach) based on their accuracy and efficiency characteristics (Section 4.3). Particularly, since simulation-based job ordering is a brute-force method, it can provide an optimal result but its efficiency is quite low, indicating that it is suitable for a small number of jobs. In contrast, the algorithm-based job ordering approach is efficient but it can only provide a sub-optimal result, which is suitable for a large number of jobs. Furthermore, the policy for the number of ordering jobs (PNJ) and time policy (TP) are correlated. We need to consider them together. We have the following two solutions:

PNJ-Dominated Solution. The user sets a threshold (n_0) for the number of jobs required to perform ordering. The ordering engine is triggered automatically when the number of arriving jobs reaches that threshold ($n \geqslant n_0$). The *TP* completely depends on the *PNJ*. It can be dynamically determined and computed by subtracting the latest-round job ordering time (or the starting time) by the current time.

TP-dominated Solution. Given a time interval Δt, the ordering engine is invoked at the time $t = \Delta t + t^{'}$, where $t^{'}$ is the latest-round time when the ordering engine was activated (or the starting time). The number of jobs n is thus equivalent to the number of arriving jobs during this time interval. The TP-dominated solution is shown in Algorithm 1.

Algorithm 1. *TP-dominated Solution with Fixed Time Interval (TP-FTI)*

```
1. Assume that MapReduce cluster start at the time $t^{curr} = 0$. For each arriving job $J_i$, it will
   be first queued in the JobDispatchingQueue. There is a boolean attribute $orderflag_i$ for
   each $J_i$. It is initialized to be $orderflag_i = false$ by default.
2. The MROrder job ordering manager waits for a time interval $\Delta t$ until the current time
   $t^{curr} = t^{curr} + \Delta t$. The policy module checks the arriving jobs queued in the JobDis-
   patchingQueue to filter out sub-set $J_A$, where $J_A = \{J_i | (J_i \in J) \wedge (t_i^a \leqslant t^{curr}) \wedge$
   $(orderflag_i = false)\}$. Thus the number of jobs at this job ordering round is $|J_A|$.
3. The job ordering engine is triggered by the policy module. It does job ordering and marks
   $orderflag_i = true$ for jobs in $J_A$.
4. The MROrder system dispatches those jobs $J_i$ with $orderflag_i = true$ in the JobDispatch-
   ingQueue and goes back to step 2.
```

Given $\Delta t = 60$ sec configured by the user, for example, the *MROrder* job ordering engine is activated every 60 secs, ordering the arriving jobs queued in the *JobDispatchingQueue* and dispatching them into MapReduce cluster. The value of Δt has a big impact on the whole performance. Too small value of Δt can make the MROrder job ordering engine work so frequently that there may be a very few jobs available (e.g., 0, 1 or 2 jobs at each job ordering round) in the *JobDispatchingQueue* at each job ordering round, losing the effect of job ordering. However, too large value of Δt will make *JobDispatchingQueue* hold lots of jobs without distributing it to the MapReduce cluster, causing MapReduce cluster keep idle without running jobs and in turn have a adverse effect on the performance. Moreover, even we have a fine configuration for Δt, it is still inflexible and not adapted to the job arrival rate. We further propose an adaptive *TP* solution to solve this problem, as shown in Algorithm 2.

Algorithm 2. *TP-dominated Solution with Adaptive Time Interval (TP-ATI)*

```
1. Let MapReduce cluster start at the time $t^{curr} = 0$. For each arriving job $J_i$, it will be first queued in the JobDispatchingQueue. There is a boolean attribute $orderflag_i$ for each $J_i$. It is initialized to be $orderflag_i = false$ by default. Initially, let $t^{wait} = \Delta t$.
2. The MROrder job ordering manager waits for a time interval $t^{wait}$ until the current time $t^{curr} = t^{curr} + t^{wait}$. The policy module checks the arriving jobs queued in the JobDispatchingQueue to filter out sub-set $J_A$, where $J_A = \{J_i | (J_i \in J) \wedge (t_i^a \leqslant t^{curr}) \wedge (orderflag_i = false)\}$. Thus the number of jobs at this job ordering round is $|J_A|$.
3. The job ordering engine is triggered by the policy module. It does job ordering and marks $orderflag_i = true$ for jobs in $J_A$.
4. The MROrder system dispatches those jobs $J_i$ with $orderflag_i = true$ in the JobDispatchingQueue.
5. The policy module updates $t^{wait}$ as follows: $t^{wait} = \max\left\{\Delta t, T_A\right\}$, where $T_A = \max_{1 \leqslant k \leqslant |J_A|} \left\{\sum_{i=1}^{k} \frac{|J_i^{\mathcal{M}}| \cdot t_i^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{M}}|} + \sum_{i=k}^{|J_A|} \frac{|J_i^{\mathcal{R}}| \cdot t_i^{\mathcal{R}}}{|\mathcal{S}^{\mathcal{R}}|}\right\}$.
6. Go back to Step 2.
```

The rationale for the adaptive waiting time adjustment based on the algorithm *TP-ATI* is that, user provides a relatively small threshold Δt for waiting time. The policy module adjusts it dynamically according to the estimated running time T_A of those workloads J_A that have been distributed to MapReduce cluster at the previous dispatching round. The MROrder tries to queue as many jobs as possible in the *JobDispatchingQueue* at each job ordering round while keeping the MapReduce cluster busy.

4.3 Ordering Engine

The ordering engine (OE) is triggered according to the policies in the policy module. The *MROrder* system provides two types of job ordering approaches, i.e., *simulation-based ordering approach* and *algorithm-based ordering approach*. The policy module is responsible for selecting the suitable ordering engine dynamically based on the number of jobs at each job ordering round. The basic idea is that the simulation-based ordering approach is chosen when there are a small number of jobs (e.g., 7 jobs), considering that it can produce an optimal result but is time-consuming. The algorithm-based ordering approach is selected for a large number of jobs.

Simulation-Based Ordering Approach (SIM). To enable simulation-based job ordering, we developed a Hadoop simulator named *HSim*. It is a tailored simulator aiming to evaluate the performance of varied job orders with a file input consisting of jobs information each with five arguments: job's ID, the number of map tasks, the number of reduce tasks, the average running time of a map task, the average running time of a reduce task. We build our simulation-based ordering approach based on *HSim*. It is a brute-force method that can enumerate all possible job orders to explore the optimal job order for a given performance metric (e.g., *makespan*, *total completion time*). Note that there are $n!$ possible job orders for n jobs. For example, there are $9! = 362880$ possible job orders for $n = 9$ jobs, which however takes 97.179 sec (refer to Table 2) for enumerating all job orders. It indicates that the simulation-based ordering approach is only feasible for a small number of jobs in practice. Moreover, instead of searching the whole space of all job orders, one might consider the Monte Calo method combined with HSim for suboptimal (rough) results by searching the partial space statistically for a large number of jobs (e.g., 50 jobs, 100 jobs). However, we argue that it is still not meaningful for a large number of jobs in practice. For example, assume that we want to control the maximum execution time of simulation not exceeding 97.179 sec (i.e., our sample space of job orders is 362880). When it comes to 20 jobs, it can only cover $\frac{362880}{20!=2432902008176640000} = 1.49 \times 10^{-13}$, which is very tiny and unmeaningful. Therefore, there is a need to explore an efficient solution for a large number of jobs in the following subsection.

Algorithm-Based Ordering Approach (ALG). We develop an algorithm-based ordering approach to deal with the job ordering for MapReduce workloads with a large number of jobs. It contains some job ordering greedy algorithms for different performance metrics. Particularly, we incorporate a greedy algorithm *MK* based on *Johnson's Rule* [9], as shown in Algorithm 3 for *makespan* optimization. It is an optimal and efficient $O(n \log n)$ job ordering algorithm for the makespan optimization for the two-stage flow shop with

Algorithm 3. Greedy algorithm based on *Johnson's Rule* (MK)

1. For each job J_i, we first estimate its map-phase processing time $T_i^{\mathcal{M}}$ and reduce-phase processing time $T_i^{\mathcal{R}}$ by using the following formula:

$$\left(T_i^{\mathcal{M}}, T_i^{\mathcal{R}}\right) = \left(\frac{|J_i^{\mathcal{M}}|}{|\mathcal{S}^{\mathcal{M}}|} \cdot t_i^{\mathcal{M}}, \frac{|J_i^{\mathcal{R}}|}{|\mathcal{S}^{\mathcal{R}}|} \cdot t_i^{\mathcal{R}}\right).$$

2. We order jobs in J based on the following principles:
   a). Partition jobs set J into two disjoint sub-sets J_A and J_B:

$$J_A = \{J_i | (J_i \in J) \wedge (T_i^{\mathcal{M}} \leqslant T_i^{\mathcal{R}})\}, \quad J_B = \{J_i | (J_i \in J) \wedge (T_i^{\mathcal{M}} > T_i^{\mathcal{R}})\}.$$

   b). Sort all jobs in J_A from left to right by non-decreasing $T_i^{\mathcal{M}}$. Order all jobs in J_B from left to right by non-increasing $T_i^{\mathcal{R}}$.
   c). Make an ordered jobs set $J^{'}$ by joining all jobs in J_A first and then J_B in order, i.e., $\phi_1 : J^{'} = \{\{J_A\}, \{J_B\}\}$.

one processor per stage. The details of Johnson's rule is as follows. Divide the jobs set J into two disjoint sub-sets J_A and J_B. Set J_A consists of those jobs J_i for which $T_i^{\mathcal{M}} < T_i^{\mathcal{R}}$. Set J_B contains the remaining jobs (i.e. $J \backslash J_A$). Sequence jobs in J_A in non-decreasing order of $T_i^{\mathcal{M}}$ and those in J_B in non-increasing order of $T_i^{\mathcal{R}}$. The optimized job order is obtained by appending the sorted set J_B to the end of sorted set J_A. Moreover, we also include a greedy algorithm *TCT* for the total completion time optimization, as shown in Algorithm 4, based on *shortest processing time first*. In comparison to the simulated-based ordering approach, the algorithm-based ordering approach is much more efficient, but it can only produce the sub-optimal result. Moreover, to support user's job ordering algorithms, *MROrder* system also provides a user interface in the algorithm-based ordering approach. Therefore, based on our *MROrder* system, user can extend the algorithm-based ordering approach for other's performance metrics.

Algorithm 4. Greedy algorithm based on *Shortest Processing Time First* (TCT)

```
1. For each job $J_i$, we first compute its processing time $T_i$ by using the formula below:
       $T_i = T_i^{\mathcal{M}} + T_i^{\mathcal{R}} = \frac{|J_i^{\mathcal{M}}|}{|\mathcal{S}^{\mathcal{M}}|} \cdot t_i^{\mathcal{M}} + \frac{|J_i^{\mathcal{R}}|}{|\mathcal{S}^{\mathcal{R}}|} \cdot t_i^{\mathcal{R}}.$
2. Order all jobs in $J$ from left to right by non-decreasing $T_i$.
```

4.4 Implementation

We have developed a prototype of the *MROrder* system. The prototype implements all components of the *MROrder* job ordering manager. The policy module provides users with all policy solutions mentioned above for choices and adopts *TP-ATI* by default. Several user's arguments are provided, including the optimization targets (e.g., *makespan*, *total completion time*), the threshold for waiting-time interval as well as the maximum number of jobs allowed at each job ordering round. The *MROrder* system automates the corresponding job ordering policy in runtime based on user's argument configuration. Moreover, our prototype adopted our simulator *HSim* as the computing component of the MapReduce cluster to simulate the computation process of online MapReduce batch jobs. The current prototype primarily aims to study various automated policy solutions for online workloads under different performance metrics. It remains as ongoing work to incorporate it into Hadoop framework for practical use.

Data Skew. In our MROrder system, we assume that the sizes and processing time of all data blocks are the same, i.e., there is no data skew among data blocks. For the case of data skew, user can use the model provided by [26] to diminish it.

Overhead. The overhead of MROrder mainly comes from the ordering engine to perform job ordering. The detailed results are given in Section 5.3. Generally, SIM takes longer time than ALG, but it provides better performance result. Thus, there is a trade-off between the performance result and overhead for the dynamic choice of job ordering approach.

5 Experimental Evaluation

In this section, we evaluate MROrder prototype and its associated policies. The detailed evaluation method is that: first, we discuss and compare the effectiveness of proposed policy solutions (e.g., TP-FTI, TP-ATI). Then we evaluate and discuss the suitable value of threshold (of the number of jobs) as the condition for switching of job ordering approach. Third, we evaluate the performance for MROrder with regard to makespan as well as total completion time. Finally, we evaluate the accuracy of our simulator *Hsim* adopted by MROrder experimentally.

5.1 Workloads

Our experiment consists of two types of synthetic workloads. One is the synthetic Facebook workload, generated based on [19,20]. Specifically, the number of map/reduce tasks as well as the arriving time for each job are based the input/output data sizes of workloads provided by [19]. We estimate the running time of map and reduce tasks per job based on the map and reduce durations in Figure 1 of [20]. More precisely, we follow the LogNormal distribution [21] with *LN*(9.9511,1.6764) for map task duration and *LN*(12.375,1.6262) for reduce task duration that fits best the Facebook task duration, given and demonstrated by [22]. It contains lots of small-size jobs (more than 58% in the number of jobs) [20]. We use it primarily to evaluate the total completion time for MROrder system.

Our second workload is a testbed workload. In contrast to synthetic Facebook workload, most of its jobs are large-size. The makespan is seriously affected primarily by the positions of large-size jobs. We use it mainly to evaluate the makespan for MROrder system.

5.2 Evaluation and Analysis of Policy Solutions

Recall that in the policy module of MROrder system, we provided several policy solutions to determine when and how to perform job ordering dynamically. Table 1 illustrates the comparison results of two policy solutions TP-FTI and TP-ATI for their suitable threshold Δt and the corresponding performance improvement of total completion time under varied sizes of synthetic Facebook workloads. Particularly, we evaluate different Δt from 10 sec, 20 sec, 30 sec till to 400 sec. We can observe that, (1). the suitable value of Δt for TP-FTI, TP-ATI is $230 \sim 350$ sec, and $10 \sim 30$ sec, respectively. It indicates that the threshold for the fixed-time interval method TP-FTI should be large, whereas it should be small for the adaptive method TP-ATI, relying on its adaptive mechanism to change the waiting time interval between two successive job ordering dynamically; (2). Under the suitable value of Δt, we note that the performance improvement of TP-ATI is much better than that of TP-FTI. This is because the TP-ATI is smarter than TP-FTI. Therefore, we take TP-ATI as the policy solution for the policy module in the following experiments.

Table 1. The comparison results of two different policy solutions for their suitable threshold Δt and the corresponding *performance improvement of total completion time* (PITCT) under varied sizes of synthetic Facebook workloads. The PITCT is a normalized ratio of performance improvement with MROrder to the unoptimized one.

JobNum	**TP-FTI**		**TP-ATI**	
	$\Delta t(sec)$	**PITCT** (%)	Δt (sec)	**PITCT** (%)
50	230	41.91	10	44.31
100	230	14.81	10	28.30
150	230	9.28	30	14.13
200	230	6.98	30	10.83
250	310	24.08	30	123.74
300	350	19.49	30	186.29
350	350	13.11	20	89.7
400	350	9.81	20	56.91

5.3 Switching Threshold for the Number of Jobs for Job Ordering Approach

In our MROrder prototype, we provide two types of job ordering approaches, namely, *SIM* and *ALG*. There is a tradeoff between the accuracy and overhead (i.e., the erased time it takes.) for these two ordering approaches (See Section 4.3 for details).

Table 2. Performance and overhead comparison of ALG versus SIM

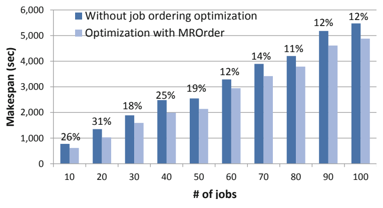
JobNum	**Makespan for ALG (sec)**	**Makespan for SIM (sec)**	**Total Completion Time for ALG (sec)**	**Total Completion Time for SIM (sec)**	**Erased Time for ALG (sec)**	**Erased Time for SIM (sec)**
1	45	45	45	45	0.001	0.002
2	170	170	240	240	0.001	0.003
3	200	198	456	453	0.003	0.003
4	338	324	799	796	0.003	0.003
5	399	394	1342	1274	0.003	0.028
6	399	396	1450	1363	0.003	0.123
7	440	437	1766	1736	0.003	0.952
8	475	471	2107	2050	0.003	9.305
9	573	564	2728	2596	0.004	97.179

Table 2 presents the comparison results under different numbers of jobs (e.g., 1-9) from our testbed workload. It consists of three parts. Column 2 and 3 give the results for makespan. Column 4 and 5 show the results for total completion time. Column 6 and 7 give the overheads for ALG and SIM ordering engines. We can observe that, (1). The results based on SIM ordering engine are better (more minimal) than that of ALGs for both makespan and total completion time. This is because SIM is a brute-force method that searches all possible job orders to get an optimal one, whereas ALGs are greedy algorithms that can only produce suboptimal results. (2). The results produced by ALG are close to SIM results, especially for makespan produced by algorithm *MK*. (3). The erased time (i.e., the overhead) consumed by ALG is very small and does not grow much as the number of jobs increases. However, the erased time for SIM grows exponentially as the number of jobs increases, especially when the number of jobs equals to 9 (e.g., 97.179 sec). It is because ALGs are $n \log n$ algorithms, whereas SIM is an $n!$ brute-force method. Based on the experimental erased time and performance results, we set the threshold for the number of jobs to be 7 as a threshold for the dynamical choice of job ordering engines.

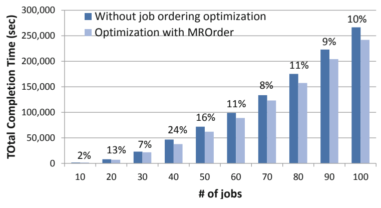
5.4 Performance Evaluation of MROrder System

We evaluate the performance of MROrder system by considering two metrics (i.e., makespan, and total completion time) and two kinds of workloads (e.g., Facebook workloads and testbed workloads). In our MROrder, we take TP-ATI for the policy module with Δt of 10 sec.

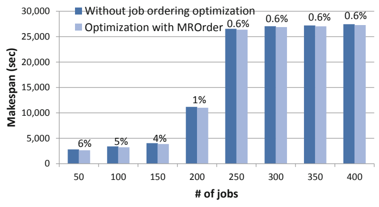
Figure 3 presents optimized performance results based on MROrder system, under varied sizes of online workloads. Specifically, the results for testbed workloads are shown in Figure 3 (a) and Figure 3 (b). The results for synthetic Facebook workloads are shown in Figure 3 (c) and Figure 3 (d). There is about $11\% - 31\%$ makespan improvement for testbed workloads in Figure 3 (a), whereas there is only 3% for Facebook workloads on average in Figure 3 (c). It is because that the makespan is affected primarily by the position of large-size jobs. The testbed workloads contain lots of large-size jobs. In contrast, the Facebook workloads consist of a large number of small-size jobs. On the other hand, for total completion time, Figure 3 (d) illustrates that the maximum performance improvement can be up to 176% for synthetic Facebook workloads. In contrast, there is a maximum of 24% performance improvement for total completion time of testbed workloads. The reason is that the total completion time is primarily



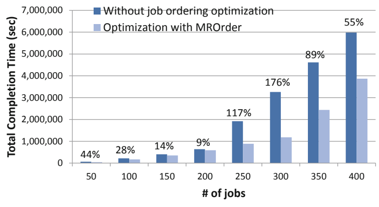
(a) Makespan under different testbed workload sizes.



(b) Total completion time under different testbed workload sizes.



(c) Makespan under different synthetic Facebook workload sizes.



(d) Total completion time under different synthetic Facebook workload sizes.

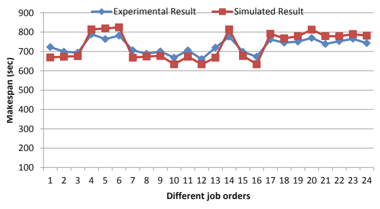
Fig. 3. The optimized performance results for MROrder system under different sizes of testbed workloads and synthetic Facebook workloads

dominated by the positions of small-size jobs. The total completion time might be poor when there are lots of small-size jobs in a workload, e.g., Facebook workload.

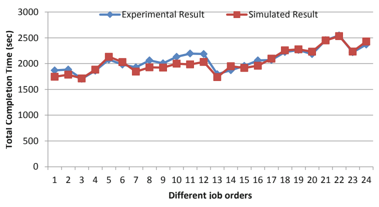
5.5 Accuracy Evaluation for Hsim

We validate the accuracy of our *Hsim* by comparing the simulation results with the experimental results of a MapReduce workload. We generate our MapReduce workload by using three representative applications, i.e., **wordcount** application (computes the occurrence frequency of each word in a document), **sort** application (sorts the data in the input files in a dictionary order) and **grep** application (finds the matches of a regex in the input files). We take Wikipedia article history dataset[2] of 10GB, as application input data. We ran experiments in Amazon's Elastic Compute Cloud(EC2) [23]. Our EC2 Hadoop cluster consists of 20 nodes each belonging to a "Extra Large" VM. We configure one node as master and namenode, and the other 19 nodes as slaves and datanodes. Each "Extra Large" instance has 4 virtual cores with 2 EC2 compute units each [23]. We configure 3 map and 1 reduce slots per slave node.

We consider the makespan as well as total completion time for all possible job orders of the MapReduce workload. Figure 4 (a) and Figure 4 (b) present the results for all $4! = 24$ job orders of a batch of 4 jobs. We note that the simulated results of both makespan and total completion time are very close (errors within 8%) to the experimental results, which validates the accuracy of our *Hsim*.



(a) *Makespan* for a batch of 4 jobs in all job orders



(b) *Total completion time* for a batch of 4 jobs in all job orders

Fig. 4. Simulated results versus experimental results for a MapReduce workload

6 Conclusion and Future Work

This paper proposed a prototype system named MROrder to perform job ordering optimization automatically for online MapReduce workloads. Several policy solutions were presented and evaluated to dynamically determine when and how to do job ordering. The MROrder system is designed to be flexible for different optimization metrics. It has implemented several algorithms to support the job ordering optimization for makespan and total completion time. It also provides an interface for users to add their job ordering algorithms into MROrder for optimization of other performance metrics.

[2] `http://dumps.wikimedia.org/enwiki/`

We are integrating MROrder into Hadoop framework. Moreover, our prototype for MapReduce only supports FIFO scheduling. In future, we will consider other schedulers such as Fair Scheduler [20], and heterogeneous environments such as [25].

Acknowledgment. This work was supported by the ”User and Domain driven data analytics as a Service framework” project under the A*STAR Thematic Strategic Research Programme (SERC Grant No. 1021580034).

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. OSDI (2004)
2. HowManyMapsAndReduces, `http://wiki.apache.org/hadoop/HowManyMapsAndReduces`
3. Moseley, B., Dasgupta, A., Kumar, R., Sarl, T.: On scheduling in map-reduce and flow-shops. SPAA, 289–298 (2011)
4. Leung, J.Y.T.: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Chapman and Hall/CRC, 25-5-25-18 (2004)
5. Dutot, P.F., Mounie, G., Trystram, D.: Scheduling parallel tasks approximation algorithms. In: Leung, J.T. (ed.) Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Chapman Hall, CRC Press (2004)
6. Dutot, P., Eyraud, L., Mounie, G., Trystram, D.: Bi-criteria Algorithm for Scheduling Jobs on Cluster Platforms. SPAA, pp. 125–132 (2004)
7. Howard, K., Siddharth, S., Sergei, V.: A model of computation for MapReduce. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 938–948 (2010)
8. Gupta, J.N.D.: Two stage hybrid flowshop scheduling problem. Journal of Operational Research Society 364, 359C–364C (1988)
9. Johnson, S.M.: Optimal two- and three-stage production schedules with setup times included. Naval Res Logist Q 1, 61–68 (1954)
10. Sanders, P., Speck, J.: Efficient Parallel Scheduling of Malleable Tasks. IPDPS, pp. 1156–1166 (2011)
11. Gupta, J.N.D., Hariri, A.M.A., Potts, C.N.: Scheduling a two-stage hybrid flow shop with parallel machines at the first stage. Annals Of Operations Research 69, 171–191 (1997)
12. Kyparisis, G.J., Koulamas, C.: A note on makespan minimization in two-stage flexible flow shops with uniform machines. European Journal of Operational Research 175, 1321–1327 (2006)
13. Hejazi, S.R., Saghafian, S.: Flowshop-scheduling problems with makespan criterion: a review. International Journal of Production Research 43, 2895–2929 (2005)
14. Gupta, J.N.D., Hennig, K., Werner, F.: Local search heuristics for two-stage flow shop problems with secondary criterion. Journal Computers and Operations Research 29, 123–149 (2002)
15. Rajendran, C.: Two-Stage Flowshop Scheduling Problem with Bicriteria. Journal of the Operational Research Society 43, 871–884 (1992)
16. Oğuz, C., Ercan, M.F., Cheng, T.C.E., Fung, Y.F.: Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop. European Journal of Operational Research 149, 390–403 (2003)

17. Oğuz, C., Ercan, M.F.: Scheduling multiprocessor tasks in a two-stage flow-shop environment. In: Proceedings of the 21st International Conference on Computers and Industrial Engineering, pp. 269–272 (1997)
18. Verma, A., Cherkasova, L., Campbell, R.: Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance. In: MASCOTS (2012)
19. Chen, Y.P., Ganapathi, A., Griffith, R., Katz, R.: The Case for Evaluating MapReduce Performance Using Workload Suites. In: MASCOTS (2011)
20. Zaharia, M., Borthakur, D., Sarma, J.: Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In: EuroSys, pp. 265–278 (2010)
21. LogNormal Distribution, http://en.wikipedia.org/wiki/Log-normal_distribution
22. Verma, A., Cherkasova, L., Kumar, V.S., Campbell, R.H.: Deadline-based Workload Management for MapReduce Environments: Pieces of the Performance Puzzle. In: NOMS (2012)
23. Amazon EC2, http://aws.amazon.com/ec2
24. He, B.S., Yang, M., Guo, Z., Chen, R.S.: Comet: batched stream processing for data intensive distributed computing. In: SOCC, pp. 63–74 (2010)
25. Tan, Y.S., Lee, B.S., Campbell, R.H., He, B.S.: A Map-Reduce Based Framework for Heterogeneous Processing Element Cluster Environments. In: CCGrid (May 2012)
26. Ibrahim, S., Jin, H., Lu, L., He, B.S., Wu, S.: Adaptive I/O Scheduling for MapReduce in Virtualized Environment. ICPP 2011, 335–344 (2011)

Leveraging Collaborative Content Exchange for On-Demand VM Multi-deployments in IaaS Clouds

Bogdan Nicolae and M. Mustafa Rafique

IBM Research, Dublin, Ireland
{bogdan.nicolae,mustafa.rafique}@ie.ibm.com

Abstract. A critical feature of IaaS cloud computing is the ability to deploy, boot and terminate large groups of inter-dependent VMs very quickly, which enables users to efficiently exploit the on-demand nature and elasticity of clouds even for large-scale deployments. A common pattern in this context is multi-deployment, i.e., using the same VM image template to instantiate a large number of VMs in parallel. A difficult trade-off arises in this context: access the content of the template on-demand but slowly due to I/O bottlenecks or pre-broadcast the full contents of the template on the local storage of the hosting nodes to avoid such bottlenecks. Unlike previous approaches that are biased towards either of the extremes, we propose a scheme that augments on-demand access through a collaborative scheme in which the VMs aim to leverage the similarity of access pattern in order to anticipate future accesses and exchange chunks between themselves in an attempt to reduce contention to the remote storage where the VM image template is stored. Large scale experiments show improvements in read throughput between 30%-40% compared to on-demand access schemes that perform in isolation.

1 Introduction

Infrastructure-as-a-Service (IaaS) cloud computing has matured over the years up to the point where it represents a potentially cost-effective solution with low entry barrier even for workloads that require huge amounts of computational resources, such as large scale scientific or data-intensive computations.

One of the main features that has contributed to the growing popularity of IaaS is the elastic on-demand provisioning of virtual machines (VMs). Users can bring up a whole virtual cluster and reconfigure it dynamically with a simple click of a button [1]. However, as the user interface grows simpler and the types of workloads diversify [2], achieving efficient on-demand VM provisioning is a non-trivial task.

A particularly difficult challenge in this context is the *multi-deployment* pattern, i.e., provisioning a large number of inter-dependent VMs concurrently from the same VM image template, which is often needed to deploy large-scale HPC and data-intensive applications. Obviously, there is a need minimize the provisioning time and guarantee scalability despite a growing number of VMs, otherwise users do not perceive IaaS as truly on-demand and lose interest, while at the same time cloud providers lose potential profit by not efficiently leveraging their computational resources. This issue is especially important in the context of spot instances [3]: users can bid for idle cloud resources at lower than regular prices, however with the risk of their VM being terminated at any moment without notice when other users bid higher. Given such a context, long provisioning time is not only inconvenient for the user, but actually leads to loss of computational time and resources that could have been otherwise leveraged at a low price.

Despite widespread need for multi-deployments, little effort has been undertaken to improve their scalability. Current techniques often pre-copy the full VM image locally on the compute nodes before launching the VM instances, which can take in the order of tens of minutes or even hours [4], not counting the time to boot the guest operating system and deploy the application itself. Although on-demand techniques have matured (e.g., locally derived copy-on-write images [5] that use a remotely stored VM image template as a backing file) and they have been shown to generate little overhead on application performance compared to the case when a local copy is available [6], they saw comparatively little attention for multi-deployments due to the fact that they generate I/O contention to the repository where the VM image template is stored.

This paper contributes with a novel technique that aims to alleviate the aforementioned issue and improve the scalability of multi-deployments by enabling efficient decentralized on-demand access that avoids bottlenecks caused by competition to the repository. To achieve this, we leverage the fact that the VMs of the group typically have highly similar access patterns (e.g., during the boot phase they access the same chunks of the virtual disk in the same order [7]) in order to build a collaborative scheme where VMs exchange chunks between themselves in anticipation of expensive concurrent I/O accesses to the remote repository that would follow, which thus can be avoided. Our approach can dynamically adapt to the access pattern: it increases the rate of exchanges when remote accesses were successfully avoided, and backs off when the success rate starts dropping due to diverging access patterns. We summarize our contributions as follows:

- We introduce a collective content exchange scheme that optimizes the multi deployment pattern by enabling efficient sharing of virtual disk image templates in an on-demand fashion and show how to integrate this approach in a typical IaaS architecture (Sections 3.1 and 3.3).
- We propose a hypervisor-transparent implementation of this scheme as an independent FUSE module that can mount a raw remote backing file locally as a mutable snapshot. This is functionally equivalent to the broadcast technique but completely removes the broadcast overhead (Section 3.4).
- We experimentally evaluate the benefits of our approach on the Grid5000 [8] testbed by performing multi-deployments on dozens of nodes (Section 4).

2 Related Work

Approaches that enable multi-deployment broadly fall into two major categories: pre-broadcast and on-demand access.

Pre-broadcast techniques fully copy the VM disk image template locally on all compute nodes before launching the VM instances themselves. This enables high I/O disk access performance inside the VM instances, because no remote access to the image repository or I/O competition with other VM instances is present. However, the broadcast step has an high overhead both in execution time and network traffic, which reduces the attractiveness of IaaS for short-lived jobs and is expensive for the provider (in terms of lost resources that could otherwise be charged for). Thus, reducing the broadcast overhead has been an active area of study, with proposals ranging from multi-cast [9] and application level broadcast-trees [10] to peer-to-peer protocols [4, 11].

At the other extreme are on-demand access approaches: VMs are instantiated on-the-fly by keeping the remote virtual disk image template read-only and storing all modifications locally using copy-on-write, which is natively supported by many hypervisors [5]. While this eliminates the broadcast step, it introduces I/O competition between VM instances because they share a single disk content source. Obviously, a centralized repository generates the highest contention, but is still a very popular choice due to simplicity [12]. Using a decentralized storage solution (such as a parallel file system [13–15] or a dedicated repository [16]) reduces contention thanks to striping, but is only partially effective in our case, because the VM instances often access the same chunks in the same order. In our previous work [7], we show how to alleviate this issue by means of adaptive prefetching, however I/O contention to the repository is still a potential problem for scalability.

Another emerging direction that relates to multi-deployment is user-level virtualization [17]. The idea here is to use a minimally configured OS and virtual disk on top of which application packages and configuration files are applied on-the-fly during boot time. In this context, the same content propagation principles that apply at low level (i.e., virtual disk chunks) can be used for higher level contents (i.e., packages and configuration files).

Finally, there are several ways to complement multi-deployments with additional optimizations. A straightforward optimization is to use the local storage available on the compute nodes as a caching layer for VM images [18]. While this does not improve first-time deployments, given the dynamicity of the cloud, many VM images pass through the same compute node during its lifetime, effectively increasing the chance to avoid a first time accesses for certain members of the multi-deployment group. However, given the large variety of VM image templates in a cloud, it is highly probable to quickly run out of local storage if full caching is attempted. Luckily, VM images share a large amount of content between each other, which makes de-duplication [19] an effective tool to leverage local storage more efficiently.

Our own approach tries to leverage the best trade-off between VM disk content broadcast and on-demand access. Much like on-demand techniques we distribute only the needed content on-the-fly, but at the same time we avoid remote I/O contention and access latency by involving the VM instances in a collaborative chunk exchange protocol in a manner similar to peer-to-peer approaches. To the best of our knowledge, we are the first to explore this direction.

3 Our Approach

3.1 Design Principles

Copy-on-Reference Local Mirroring: To facilitate on-demand VM disk image access, we leverage *copy-on-reference*, initially introduced for process migration in the V-system [20]. To this end, our approach exposes a private local view of the virtual disk image stored remotely on the VM repository to the hypervisor. We call this local view a *mirror*. From the perspective of the hypervisor, the local mirror behaves like the original and it is functionally equivalent to a local copy using pre-broadcast. The mirror is logically partitioned into fixed-sized *chunks*. Whenever the hypervisor needs to read a region of the image, all chunks covered by the region that are not already locally available are fetched remotely from the original source and copied (i.e., "mirrored") locally. Once all contents is available locally, the read can proceed. Writes behave in a similar fashion, except for those chunks that are totally overwritten: in this case no remote fetch is necessary.

Preventive Peer-to-Peer Content Exchange: As explained in Section 2, on-demand access has a serious disadvantage as it generates I/O access contention to the remote repository where the VM disk image is stored. Although copy-on-reference limits this effect to first-time reads only (because the local mirror gradually becomes populated), by itself this is often not enough, as most access patterns need to read data only once (e.g., read configuration files during the boot process or sweep through an input data set in order to perform a computation). Thus, optimizing first-time reads becomes a prime concern. Since the VM instances of multi-deployments often follow a similar access pattern, a natural idea in this context is to enable the VM instances to talk to each other and "help" each other out in order to reduce the pressure on the remote repository. Based on the observation that I/O contention leads to jitter [7] (i.e., slight differences in time when the same chunk is accessed), we propose to organize the VM instances in a peer-to-peer topology where each VM has a set of neighbors, with whom it "gossips" about the chunks that should be fetched on-demand. Based on this information, VMs are able to anticipate future trends in access pattern and obtain chunks from their neighbors before they are actually needed, effectively preventing costly remote accesses if the anticipation was successful.

Access Pattern Aware Content Exchange Throttling: Preventive peer-to-peer content exchange however is not without drawbacks. Although the performance overhead of exchanging chunks can be masked by decoupling it from on-demand access and running it as a background process, it invariably leads to network bandwidth utilization. This steals away bandwidth from the application running inside the VM instance and might even impact the on-demand access bandwidth. Therefore, it is crucial to "focus the gossiping" such that it maximizes the prediction rate, and thus minimizes the bandwidth wasted on obtaining chunks that were never needed. To this end, we propose to monitor the success rate in terms of number of chunks that were fetched locally but not yet accessed (which we refer to as *unmatched*). When the number of unmatched chunks reaches a predefined threshold, we assume the VM has started to exhibit an access pattern that diverges from the rest of the neighborhood and as such it will avoid sending chunks until the number of unmatched chunks falls below the threshold. Using this scheme, each VM dynamically adapts to the access pattern in relationship to the other VMs, nurturing its collaborations when it senses a common pattern, and backing off when it senses a divergence. To avoid the case when VMs converge again without lowering the amount of unmatched chunks accumulated in the past, one solution is to automatically eliminate unmatched chunks older than a predefined time window. For simplicity, we opted not to address this case for the purpose of this work.

3.2 Algorithmic Description

In this section, we zoom on the design principles presented in Section 3.1 by providing an algorithmic description. For simplicity, we insist only on the most important aspects, in particular how a read and a write is performed and how to decouple the peer-to-peer preventive exchange scheme from on-demand access and perform it asynchronously in the background.

Algorithm 1. Read the range $(offset, size)$ into $buffer$ from disk image

```
1: function READ($buffer, offset, size$)
2:     for all $chunk \in Image | chunk \cap (offset, size) \neq \emptyset$ do
3:         if $ChunkState[chunk] = REMOTE$ then
4:             fetch $chunk$ from repository and mirror it locally
5:             if $unmatched < THRESHOLD$ then
6:                 $HintQueue \leftarrow HintQueue \cup \{chunk\}$
7:             end if
8:             $ChunkState[chunk] \leftarrow READ$
9:         else if $ChunkState[chunk] = LOCAL$ then
10:            $unmatched \leftarrow unmatched - 1$
11:            $ChunkState[chunk] \leftarrow READ$
12:        end if
13:    end for
14:    return read $(offset, size)$ into $buffer$ from $Mirror$
15: end function
```

Each chunk of the virtual disk image can be in one of the four possible states (denoted $ChunkState$): $REMOTE$ (the chunk was not yet locally fetched), $LOCAL$ (the chunk was obtained through gossiping and is locally present, but was not yet needed), $READ$ (the chunk was requested by a read operation) and $WRITTEN$ (the chunk was overwritten either totally or partially).

The READ operation is detailed in Algorithm 1. In a nutshell, it determines all chunks that are missing locally and fetches them from the remote repository, after which it redirects the read request to the local mirror. If any chunk triggered on-demand access (i.e., was in the $REMOTE$ state), it is scheduled to be sent to the neighbors through $HintQueue$, which is then used by the preventive exchange. This happens only if the number of unmatched chunks is lower than the threshold. On the other hand, if any chunk is already available locally thanks to preventive exchange, the number of unmatched chunks is decremented. In both cases, $ChunkState$ transitions into $READ$, in order to reflect the new state.

Algorithm 2. Write the range $(offset, size)$ from $buffer$ to disk image

```
1: function WRITE($buffer, offset, size$)
2:     for all $chunk \in Image | chunk \cap (offset, size) \neq \emptyset$ do
3:         if $ChunkState[chunk] = REMOTE$ and $chunk \not\subset (offset, size)$ then
4:             fetch $chunk \setminus (offset, size)$ from repository and mirror it locally
5:         end if
6:         $ChunkState[chunk] \leftarrow WRITTEN$
7:     end for
8:     return write $(offset, size)$ from $buffer$ to $Mirror$
9: end function
```

The WRITE operation, depicted in Algorithm 2 simply needs to make sure that there are no missing chunks that are only partially overwritten and thus will generate gaps. If this is not the case, it fetches the missing content from the remote repository in order to fill those gaps. In either case, it marks all involved chunks as $WRITTEN$ and finally redirects the write to the mirror.

Algorithm 3. Asynchronous preventive chunk exchange with other peers

```
1: procedure BACKGROUND_EXCHANGE
2:     while true do
3:         if $chunk$ received from neighbor and $ChunkState[chunk] = REMOTE$ then
4:             mirror $chunk$ locally
5:             $ChunkState[chunk] \leftarrow LOCAL$
6:             $unmatched \leftarrow unmatched + 1$
7:         end if
8:         if $HintQueue \neq \emptyset$ then
9:             $chunk \leftarrow$ POP_FRONT($HintQueue$)
10:            if $ChunkState[chunk] \neq WRITTEN$ then
11:                send $chunk$ to all neighbors
12:            end if
13:        end if
14:    end while
15: end procedure
```

Finally, the preventive chunk exchange scheme is performed asynchronously inside BACKGROUND_EXCHANGE, detailed in Algorithm 3. In a nutshell, it listens for gossips about new chunks from all its neighbors and whenever it receives one that corresponds to a chunk that is missing locally, it fetches that chunk and mirrors it locally,

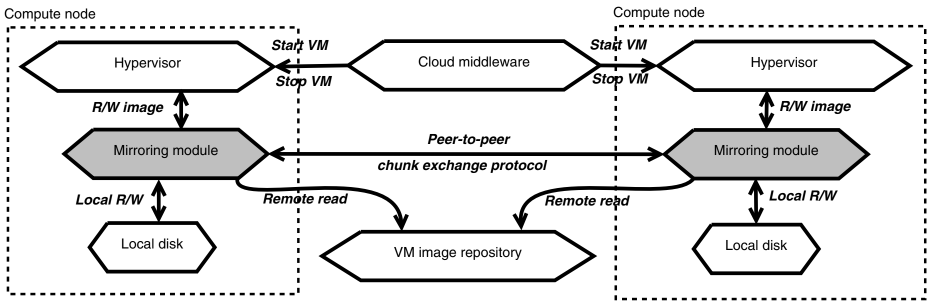


Fig. 1. Cloud architecture that integrates our approach (dark background)

incrementing the number of unmatched chunks. At the same time, if the READ operation enqueued any hints about on-demand chunks inside $HintQueue$ and the corresponding chunks were not overwritten in the mean time, then it informs all its neighbors about these new chunks.

Note that we opted for an optimistic scheme based on push, which improves latency compared to sending just the chunk id and waiting for a pull request. This technique favors the setting we explored in this work: small neighborhoods with a simple ring topology (See Section 4). However, with increasing neighborhood size it is more likely that a VM receives the same chunk from multiple sources, thus a push approach might unnecessarily waste bandwidth and create more overhead. Nevertheless, our algorithms require minimal changes to support a pull approach.

3.3 Architecture

We depict a simplified IaaS cloud architecture that integrates our approach in Figure 1. For better clarity, the building blocks that correspond to our own approach are emphasized with a darker background.

The *VM image repository* is the storage service responsible to hold the VM disk image templates used as the source of multi-deployments. The only requirement for the VM image repository is to be able to support random-access remote reads, which gives our approach high versatility to adapt to a wide range of options: centralized approaches (e.g., NFS server), parallel filesystems or other dedicated services that specifically target VM storage and management [16, 21].

The *cloud client* has direct access to the VM image repository and is allowed to upload and download VM images from it. Furthermore, the cloud client also interacts with the *cloud middleware* through a control API that enables launching and terminating multi-deployments. In its turn, the cloud middleware will interact with the *hypervisors* deployed on the compute node to instantiate the VM instances that are part of the multi-deployment.

Each *hypervisor* interacts with the local mirror of the VM disk image as if it were a full local copy of the VM disk image template. To facilitate this behavior, the *mirroring module* acts as a proxy that traps all reads and writes of the hypervisor and takes the appropriate action: it populates the local mirror on-demand only in a copy-on-reference fashion while using the peer-to-peer chunk exchange protocol described in Section 3.1 to pre-populate regions that are likely to be accessed in the future based on the collective access pattern trend.

3.4 Implementation

We implemented the mirroring module as file system in userspace on top of FUSE [22]. This has several advantages in our context: (1) it is transparent to the hypervisor (and thus portable); (2) it enables easy interfacing with any remote storage repository (since it is a userspace implementation) and (3) it is easy to integrate into existing cloud middleware, as it enables us to emulate a behavior that is functionally equivalent to pre-broadcast.

To facilitate efficient on-demand access and copy-on-write support at kernel level, we map the remote VM disk image template into the memory of the host using the mmap system call. Since the kernel automatically manages memory page faulting and implicitly fetches any missing remote content, reads and writes are greatly optimized as they effectively translate to simple memory copy operations. Similarly, any chunks that were obtained through the preventive chunk exchange scheme can be mirrored locally again by simple memory copy operations.

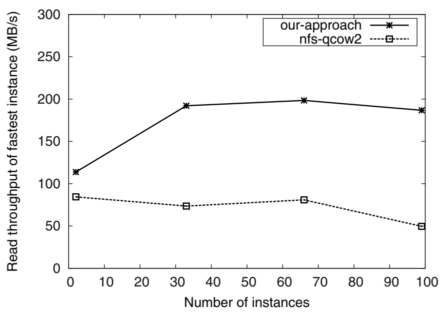
The preventive copy-on-write chunk exchange scheme runs in its own thread, which communicates with the main FUSE thread through the data structures presented in Section 3.2. The communication between the mirroring modules is implemented on top of Boost ASIO [23], a high performance asynchronous event-driven library which is part of the Boost C++ collection of libraries. Since the preventive peer-to-peer exchange scheme is not a pre-condition for correctness (i.e. our approach works even when no exchange is happening), we have opted for a lightweight solution that performs gossiping through UDP sockets. This has the potential to significantly reduce networking overhead at the cost of unreliable communication, which is a perfectly acceptable trade-off in our case, as we can afford to occasionally lose hints about chunks.

4 Evaluation

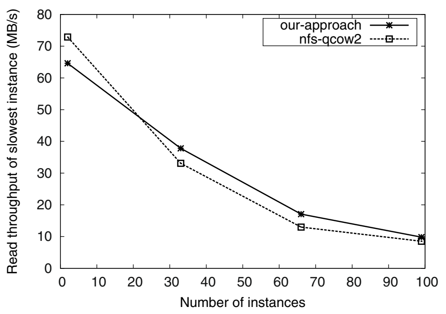
This section evaluates the scalability of our approach experimentally for a series of multi-deployment scenarios.

4.1 Experimental Setup

The experimental platform used to run our experiments is Grid'5000 [8]. For the purpose of this work, We reserved 100 nodes of the graphene cluster. The nodes are outfitted with x86_64 CPUs offering hardware support for virtualization, local disk storage of 277 GB (access speed $\simeq$55 MB/s) and 16 GB of RAM. The nodes are interconnected with Gigabit Ethernet (measured: 117.5 MB/s for TCP sockets with MTU = 1500 B with a latency of $\simeq$0.1 ms). The hypervisor running on all compute nodes is QEMU/KVM 1.2.0, while the operating system is a recent Debian Sid Linux distribution. For all experiments, a 4 GB raw disk image file based on the same Debian Sid distribution was used.



(a) Overall throughput of the fastest instance in the VM group (higher is better)



(b) Overall throughput of the slowest instance in the VM group (higher is better)

Fig. 2. Scalability of multi-deployments: throughput under concurrent on-demand access when reading 512 MB of the virtual disk using dd

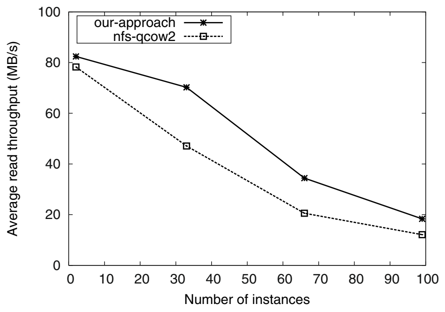
4.2 Methodology

We compare our approach (presented in Section 3.4) to the most widely used on-demand access technique in practice: local copy-on-write files that are derived from a shared backing file that is stored remotely. To enable copy-on-write, we rely on the *QCOW2* [5] image format, which is part of standard QEMU/KVM distribution. The backing file is shared through a NFS server. In order to deploy a VM instance, we create a fresh qcow2-derived file on the local disk of the compute node where the hypervisor is running and use this file as the VM disk image. For the rest of this paper, we denote this approach nfs−qcow2. Our approach uses the same backing file for local mirroring. With respect to the peer-to-peer topology, we opted for a ring: each mirroring module is linked to the mirroring module that is deployed on next compute node in a predefined ordering of all compute nodes. The $Threshold$ was fixed at 128, while the chunk size is fixed at 32 KB. We denote this setting our−approach.

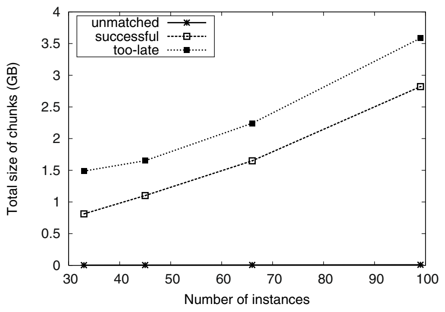
The experiments consist in deploying an increasing number of VM instances concurrently, each on a dedicated compute node. Each VM instance boots and then reads the first 512 MB of its virtual disk (dd if=/dev/sda of=/dev/null count=1M). This simulates a read-intensive data access pattern (e.g. as sweeping through an input file) that is exhibited by all members of the multi-deployment in parallel and that generates high contention to the remote repository. We record the throughput of dd for every VM instance, as well as various other statistics gathered during the runtime of our−approach that relate to its internal workings.

4.3 Results

Figure 2 depicts the throughputs of the fastest and respectively the slowest VM instance for an increasing multi-deployment size. As can be observed, the fastest instance in the case of nfs−qcow2 experiences a significant drop in throughput with increasing number of VM instances (Figure 2(a)). This is expected because of increasing I/O pressure on



(a) Average throughput of the VM multi-deployment group as a whole (higher is better)



(b) Statistics about chunk accesses for our−approach

Fig. 3. Scalability of multi-deployments: average throughput and aggregated statistics under concurrent on-demand access when reading 512 MB of the virtual disk using dd

the NFS server. Our approach on the other hand achieves the exact opposite: not only does it start better (showing that preventive chunk exchange successfully avoids access to the NFS server even in a group of 2), but it also exhibits a dramatic increase in throughput as the number of VM instances is growing (showing that more VMs provide better chunk exchange opportunities). At the other extreme, even the slowest instance (Figure 2(b)) benefits from chunk exchange, albeit at lesser extent.

To put these results in perspective, Figure 3(a) depicts the average throughput achieved by the VM multi-deployment group as a whole. As expected, the increasing I/O pressure on the NFS server leads to a noticeable drop in both cases. However, when increasing the multi-deployment size beyond two, a stable gain of at least 30% more throughput is noticeable for our−approach when compared to nfs−qcow2.

To better understand the contribution of preventive chunk exchange to this result, we illustrate in Figure 3(b) the total size corresponding to how many chunks were unmatched (i.e., exchanged but never needed for a read), successful (i.e., exchanged and later contributed to avoid a remote NFS access) and too−late (i.e., exchanged but arrived too late to avoid a remote NFS access during an on-demand read). It can be observed that due to high similarity in the access pattern, almost all chunks are matched to a later read. This is observable by inspecting unmatched, which maintains a negligible level (less than $\simeq$10 MB) even for a large 100 node multi-deployment. Combined with a steady increasing trend for successful, this effectively explains the why the average throughput has a steady gain itself. However, it can also be noted that too−late is larger than successful, which shows that under high read pressure (as is our case) there is a high chance that a chunk is needed before it can be exchanged.

5 Conclusions

This paper introduced a novel multi-deployment technique based on augmented on-demand remote access to the VM disk image template. Being on-demand, it avoids an expensive full pre-broadcast, while at the same time it pioneers the idea of exchanging chunks between multi-deployment members on-the-fly, in an effort to anticipate and prevent bottlenecks due to concurrent access to the remote repository where the VM disk image template is stored.

Our scheme is highly scalable, maintaining on the average a steady 30-40% improvement in read throughput compared to simple on-demand schemes in which the members of the multi-deployment are independent of each other. This is possible thanks to jitter between the VM instances, which enables the faster instances to effectively forward their chunks to slower instances in order to help them out. The results of this effect are dramatic: some instances become up to 4x faster compared to simple on-demand access.

Thanks to these encouraging results, we plan to further investigate the potential benefits of collaborative chunk exchange. In particular, we experienced a high number of chunks that arrived too late to be of use and thus an interesting direction to explore is how to avoid such chunks. Furthermore, as discussed in Section 3.2, we did not explore how to choose the optimal neighborhood size / topology and whether a pull scheme (i.e. push only chunk id as hint to others and prefetch chunk contents from others) might work better under the right circumstances. We plan to perform a deeper analysis in these areas.

Acknowledgments. The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see `http://www.grid5000.fr/`).

References

1. Amazon: Amazon Elastic Compute Cloud (EC2), `http://aws.amazon.com/ec2/`
2. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: SoCC 2012: Proceedings of the 3rd ACM Symposium on Cloud Computing, pp. 1–7. ACM, San Jose (2012)
3. Andrzejak, A., Kondo, D., Yi, S.: Decision model for cloud computing under sla constraints. In: MASCOTS 2010: Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 257–266. IEEE Computer Society, Washington (2010)
4. Wartel, R., Cass, T., Moreira, B., Roche, E., Guijarro, M., Goasguen, S., Schwickerath, U.: Image distribution mechanisms in large scale cloud providers. In: CloudCom 2010: Proceedings of the 2nd IEEE Second International Conference on Cloud Computing Technology and Science, pp. 112–117. IEEE Computer Society, Indianapolis (2010)
5. Gagné, M.: Cooking with linux: still searching for the ultimate linux distro? Linux J. (161), 9 (2007)
6. Chen, H., Kim, M., Zhang, Z., Lei, H.: Empirical study of application runtime performance using on-demand streaming virtual disks in the cloud. In: MIDDLEWARE 2012: Proceedings of the 13th ACM/IFIP/USENIX International Middleware Conference (Industrial Track), Montreal, Canada, pp. 5:1–5:6 (2012)
7. Nicolae, B., Cappello, F., Antoniu, G.: Optimizing multi-deployment on clouds by means of self-adaptive prefetching. In: Euro-Par 2011: 17th International Euro-Par Conference on Parallel Processing, Bordeaux, France, pp. 503–513 (2011)

8. Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.-G., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. Int. J. High Perform. Comput. Appl. 20, 481–494 (2006)
9. Hibler, M., Stoller, L., Lepreau, J., Ricci, R., Barb, C.: Fast, scalable disk imaging with frisbee. In: Proc. of the 2003 USENIX Annual Technical Conference, San Antonio, USA, pp. 283–296 (2003)
10. SCPWave, `http://code.google.com/p/scp-wave/`
11. Schmidt, M., Fallenbeck, N., Smith, M., Freisleben, B.: Efficient distribution of virtual machines for cloud computing. In: PDP 2010: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, pp. 567–574. IEEE Computer Society, Washington, DC (2010)
12. Robison, N.A., Hacker, T.J.: Comparison of vm deployment methods for hpc education. In: RIIT 2012: Proceedings of the 1st Annual conference on Research in Information Technology, pp. 43–48. ACM, Calgary (2012)
13. Carns, P.H., Ligon, W.B., Ross, R.B., Thakur, R.: Pvfs: A parallel file system for Linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, pp. 317–327. USENIX Association, Atlanta (2000)
14. Schmuck, F., Haskin, R.: Gpfs: A shared-disk file system for large computing clusters. In: FAST 2002: Proceedings of the 1st USENIX Conference on File and Storage Technologies, USENIX Association, Berkeley (2002)
15. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: OSDI 2006: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 307–320. USENIX Association, Berkeley (2006)
16. Nicolae, B., Bresnahan, J., Keahey, K., Antoniu, G.: Going back and forth: Efficient multi-deployment and multi-snapshotting on clouds. In: HPDC 2011: 20th International ACM Symposium on High-Performance Parallel and Distributed Computing, San José, USA, pp. 147–158 (2011)
17. Zhang, Y., Li, Y., Zheng, W.: Automatic software deployment using user-level virtualization for cloud-computing. Future Gener. Comput. Syst. 29(1), 323–329 (2013)
18. De, P., Gupta, M., Soni, M., Thatte, A.: Caching vm instances for fast vm provisioning: a comparative evaluation. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 325–336. Springer, Heidelberg (2012)
19. Jin, K., Miller, E.L.: The effectiveness of deduplication on virtual machine disk images. In: SYSTOR 2009: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference, pp. 7:1–7:12. ACM, Haifa (2009)
20. Theimer, M.M., Lantz, K.A., Cheriton, D.R.: Preemptable remote execution facilities for the v-system. In: SOSP 1985: Proceedings of the Tenth ACM Symposium on Operating Systems Principles, pp. 2–12. ACM, New York (1985)
21. Hansen, J.G., Jul, E.: Scalable virtual machine storage using local disks. SIGOPS Oper. Syst. Rev. 44, 71–79 (2010)
22. File System in UserspacE (FUSE), `http://fuse.sourceforge.net/`
23. The Boost C++ collection of libraries, `http://www.boost.org/`

Energy and Carbon-Efficient Placement of Virtual Machines in Distributed Cloud Data Centers

Atefeh Khosravi, Saurabh Kumar Garg[*], and Rajkumar Buyya

Cloud Computing and **D**istributed **S**ystems (CLOUDS) Laboratory
Department of Computing and Information Systems
The University of Melbourne, Australia
atefehk@student.unimelb.edu.au, {sgarg,raj}@csse.unimelb.edu.au

Abstract. Due to the increasing use of Cloud computing services and the amount of energy used by data centers, there is a growing interest in reducing energy consumption and carbon footprint of data centers. Cloud data centers use virtualization technology to host multiple virtual machines (VMs) on a single physical server. By applying efficient VM placement algorithms, Cloud providers are able to enhance energy efficiency and reduce carbon footprint. Previous works have focused on reducing the energy used within a single or multiple data centers without considering their energy sources and Power Usage Effectiveness (PUE). In contrast, this paper proposes a novel VM placement algorithm to increase the environmental sustainability by taking into account distributed data centers with different carbon footprint rates and PUEs. Simulation results show that the proposed algorithm reduces the CO_2 emission and power consumption, while it maintains the same level of quality of service compared to other competitive algorithms.

Keywords: Cloud computing, Data center, Energy efficiency, Carbon footprint, Virtual machine placement.

1 Introduction

The information and communication technology industry (ICT) consumes an increasing amount of energy and most of it is consumed by data centers [8]. A major consequence of this amount of energy consumption by data centers is a significant increase in ecosystem carbon level. According to Gartner, the ICT industry produces 2% of global CO_2 emission, which places it on par with the aviation industry [22]. Therefore, reducing even a small fraction of the energy consumption in ICT, results in considerable savings in financial and carbon emission of the ecosystem.

Cloud computing offers a wide range of services and applications to its users. Three main services that Clouds provide are infrastructure, platform, and software as a service. Infrastructure as a service (IaaS) allows users to run their applications in form of virtual machines (VMs) on a shared infrastructure. Cloud

[*] Currently working in IBM Research, Melbourne, Australia.

data centers take advantage of virtualization technology [7] to share a physical server's resources among multiple VMs. Each VM has its own characteristics and depending on the resource usage, it consumes energy and leaves carbon footprint. By the arrival of each VM request, the Cloud manager selects the physical resource to instantiate the request. VM placement in Cloud computing system is a complex task and if cannot be done effectively, it leads to high energy usage and high carbon footprint.

Thus, wisely taking into account parameters that affect VM placement and physical server selection result in less energy consumption and less carbon footprint. Distributed Cloud data centers, alongside with bringing high availability and disaster recovery, provide the opportunity to have different energy sources. *Carbon footprint rate* of energy sources is an important parameter, since data centers use electricity driven by these sources to run VMs. By having different energy sources in different data center sites or within a data center site, Cloud providers should increase the use of more clean and off-grid renewable energies [24]. *Power usage effectiveness (PUE)* is coined by the Green Grid consortium [14] and indicates the energy efficiency of a data center. PUE is a ratio of total power consumed by the data center to its power consumed by IT devices. Providers can consider PUE as a parameter to perform VM placement among different data center sites. *Proportional power* is another parameter that can be taken into account for VM placement. Server proportional power has a cubic relation with CPU frequency [17]. Therefore, considering the increase in CPU frequency, which is related to increase in CPU utilization upon new request arrival, will have a great impact on the amount of energy consumption in data centers.

This paper proposes a VM placement algorithm by considering distributed Cloud data centers with the objective of minimizing carbon footprint. Our proposed Cloud computing system, Energy and Carbon-Efficient (ECE) Cloud architecture, benefits from distributed Cloud data centers with different carbon footprint rates, PUE value, and different physical servers' proportional power. ECE Cloud architecture places VM requests in the best suited data center site and physical server. The main contributions of this paper are: an Energy and Carbon-Efficient Cloud architecture, based on distributed Cloud data centers; an efficient VM placement algorithm that integrates energy efficiency and carbon footprint parameters; a comprehensive comparison on carbon footprint and power consumption for different VM placement algorithms with respect to quality of service (number of rejected VMs).

The reminder of the paper is organised as follows. In Section 2 the related work is discussed. Section 3 presents the proposed Cloud architecture with its components, VM placement algorithm, and formulates the objective. The performance evaluation results and the experimental environment are presented in Section 4. Section 5 concludes the paper and presents future works.

2 Related Work

There is a growing body of literature that aims to reduce the amount of carbon dioxide of Cloud services in data centers. Most of the works in this area focus on reducing the energy consumption in a single data center or considering the data center hardware aspects [6] [5]. Well-known technologies that data centers benefit from by applying virtualization technology [7] are VMs migration [15] and consolidation [23]. The main problem with migration and consolidation is that they are complex and, due to the need for resuming and suspending VMs cause overhead to the system [10]. Moreover, these technologies act reactive whereas applying preventive technologies are more efficient.

As idle servers consume almost half of the power when they are in the peak power state [4], work by Lin et al. [18] uses a dynamic right-sizing on-line algorithm to predict the number of active servers that are needed for the arriving workload to the data center. Based on their experiments, dynamic right-sizing can achieve significant energy savings in the data center, but it requires servers to have different power levels and be able to transit to different states. A similar work done by Lefevre et al. [16] proposes Green Open Cloud (GOC) architecture, with advance resource reservation for users to improve the prediction of the arrival requests.

The above mentioned technologies are adopted within a data center and intend to reduce the energy consumption, whilst they do not particularly consider carbon emission. Reducing data center energy consumption does not necessarily lead to reduce in carbon footprint. Works by Aksanli et al. [3] and Goiri et al. [12] consider the availability of both non-polluting (green) and polluting (brown) energy sources in a single data center. They use prediction-based scheduling algorithms to increase usage of green energy sources.

Liu et al. [19] consider reducing the carbon footprint of data centers by considering multiple data center sites. They proposed an algorithm to efficiently use the renewable energies, such as wind and solar, in different places. This algorithm uses the idea of geographic diversity of data center sites and unpredictability of renewable energies to find the optimal percentage of wind/solar energies in order to reduce the brown energy consumption. Garg et al. [11] also consider reducing carbon footprint of Cloud data center sites. They proposed a novel carbon-aware green Cloud architecture, which uses two directories for Cloud providers to register their offered services.

Our work is different from the previous works, since we address the problem of increase in carbon footprint of the Cloud data centers by performing efficient VM placement. Our proposed method accommodates VM requests by considering distributed data center sites of a Cloud provider, with various energy sources and carbon footprint rates. Moreover, we consider data centers' PUE, physical servers' proportional power usage, and user VM requests of different types. Finally, we present an energy and carbon-efficient algorithm that uses two level decision making for VM placement.

3 System Model

In this section, Energy and Carbon-Efficient (ECE) Cloud architecture is described. This architecture assures system quality of service, while minimizing the Cloud carbon footprint by applying an energy and carbon-efficient heuristic for VM placement.

3.1 ECE Cloud Architecture

The proposed architecture is represented in Figure 1. The system consists of the following components and symbols used in this paper are presented in Table 1:

Table 1. Description of Symbols

Symbol	Description	Symbol	Description
d	Number of Data Center Sites	P	Proportional Power
c	Number of Clusters at each Data Center	P_{fixed}	Server Power Consumption in Idle State
h	Number of Hosts at each Cluster	P_f	Server Power Consumption at Frequency f
cf	Data Center/Cluster Carbon Footprint Rate	f_u	CPU Operating Frequency at Utilization u
CF	Cloud Total Carbon Footprint	$type$	Virtual Machine Instance Type
ht	Virtual Machine Holding Time	$core$, pu	CPU Cores and Total Processing Unit
ram,$storage$	RAM and Storage	bw	Network Bandwidth

Cloud Users: Cloud Users send their VM requests based on predefined requirements to the Cloud provider. Virtual machine types and configurations are inspired by Amazon Elastic Compute Cloud (EC2) [1]. The expected requirements for each VM are specified by its predefined configurations in terms of required number of cores, processing unit of each core, storage, RAM, and network bandwidth. In addition, holding time of a VM depends on the application runs on that VM. We consider two types of applications in this paper: bag-of-tasks and web-requests. Every requested VM by users has the following requirements: $(ht, type)$, where each $type$ consists of the following components: $\{cores, pu, ram, storage, bw\}$. Cloud computing system load at time t, according to the running VMs, is represented as:

$$load = \sum_{i=1}^{d} \sum_{j=1}^{c} \sum_{k=1}^{h} vm_{(i,j,k,t)}.$$

Cloud Provider: A Cloud provider has several geographically distributed data center sites. Each data center is composed of several clusters with various heterogeneous physical servers. Physical servers are characterised by CPU cores, CPU processing unit, amount of RAM, storage, and network bandwidth. In addition to the physical servers configuration, each data center has its own energy-related parameters, shown by PUE and proportional power. Moreover, each data center can have one or more energy sources with different carbon footprint rates.

ECE Cloud Information Service: Each data center site registers its characteristics in the ECE Cloud information service (ECE-CIS) and they keep their information updated. This information includes available physical resources and

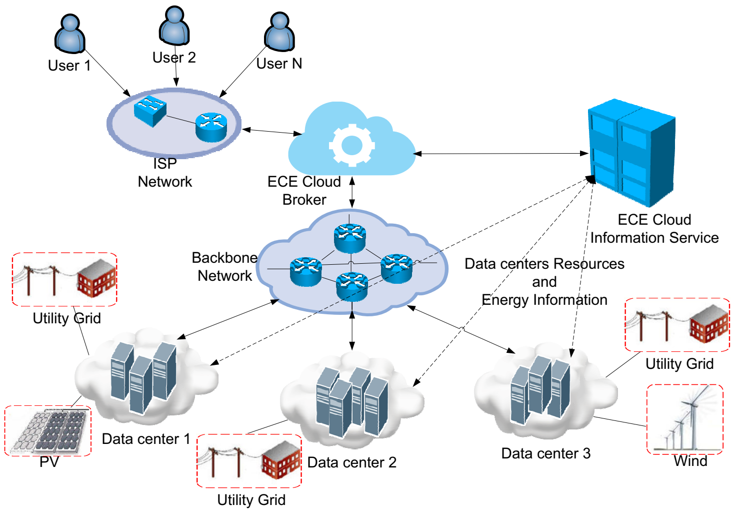


Fig. 1. Energy and Carbon-Efficient (ECE) Cloud Architecture

energy related parameters; such as data center PUE, energy source(s), carbon footprint rate, and physical servers' current utilization and power consumption. Cloud broker uses this information to perform ECE VM placement in Cloud computing environment.

ECE Cloud Broker: ECE Cloud Broker is the Cloud provider's interface with Cloud users. It receives user requests and schedules them based on their predefined requirements. Despite users request scheduling, broker should also ensure energy efficient data centers with minimum carbon footprint for Cloud providers.

Resources on the Cloud provider are physical servers in the clusters within each data center. The broker receives the current status of data centers' physical resources and their energy information from ECE-CIS, and based on this information, assigns the VM to a physical server in a data center site. Based on [25], in today's Internet and core networks design, average number of hops a packet passes from source to destination is between 12-14 hops. Therefore, we can have data center site selection without considering network distance; especially for sites that are located in a region, such as different states in USA, as we considered in this paper.

3.2 Placement Decision

As stated before, the broker makes the placement decision based on the data centers' power usage effectiveness (PUE), energy sources carbon footprint rate, and proportional power.

The PUE indicates the energy efficiency of a data center and is a metric to compare different data center designs in terms of electricity consumption. Data center's PUE is calculated as:

$$PUE_i = \frac{Datacenter_i TotalPowerConsumption}{Datacenter_i ITDevicesPowerConsumption},$$

where the total power consumption is sum of power drawn by cooling, lightening, and IT equipment. PUE is a value larger than 1 ($PUE \geq 1$). PUE of 1.0 means 100% of the data center's electricity goes to the IT part and is ideal for any data center, but is unattainable pragmatically. In other words, the smaller the PUE, the more energy efficient the data center.

Data center proportional power is the next important metric in physical server selection. According to the experiments by Lien et al. [17] server's power consumption depends on the system base power and the CPU frequency, and the CPU frequency itself depends on the CPU utilization. The data center proportional power, also known as dynamic power, is calculated as: $P = P_{fixed} + P_f \times f_u^3$. The power consumption for a VM on physical server k in cluster j of data center i at time t is modeled as: $P(vm_{(i,j,k,t)})$.

According to the above mentioned metrics the objective is to minimize total carbon footprint of the Cloud provider, CF, for time interval $[0, T]$, and is computed as follows:

$$CF = \sum_{t=1}^{T} \sum_{i=1}^{d} (PUE_i \times \sum_{j=1}^{c} (cf_j \times \sum_{k=1}^{h} (P(vm_{(i,j,k,t)}) \times ht))),$$

subject to following constraints:

$$\sum_{i=1}^{d} \sum_{j=1}^{c} \sum_{k=1}^{h} vm_{(i,j,k)}^{core} \leq host_{(i,j,k)}^{core}, \qquad \sum_{i=1}^{d} \sum_{j=1}^{c} \sum_{k=1}^{h} vm_{(i,j,k)}^{pu} \leq host_{(i,j,k)}^{pu},$$

$$\sum_{i=1}^{d} \sum_{j=1}^{c} \sum_{k=1}^{h} vm_{(i,j,k)}^{ram} \leq host_{(i,j,k)}^{ram}, \qquad \sum_{i=1}^{d} \sum_{j=1}^{c} \sum_{k=1}^{h} vm_{(i,j,k)}^{storage} \leq host_{(i,j,k)}^{storage}.$$

The above mentioned constraints ensure that allocated resources to the VMs on a physical server do not exceed the total capacity of the server.

3.3 Energy and Carbon-Efficient (ECE) Heuristic for VM Placement

By the arrival of each VM request the broker has $(d \times c \times h)$ different VM placement options. The VM placement problem can be seen as a bin-packing problem with different bin sizes (physical servers). Therefore, we propose the Energy and Carbon-Efficient (ECE) VM placement algorithm (Algorithm 1), which is a derivation of the best-fit heuristic to place the VMs in the data center, cluster, and host with the minimum carbon footprint, PUE, and minimum increase in physical server's power consumption.

Algorithm 1. Energy and Carbon-Efficient (ECE) VM Placement Algorithm

```
Input: $datacenerList$, $clusterList$, $hostList$
Output: $destination$
1  while $vmRequest$ do
2      Get data centers' Information from ECE-CIS;
3      foreach $datacenter$ in $datacenterList$ do
4          Add $clusterList$ into $aggregateClusterList$;
5      Sort $aggregateClusterList$ in an ascending order of $(PUE \times cf)$;
6      foreach $cluster$ in $aggregateClusterList$ do
7          foreach $host$ in $hostList$ do
8              $P_1 \leftarrow$ Get current $hostDynamicPower$;
9              $P_2 \leftarrow$ Calculate $hostDynamicPower$ after initiating the $vm$;
10             $\triangle P \leftarrow P_2 - P_1$;
11         Sort $hostList$ in an ascending order of $\triangle P$;
12         foreach $host$ in $hostList$ do
13             if $host$ is suitable for $vm$ then
14                 $destination \leftarrow (datacenter, cluster, host)$;
15                 return $destination$;
16     $destination \leftarrow null$; //rejection of request;
17     return $destination$;
```

The broker receives a VM request and selects the best physical server for the VM. Its objective is to minimize the data centers' carbon footprint and accordingly power consumption. Therefore, ECE placement algorithm gets data centers' resources and energy status from ECE-CIS, upon the arrival of a new VM request (Line 2). According to the received information, ECE adds the clusters of all the data centers into an aggregated cluster list (Lines 3-4), and sorts the new list based on the minimum $(PUE \times cf)$ (Line 5). By receiving the data centers and clusters status, ECE calculates the amount of power consumption that will be added to each host after initiating the new VM (Lines 8-10). Afterwards, ECE sorts the hosts list based on the estimated $\triangle P$ (Line 11), and if the host has enough resources for the VM (Line 13), it submits the VM to the selected data center, cluster, and host.

In order to show the time complexity of Algorithm 1, we consider v VM requests. Line 3-4 take $O(d)$, and the sort function at Line 5 can be done in $O(dc\log(dc))$. Lines 7-9 need $O(h)$ time, and the sort function for hosts at Line 11 needs $O(h\log(h))$ to be done. Lines 12-15 take $O(h)$, in the worst case. Thus, the total running time of the algorithm is $O(v(d+dc\log(dc)+dc(h+h\log(h)+h)))$. Since there are small number of data center sites and clusters (dc) for a Cloud provider, the complexity of this algorithm is dominated by the number of VM requests and hosts sort function. The total time complexity of the algorithm is $O(vdch\log(h)))$.

4 Performance Evaluation

We use the CloudSim toolkit [9] to evaluate the Cloud computing virtualized environment. We have extended CloudSim to enable energy and carbon-efficient VM placement simulations. Apart from being aware of data center's PUE, carbon footprint rate, and dynamic power, the extended package has the ability to simulate dynamic VM requests with different instance types.

In order to evaluate the proposed algorithm, we modeled an IaaS provider with 4 data center sites, and each site with 90 heterogeneous physical servers. Each data center has a unique PUE value and 2 clusters with different carbon footprint rates. Table 2 shows data centers' PUE value and carbon footprint rate for different group of clusters. The PUE value is based on the work by Greenberg et al. [13], and is in the range [1.56, 2.1]. Data centers' carbon footprint rates, the last column of Table 2, are extracted based on the information from US Department of Energy, Appendix F, Electricity Emission Factors [2]. In this simulation, we use 5 different physical servers whose characteristics are given in Table 3. According to the linear relationship between CPU utilization and frequency, and the cubic relation between CPU frequency and system proportional power, the following is the power models for the platforms:

$$\text{CPU Frequency(in GHz)}: \{f(u) : (1.4, 1.57, 1.74, 1.91, 2.08, 2.25, 2.42, 2.6, 2.77, 2.94, 3.11)\}$$
$$\text{Power Model1(in Watt)}: \{P_f : (60, 63, 66.8, 71.3, 76.8, 83.2, 90.7, 100, 111.5, 125.4, 140.7)\}$$
$$\text{Power Model2(in Watt)}: \{P_f : (41.6, 46.7, 52.3, 57.9, 65.4, 73, 80.7, 89.5, 99.6, 105, 113)\}$$

VM characteristics are inspired by Amazon EC2 instance types given in Table 4. The physical resources to the VMs are allocated based on the VM resource requirements and all the VMs are considered to perform at the maximum utilization during their lifetime. The VM type and the number of VMs requested by users depend on the user type (bag-of-tasks or web-requests), and are based on the related probabilities. The VM type related probability is shown in the last column of Table 4 and is derived from the work by Mills et al. [21].

In order to generate the workload, we need VM requests arrival rate and holding time. The Lublin-Feitelson [20] workload model is employed to generate the bag-of-tasks VM requests. We take benefit of Lublin to set arrival request parameters, including simulation duration, number of requests, requests arrival time, and request holding time. We consider each generated request in Lublin as a VM request. In order to generate VMs with longer holding time, we increased the first parameter of the Gamma distribution and left other Lublin parameters with their default value. To generate the web-requests, we use the same arrival time model of bag-of-tasks requests, and for the holding time we use a hyper gamma distribution with expectation value 73 and variance 165. For both workloads, we omit 5% of created requests at the start (warm-up period) and end (cool-down period) of the simulation to get a steady environment. We apply 240-hour long workload with different number of requests. Finally, for the purpose of accuracy, each experiment is repeated 30 times and the mean is reported for measured values for experimental results.

Table 2. Data Centers Characteristics

Data Center Site	**PUE**	**Carbon Footprint Rate** (Tons/MWh)
DC1 -Oregan, USA	1.56	0.124, 0.147
DC2 -California, USA	1.7	0.350, 0.658
DC3 -Virginia, USA	1.9	0.466, 0.782
DC4 -Dallas, USA	2.1	0.678, 0.730

Table 3. Platform Types Characteristics

Platform Type	**Number of Cores**	**Core Speed** (GHz)	**Memory** (GB)	**Storage** (GB)	**Network Bandwidth** (Mbps)	**Bits**	**Power Model**
Platform1	2	2	16	2000	1000	B32	PowerModel1
Platform2	4	4	32	6000	1000	B64	PowerModel1
Platform3	8	4	32	7000	2000	B64	PowerModel2
Platform4	8	8	64	7000	4000	B64	PowerModel2
Platform5	8	16	128	9000	4000	B64	PowerModel2

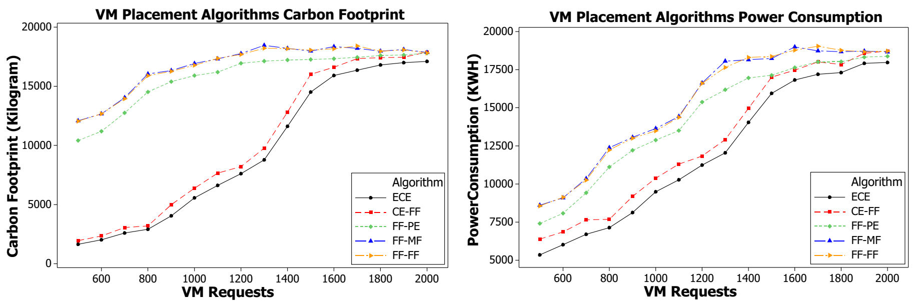
4.1 Results

We use the described workload data to compare the proposed VM placement algorithm with respect to carbon and power efficiency with four competing algorithms. The first algorithm is a version of ECE, that its data center and cluster selection is same as ECE, and uses first-fit bin-packing for host selection. We refer to this algorithm as Carbon-Efficient First-Fit (CE-FF). The other group of algorithms are three bin-packing heuristics that use first-fit heuristic for data center/cluster selection, without considering carbon footprint parameters. First-Fit Power-Efficient (FF-PE) uses power-efficient policy for host selection (same as ECE host selection). First-Fit Most-Full First (FF-MF) selects the physical server with least available resources. Finally, the last algorithm uses first-fit heuristic for data center, cluster, and host selection (FF-FF).

Figure 2a illustrates the carbon footprint of ECE in comparison with other placement algorithms under different number of VM requests. Based on the experiments, as the number of VMs increases, the system utilization increases as well to the point that system performs with highest utilization and reaches to the saturation point. Therefore, increase in system load leads to increase in the total carbon footprint in data centers. Based on the Figure 2a, ECE in comparison to CE-FF (carbon-efficient) and other heuristics (non carbon-efficient) reduces carbon footprint with an average of 10% and 45% respectively. The same behaviour can be seen for the data centers' power consumption in Figure 2b. The ECE algorithm has lower power consumption in comparison to the other algorithms and consumes on average 8% and 20% less power than CE-FF and other heuristics placement algorithms respectively. Considering the differences between algorithms behaviour in both figures, we can infer that just considering power-efficient parameters is not enough to reduce the data centers' carbon footprint. However, taking into consideration data centers' energy and carbon rate parameters, at the same time, leads to significant reduction in terms of Cloud computing system carbon footprint and consumed power.

Table 4. VM Types and Simulated User Types; (Bag-of-Task Users (BT) and Web-Request Users (WR))

VM Type		Number of Cores	Core Speed (GHz)	Memory (MB)	Storage (GB)	Network Bandwidth (Mbps)	Bits	Probability and UserType
Standard Instances	M1Small	1	1	1740	160	500	B32	0.25-BT
	M1Large	2	4	7680	850	500	B64	0.12-WR 0.25-BT
	M1XLarge	4	8	15360	1690	1000	B64	0.08-WR
High Memory Instances	M2XLarge	2	6.5	17510	420	1000	B64	0.12-WR
	M22XLarge	4	13	35020	850	1000	B64	0.08-WR
High CPU Instances	C1Medium	2	5	1740	320	500	B32	0.1-BT



(a) Carbon Footprint (b) Power Consumption

Fig. 2. Comparison of ECE Algorithm with other VM Placement Algorithms

Table 5. SLA Violation for Different VM Placement Algorithms

VM Placement Algorithm	SLA Violation Under Different VM Requests					
	1000	1200	1400	1600	1800	2000
ECE	0.0%	0.05%	0.4%	2.9%	8.6%	13.0%
CE-FF	0.0%	0.0%	0.3%	0.7%	6.0%	11.4%
FF-PE	0.0%	0.0%	0.3%	2.5%	9.4%	15.3%
FF-MF	0.0%	0.0%	0.2%	2.5%	9.7%	15.3%
FF-FF	0.0%	0.0%	0.1%	2.6%	9.7%	15.3%

Table 5 shows the SLA violation under different system loads for different VM placement algorithms. It shows that, the SLA violation (number of rejected VMs) under low system load for ECE is slightly higher than the other algorithms. However, by increasing system load, ECE will have lower SLA violation. Overall, all the VM placement algorithms have close values for violation, while ECE considerably reduces carbon footprint and power consumption.

5 Conclusion and Future Work

In this paper, the problem of VM placement to reduce Cloud computing energy consumption and carbon footprint is investigated. We used ECE Cloud information service (ECE-CIS), as part of next generation Cloud computing environment. ECE-CIS obtains energy and carbon related information from data centers and enables the broker to carry out carbon and power-efficient VM placement. We introduced the energy and carbon-efficient (ECE) VM placement algorithm, and compared it with a carbon-efficient algorithm (CE-FF) and three other heuristic algorithms (FF-PE, FF-MF, FF-FF). We performed the simulations by extending CloudSim and used different VM instance types with different holding times for the system workload. Based on the experiment results, ECE can on average save up to 10% and 45% carbon footprint in the ecosystem in comparison to CE-FF and three other heuristics respectively, while keeping SLA violation level as the same. Moreover, ECE reduces the power consumption in data centers by an average of 8% and 20% in comparison to CE-FF and other three algorithms respectively; which illustrates the importance of considering data centers' carbon footprint rate and PUE to reduce Cloud computing carbon footprint.

In the future, we plan to study the impact of different user applications and VM holding times on the VM placement policies. Moreover, we want to explore the effect of inter-data centers network distance and data locality on the Cloud computing system carbon footprint.

Acknowledgments. We would like to thank Amir Vahid Dastjerdi, Adel Nadjaran Toosi, Rodrigo N. Calheiros, and Deepak C Poola for their help and constructive suggestions.

References

1. Amazon EC2 instance types, `http://aws.amazon.com/ec2/instance-types/`
2. US Department of Energy, Appendix F, Electricity Emission Factors, `http://www.eia.doe.gov/oiaf/1605/pdf/Appendix`
3. Aksanli, B., Venkatesh, J., Zhang, L., Rosing, T.: Utilizing green energy prediction to schedule mixed batch and service jobs in data centers. In: Proc. of the 4th Workshop on Power-Aware Computing and Systems, pp. 5:1–5:5. ACM (2011)
4. Barroso, L., Holzle, U.: The case for energy-proportional computing. IEEE Computer 40(12), 33–37 (2007)
5. Beloglazov, A., Buyya, R., Lee, Y., Zomaya, A.: A taxonomy and survey of energy-efficient data centers and cloud computing systems. Advances in Computers 82(2), 47–111 (2011)
6. Berl, A., Gelenbe, E., Di Girolamo, M., Giuliani, G., De Meer, H., Dang, M., Pentikousis, K.: Energy-efficient cloud computing. The Computer Journal 53(7), 1045–1051 (2010)
7. Brey, T., Lamers, L.: Using virtualization to improve data center efficiency. The Green Grid, Whitepaper 19 (2009)
8. Brown, R., et al.: Report to congress on server and data center energy efficiency, pp. 109–431. Public law (2008)
9. Calheiros, R., Ranjan, R., Beloglazov, A., De Rose, C., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software: Practice and Experience 41(1), 23–50 (2011)

10. Clark, C., Fraser, K., Hand, S., Hansen, J., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: Proc. of the 2nd Conference on Symposium on Networked Systems Design & Implementation, vol. 2, pp. 273–286. USENIX Association (2005)
11. Garg, S., Yeo, C., Buyya, R.: Green cloud framework for improving carbon efficiency of clouds. In: Proc. of the 17th International Conference on Parallel Processing, pp. 491–502 (2011)
12. Goiri, Í., Beauchea, R., Le, K., Nguyen, T., Haque, M., Guitart, J., Torres, J., Bianchini, R.: Greenslot: scheduling energy consumption in green datacenters. In: Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 20:1–20:11. ACM (2011)
13. Greenberg, S., Mills, E., Tschudi, B., Rumsey, P., Myatt, B.: Best practices for data centers: Lessons learned from benchmarking 22 data centers. In: Proc. of the ACEEE Summer Study on Energy Efficiency in Buildings in Asilomar, CA. ACEEE, August 3, vol. 87, pp. 76–87 (2006)
14. Haas, J., Froedge, J., Pflueger, J., Azevedo, D.: Usage and public reporting guidelines for the green grids infrastructure metrics, pue/dcie (2009)
15. Harney, E., Goasguen, S., Martin, J., Murphy, M., Westall, M.: The efficacy of live virtual machine migrations over the internet. In: Proc. of the 2nd International Workshop on Virtualization Technology in Distributed Computing. ACM (2007)
16. Lefèvre, L., Orgerie, A.: Designing and evaluating an energy efficient cloud. The Journal of Supercomputing 51(3), 352–373 (2010)
17. Lien, C., Bai, Y., Lin, M.: Estimation by software for the power consumption of streaming-media servers. IEEE Transactions on Instrumentation and Measurement 56(5), 1859–1870 (2007)
18. Lin, M., Wierman, A., Andrew, L., Thereska, E.: Dynamic right-sizing for power-proportional data centers. In: Proc. of the IEEE INFOCOM, pp. 1098–1106. IEEE (2011)
19. Liu, Z., Lin, M., Wierman, A., Low, S., Andrew, L.: Geographical load balancing with renewables. ACM SIGMETRICS Performance Evaluation Review 39(3), 62–66 (2011)
20. Lublin, U., Feitelson, D.: The workload on parallel supercomputers: modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing 63(11), 1105–1122 (2003)
21. Mills, K., Filliben, J., Dabrowski, C.: Comparing vm-placement algorithms for on-demand clouds. In: Proc. of the Third International Conference on Cloud Computing Technology and Science, pp. 91–98. IEEE (2011)
22. Pettey, C.: Gartner estimates ict industry accounts for 2 percent of global co2 emissions (2007)
23. Srikantaiah, S., Kansal, A., Zhao, F.: Energy aware consolidation for cloud computing. In: Proc. of the 2008 Conference on Power Aware Computing and Systems, p. 10. USENIX Association (2008)
24. Stewart, C., Shen, K.: Some joules are more precious than others: Managing renewable energy in the datacenter. In: Proc. of the Workshop on Power Aware Computing and Systems (2009)
25. Van Mieghem, P.: Performance analysis of communications networks and systems. Cambridge University Press (2006)

Reconfiguration Stability of Adaptive Distributed Parallel Applications through a Cooperative Predictive Control Approach

Gabriele Mencagli, Marco Vanneschi, and Emanuele Vespa

Department of Computer Science
University of Pisa
Largo B. Pontecorvo, 3, I-56127, Pisa, Italy
{mencagli,vannesch,vespa}@di.unipi.it

Abstract. Distributed parallel applications executed on heterogeneous and dynamic environments need to adapt their configuration (in terms of parallelism degree and parallelism form for each component) in response to unpredictable factors related to the physical platform and the application semantics. On emerging Cloud computing scenarios, reconfigurations induce economic costs and performance degradations on the execution. In this context, it is of paramount importance to define smart adaptation strategies able to achieve properties like control optimality (optimizing the application global QoS) and reconfiguration stability, expressed in terms of number of reconfigurations and the average time for which a configuration is not modified. In this paper we introduce a methodology to address this issue, based on Control Theory and Optimal Control foundations. We present a first validation of our approach in a simulation environment, outlining its effectiveness and feasibility.

Keywords: Distributed Parallel Computations, Reconfigurations, Autonomic Computing, Model-based Predictive Control.

1 Introduction

With the emergence of computational paradigms like Grid and Cloud Computing, properties like reconfigurability and ***adaptiveness*** have gained more importance [1,2]. In scenarios characterized by variable workload conditions and dynamic execution environments the achievement of the desired *Quality of Service* (QoS) requires to adapt the application configuration expressed in terms of component identification, mapping onto physical resources, and proper selection of a parallelism degree and a parallelism form (e.g. farming and data-parallelism paradigms) for each component [3]. Such choices need to be applied to computations by performing run-time *reconfiguration activities.*

Reconfiguration processes can induce costs on the execution [3]. During a switching from a configuration to another, it is important to take into account several factors, such as the cost of the newly selected configuration (e.g. in a pay-per-use execution environment dependent on the classes of dynamically provisioned computing resources [1]). The reconfiguration cost can also be proportional to the "*amplitude*" of the switch [2], i.e. a monetary charge and/or a performance overhead proportional to the amount of allocated/deallocated computing resources. Consequently, *it is clear that reconfigurations do not come for free, but they should be executed only when real benefits in terms of QoS can be achieved.*

Over the last years, many studies [3,4] about adaptiveness for distributed parallel applications have focused on providing run-time supports to dynamic reconfigurations with the minimum impact on the computation performance (often assuming dedicated execution environments). On the other hand, the problem of defining powerful *Adaptation Strategies* is still an open research issue which requires innovative approaches to performance modeling, to the achievement of agreements between control decisions of different controllers, and to ensure the minimization of operating costs related to the application execution.

In this paper we propose a novel approach based on a control-theoretic strategy known as *Model-based Predictive Control* [5]. We control the parallelism degree of distributed parallel computations organized as graphs of parallel components. Each component features a local control sub-problem and cooperates with the others in order to reach the optimal application QoS (expressed in terms of effective performance and resource utilization cost). The cooperation is enforced using the *Distributed Subgradient Method* [6]. We propose different formulations of the adaptation strategy and we show interesting results in terms of *reconfiguration stability*. This property is expressed as the amount of performed reconfigurations and the average time for which a newly selected configuration is not modified.

This paper is organized as follows. The next section reviews research work on adaptiveness for distributed parallel computations. Section 3 presents our methodology. Section 4 shows a first evaluation of our approach developed in a simulation environment. Finally, Section 5 gives the conclusion of this work.

2 Related Work

Providing computing systems with run-time supports to dynamic reconfigurations has been the subject of several researches in different fields like Mobile, Grid and Cloud Computing. On such environments, it is of great importance to dynamically provide computing resources to applications featuring variable QoS requirements and characterized by irregular workload conditions. Examples are described in [7], in which provisioning mechanisms of virtual machines are provided to accelerate compute-intensive jobs submitted into Cloud platforms.

Besides efficient run-time supports [3,4], emerging computing environments raise critical problems related to when reconfigurations should be executed in order to optimize performance and economic aspects. Therefore, adaptation strategies have gained much attention. The pro-active adaptation to future workload variations goes in the direction of defining powerful strategies able to optimize performance requirements and operating costs by avoiding unnecessary reconfigurations. Despite the existence of first activities in this area [8], this research direction is open and still lacks from systematic approaches.

Along this line, this paper introduces an approach based on the application of Control Theory and Optimal Control foundations.

3 Methodology and Problem Statement

The control problem of distributed parallel applications can be decomposed into a set of sub-problems associated with each application module (we use module as a synonym of component). The distributed control logic should be organized in order to state how individual solutions of local control problems are combined and directed towards a solution that optimizes the application global QoS.

The core element of our methodology is the concept of *adaptive parallel module* (i.e. shortly ***ParMod***), an active unit featuring a parallel computation and an adaptation strategy to respond to dynamic execution conditions. A ParMod is structured into two interconnected parts following a closed-loop control scheme:

– the **Operating Part** performs the *functional logic* of the module, i.e. a parallel computation that instantiates a structured parallelism pattern (e.g. task-farm and data-parallel schemes) [3,4]. The computation is activated by receiving tasks from input data streams. Results are transmitted onto output data streams directed to specific destinations;
– the **Control Part** (controller) observes the Operating Part execution and performs reconfiguration activities. The Control Part implements the *adaptation strategy* that drives the reconfiguration selection.

At discrete time intervals (called *control steps*), the Operating Part exchanges measurements representing the actual behavior of the parallel computation (e.g. memory usage, resource utilization, service time and computation latency) with the Control Part. In order to take effective reconfiguration decisions, at each control step Control Parts of different ParMods (belonging to the same application) exchange *control information* in order to reach specific agreements between control decisions. The result is a set of reconfigurations able to change the parallel computation, e.g. modifications of the current parallelism degree (number of threads/processes of the current implementation).

3.1 Distributed Model-Based Predictive Control

An important precondition to apply control-theoretic techniques is the existence of a mathematical model of the controlled system. For each application module M_i we identify a *local model* involving the following set of variables:

– ***QoS variables*** ($\mathbf{x}_i(k) \in \mathbb{R}^n$) are metrics describing the current behavior of the parallel computation, such as the performance, memory usage and resource consumption assumed at the beginning of control step k;
– ***control variables*** ($\mathbf{u}_i(k) \in \mathcal{U}_i$) are parameters that identify the ParMod configuration used throughout the k-th control step;

– ***disturbances*** ($\mathbf{d}_i(k) \in \mathbb{R}^m$) model exogenous uncontrollable factors influencing the relationship between control and QoS variables (e.g. the arrival rate from external sources).

A formal representation of the local model can be described by the following discrete-time expression:

$$\mathbf{x}_i(k+1) = \Phi_i\Big(\mathbf{x}_i(k),\, \mathbf{d}_i(k),\, \mathbf{u}_i(k),\, \mathbf{u}_{j\neq i}(k)\Big) \tag{1}$$

The model allows Control Part to predict future values assumed by local QoS variables as a function of local control inputs, disturbances and (in dynamic models) present values of QoS variables. Furthermore, the next QoS of each sub-system is still related to the remaining control variables of the other sub-systems (or a sub-set of them). Therefore *the control problem of the whole application can be viewed as a set of coupled sub-problems.*

In this paper we present adaptation strategies based on a control-theoretic technique named Model-based Predictive Control (shortly **MPC**) [5] . MPC is a method in which the current reconfiguration decision is taken by solving, at the beginning of each control step, a finite-horizon optimal control problem using the current value of QoS variables and statistical multiple-step ahead predictions of future disturbances. To be robust in dynamic and uncertain environments, only the first element of the optimal reconfiguration sequence (trajectory) is passed to the Operating Part, and the same procedure is repeated at the next control step.

Distributed MPC schemes can be applied to control large-scale systems such as distributed computations. In this case the optimization problem is composed of a set of coupled sub-problems each one formed by a local objective function, a local model and a set of local constraints. In a *cooperative scenario*, the goal of the decomposition is to reach a sequence of globally optimal control decisions, i.e. the solutions of the sub-problems should optimize the following global problem:

$$\begin{aligned} &\underset{\overline{U}_1(k),\ldots,\overline{U}_N(k)}{\arg\min}\; J_G = \sum_{i=1}^{N} w_i\, J_i\Big(\overline{X}_i(k+1),\, \overline{U}_i(k),\, \overline{U}_{j\neq i}(k)\Big) \\ &s.t. \\ &\mathbf{x}_i(k+1) = \Phi_i\big(\mathbf{x}_i(k),\, \mathbf{d}_i(k),\, \mathbf{u}_i(k),\, \mathbf{u}_{j\neq i}(k)\big) \;\; i = 1,2,\ldots,N \\ &\mathbf{u}_i(k) \in \mathcal{U}_i \;\; i = 1,2,\ldots,N \end{aligned} \tag{2}$$

where J_G is the global objective function, defined as the weighted sum of local objectives (w_i is a positive weight), and an uppercase overlined letter represents a trajectory over a prediction horizon of h future control steps.

3.2 Addressing the Stability of Control Decisions

In dynamic execution contexts, distributed parallel applications should adapt the amount of used resources to provide acceptable levels of performance and a reasonable resource utilization cost. For each ParMod M_i, the configuration parameter (control input) is the current parallelism degree $n_i(k) \in \mathcal{U}_i$ (number of used computing nodes), where $\mathcal{U}_i$ is the closed interval $[1, n_i^{max}]$ of integers. Disturbances are parameters that may change due to environmental or application-dependent reasons. Examples are the mean calculation time per task $T_{calc\text{-}i}(k)$ and the probabilities of task transmission between modules, i.e. $p_{i,j}(k)$ is probability to transmit a task from ParMod M_i to M_j during control step k. The mean *ideal service time* of a module must be defined as a function of its parallelism degree, e.g. $T_{S_i}(k) = T_{calc\text{-}i}(k)/n_i(k)$ (*perfect scalability assumption*).

The interaction between distributed modules usually follows the message-passing paradigm. Communications resort on *blocking mechanisms* to address the finiteness of the input buffers. If a message attempts to enter a full capacity destination queue upon the completion of a service at M, it is forced to wait in that component until the destination has a free position. We call the mean *inter-departure time*, the steady-state average time between two successive result departures. We denote with $T_{D_i}(k)$ the QoS variable representing the mean inter-departure time of M_i at the beginning of control step k (it refers to the average value assumed during the last step $k-1$).

To exemplify our approach, we adopt a simple yet powerful performance model already discussed in [9]. The method is valid for a large class of computation graphs, i.e. *acyclic graphs with a single source module.* The main result is summarized by the following theorem (the proof can be found in [9]):

Theorem 1 (Steady-State Analysis). *Given a single source acyclic graph G of N modules, the inter-departure time T_{D_i} from M_i can be expressed as:*

$$T_{D_i}(k+1) = \max\Big\{ f_{i,1}\big(T_{S_1}(k)\big),\, f_{i,2}\big(T_{S_2}(k)\big), \ldots, f_{i,N}\big(T_{S_N}(k)\big)\Big\} \tag{3}$$

Each term $f_{i,j}$ with $j = 1, 2, \ldots, N$ expresses the inter-departure time of M_i if module M_j is the bottleneck of the graph. $f_{i,j}$ is defined as a function of the service time of M_j:

$$f_{i,j}\big(T_{S_j}(k)\big) = T_{S_j}(k)\, \frac{\displaystyle\sum_{\forall \pi \in \mathcal{P}(M_1 \to M_j)} \left(\prod_{\forall (s,d) \in \pi} p_{s,d}(k) \right)}{\displaystyle\sum_{\forall \pi \in \mathcal{P}(M_1 \to M_i)} \left(\prod_{\forall (s,d) \in \pi} p_{s,d}(k) \right)} \tag{4}$$

where M_1 denotes the source, $\mathcal{P}(M_1 \to M_i)$ is the set of all the paths starting from M_1 and reaching M_i, and (s,d) is an edge of the path π. Since we do not know which module will be the bottleneck, the inter-departure time is calculated by taking the maximum between the functions $f_{i,j}$ for $j = 1, \ldots, N$.

We study two different formulations of the MPC strategy. In the first one we do not model any abstract term related to the reconfiguration cost (we refer to this as *Non-Switching Cost Formulation* for brevity):

Definition 1 (Non-Switching Cost Formulation). *Each parallel module has a local cost function defined over a horizon of one future step:*

$$J_i(k) = \underbrace{\alpha_i \, T_{D_i}\big(k+1\big)}_{performance\ cost} + \underbrace{\beta_i \, n_i(k)}_{resource\ cost} \tag{5}$$

The performance cost discourages configurations that compromise the capability to process incoming tasks. The second part expresses a cost proportional to the number of used nodes. α_i and β_i are two positive coefficients establishing the desired trade-off between the two contrasting aspects of the cost function.

In Grid and Cloud environments the reconfiguration process can induce costs on the computation, both in terms of a performance degradation (e.g. parallel modules could be blocked waiting for the reconfiguration process to complete) as well as in terms of a monetary charge due to the dynamic provisioning of resources. In the second formulation we account for an abstract cost term:

Definition 2 (Switching Cost Formulation). *The local cost function of each ParMod M_i is defined over a prediction horizon of h control steps (with $h \geq 1$):*

$$J_i(k) = \underbrace{\sum_{q=k}^{k+h-1} \alpha_i \cdot T_{D_i}(q+1)}_{performance\ cost} + \underbrace{\sum_{q=k}^{k+h-1} \beta_i \cdot n_i(q)}_{resource\ cost} + \underbrace{\sum_{q=k}^{k+h-1} \gamma_i \cdot \Delta_i(q)^2}_{switching\ cost} \tag{6}$$

where $\Delta_i(k) = n_i(k) - n_i(k-1)$. The switching cost term is defined as a function of the square of parallelism degree variations over the horizon (γ_i is a positive coefficient), and binds control decisions between consecutive steps allowing to express formulations with a parametric horizon length.

This formulation is aimed at improving the reconfiguration stability by discouraging reconfigurations with large amplitude and avoiding fluctuating behaviors due to disturbances with high variance and featuring trend patterns.

We solve the cooperative distributed MPC problem using the ***Distributed Subgradient Method***, originally proposed in [6] for multi-agent environments. The method addresses the problem of optimizing in a distributed fashion the sum $J_G(k) = \sum J_i(k)$ of non-smooth convex functions known only by their agents. This method suits particularly well our needs, since:

- each Control Part knows only its local cost function and the model to predict the steady-state performance of its Operating Part;
- in both of our formulations each local cost is expressed by a *non-differentiable convex* function (we recall that the inter-departure time is defined as the point-wise maximum of a set of convex functions $f_{i,j}$);
- Control Parts are directly interconnected only between neighbors.

Each Control Part computes and maintains a local estimate of the optimum ***strategy profile matrix*** $\mathcal{S}(k) \in \mathbb{R}^{h \times N}$, where each column i corresponds to the reconfiguration trajectory of ParMod M_i (parallelism degrees are considered real values for feasibility reasons). Neighboring controllers iteratively exchange their local estimates and compute the next estimate using the following rule:

$$\mathcal{S}_{[i]}^{(q+1)}(k) = \mathcal{P}_{\mathcal{U}_f}\left[\sum_{j=1}^{N}\Big(\mathcal{W}[i,j]\,\mathcal{S}_{[j]}^{(q)}(k)\Big) - a^{(q)}\,\mathcal{G}_i\right] \tag{7}$$

where q is the current iteration, $a^{(q)} > 0$ is the *step-size* and $\mathcal{G}_i$ is a *subgradient* of J_i at point $\mathcal{S}_{[i]}^{(q)}(k)$[1]. $\mathcal{P}_{\mathcal{U}_f}$ is the Euclidean projection onto the convex set of admissible strategy profiles defined by: $\mathcal{U}_f = \mathcal{U}_1^h \times \mathcal{U}_2^h \times \ldots \times \mathcal{U}_N^h$.

Each controller maintains a set of weights representing the importance given to the estimates received by the controllers (zero is assigned to non-neighbor controllers). To prove the convergence to the global optimum, in [6] the authors state a condition about how the weights should be assigned: the weight matrices $\mathcal{W} \in \mathbb{R}^{N\times N}$ should be *doubly stochastic*, i.e. all the columns and rows sum to 1.

The MPC strategy based on the Distributed Sub-gradient Method consists in a sequence of actions performed by the controllers at each control step k:

– each controller acquires monitoring information from its Operating Part and calculates statistical predictions of disturbances over the prediction horizon;
– each controller uses a specific initial estimate of the strategy profile matrix and applies the iterative protocol for a fixed number of iterations;
– at each iteration, controllers receive the local estimates from their neighbors, apply the update rule (7) and transmit the next estimate;
– after the last iteration, each controller knows its optimal reconfiguration trajectory and applies the first element of that trajectory (properly rounded to the nearest integer) as the new parallelism degree for control step k.

This method allows us to consider also *non-ideal performance behaviors* of parallel modules, providing that the ideal service time is modeled as a convex function of the parallelism degree. An example is when the service time stops to decrease or even increases using parallelism degrees larger than a specific value.

4 Evaluation of the Approach

We have developed a ParMod simulation environment based on the OmNeT++ discrete event simulator. A ParMod is simulated by two OmNeT components modeling the Operating Part and the Control Part. The Operating Part implements a queue logic in which buffered elements represent input tasks. To reproduce a blocking semantics, we have implemented a communication protocol based on the transmission of send and ack messages. The Operating Part can adopt two working logics: (i) a *task-farm semantics*, in which at most p tasks in parallel can be executed, where p is the current parallelism degree; (ii) a *data-parallel semantics*, in which only one task at a time is processed with an execution time equal to the calculation time divided by the parallelism degree.

[1] The subscript $[i]$ denotes that $\mathcal{S}_{[i]}^{(q)}(k)$ is the estimate of the i-th controller.

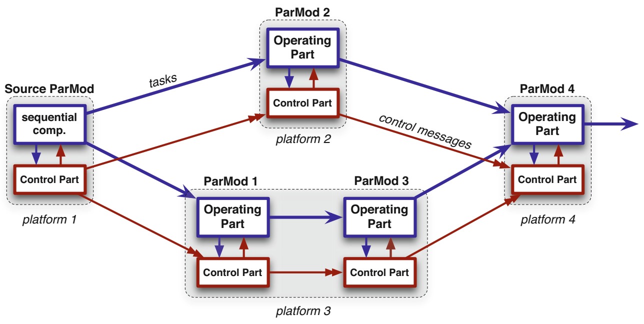


Fig. 1. Computation graph of the experiment

We consider the computation graph depicted in Figure 1. The source module, implementing a sequential computation, transmits tasks with a variable rate to ParMod 1 and 2 according to the same probability. ParMods need to dynamically adapt their parallelism degree in order to sustain the current arrival rate and to avoid using computing nodes unnecessarily. To exemplify a dynamic situation, Figure 2 shows a time-series of the ideal service time (the inverse of the service rate) of the source module, which is modeled as a measured disturbance.

In order to apply the distributed MPC strategy, we need multiple-step ahead predictions of disturbances. We exploit the well-known *Holt-Winters* filtering technique [10], an effective method accounting for time-series featuring non-stationarities such as trends and seasonal patterns. In this example we achieve accurate predictions: over the entire execution the mean relative error between the real trajectories and the predicted ones at each control step is of 8.83%, 9.38%, 10.06% and 10.77% with a horizon length equal to 1, 2, 3 and 4 steps.

We compare the Non-Switching Cost Formulation with the strategy in which we consider a switching term in the local cost functions. Moreover, in order to have a performance upper-bound, we consider the static case (named `MAX`) in which ParMods do not perform any reconfiguration, but they are configured to use their maximum parallelism degree for the entire execution. Table 1 shows a set of user-defined parameters representing the importance of the different

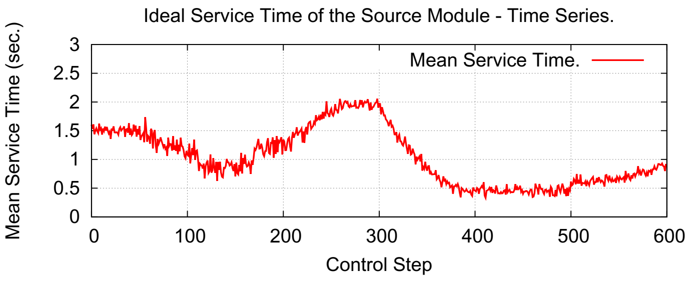


Fig. 2. Ideal service time of the source: 600 control steps each one of 240 seconds

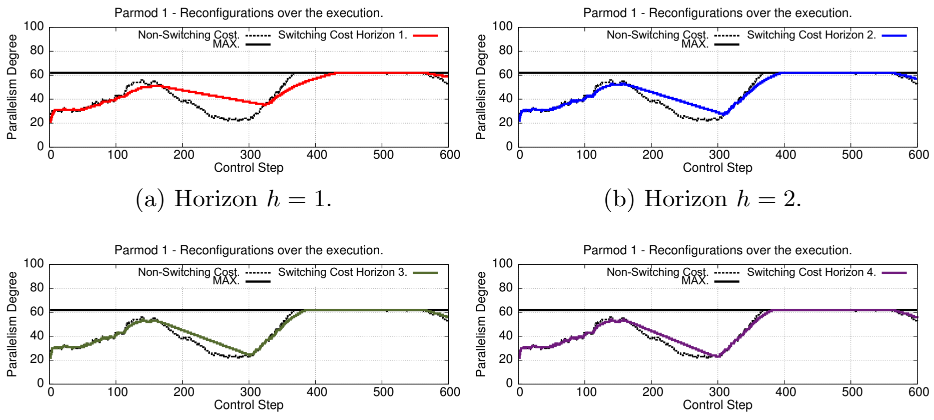
Table 1. Configuration parameters of the experiment

	Source	**ParMod 1**	**ParMod 2**	**ParMod 3**	**ParMod 4**
$\boldsymbol{T_{calc}}$	Fig. 2	90 sec.	25 sec.	70 sec.	35 sec.
$\boldsymbol{\alpha}$	20	20	20	20	20
$\boldsymbol{\beta}$	0.5	0.3	0.8	0.3	0.4
$\boldsymbol{\gamma}$	-	1.5	1.2	1.5	1.2
$\boldsymbol{n_i^{max}}$	1	64	48	64	48

control objectives (performance vs. resources), the mean calculation times and maximum parallelism degrees.

Figure 3 shows the reconfiguration sequence of ParMod 1. For the sake of space we omit the results of the other modules (which are qualitatively similar). The reconfiguration sequence with the Non-Switching Cost Formulation follows the behavior of the source module. Execution phases in which the arrival rate to ParMod 1 decreases (i.e. from step 150 to 300) correspond to time intervals in which the module releases computing resources (its parallelism degree is oversized). The opposite behavior can be noticed after control step 300: due to a decreasing trend of the source service time, ParMod 1 starts to increase its parallelism degree. After control step 370 it reaches n_1^{max} (64 nodes), i.e. it can not acquire any other resource and becomes the graph bottleneck.

As we consider the switching cost, we achieve smoother reconfiguration sequences. *The switching cost acts as a disincentive to reconfigurations*: i.e. during execution phases in which the workload is lighter, it slows down the release of computing resources, while in phases of heavy workload it slows down the allocation of new resources. With longer horizons, controllers have a better degree of foresight and can more precisely evaluate if the acquisition/release of resources is effectively useful (e.g. avoiding to release/re-acquire them nearly in the future).



(a) Horizon $h = 1$. (b) Horizon $h = 2$.

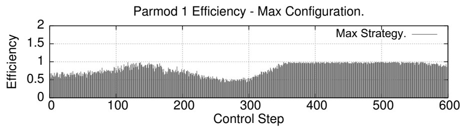
(c) Horizon $h = 3$. (d) Horizon $h = 4$.

Fig. 3. Parallelism degree variations of ParMod 1

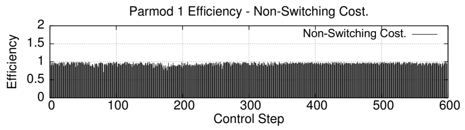
This justifies a more rapid capability of longer-horizon strategies to respond to a pronounced change in the disturbance trend. Moreover, in execution phases characterized by a high level of uncertainty, the Switching Cost Formulation avoids many small reconfigurations of little amplitude (as happens from step 0 to 300). Therefore, *we can say that the switching cost acts also as a stabilizer in presence of disturbances with a significant variance.*

The horizon length has also important consequences on the *efficiency* of resource utilization, measured as the ratio between the ideal service time of a ParMod and its inter-departure time. An efficiency smaller than 1 means that the parallelism degree is over-sized. Figure 4 outlines the efficiency of ParMod 1. The `MAX` configuration (Figure 4a) suffers from a severe degradation from step 0 to 370. After step 370 the efficiency rises to 1 because the ParMod becomes the application bottleneck and it begins to fully exploit its maximum parallelism degree. Extremely interesting is the behavior of the Non-Switching Cost Formulation. In this case the efficiency is near to 1 throughout the execution. The reason is given by the structure of the local cost function (Definition 1): if a module is adopting an over-sized parallelism degree, it can release some computing resources without affecting its effective performance, but improving the value of its local cost without making the other cost functions worse off.

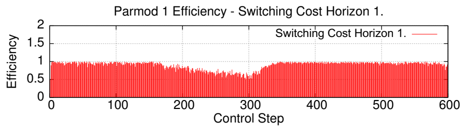
By introducing the switching cost we have a break that causes a slower release of computing resources. This induces a slight degradation of the efficiency from step 150 to 370. Using longer horizons (providing that the disturbance predictions are sufficiently accurate) we are able to mitigate this effect, and the efficiency tends to 1 again as Figure 4f depicts. To compare different strategies in terms of their reconfiguration stability, we introduce the following metric:



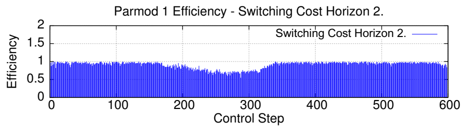
(a) Max Strategy.



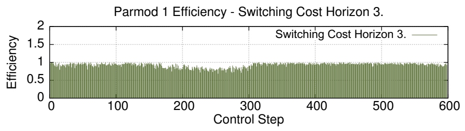
(b) Non-Switching Cost.



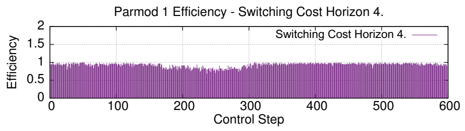
(c) Switching Cost $h = 1$.



(d) Switching Cost $h = 2$.



(e) Switching Cost $h = 3$.



(f) Switching Cost $h = 4$.

Fig. 4. Efficiency of ParMod 1 with the different strategies

Definition 3. *We denote as **Mean Stability Index** (shortly* MSI*) the average number of control steps for which a reconfiguration remains active.*

Table 2 shows the total number of reconfigurations and the number of tasks that leave the system. The global MSI is the average of the individual indices of each module. With the MAX configuration we are able to maximize the number of completed tasks at the expense of a big waste of computing resources during the first part of the computation. With the Non-Switching Cost strategy the difference in completed tasks (the "no delay" column) is only of 8.25% but with a significant benefit in terms of efficiency. The Switching Cost Formulation has a large degree of configurability: with short horizons it heavily reduces the number of reconfigurations and the stability index with a small loss in terms of completed tasks. As we use longer horizons, the performance difference becomes negligible (only 0.5% of tasks reduction) but with a great improvement (34%) in terms of reconfigurations with a horizon of four steps. We conclude that the Switching Cost strategy is extremely powerful: *it makes it possible to significantly reduce the number of reconfigurations with a negligible performance reduction.*

Table 2. Number of reconfigurations, completed tasks and Mean Stability Index

Strategy	**Reconf.**	**MSI**	**Compl. Tasks (no delay)**	**Compl. Tasks (with delay)**
MAX	-	-	144,403	144,403
Non-Switch. Cost	870	2.77	132,482	123,643
Switch. Cost $h = 1$	389	6.28	129,560	124,898
Switch. Cost $h = 2$	524	4.65	131,176	124,911
Switch. Cost $h = 3$	559	4.34	131,641	125,030
Switch. Cost $h = 4$	574	4.25	131,808	125,823

When a performance overhead is introduced, performing fewer reconfigurations could be useful also from the performance viewpoint. To prove this insight we modify our simulator: every time a ParMod applies a reconfiguration, it suspends to process incoming tasks for an amount of time modeled by a random variable *delay*. In order to reproduce a Cloud scenario, in which the time-to-deploy of a virtual machine can reach tens of seconds [1], we repeat the simulations using a delay of 30 seconds. Now we have a different tendency in terms of completed tasks (the "with delay" column): *using the Switching Cost Formulation we achieve better performance saving a consistent number of reconfigurations.*

We conclude by pointing out the feasibility of our approach. Although the subgradient method can be rather slow, we can limit the number of iterations (we used 125 iterations per step). This can be done by considering two aspects: (i) firstly each Control Part applies an integer rounding of the parallelism degree, thus a high precision is not necessary actually; (ii) since between consecutive control steps optimal solutions are likely close, we use as starting estimate the optimal strategy profile matrix calculated at the previous step. In this way we can drastically reduce the number of iterations maintaining an acceptable precision.

5 Conclusion

This paper provides a description of our approach. The control logic of each module consists of a performance model and a local cost function. Reconfigurations are applied following the receding horizon principle and the MPC strategy. Controllers cooperate to reach globally optimal decisions using the Distributed Subgradient Method. In order to enforce the stability of control decisions, and measuring the impact of stability w.r.t QoS goals, we evaluate different MPC formulations. Simulation results show the effectiveness of our approach. In the future we plan to apply our techniques in real-world distributed environments.

References

1. Costa, R., Brasileiro, F., Lemos, G.: Analyzing the impact of elasticity on the profit of cloud computing providers. Future Generation Computer Systems (2013)
2. Han, R., Ghanem, M.M., Guo, L., Guo, Y., Osmond, M.: Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. Future Generation Computer Systems (2012)
3. Vanneschi, M., Veraldi, L.: Dynamicity in distributed applications: issues, problems and the assist approach. Parallel Comput. 33(12), 822–845 (2007)
4. Aldinucci, M., Campa, S., Danelutto, M., Vanneschi, M.: Behavioural skeletons in gcm: Autonomic management of grid components. In: Parallel, Distributed and Network-Based Processing, PDP 2008, pp. 54–63 (February2008)
5. Garcia, C.E., Prett, D.M., Morari, M.: Model predictive control: theory and practice a survey. Automatica 25, 335–348 (1989)
6. Nedic, A., Ozdaglar, A.: Distributed subgradient methods for multi-agent optimization. IEEE Transactions on Automatic Control 54(1), 48 (2009)
7. Warneke, D., Kao, O.: Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. IEEE Trans. Parallel Distrib. Syst. 22(6) (2011)
8. Islam, S., Keung, J., Lee, K., Liu, A.: Empirical prediction models for adaptive resource provisioning in the cloud. Future Generation Computer Systems 28(1), 155–162 (2012)
9. Mencagli, G.: A Control-Theoretic Methodology for Controlling Adaptive Structured Parallel Computations. Ph.D Thesis, University of Pisa, Italy (2012)
10. Chatfield, C., Yar, M.: Holt-winters forecasting: Some practical issues. Journal of the Royal Statistical Society. Series D (The Statistician) 37(2), 129–140 (1988)

On the Use of a Proportional-Share Market for Application SLO Support in Clouds

Stefania Costache[1,2,⋆], Nikos Parlavantzas[2,3], Christine Morin[2], and Samuel Kortas[1]

[1] EDF
[2] INRIA Centre Rennes - Bretagne Atlantique
[3] INSA Rennes

Abstract. Virtualization provides increased control and flexibility on how resources are allocated to applications. However, common resource provisioning mechanisms do not fully use these advantages; either they provide limited support for applications demanding quality of service, or the resource allocation complexity is high. To address these issues we developed Themis, a market-based application management platform. By limiting the coupling between the applications and resource management, Themis can support diverse types of applications and performance goals while ensuring maximized resource usage. In this paper we present the performance of Themis when users execute batch applications with different Service Level Objectives such as deadlines.

1 Introduction

Cloud computing is attractive to execute increasingly dynamic and complex applications on a High Performance Computing infrastructure. An organization can efficiently share its physical resources between different application types (e.g., MapReduce, MPI, or other legacy applications) by allowing each application to run in its own virtual cluster (a set of virtual machines configured with the software packages needed by the user) with limited interference from the infrastructure's administrator. Recent Platform-as-a-Service solutions, both commercial [1, 8] and research [16] hide the complexity of deploying and configuring these virtual clusters, providing users with support to develop and run their applications with no concerns regarding infrastructure's resource management complexities.

However, a remaining challenge is the design of resource management policies to share resources fairly between applications, in terms of their priority and user-specified Service Level Objectives (SLOs). The common cloud resource-provisioning model is "on-demand" virtual machine (VM) provisioning. This model relies on a First-Come-First-Served policy to schedule virtual machines. This would not be a problem if the infrastructure capacity were enough for all

⋆ The author is currently a PhD student at INRIA and EDF R &D. This work is supported by ANRT through the CIFRE sponsorship No. 0332/2010.

user requests in the highest demand periods. Nevertheless, this is rarely the case, as expanding the resource pool is expensive. Thus, it is preferable to solve the contention periods when they appear, around deadlines (e.g., conference or project deliverable), rather than acquiring more resources.

To address this challenge we proposed Themis [10], a platform for application and resource management. To allocate VMs to applications, Themis uses a proportional-share market on top of a virtualized infrastructure. VMs are bought from the market at a (user or application) specified cost (i.e. bid) while the CPU time and memory amount allocated to each of them varies in time according to the total resource demand and the costs of other VMs. To keep the correct CPU and memory amounts allocated to each VM, Themis migrates them between physical nodes. We evaluated the resource allocation algorithms of Themis when applications compete for CPU time [10] and implemented it in a real prototype.

In this paper we analyze the performance of the implemented proportional-share market when applications buy amounts of *different* resources (i.e., CPU and memory) to meet *different* SLOs. Simulations with a real workload trace show that even with simple adaptation policies and using only current knowledge, the system behaves well in terms of application performance and number of VM operations (e.g., migrations, suspend/resume).

This paper is organized as follows. Section 2 presents the context of our work and introduces Themis. Section 3 details the proposed resource management policies. Section 4 describes the evaluation of the proposed resource management policies and Section 5 concludes the paper.

2 Context

This Section describes the context of our work. We first introduce the application model considered in this paper. Then we give an overview of Themis.

2.1 Application Model

Although Themis supports a wide variety of applications, in this paper we focus on batch applications composed of a fixed number of tasks. Each task requires one CPU core and a specified amount of memory. We don't model the communication between tasks. To finish their execution, applications need to perform a certain computation amount (e.g. 1000 iterations). There is a large number of iterative applications that follow this model, for example scientific simulators (e.g., Code_Saturne [9]). These applications have a relatively stable iteration execution time. The iteration execution time can be tuned by modifying the resource allocation received by each task. For example, if each task receives one full core, one iteration can take 1 second. If the resource allocation drops at half, the same iteration can take 2 seconds.

2.2 Themis Overview

We previously developed Themis, a market-based platform for application and resource management on clouds [10]. A variety of solutions were proposed to use

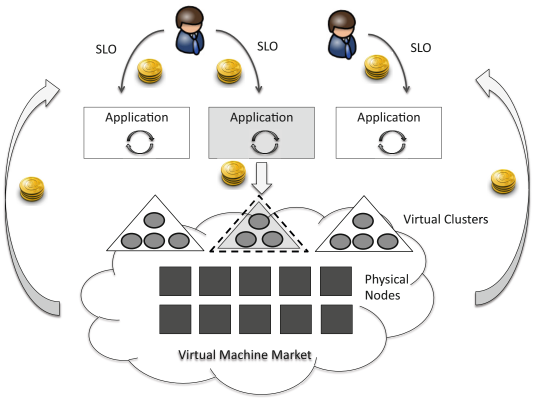


Fig. 1. Themis eco-system

a market to schedule jobs on clusters [11, 21, 20, 7, 22, 6, 3, 19], to schedule jobs on grids [4] or to run parallel applications [17]. Market-based systems force users to assign throughful priorities to their applications as they have to pay for their execution. If the system is too loaded, a user that doesn't need to run her application immediately will postpone its execution until a less loaded period, as the execution cost is lower. Works that apply market mechanisms focus on providing better user satisfaction than traditional resource management systems, usually batch schedulers. However, these systems neither consider application adaptation, nor user SLOs.

Figure 1 gives an overview of Themis. Users receive budgets of credits from a central banking service and use them to run their applications on the cloud. In Themis, applications are funded at a rate established by users, representing the maximum execution cost supported by the user, i.e., budget.

Themis regulates the resource allocation between applications by using a proportional-share market [13, 7, 14, 18]. Applications provision VMs from this market by specifying bids for them. Users are charged for the cost of used resources, i.e., the VM bids, at each scheduling period. In Themis resources (i.e. CPU and memory) are allocated to VMs by using a proportional-share allocation rule. With the proportional-share scheduling mechanisms, each VM i is assigned a bid b_i and receives a share of $b_i/\Sigma b$ of the infrastructure resources. For example, let's take two VMs A and B that want to use the CPU resource of a physical node. The VM A has a bid of 1 and the VM B has a bid of 2. In this case, A receives 33% CPU time and B receives 66%. This mechanism is already supported by current hypervisors and achieves good system utilization through fine-grained resource allocation.

Themis runs each application in a private virtual cluster and allows the application to adapt its resource demand by changing the number of VMs or the resource allocation (i.e., CPU and memory) for each VM by changing the VM bids. Applications individually adapt their resource demand to meet their SLOs (e.g. deadlines) and to react to fluctuations in the resource prices.

VM Allocation on the Proportional-Share Market. We apply the proportional-share market to allocate to each VM a fraction of CPU time and physical memory on a physical node. Themis periodically applies the proportional-share allocation rule system-wide. For simplicity, we consider that the storage capacity is sufficient to accommodate all the VM images.

As the system load or the bids change in time, to ensure an allocation corresponding to the submitted bids, VMs might need to be migrated between nodes. To limit the number of migrations, each VM is allowed to have a certain allocation error. To explain what the allocation error is, we introduce the following terms. We define a_{ri} as the maximum resource allocation a VM receives for a resource r from the capacity of its current node proportional to the bids of all the VMs running on this node. We define a_{rh} as the resource allocation the VM would receive for a resource r if the total infrastructure capacity is considered. Then, if the VM is not migrated, its allocation error is the maximum error over all resources: $max_r(e_r)$, where $e_r = \frac{a_{ri} - a_{rh}}{a_{ri}}$. The goal of the migration algorithm is to ensure that each VM has an allocation error below a given threshold (e.g. 10%). To select the VMs to be migrated at each scheduling interval, Themis uses a tabu-search heuristic [12]. Tabu-search is a local-search method for finding the optimal solutions of a problem. The heuristic runs for a specific number of iterations. At each iteration the heuristic tries to move the VM with the maximum allocation error that is not in the tabu list to the physical node that minimizes it. The heuristic stops if it cannot improve the VM allocation error for the last iterations (i.e. 100 iterations) or if reaching a better solution involves a number of migrations higher than a threshold.

Application Resource Demand Adaptation Policy. On top of the proportional-share market, applications can adapt in two ways: (i) by changing their resource demand (i.e. number of virtual machines) to cope with modifications in their workload (e.g. changes of computation algorithms, additional started modules, etc.); (ii) by changing their bids (which are re-considered at each scheduling period) to cope with fluctuations in price. In this paper we focus on the last case, for which we developed a simple policy. The application uses only information regarding its current CPU and memory allocation and resource prices. Based on this information, the application tries to keep the value for its remaining execution time, or the predicted time for the next iteration, close to a user-defined reference metric. For this we use simple heuristics: the application decreases the bids for its resources when its performance value drops below the given target (i.e. the predicted execution time becomes smaller than the remaining time to deadline) and it increases them otherwise.

In this paper we extend our previous work in several ways. First, we extend the resource allocation algorithms to support for multiple resource allocation. Second, we define a set of adaptation policies for different deadline-based SLOs. These policies use the current application resource allocation and resource price to adapt the application's bids for VMs and obtain the desired resource allocation for meeting the application's SLO (they vertically scale the VMs). Finally, we show how Themis can support these different user SLOs and how its resource allocation algorithms cope with both CPU and memory dynamic allocation.

3 SLO Policies on the Proportional-Share Market

In this section we define a set of SLOs users typically require for their applications and a set of policies to adapt the application resource demand to these SLOs.

User and SLO Modeling. After studying the needs of HPC users from an organization (e.g. at Electricité de France), we determined two classification criterias: the required time of their application results and the required application results.

Based on the required time of their application results, we found that there are two classes of users:

- deadline users: they want the application results by a specific deadline. For example, a user needs to send her manager a simulation result by 7pm.

- performance users: they want the results as soon as possible but they are also ready to accept a bounded delay. This delay is defined by the application deadline too. For example, a developer wants to test a newly developed algorithm. She wants the results as fast as possible, but if the system is not capable to provide them, she might be willing to wait until the morning.

Based on the required application results, we found two classes of preferences:

- strict results: to provide useful results the application needs to finish all its computation before its deadline.

- partial results: some users might value *partial* application results at their given deadline; for example, for a user who implemented a scientific method and needs to run 1000 iterations of her simulation to test it, finishing 900 iterations is also sufficient to show the good method behavior.

We combined these classes and obtained four user types: (i) deadline-strict; (ii) deadline-partial; (iii) performance-strict; (iv) performance-partial. We think that these categories can be representative for other organizations as well.

Additional Mechanisms. Besides adapting their bids, to minimize the execution cost applications can apply two policies: (i) delay their execution if the price is too high; (ii) suspend their execution when the price becomes too high, and resume it later when the price drops.

Algorithm 1 describes the conditions the application uses to start, resume or suspend its execution. The suspend policy is run at each scheduling interval by the application. The start/resume policies are run by Themis on behalf of the application, if the application hasn't started or is suspended.

The *StartResume* policy computes the initial payment for *nvms* VMs with T_{alloc} allocated resources by using the current market price. If this payment is greater than the maximum afforded budget, bid_{max}, then the application postpones its execution/resume with a random amount of time bounded by t_{wait}.

Algorithm 1. Application Start/Resume/Suspend Policies

```
1: **StartResume**($bid_{max}, nvms, T_{alloc}, price_{current}, t_{wait}$)
2: bid = compute initial payment(price_current, $T_{alloc}$, nvms)
3: start = False
4: if $bid[CPU] + bid[memory] < bid_{max}$ then
5:     start = True
6: if start = False then
7:     wait random time period between now and $t_{wait}$
8:
9: **Suspend**($value, T, bid, bid_{max}$)
10: if ( $value < T$) and
    $bid[CPU] + bid[memory] < bid_{max}$ then
11:     if $suspend\_iterations < max\_iterations$ then
12:         $suspend\_iterations = suspend\_iterations + 1$
13:         suspend = False
14:     else
15:         suspend = True
```

The *Suspend* procedure describes the conditions the application uses to suspend its execution. To minimize the execution cost, the application suspends itself when its performance metric, *value*, is bigger than the reference T and the application cannot afford to improve it. To avoid cases in which all applications would suspend at the same time, before suspending its execution, the application waits for a random number of scheduling periods, defined by *max_iterations*.

Application Policies. We derive a set of application specific policies that consider the types of users and goals previously presented. These policies run periodically during the application execution. To ensure the best chance to finish its execution before a deadline, an application starts as soon as the price drops enough so the application can afford a minimum allocation for each VM (e.g. 25% of CPU time). Then, during its execution it applies the different policies according to its SLO. These policies are the following:

- deadline-strict: Applications start when the price is low enough to ensure a good allocation (i.e. 75% CPU time). During their execution they adapt their bids to keep a low price in low utilization periods and to use as much resource as their SLO allows in high utilization periods. If the application cannot pay for the resources needed to meet its SLO it suspends its execution. In this way it leaves resources for other applications and avoids wasting credits for nothing. The application resumes if the price drops enough to allow it to finish its execution within the deadline. If, during its execution, the application sees it cannot miss the deadline it stops.

- deadline-partial: This policy is similar to the previous policy. Nevertheless, there are two differences: (i) the application suspends only when a minimum allocation cannot be ensured (e.g. 30% cpu time or 30% physical allocated memory); (ii) as any work done at the deadline is useful, the application does not stop its execution when it sees it cannot meet its deadline anymore.

- performance-strict: The policy is similar to the **deadline-strict** policy and it follows the same algorithm. However, during its execution, the application, instead of tracking a performance reference metric, tries to keep a maximum allocation given its budget. When the application cannot have a minimum allocation at the current price, the application suspends.

- performance-partial: This policy is similar to the previous one but is used by users accepting partial results. However, as for the *deadline-partial* policy, the application does not terminate before the deadline is reached.

4 Evaluation

This section describes the evaluation of our proposed resource management policies. With our evaluation we seek an answer to the following two questions:

- How does the system perform in terms of user satisfaction when their applications behave strategically and adapt their resource demands?
- What is the performance overhead of the application adaptation, considering that application adaptation leads to VM operations?

4.1 Implementation

Themis is implemented in CloudSim [5], a Java event-based cloud simulator. CloudSim can be used to model applications, workload submission scenarios and varios resource management policies. The simulated environment is composed of a datacenter, its VM allocation policy that runs periodically, and multiple applications, created dynamically during the simulation run. Applications are created according to their submission times, taken from a workload trace, and are destroyed when they finish their execution. During its lifetime, each application runs the resource demand adaptation policy and interacts with the datacenter to change the bids for its VMs. We extended our previous implementation [10] by introducing dynamic memory allocation and overheads for VM boot and suspend/resume operations; then we implemented the proposed policies.

4.2 Evaluation Metrics

The performance of a resource management system can be measured in different ways. Traditional metrics include application wait time, resource utilization or number of missed deadlines. Nevertheless, these metrics do not reflect accurately the total user satisfaction, which represents an important metric in showing how well resources are managed. To quantify the total user satisfaction, the aggregate user satisfaction can be used. We model the user satisfaction as a function of the budget assigned by the user to its application and the application execution time, i.e., a utility function. As there are four different user types, we obtain four utility functions.

Nevertheless, before discussing the signification of utility functions, we define the following terms. t_{exec} is the application execution time. $t_{deadline}$ is the time from the submission to deadline. t_{ideal} is the ideal execution time, i.e., if the application runs on a dedicated infrastructure. $work_{done}$ represents the number of iterations the application managed to execute until it was stopped and

Table 1. Utility functions

User	Utility Function
Deadline-strict	B, if $t_{exec} \leq t_{deadline}$, 0 otherwise
Deadline-partial	B if $t_{exec} \leq t_{deadline}$, $B \cdot \frac{work\_done}{work\_total}$ otherwise
Performance-strict	$max(0, B \cdot \frac{(t_{deadline} - t_{exec})}{(t_{deadline} - t_{ideal})})$

$work_{total}$ represents the total number of iterations. B is the application's budget per time scheduling period and per task. B is assigned by the user and reflects the application's importance.

Table 1 summarizes the used utility functions. The deadline-strict user values the application execution at her budget rate if the application finishes before deadline, otherwise she assigns a value of zero. The deadline-partial user is satisfied with the amount of work done until the deadline. Thus the value of the application execution is proportional with this amount. The performance-strict user becomes dissatisfied proportional to her application execution slowdown. We bound the value of her dissatisfaction at zero. The utility function for the performance-partial user is a combination between the deadline-partial and the performance-strict functions.

4.3 VM Operations Modeling

We implemented in CloudSim a model for several VM performance overheads:

- resource allocation. We assume pessimistically that the application's performance degrades proportionally with the allocated memory fraction, when this fraction is less than the demanded memory.

- VM boot/resume. We simulate the VM boot and resume times separately. The VM boot time is modeled by sending the application a message that the VM was created with a delay of 30 seconds. The VM resume time is modeled by assigning to the VM a processing capacity of 0 for 30 seconds.

- VM migration. We compute the migration time as the time to transfer the VM memory state by using the available network bandwidth of the current host. We assume that the bandwidth is shared fairly between all the requests. The available network bandwidth is computed by considering all the suspend and resume operations that occur on the considered host at the current scheduling period. We model the migration performance overhead as 10% of CPU capacity used by the virtual machines in which the application is running. We chose this overhead as previous work found that migration brings 8% performance degradation for HPC applications [15].

4.4 Workload Modeling

To evaluate the system performance we use a real workload trace as it reflects the user behavior in a real system. Such traces are archived and made publicly available [2]. As a workload trace, we chose the HPC2N file as it has detailed information regarding memory requirements of the applications. This file contains information regarding applications submitted to a Linux cluster from Sweden. The cluster has 120 nodes with two 240 AMD Athlon MP2000+ processors each. We assigned to each node 2 GB of memory. For applications with no memory information, we assigned a random memory amount, between 10% and 90% of the node's memory capacity. We ran each experiment by considering the first 1000 jobs, which were submitted over a time period of 18 days. We scaled the inter-arrival time with a factor between 0.1 and 1 and we obtained 10 traces with different contention levels. A factor of 0.1 gave a highly contended system while a factor of 1 gave a lightly loaded system.

We consider that all applications have a deadline and a re-chargeable budget. As we couldn't find any information regarding application deadlines, we assigned synthetic deadlines to applications. The budgets assigned to applications are inversely proportional to the application's deadline factor.

4.5 Results

Figure 2(a) describes the total satisfaction that the system provides to users when applications use the *strict-deadline* policy compared to well-known algorithms like FCFS and EDF. We selected this policies as we wanted to use the same comparison criteria as in previous work [10] and this policy performed good in terms of user SLO satisfaction. We didn't include the other three policies due to lack of space. To see the behavior of our algorithms when both CPU and memory need to be dynamically allocated, we measure the total user satisfaction when the memory is enough for all the requests and when the memory is constrained at 2GB RAM per physical node. We notice that when both CPU and memory need to be allocated our market out-performs FCFS much more than in the case of CPU only allocation. The proportional-share mechanism allows applications to run with a less than required amount of resources.

Compared to EDF, we notice a performance degradation that increases with the system load. As the inter-arrival time decreases, EDF is capable to take better scheduling decisions: more applications with smaller deadlines, and in the same time higher budgets, get to run on time. This provides better user satisfaction than our system and FCFS. Then, our system is decentralized: each application acts selfishly and independently to meet its own application SLO while with EDF, the central scheduler sorts applications by their deadline and executes the application with the smallest deadline first.

Figure2(b) describes the total satisfaction that the system provides to users when applications use different policies. Themis allocates both CPU and memory. In this scenario, each application selects a policy randomly from the four ones we provide. We notice a performance degradation compared to the case when all applications use a *strict-deadline* policy. Applications using a policy like *strict-performance* spend all their budget to try to receive a maximum allocation, leading to other applications to miss their own deadlines. In the case of

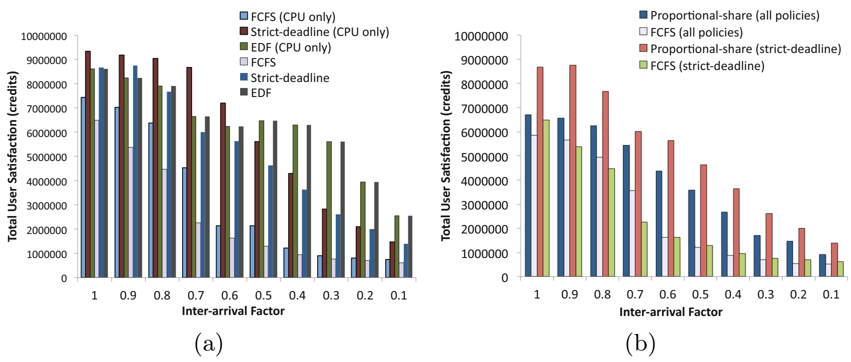


Fig. 2. Proportional share market performance in terms of total user satisfaction in two cases: (a) when the strict-deadline policy is considered and (b) when all the policies are considered

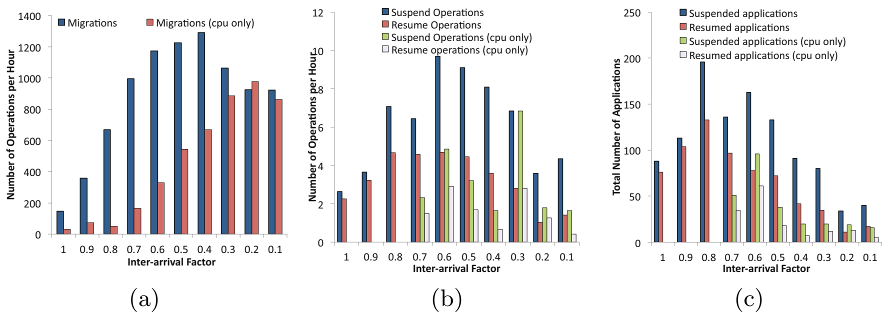


Fig. 3. Performance of VM allocation algorithm in terms of (a) average number of migrations per hour (b) average number of suspend/resume operations per hour and (c) total number of applications suspended and resumed

the *strict-deadline* policy, applications are cost effective, leaving a larger share of resources for other applications.

Figure 3 describes the number of VM operations performed by the VM allocation algorithms and the total number of applications suspended/resumed during the experiment. To perform this experiment we selected the *strict-deadline* policy, but any other policy would have been appropriate too. We make two observations: (i) the number of VM operations decreases when the system is highly loaded; (ii) the number of VM operations when there are multiple allocated resources is significantly higher than in the case of one resource. The first observation is explained by the fact that when there is a high load, more applications don't start or resume their execution. The second observation is intuitive: applications adapt their bids for multiple resources, leading to more errors in VM allocations and thus, more migrations.

To conclude our results, we need to stress that each application that runs on Themis adapts individually to the market condition and its SLO (e.g. deadline). This selfish application behavior leads to a performance degradation (or also known as the Price of Anarchy), compared to when applications collaborate or when they operate under strict, centralised control. To illustrate why this uncoordinated behavior can be bad, let us take the case of suspend/resume at price fluctuations. When multiple applications suspend, the other running applications receive a higher resource allocation and drop their bids. This creates a favorable condition for the suspended applications to resume again. However, when the other applications resume, the price increases leading them to another suspend. The performance degradation is the "price" payed by the nature of our system, that allows applications to behave selfishly.

5 Conclusions

In this paper we analyzed the performance of a proportional-share market mechanism implemented in Themis, an application and resource management platform. In Themis, applications autonomously adapt their resource demand to meet their SLOs, disregarding the other infrastructure occupants.

We extended Themis with multi-resource allocation algorithms and we simulated the application behavior by considering four SLO-driven resource demand adaptation policies. Our simulation results show the efficiency of the proportional-share market. Our policies behave reasonably in terms of application performance and number of VM operations. When the system is lightly loaded our policies lead to a better system performance than well-known scheduling schemes. As each application adapts autonomously to its own SLO, it is intuitive that the system's performance degrades in high load periods.

As future work we plan to implement a resource regulation mechanism in which applications can be more aware of the other infrastructure occupants. For example, when there are not enough free resources to satisfy all arriving applications we can use a double auction in which applications already running on the infrastructure can sell their resources to more urgent applications. We also plan to do more experiments with Themis on a real testbed.

References

[1] AmazonEBS, `http://aws.amazon.com/`
[2] Archive, P.W., `http://www.cs.huji.ac.il/labs/parallel/workload/`
[3] AuYoung, A., Chun, B., Snoeren, A., Vahdat, A.: Resource allocation in federated distributed computing infrastructures. In: Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure, vol. 9 (2004)
[4] Buyya, R., Abramson, D., Giddy, J., Stockinger, H.: Economic models for resource management and scheduling in grid computing. Concurrency and computation: practice and experience 14(13-15), 1507–1542 (2002)

[5] Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software Practice and Experience 41(1), 23–50 (2011)
[6] Chun, B.N., Buonadonna, P., AuYoung, A., Ng, C., Parkes, D.C., Shneidman, J., Snoeren, A.C., Vahdat, A.: Mirage: A microeconomic resource allocation system for sensornet testbeds. In: Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors (2005)
[7] Chun, B.N., Culler, D.E.: REXEC: A decentralized, secure remote execution environment for clusters. In: Falsafi, B., Lauria, M. (eds.) CANPC 2000. LNCS, vol. 1797, pp. 1–14. Springer, Heidelberg (2000)
[8] CloudFoundry, `http://www.cloudfoundry.com/`
[9] CodeSaturne. Codesaturne: a finite volume code for the computation of turbulent incompressible flows. International Journal on Finite Volumes (2004)
[10] Costache, S.V., Parlavantzas, N., Morin, C., Kortas, S.: Themis: Economy-based automatic resource scaling for cloud systems. In: Proceedings of IEEE International Conference on High Performance Computing and Communications (2012)
[11] Ferguson, D., Yemini, Y., Nikolaou, C.: Microeconomic algorithms for load balancing in distributed computer systems. In: International Conference on Distributed Computer Systems, vol. 499 (1988)
[12] Glover, F., Laguna, M., et al.: Tabu search, vol. 22. Springer (1997)
[13] Lai, K.: Markets are dead, long live markets. SIGecom Exch. 5, 1–10 (2005)
[14] Lai, K., Rasmusson, L., Adar, E., Zhang, L., Huberman, B.: Tycoon: An implementation of a distributed, market-based resource allocation system. Multiagent and Grid Systems 1(3), 169–182 (2005)
[15] Nagarajan, A.B., Mueller, F., Engelmann, C., Scott, S.L.: Proactive fault tolerance for hpc with xen virtualization. In: Proceedings of the 21st International Conference on Supercomputing (ICS), p. 23 (2007)
[16] Pierre, G., Stratan, C.: Conpaas: a platform for hosting elastic cloud applications. In: IEEE Internet Computing (2012)
[17] Regev, O., Nisan, N.: The popcorn market. online markets for computational resources. Decision Support Systems 28(1), 177–189 (2000)
[18] Sandholm, T., Lai, K.: Dynamic proportional share scheduling in hadoop. In: 15th Workshop on Job Scheduling Strategies for Parallel Processing (2010)
[19] Sherwani, J., Ali, N., Lotia, N., Hayat, Z., Buyya, R.: Libra: a computational economy-based job scheduling system for clusters. Software Practice and Experience 34, 573–590 (2004)
[20] Stoica, I., Abdel-Wahab, H., Pothen, A.: A microeconomic scheduler for parallel computers. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 200–218. Springer, Heidelberg (1995)
[21] Waldspurger, C.A., Hogg, T., Huberman, B.A., Kephart, J.O., Stornetta, W.S.: Spawn: A distributed computational economy. IEEE Transactions on Software Engineering 18(2), 103–117 (1992)
[22] Yeo, C.S., Buyya, R.: Pricing for utility-driven resource management and allocation in clusters. International Journal of High Performance Computing Applications 21(4), 405–418 (2007)

Topic 7: Peer-to-Peer Computing (Introduction)

Damiano Carra, Thorsten Strufe, György Dán, and Marcel Karnstedt

Topic Committee

In Peer-to-peer (P2P) systems computers form an overlay network and share their resources (storage, CPU, bandwidth) to implement a service on top of the Internet. P2P computing has a great potential for creating systems that are self-organizing, efficient, and scalable, but it also faces many challenges: dynamic peer arrivals and departures, which may be correlated (e.g., flash crowd effects, or software failures), high variability of resources, and resource heterogeneity. This topic provides a forum for researchers to present new contributions to P2P systems, technologies, middleware, and applications that address key research issues and challenges.

This year, one paper, which was evaluated by four referees, has been accepted for publication in the peer-to-peer track: "Design and Implementation of a Scalable Membership Service for Supercomputer Resiliency-Aware Runtime", by Yoav Tock, Benjamin Mandler, Jose Moreira and Terry Jones from IBM Haifa Research Laboratory, IBM T.J. Watson Research Center and Oak Ridge National Laboratory. The paper presents the design and implementation of two services for ultra-large HPC systems: a node membership service and an attribute replication service. For such services, the design uses techniques from Peer-to-Peer computing. To deal with the very large number of nodes, the design is based on a hierarchical structure of the nodes and uses an eventual consistency model. The proposed approach also supports versioning.

We would like to take the opportunity of thanking the authors who submitted a contribution, as well as the Euro-Par Organizing Committee, and the external referees with their highly useful comments, whose efforts have made this conference and this topic possible.

Design and Implementation of a Scalable Membership Service for Supercomputer Resiliency-Aware Runtime*

Yoav Tock[1], Benjamin Mandler[1], José Moreira[2], and Terry Jones[3]

[1] IBM Haifa Research Laboratory, Haifa, Israel
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
[3] Oak Ridge National Laboratory, Oak Ridge, TN, USA

Abstract. As HPC systems and applications get bigger and more complex, we are approaching an era in which resiliency and run-time elasticity concerns become paramount. We offer a building block for an alternative resiliency approach in which computations will be able to make progress while components fail, in addition to enabling a dynamic set of nodes throughout a computation lifetime. The core of our solution is a hierarchical scalable membership service providing eventual consistency semantics. An attribute replication service is used for hierarchy organization, and is exposed to external applications. Our solution is based on P2P technologies and provides resiliency and elastic runtime support at ultra large scales. Resulting middleware is general purpose while exploiting HPC platform unique features and architecture. We have implemented and tested this system on BlueGene/P with Linux, and using worst-case analysis, evaluated the service scalability as effective for up to 1M nodes.

1 Introduction

Current trends dictate increasing complexity and component counts on supercomputers and mainstream commercial systems alike [1]. This trend exposes weaknesses in the underlying clustering infrastructure needed for continuous availability, maximizing utilization, and efficient administration of such systems [2]. These issues can properly be tackled by having a resiliency supportive run-time for ensuring continuous availability and elastic run-time support for utilization maximization through proper jobs placement and load balancing. System elasticity can be an important factor also for the conservation of power which is a growing concern in the HPC world. The issue of resiliency has been identified as one of the big future challenges in the HPC world [3].

Current HPC execution environments do not provide the hosted parallel applications with a fault-free guarantee. Rather, developers need to specifically take appropriate actions in the presence of such faults (for example by using checkpoint-restart). Moreover, the most popular programming paradigm for HPC, MPI, assumes all interruptions, including single core failures, are fatal to the entire parallel application [4]. It has been identified that as systems grow, failure rates will reach a level that will render current resiliency models ineffective [5].

* This research received funding from the U.S. DoE under award No. DE-SC0002107; the European Community's FP7/2007-2013 Programme under grant agreement No. 317862; and used resources of the Oak Ridge Leadership Computing Facility at ORNL, which is supported by the U.S. DoE Office of Science under Contract No. DE-AC05-00OR22725.

We propose a step to alleviate these problems by providing a highly scalable clustering infrastructure for supporting a resiliency-aware elastic runtime and the hosted applications (see Fig. 1-A). The most important service a clustering infrastructure must provide is a membership service, which delivers to every node, or a select subset of the nodes, a view of the all the nodes that belong to the cluster. The membership service must detect failures of members, and support leave, join, and discovery functions. Although membership services were widely investigated, the characteristics of HPC pose unique requirements:

Ultra Large Scale: To deal with 1M nodes, we apply two principles: (1) use of a relaxed consistency model, namely eventual consistency, and (2) a hierarchical architecture.

Coupling: HPC workloads are mostly tightly coupled. That is, a single failure may cause a huge amount of other nodes to wait for recovery. Such parallel computations require *fast failure detection* in order for the failed computation to migrate to a new location with minimal disruption to the entire computation. To accommodate that we expedite failure notification to certain privileged “high priority monitors”.

Churn: Current HPC workloads are characterized by static, a-priori defined groups. Even programming models that theoretically permit dynamic membership [6], are currently implemented in a way that assumes static a-priori membership. Dynamic membership is primarily needed to deal with faults, yet opens up the possibility to dynamically shrink and expand a computation. The expected churn rate in HPC is lower than in Internet or data-center settings. However, the dense integration of modern machines increases the likelihood of correlated failures, where a failure of one component (e.g. IO-node) causes cascading failures (e.g. of compute nodes). This requires a membership service that tolerates concurrent failures without significant loss of performance.

Failure Model: Recent advancements in HPC resiliency support include CIFTS and MPI3 [7], which are based on a failure model assuming fail-stop failures, no network partitions, and a perfect failure detector. Our system takes it a step further by removing these restrictions on the failure model.

In order to support resilient and elastic HPC runtime and applications we expose several services: (1) a membership service, (2) an attribute replication service, and (3) group communication services [8]. The runtime will be able to take advantage of this suite of services in order to achieve, among other aspects, adequate tasks placement, scheduling, load balancing, migration, and performance monitoring.

In this paper we concentrate on the membership and the attribute replication services, which form the foundation on which other group communication services are built. The contribution of this paper is centered around the design, implementation, and evaluation of these services. The system was evaluated to reach a higher scale than known methods (see Sec. 6), while achieving good results both for massive start-up as well as for failure detection, while assuming a general non restrictive failure model. Using worst case analysis, a full implementation of these services was evaluated as effective for up to a million nodes. Both the architecture and the membership and attribute replication services present innovations that facilitate this achievement.

In the remainder of the paper, Sec. 2 provides an overview of the system architecture, while Sec. 3 & 4 dive deeper into the major components of the system. Section 5 details

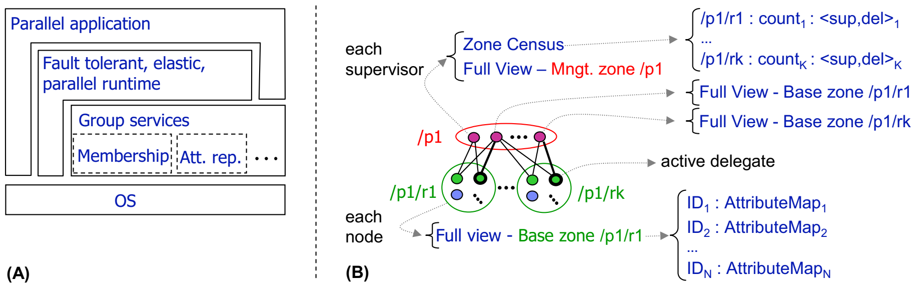

Parallel application
Fault tolerant, elastic, parallel runtime
Group services
Membership
Att. rep.
...
OS
(A)
each supervisor
Zone Census
Full View – Mngt. zone /p1
/p1/r1 : count1 : <sup,del>1
...
/p1/rk : countK : <sup,del>K
Full View - Base zone /p1/r1
Full View - Base zone /p1/rk
/p1
/p1/r1
/p1/rk
active delegate
each node
Full view - Base zone /p1/r1
ID1 : AttributeMap1
ID2 : AttributeMap2
...
IDN : AttributeMapN
(B)

Fig. 1. (A) The software stack of a fault tolerant, elastic parallel runtime. (B) A fault tolerant hierarchy with redundant connectivity, and the data structure distribution along the two levels.

a thorough evaluation of the system, related work is in Sec. 6, and concluding remarks are presented in Sec. 7.

2 System Architecture

The requirements and characteristics of HPC systems, detailed in the previous section, led us to a hierarchical peer-to-peer design (see Fig. 1-B). The basic building block of the system is that of a zone. Within a zone, full membership is maintained with an eventual consistency semantic (detailed in Sec. 3). In addition, an attribute replication service allows each node to write a key-value table, which is then replicated to all other nodes. This allows each node to communicate its state to every other node in the zone. This design scales to zones consisting of approximately a few thousand members. In order to achieve the 1M nodes target, the membership and attribute replication services are used as building blocks of a two layers hierarchy, composed of base-zones federated by a management-zone (see Fig. 1-B). On Blue Gene/P [9] for example, zone affiliation be can derived from the "personality" of each node, organizing the compute nodes of a rack into a base-zone, selecting IO-nodes for the management zone, and assigning co-located IO-nodes to the respective base-zones. Each base zone elects, using the attribute service, delegates to connect to the management zone's appropriate supervisor. An active delegate sends the supervisor updates about the base-zone membership. A supervisor shares information with its peers, using the attribute replication service, publishing the names of guarded base-zones, the number of active delegates per such zone, and the view size of each base-zone (see Fig. 1-B). This enables each member of the management zone to have a "system census", which is an up-to-date eventually consistent summary view of the entire cluster. In addition, if further details are needed, there are mechanisms in place which enable every management node to obtain more specific information regarding each base zone, such as the detailed full membership of that zone. Since components, including delegates, supervisors, and the links between them can fail, the hierarchy contains redundancy in each of these elements. Whenever a component fails, it is immediately and automatically replaced, so that the integrity of the hierarchy is maintained at all times.

In order to ensure fast failure detection, the active delegate in each zone serves as the zone's "high priority monitor" (HPM). Failure detection information is sent to the HPM node directly, although possibly using unreliable communication, in addition to the reliable yet relatively slow gossip-based dissemination mechanism. These notifications traverse the hierarchical structure as well. Thus, a single monitor or a hierarchy thereof can take fast actions such as re-spawning the failed computation.

3 Zone Membership

In this section we describe in detail the membership protocol implemented in both base and management zones. Each node n is associated with an *ID*, which carries the node name and network endpoints. Each *ID* is accompanied by a version $\mathit{Ver} = \langle inc, minor \rangle$, where: (1) inc identifies different incarnations of the same node, separated by crash failures or restarts, and is strictly increasing; and (2) $minor$ starts from 1 on every incarnation and is incremented by n according to the protocol described below. The local membership *View* is a map $\mathit{ID} \rightarrow \mathit{Ver}$, where $\mathit{View}_n[m]$ is the most current *Ver* of node m known to n. Changes to the view are delivered differentially: $\textsc{Notify-Join}(\langle p, v \rangle)$ is invoked when node p with version v joins the overlay or after a network partition heals; $\textsc{Notify-Leave}(\langle p, v \rangle)$ is invoked when $\langle p, v \rangle$ leaves, fails, or is behind a network partition. The semantic is that of eventual consistency: $\forall m, n$ that remain in the same group, $\mathit{View}_n = \mathit{View}_m$, some finite time after no more nodes join or leave, and the network is stable.

Bootstrap & History: When a node starts, it is given a set *B* of *ID*s to bootstrap from. The history map *H* contains the *ID*s of nodes which were recently removed from the *View*, along with their removal time and version when removed. If a node re-enters the view, it is removed from *H* (i.e. $H \cap \mathit{View} = \emptyset$). The history map is used to identify stale messages, that arrive after a node fails. In case the history map *H* grows beyond a certain limit, it is pruned by removing nodes older than some threshold.

Discovery: The discovery task selects target $m \in \{B \setminus \mathit{View} \cup H\}$ randomly, and sends a discovery-request message that contains its own *ID* and *Ver*, as well as a *boot* flag that encodes whether the target m was selected from *B* or from *H*. The discovery-reply consists of the full view of the target m, and the flag *boot*. For both request and reply, in case $\neg \mathit{boot}$, the receiver p performs $\mathit{Ver}_p.min \leftarrow \mathit{Ver}_p.min + 1$, and then processes each pair of $\langle \mathit{ID}, \mathit{Ver} \rangle$ from the message using $\textsc{ProcessAlive}(\langle \mathit{ID}, \mathit{Ver} \rangle)$ (see Alg. 1). This process both heals partitions and discovers new peers. The discovery targets are not kept as permanent neighbors. The discovery task is performed frequently at bootstrap, and as time passes its frequency decreases.

Topology: As the view begins to fill up, the topology component starts choosing and connecting to long-term neighbors. The topology built from the view has two ingredients: (1) a robust ring where each node is connected to K_s successors (ring order based on SHA1 of *ID*), and (2) K_r random neighbors. The ring ensures that eventually all failed nodes will be discovered by their predecessor(s). This ensures eventual completeness of the view [10]. The random nodes are selected according to a protocol which approximates a K_r-connected random graph [11], yielding a robust and well connected overlay.

Algorithm 1. Processing of Alive, Leave, and Suspicion events, at node n

```
procedure ProcessAlive(ID p, Ver v)
    if $(p \notin View_n) \wedge ((p \notin H_n) \vee (p \in H_n \wedge v > H_n[p].Ver))$ then    ▷ join
        $View_n[p] \leftarrow v$; remove $p$ from $H_n$; add $\langle p, v\rangle$ to $\mathbf{\Delta}.\mathcal{A}$; Notify-Join($\langle p,v\rangle$);
    if $(p \in View_n) \wedge (v > View_n[p])$ then    ▷ newer version
        add $\langle p, v\rangle$ to $\mathbf{\Delta}.\mathcal{A}$;
        for all $r \in S_n[p], S_n[p][r] < v$ do    ▷ prune refuted suspicions remove $r$ from $S_n[p]$;
        if $View_n[p].inc = v.inc$ then $View_n[p] \leftarrow v$;
        else    ▷ new incarnation
            Notify-Leave($\langle p, View_n[p]\rangle$); $View_n[p] \leftarrow v$; Notify-Join($\langle p,v\rangle$);
procedure ProcessLeave(ID p, Ver v)
    if $(p \in View_n) \wedge (v \geq View_n[p])$ then    ▷ in-view leave
        remove $p$ from $View_n$ and $S_n$; $H_n[p] \leftarrow \langle v, time\rangle$; add $\langle p, v\rangle$ to $\mathbf{\Delta}.\mathcal{L}$;
        NotifyLeave($\langle p,v\rangle$);
    if $(p \notin View_n) \wedge (p \in H_n) \wedge (v > H_n[p].Ver)$ then    ▷ out-of-view leave
        $H_n[p] \leftarrow \langle v, time\rangle$; add $\langle p, v\rangle$ to $\mathbf{\Delta}.\mathcal{L}$;
procedure ProcessSuspicion(ID r, ID s, Ver v)
    if $s = n$ then    ▷ refute suspicion on self
        $Ver_n.min \leftarrow Ver_n.min + 1$; add $\langle n, Ver_n\rangle$ to $\mathbf{\Delta}.\mathcal{A}$;
    else if $s \in View_n \wedge v \geq View_n[s]$ then
        if $r \notin S_n[s] \vee S_n[s][r] < v$ then    ▷ valid, new suspicion
            $S_n[s][r] \leftarrow v$; add $\langle r, s, v\rangle$ to $\mathbf{\Delta}.\mathcal{S}$;
            if $|S_n[s]| \geq \Theta$ then    ▷ enough evidence! correction for small views omitted
                remove $s$ from $View_n$ and $S_n$; $H[s] \leftarrow \langle v, time\rangle$; NotifyLeave($\langle s,v\rangle$);
```

Failure Detection and Orderly Leaves: Failure detection is based on neighbors monitoring each other using heartbeats. Node r creates a failure suspicion report on node s if (1) an established connection between r and s fails, or (2) a heartbeat timeout is reported on s, or (3) s is member of r's successor list, and a connect attempt from r to s fails. The last condition ensures the view's eventual completeness [10]. A suspicion report consists of the tuple $\langle r, s, v\rangle$ for the reporter's *ID*, the suspect's *ID*, and suspect's *Ver*. The reporter calls ProcessSuspicion$(r, s, View_r[s])$ in order to spread the report further (see Alg. 1). Adjacent to $View_n[m]$ is the suspicion repository 2D map $S_n[s][r] \rightarrow v$, which stores every unique suspicion report received. Note that suspicion reports with a version lower than the version of the suspect in the view are discarded. In order to decrease the false detection rate, a node is declared as "failed" only after the number of reporters suspecting the same node exceeds a threshold, Θ. To ensure eventual completeness, $K_s \geq \Theta$ must hold. When a node orderly leaves the overlay, it sends all its neighbors a leave message, which contains the node *ID* and *Ver* (and possibly an exit code). Upon reception leave messages are processed by ProcessLeave(*ID*, *Ver*) and added to the update database for further dissemination.

Membership Updates: Node membership information is disseminated over the $K_r + K_s$ long-term overlay links. When node n acquires a new neighbor m, it will send m a membership message that contains: (1) the entire $View_n$, and (2) all the current suspicions (all the tuples $\langle r, s, S_n[s][r]\rangle$). After the first "base-view message" a neighbor is sent only "update" messages, which differentially capture the difference from the base-view. The update-database $\boldsymbol{\Delta}$ contains the following three sets: (1) $\mathcal{A}$ – The $\langle ID, Ver\rangle$ of nodes on which fresh *Alive* information was received (may include the current node); (2) $\mathcal{S}$ – received suspicion reports ⟨r,s,v⟩ ; (3) $\mathcal{L}$ – The $\langle ID, Ver\rangle$ of received *Leave* messages. The update database starts empty and accumulates alive, leave, and suspicion events (see Alg. 1). A membership update message is sent to all neighbors after a configurable aggregation interval (τ) from the first such event that hits an empty $\boldsymbol{\Delta}$. After $\boldsymbol{\Delta}$ is sent to all the neighbors, it is cleared. Upon receiving a membership message (base or update), every $\langle p, v\rangle \in \mathcal{L}$, every $\langle p, v\rangle \in \mathcal{A}$, and every $\langle r, s, v\rangle \in \mathcal{S}$ is processed by Alg. 1 #10,#1,#16, respectively. This order minimizes the chance of notifying a false suspicion.

High Priority Monitoring: In the protocol described above, membership updates are expected to propagate to all nodes in time proportional to τ and the overlay diameter, which is $O(\log_{K_r} N)$ [11]. In many applications there is only a single or a small number of monitors that take decisions based on membership events. It is possible to decrease the monitor's failure detection time by sending the original suspicion reports directly to the monitor (e.g. using UDP), in addition to the reliable propagation mechanism described in Alg. 1. We allow a small number of selected nodes to declare themselves (automatically or programmatically) as monitors using the attribute replication service. This lets every node in the zone know who the monitors are. When node n suspects the failure of a neighbor m with version v, it will immediately send the monitors a membership message containing suspicion $\langle n, m, v\rangle$. This message is processed just like any other suspicion (see Alg. 1). A monitor will receive up to $K_r + K_s$ such messages on every failure.

4 Attribute Replication Service

Each node n has an attribute map A_n of key-value pairs it can write to. Each key-value pair ⟨k,t⟩ is associated with a version number $u \in \mathbb{N}$, such that newer values of the same key carry larger version number. The goal is to replicate the attribute map A_n of node n to all other nodes. Let us denote by $M_n(m)$ the map replica of node m, that node n holds. Thus, node n holds one map it can write to directly $M_n(n) \equiv A_n$, and read-only replicas of the maps of every other node $M_n(m)$, $\forall m \neq n$. Thus, the ultimate goal is to reach $M_n = M_m, \forall n, m$ (some finite time after the end of writes). When node n learns about the attribute changes of node p, it notifies the user by invoking NOTIFY-ATTCHANGE($M_n(p)$). The attribute dissemination protocol is inspired by Anti-Entropy (AE) protocols [12,13], where in every round a node reconciles its state with a randomly selected gossip peer. The disadvantage is that on every AE round, the entire membership of node n ($O(|View_n|)$) has to be transmitted. This step has to be repeated periodically even if no updates to the map were made, consuming bandwidth even in idle state. Moreover, running AE reconciliation with two peers in parallel carries the cost of potentially getting duplicate copies of the same data entries. In our topology each node has a stable set of neighbors, connected by reliable connections. Thus, remembering what was exchanged in the last round and transmitting just the difference can save a lot of bandwidth. We therefore designed an improved protocol, which: (1)

Algorithm 2. Attribute replication message handlers: received at n, sent from m

```
1: procedure UPON-ATTUPDATE($\Lambda\, update$)
2:     $\Lambda_n \leftarrow \emptyset$
3:     for all $\langle p, u\rangle \in update$ do
4:         if $u > M_n(p).u \wedge u > M_n(p).u\_pend$ then
5:             $\Lambda_n \leftarrow \Lambda_n \cup \langle p, M_n(p).u\rangle$; $M_n(p).\langle u\_pend, trg\_pend\rangle \leftarrow \langle u, m\rangle$;
6:     if $\Lambda_n \neq \emptyset$ then send AttRequest($\Lambda_n, false$) to $m$;
7: procedure UPON-ATTREQUEST($\Lambda\, request, push$)
8:     $AttData_n \leftarrow \emptyset$;
9:     for all $\langle p, u\rangle \in request$ do
10:        if $u < M_n(p).u$ then
11:            for all $\langle k, t, u'\rangle \in M_n(p) : u' > u$ do $AttData_n \leftarrow AttData_n \cup \langle p, k, t, u'\rangle$;
12:        else if $\neg push$ then $AttData_n \leftarrow AttData_n \cup \langle p, \perp, \perp, \perp\rangle$;
13:    send AttReply($AttData_n$) to $m$
14: procedure UPON-ATTREPLY(AttData $data$)
15:    $\Lambda_n \leftarrow \emptyset$;
16:    for all $d_m(p, k, t, u) \in data$ do
17:        if $u = \perp$ then
18:            $\Lambda_n \leftarrow \Lambda_n \cup \langle p, M_n(p).u\rangle$;
19:            $M_n(p).\langle u\_pend, trg\_pend\rangle \leftarrow \langle M_n(p).u, \emptyset\rangle$;
20:        else if $M_n(p)[k] = \emptyset \vee M_n(p)[k].u < u$ then
21:            $M_n(p)[k] \leftarrow \langle t, u\rangle$; $M_n(p).u \leftarrow \max(M_n(p).u, u)$;
22:    if $\Lambda_n \neq \emptyset$ then send AttRequest($\Lambda_n, true$) to all neighbors;
23:    for all $M_n(p).u > M_n(p).u\_notified$ do
24:        NOTIFY-ATTCHANGE($M_n(p)$); $M_n(p).u\_notified \leftarrow M_n(p).u$;
```

avoids sending full $O(|View_n|)$ sized messages in each reconciliation; (2) lets traffic reduce to zero when there are no writes and the overlay is stable; and (3) minimizes the reception of duplicate data updates.

An entry in the map is $A_n[k] = \langle t, u\rangle$. The map itself has a version number $A_n.u \in \mathbb{N}$ which starts at 0 when the map is empty. The map is written one key at a time. Every time a key is written the version number of the map is incremented, and the version of the corresponding $\langle k, t\rangle$ entry is set to $A_n.u$. Thus no two entries carry the same version, and $A_n.u$ equals the maximal version number in the map. This versioning scheme permits a protocol with *per source sequential consistency*, meaning that writes to A_n are delivered in the same order in every other node p. The tables $M_n(p)$ are augmented with the following fields:

1. $M_n(p).u = \max\{u : \langle k, t, u\rangle \in M_n(p)\}$ – the last version of A_p known to n;
2. $M_n(p).u\_sent$ – the version sent by n to all its neighbors during the last round;
3. $M_n(p).pend\_trg$ – the neighbor *ID* to which a request was sent;
4. $M_n(p).u\_pend$ – a version known to $M_n(p).pend\_trg$ to which a request was sent;
5. $M_n(p).u\_notified$ - the last version delivered to the application.

In every round, node n will reconcile its state with its overlay neighbors, so that eventually $M_n = M_m$ for every neighbor m. Let a digest Λ_n be a list of identifier and table version pairs $\langle p, M_n(p).u\rangle$; and let *AttData*$_n$ be list data entries $d_n(p, k, t, u)$, where each entry is a single key-value-version tuple from $M_n(p)$. The reconciliation protocol has three stages – *Update*, *Request*, and *Reply*. (1) In the *Update* stage, node n will prepare, at configurable periodic intervals, a differential digest of its state, containing tables that changed since the last round: $\Lambda_n \leftarrow \{\langle p, M_n(p).u\rangle : M_n(p).u > M_n(p).u\_sent\}$. If $\Lambda_n \neq \emptyset$, it will be sent to all n's neighbors, and the tables will be marked as sent: $\forall\langle p, u\rangle \in \Lambda_n, M_n(p).u\_sent \leftarrow u$. (2) In the *Request* stage (Alg. 2 #1), after processing an update digest from m, node n sends m a request containing a digest of its parts of the state that are less recent than those of m. The validity of the request w.r.t. the preceding update is given special care (Alg. 2 #12,17-19). (3) In the *Reply* stage (Alg. 2 #7), node n sends to node m the data entries that are more recent than what node m declared it knows and needs. (4) Finally, when node n receives the reply (Alg. 2 #14), it merges the incoming data into the existing tables, and notifies the application on the respective attribute changes.

When node n acquires a new neighbor m, the full digest $\Lambda_n \leftarrow \{\langle p, M_n(p).u\rangle, \forall p \in View_n\}$ is sent to m. In case a neighbor m disconnects (fails, leaves, or changes neighbors), its *ID* is searched in all the $M_n(p).pend\_trg$ fields. If found, it means that a request sent to it will not be answered. Thus, the pending request (i.e. $\langle p, M_n(p).u\rangle$) will be resent to all neighbors. The attribute map of a node is valid to other nodes only when the node is "alive". Thus, when a node p leaves the overlay, all $M_n(p), \forall n \neq p$, are deleted. When a node joins the overlay, an empty replica is initialized in all other nodes. As it acquires new neighbors, the joining node will push its state digest to its neighbors, and vice versa. Key deletion is translated into a write $A_n[k] \leftarrow \langle\perp, A_n.u + 1\rangle$, which is a kind of "death-certificate" for the key (see [12]).

5 Evaluation

Developing hardware and software for much larger systems than presently available has always posed difficult challenges for those who must assess performance before full scale measurements are possible. Our design lends itself to encapsulated component performance testing even though supercomputers with one million nodes do not exist yet. We divide our entire system into three components: management zone, base zones, and the communication links between them (see Fig. 1-B). We isolated the required relevant performance metrics for each component. Then we are able to devise tests for each component at the required performance to achieve successful systems of one million nodes. We fully implemented the hierarchical membership and attribute services in C++. Our test bed was a rack of Blue Gene/P[1]. We used regular Linux on compute nodes, rather than CNK[2]. The version we used provides full TCP/IP functionality on all the networks, including the torus. We set out to test whether our system is up to the task of managing the target scale by first testing a single zone to its full scale. Then we test a hierarchical system that has the full number of base zones and management nodes although with "stub" base zones. Each "stub" base zone is represented by a single node, which injects in to the management zone the same traffic as a full base-zone. We used

[1] Which includes 1024/64 compute/IO nodes, 32 bit integer, 850MHz, 4GB RAM [9].

[2] Version 2.6.29.1 with IBM modifications for Blue Gene/P, available: `http://git.anl-external.org/bg-linux.repos/linux-2.6.29.1-BGP.git/`

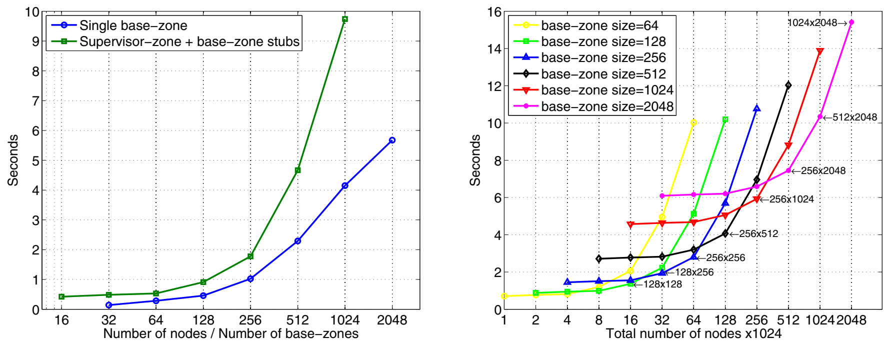


Fig. 2. Left: Boot time of (1) a single base zone with a variable number of nodes, (2) a management zone with a variable number of small base zones, one supervisor per base zone. **Right**: Projected boot time of a complete system as a function of total size and base zone size. Arrows indicate the optimal configuration for each size.

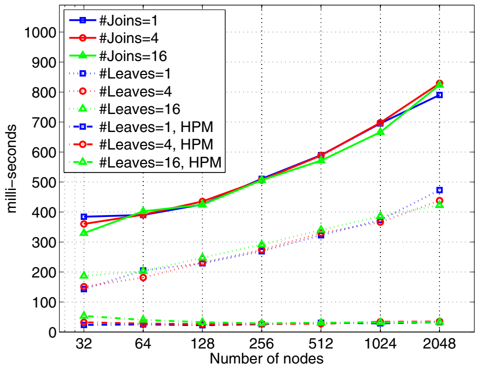


Fig. 3. Average time for a view with nodes joining and leaving, as a function of zone size and number of concurrent joins/leaves

$\tau = 200\text{ms}$, $\Theta = 1$, $K_r = 3$, $K_s = 1$ in all the experiments. The metrics chosen for measuring our system‘s performance are biased towards the HPC use case. The main difference between this use case and traditional (Internet and data-center) scenarios is the way in which the system is brought up, the frequency of nodes joining and leaving (churn), and the coupling between the nodes.

5.1 Boot Time

Unlike conventional Internet-scale or data-center based systems in which the nodes gradually join until the system gains size, a supercomputer or a partition of a supercomputer, usually boots all its nodes at once. Thus we want to make sure that the time it takes to form a stable view upon boot is reasonable, in line with the time it takes for the other processes that happen during boot (daemon startup, file system mount, etc).

First, we measure the time to a stable view of a single base zone, versus the number of nodes (32-2048), assuming all the nodes boot at once (Fig. 2-Left). Results indicate that a 2048-node zone yields a stable view in $T_{Base}^{Boot}(n = 2048) \approx 5.7$s (we used 2 processes per machine, with 1 process per machine results are approximately $\approx 30\%$ better). This tests the performance of the membership protocol. The results indicate that the boot time for a single zone is linear in the number of nodes. This is to be expected because every node has to get at least a single "alive" indication from every other node, directly or indirectly. Next, we measure the time to a stable view of a 2-layer setup with the number of base zones increasing from 16 to 1024, with 1 nodes in each base zone serving as a stub, and 1 supervisor per base zone (Fig. 2-Left). Results indicate that a 1024-node supervisor zone with 1024-base zones (1 stub node) boots in $T_{Sup}^{Boot}(m = 1024, n = 1) \approx 9.7$s. The boot stabilization time of the management zone includes: (1) stabilization of the management zone membership view and topology, (2) connecting with the base zone delegates, (3) receiving the views of the base-zones, and (4) distributing the summaries to all the members of the management zone using the attribute service. This takes longer than the stabilization time of a base zone with the same number of nodes, and shows linear scaling as well. These two measurements allow us to make a worst-case estimate of the full system stabilization time, for different combinations of management zone size (m) and base zone size (n), by making the following worst case assumptions: (1) that the stabilization time of a stub base zone is negligible, and (2) that the management zone starts after the base zones have stabilized. This results in $T_{Sup}^{Boot}(m, n) \lessapprox T_{Sup}^{Boot}(m, 1) + T_{Base}^{Boot}(n)$. Figure 2-Right shows that a 1M system with 2048-node base-zones and a 512-node management zone would boot in $\approx T_{Sup}^{Boot}(512, 1) + T_{Base}^{Boot}(2048) \sim 10.3$s. Figure 2-Right also shows what would be the optimal configuration of a management- and base-zones, for every system size, in terms of boot time.

5.2 Leave-Join Performance

We evaluate the leave-join performance by first booting a zone, and then forcing a number of nodes to fail concurrently. The victim nodes concurrently rejoin after a while. In both cases we measure the average time to a stable view; in case of leaves this includes failure detection time. The time to stable view (Fig. 3, top 6 curves) measures the propagation time of concurrent membership changes using the normal membership and attribute dissemination protocol. Membership stabilization time follows a logarithmic relation with zone size, since our topology creates a graph with logarithmic diameter and average path length. (We verified that $\forall N$ the diameter is $\leq \log_{K_r} N$). The number of nodes leaving or joining has hardly any effect, since as long as the ratio of transient nodes to total zone size is not too high, the topology retains its desirable robust "logarithmic" features (diameter, average path length) in the face of churn [11]. Join events take exactly 1 aggregation delay (τ) longer than leaves, since the joining node takes one round to discover peers before it spreads its identity, in an attempt to build a "good" topology right away. Membership events are propagated as node-census attribute events in the supervisor zone. The propagation time follows the same rule as in base zones, since it is influenced by the (identical) topology of the overlay and the aggregation time. Therefore we can estimate the stabilization time of a full system, as in Sec. 5.1, by adding the stabilization times of the base- and supervisor-zones, according to their respective sizes. For example, in a 1M node system with a 512/2048 supervisor/base-zone configuration, a leave event would propagate to all the supervisors in $\approx$700ms.

5.3 High Priority Monitoring

Figure 3 (bottom 3 curves) displays the time it takes an HPM node to reach a stable view after leave events (same experiment as above). The delay is almost independent of zone size, and is $\approx$30ms for almost all cases. The round-trip delay between any node in Blue Gene/P is below 1ms; we therefore conclude that this time is mainly failure detection time, which is independent of size. When the ratio of failed nodes to the zones size is too large (e.g. 16 out of 32), detection time grows because of the likelihood that some failed node X would have all its neighbors failing as well. The failure of node X is then discovered by some surviving node that tries to connect to it as a successor, and fails.

6 Related Work

Membership services present a wide spectrum of semantics [14], which vary from consistent views like Virtual Synchrony [15], to eventual consistency [16] as implemented in Cassandra [17], and to partial views either randomized as in SCAMP [18] or structured as in Chord [19]. Scalability increases as semantics become less strict, from hundreds in VS, thousands in eventual consistency, and tens of thousands for partial views. An important class of these services relies on "gossip" protocols [20], where peers periodically exchange parts of their state with a random selection of peers. Gossip based protocols are extremely robust. However, they are slow to detect failures [10], and generate traffic even when no membership changes occur. Overlay networks in which peers have stable connections retain similar robustness by choosing peers such that the resulting overlay network remains well connected in the face of failures, as is Araneola [11] and Symphony [21]. The advantages of stable peers are (1) efficient distributed failure detection [10,22]; (2) the ability to minimize "OS jitter" [23]; and (3) the ability to implement additional functions like a publish-subscribe service [8], and a key-value store [19,17]. Hierarchical membership schemes were proposed by HiSCAMP [24], where the focus is on a dynamic self organizing hierarchy, and more recently by Census [25] where the focus is on self-organization reflecting the geographic distribution of the nodes, and on delivering consistent views. Our implementation focuses on scalability and is an order of magnitude greater than Census and HiScamp (1M vs. 10K and 50K, resp.); and fast failure detection, which at 1M nodes is less than 100ms for the HPM and takes around 800ms to form a consistent view (Census [25] chooses to provide a consistent view every 30 seconds for a system of 10K nodes). An early attempt to develop a membership service specifically for HPC introduced a flat tree-based membership algorithm for MPI environments, and was evaluated only up to 1024 nodes [4]. More recently the CIFTS project (e.g [7]) was devoted to fault-tolerance in HPC systems. Our approach adopts a much more general failure model than the one adopted by CIFTS, and therefore uses a different overlay topology (expander vs. tree) and different algorithms.

7 Conclusions

We demonstrated that membership services can scale effectively to upcoming HPC system sizes, supporting continuous availability, for a next generation of HPC run-time support, and system administration. We believe that ExaScale size deployments can be made resiliency aware by employing this work. We are currently working to integrate our membership and attribute replication services with Charm++ [6], in order to demonstrate a true fault tolerant, elastic, parallel runtime.

References

1. Kogge, P.M., Dysart, T.J.: Using the TOP500 to trace and project technology and architecture trends. In: SC 2011(2011)
2. Sarkar, V., et al.: ExaScale Software Study: Software Challenges in Extreme Scale Systems. Technical report, DARPA IPTO, AFRL (September 2009)
3. Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., Snir, M.: Toward Exascale Resilience. Int. J. High Perform. Comput. Appl. 23(4), 374–388 (2009)
4. Varma, J., Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: Scalable, fault tolerant membership for MPI tasks on HPC systems. In: ICS 2006 (2006)
5. Ferreira, K., Stearley, J., Laros, J.H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D.: Evaluating the viability of process replication reliability for exascale systems. In: SC 2011 (2011)
6. Charm++ website (2012), http://charm.cs.uiuc.edu/
7. Buntinas, D.: Scalable distributed consensus to support MPI fault tolerance. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 325–328. Springer, Heidelberg (2011)
8. Tock, Y., Mandler, B., Moreira, J., Jones, T.: Poster: scalable infrastructure to support supercomputer resiliency-aware applications and load balancing. In: SC 2011 Companion, pp. 9–10 (2011)
9. IBM-Blue-Gene-Team: Overview of the IBM Blue Gene/P project. IBM J. of Res. and Dev. 52(1/2), 199–220 (2008)
10. Gupta, I., Chandra, T.D., Goldszmidt, G.S.: On scalable and efficient distributed failure detectors. In: PODC 2001, pp. 170–179 (2001)
11. Melamed, R., Keidar, I.: Araneola: a scalable reliable multicast system for dynamic environments. In: NCA 2004, pp. 5–14 (2004)
12. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible Update Propagation for Weakly Consistent Replication. In: SOSP 1997, pp. 288–301 (1997)
13. van Renesse, R., Dumitriu, D., Gough, V., Thomas, C.: Efficient reconciliation and flow control for anti-entropy protocols. In: LADIS 2008 (2008)
14. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Computing Surveys 33(4), 427–469 (2001)
15. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: SOSP 1987 (1987)
16. van Renesse, R., Minsky, Y., Hayden, M.: A gossip-style failure detection service. In: Middleware 1998, pp. 55–70 (1998)
17. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (2010)
18. Ganesh, A., Kermarrec, A.M., Massoulie, L.: Peer-to-Peer Membership Management for Gossip-Based Protocols. IEEE Trans. Comput. 52(2), 139–149 (2003)

19. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM 2001, pp. 149–160 (2001)
20. Kermarrec, A.M., van Steen, M.: Gossiping in distributed systems. SIGOPS Oper. Syst. Rev. 41(5), 2–7 (2007)
21. Manku, G.S., Bawa, M., Raghavan, P.: Symphony: distributed hashing in a small world. In: USITS 2003: Proc. USENIX Symp. on Internet Tech. and Sys., vol. 4, pp. 10–23 (2003)
22. Zhuang, S.Q., Geels, D., Stoica, I., Katz, R.H.: On failure detection algorithms in overlay networks. In: INFOCOM 2005, vol. 3, pp. 2112–2123 (2005)
23. Jones, T.: Linux Kernel Co-Scheduling and Bulk Synchronous Parallelism. The Int'l J. of High Performance Computing Applications 26(2), 136–145 (2012)
24. Ganesh, A.J., Kermarrec, A.M., Massoulié, L.: HiScamp: self-organizing hierarchical membership protocol. In: SIGOPS 2002 European Workshop, pp. 133–139 (2002)
25. Cowling, J., Ports, D.R.K., Liskov, B., Popa, R.A., Gaikwad, A.: Census: Location-Aware Membership Management for Large-Scale Distributed Systems. In: USENIX (2009)

Topic 8: Distributed Systems and Algorithms (Introduction)

Achour Mostefaoui, Andreas Polze, Carlos Baquero,
Paul Ezhilchelvan, and Lars Lundberg

Topic Committee

Distributed Computing is becoming more and more led by technological and application advances. Many works consider new computing models compared to the classical closed model with a fixed number of participants and strong hypothesis on communication and structuration. Indeed, it is hard to imagine some application or computational activity and process that falls outside Distributed Computing. Internet and the web (e.g. social networks, clouds) are becoming the main application field for distributed computing. In addition to the classical challenges that developers have to face (asynchrony and failures) they have to deal with load balancing, malicious and selfish behaviors, mobility, heterogeneity and the dynamic nature of participating processes.

Topic 8 of Euro-Par (Distributed Systems and Algorithms) makes a good mix between research and development. Papers submitted to Topic 8 gave a good overview of the spectrum of Distributed Systems. They focussed on a range of interesting research areas, such as web oriented applications, data managements (data bases) and fault-tolerance. The accepted papers also represent this diverse research landscape, thus making Topic 8 of Euro-Par a good forum to discuss both novel approaches and connections between sub-areas of research in Distributed Systems.

This year five papers were accepted. The paper “Gunther: Search-Based Autotuning of MapReduce”, by Guangdeng Liao, Kushal Datta and Theodore Willke proposes a novel approach to parameter optimization in Hadoop clusters based on global search algorithms. Namely, the authors present a tool (Gunther) for automatically finding suitable values for some of the more than 200 configurable parameters in Hadoop. Their approach uses trial execution of Hadoop applications and based on these trial executions they use Genetic Algorithms for finding suitable parameter values.

The paper “Multi-criteria checkpointing strategies: optimizing response-time versus resource utilization”, by Aurelien Bouteiller, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault and Yves Robert, discusses the optimization of system utilization during exhaustive checkpoint rollback operations, with a specific focus on Exascale computing. In uncoordinated checkpointing, when one process rolls-back due to exception, the other cooperating processes need not; however they can be blocked from making progress until the rollback and subsequent restoration are complete. The presented work prevents this by running another application in the meantime so that resource usage is maximized.

The paper “Efficient event prewarning for sensor networks with multi microenvironments”, by Yinglong Li, describes an approach for structuring sensor networks into communication clusters for better performance behavior. To this respect, it proposes an algorithm that eliminates erroneous data by identifying outliers. The paper also proposes a spatial correlation algorithm for generating prewarnings as a function of the location where it was sensed and assesses its efficiency by comparing it with an idealized protocol. In order to improve the quality of these prewarnings the authors suggest spacial correlation analysis.

The paper “On the Scalability of Snapshot Isolation”, by Masoud Saeida Ardekani, Pierre Sutra, Nuno Preguica and Marc Shapiro, presents an impossibility result, namely, the proof that genuine partial replication (GPR) cannot be achieved whilst guaranteeing snapshot isolation (SI) consistency as strategies for transaction performance improvements. For this purpose, the authors prove that snapshot isolation (a consistency property) can be broken down into four properties. Then, they show that two of these properties conflict with the property genuine partial replication (GPR). This means that SI and GPR can not be obtained in the same system.

Finally, the paper “Efficient Parallel Block-Max WAND Algorithm”, by Veronica Gil Costa, Oscar Rojas and Mauricio Marin, proposes different ways of increasing the performance of web searches (top-k service) through parallel execution on a Master/Slave architecture. The authors consider three different approaches: a distributed two-stage algorithm, and two multi-threaded algorithms using different synchronization schemes (called local and shared heap). The two-stage algorithm is targeted for clusters of distributed computers and the multi-threaded versions are targeted for multi-core computers. This means that the distributed approach and the multi-threaded approaches could be used together if one is using clusters of multiprocessors.

We would like to take this opportunity to thank all authors who submitted their work to Topic 8 of Euro-Par 2013, all external referees who assisted us, and all people involved in organizing the review process for their hard work.

On the Scalability of Snapshot Isolation[⋆]

Masoud Saeida Ardekani[1], Pierre Sutra[2], Marc Shapiro[3], and Nuno Preguiça[4]

[1] Université Pierre et Marie Curie, Paris, France
[2] University of Neuchâtel, Switzerland
[3] INRIA & Université Pierre et Marie Curie, Paris, France
[4] CITI, Universidade Nova de Lisboa, Lisbon, Portugal

Abstract. Many distributed applications require transactions. However, transactional protocols that require strong synchronization are costly in large scale environments. Two properties help with scalability of a transactional system: genuine partial replication (GPR), which leverages the intrinsic parallelism of a workload, and snapshot isolation (SI), which decreases the need for synchronization. We show that under standard assumptions (data store accesses are not known in advance, and transactions may access arbitrary objects in the data store), it is impossible to have both SI and GPR. Our impossibility result is based on a novel decomposition of SI which proves that, like serializability, SI is expressible on plain histories.

1 Introduction

Large scale transactional systems have conflicting requirements. On the one hand, strong transactional guarantees are fundamental to many applications. On the other, remote communication and synchronization are costly and should be avoided.[1]

To maintain strong consistency guarantees while alleviating the cost of synchronization, Snapshot Isolation (SI) is a popular approach in both distributed database replications [1–3], and software transactional memories [4, 5]. Under SI, a transaction accesses its own *consistent snapshot* of the data, which is unaffected by concurrent updates. A read-only transaction always commits unilaterally and without synchronization. An update transaction synchronizes on commit to ensure that no concurrent conflicting transaction has committed before it.

Our first contribution is to prove that SI is equivalent to the conjunction of the following properties: *(i)* no cascading aborts, *(ii)* strictly consistent snapshots, i.e., a transaction observes a snapshot that coincides with some point in (linear) time, *(iii)* two concurrent write-conflicting update transactions never both

[⋆] This work is partially supported by FCT/MCT projects PEst-OE/EEI/UI0527/2011 and PTDC/EIA-EIA/108963/2008, and the European Commission's Seventh Framework Program (FP7) under grant agreement No. 318809 (LEADS).

[1] We address general-purpose transactions, i.e., we assume that a transaction may access any object in the system, and that its read- and write-sets are not known in advance.

commit, and *(iv)* snapshots observed by transactions are monotonically ordered. Previous definitions of SI [6, 7] extend histories with abstract snapshot points. Our decomposition shows that in fact, like serializability, SI can be defined on plain histories [8].

Modern data stores replicate data for both performance and availability. Full replication does not scale, as every process must perform all updates. *Partial replication* (PR) aims to address this problem, by replicating only a subset of the data at each process. Thus, if transactions would communicate only over the minimal number of replicas, synchronization and computation overhead would be reduced. However, in the general case, the overlap of transactions cannot be predicted; therefore, many PR protocols perform system-wide global consensus [2, 3] or communication [9]. This negates the potential advantages of PR; hence, we require *genuine* partial replication [10] (GPR), in which a transaction communicates only with processes that replicate some object accessed in the transaction. With GPR, independent transactions do not interfere with each other, and the intrinsic parallelism of a workload can be thus exploited.

Our second contribution is to show that SI and GPR are incompatible. More precisely, we prove that an asynchronous message-passing system supporting GPR, even if it is failure-free, cannot compute monotonically ordered snapshots, nor strictly consistent ones.

This paper proceeds as follows. We introduce our system model in Section 2. Section 3 presents our decomposition of SI. Section 4 shows that GPR and SI are mutually incompatible. We discuss implications of this result and related work in Section 5. Section 6 closes this paper. Due to space constraints, some proofs are deferred to our companion technical report [11].

2 Model

This section defines the elements in our model and formalizes SI and GPR .

2.1 Objects and Transactions

Let *Objects* be a set of objects, and $\mathcal{T}$ be a set of transaction identifiers. Given an object x and an identifier i, x_i denotes *version* i of x. A *transaction* $T_{i \in \mathcal{T}}$ is a finite permutation of read and write operations followed by a *terminating* operation, commit (c_i) or abort (a_i). We use $w_i(x_i)$ to denote transaction T_i writing version i of object x, and $r_i(x_j)$ to mean that T_i reads version j of object x. In a transaction, every write is preceded by a read on the same object, and every object is read or written at most once.[2] We note $ws(T_i)$ the write set of T_i, i.e., the set of objects written by transaction T_i. Similarly, $rs(T_i)$ denotes the read set of transaction T_i. The *snapshot* of T_i is the set of versions read by T_i. Two transactions *conflict* when they access the same object and one of them modifies it; they *write-conflict* when they both write to the same object.

[2] These restrictions ease the exposition of our results but do not change their validity.

2.2 Histories

A *complete history* h is a partially ordered set of operations such that (1) for every operation o_i appearing in h, transaction T_i terminates in h, (2) for every two operations o_i and o'_i appearing in h, if o_i precedes o'_i in T_i, then $o_i <_h o'_i$, (3) for every read $r_i(x_j)$ in h, there exists a write operation $w_j(x_j)$ such that $w_j(x_j) <_h r_i(x_j)$, and (4) any two write operations over the same objects are ordered by $<_h$. A *history* is a prefix of a complete history. For some history h, order $<_h$ is the *real-time order* induced by h. Transaction T_i is *pending* in history h if T_i does not commit, nor abort in h. We note $\ll_h$ the version order induced by h between different versions of an object, i.e., for every object x, and any two transactions T_i and T_j, $x_i \ll_h x_j = w_i(x_i) <_h w_j(x_j)$. Following Bernstein et al. [12], we depict a history as a graph. We illustrate this with history h_1 below in which transaction T_a reads the initial versions of objects x and y, while transaction T_1 (respectively T_2) updates x (resp. y).[3]

$$\begin{tikzcd}
h_1 = r_a(x_0) \arrow[r] \arrow[dr] & r_1(x_0).w_1(x_1).c_1 & \\
& r_a(y_0).c_a \arrow[r] & r_2(y_0).w_2(y_2).c_2
\end{tikzcd}$$

When order $<_h$ is total, we shall write a history as a permutation of operations, e.g., $h_2 = r_1(x_0).r_2(y_0).w_2(y_2).c_1.c_2$.

2.3 Snapshot Isolation

Snapshot isolation (SI) was introduced by Berenson et al. [8], then later generalized under the name GSI by Elnikety et al. [7]. In this paper, we make no distinction between SI and GSI.

Let us consider a function $\mathcal{S}$ which takes as input a history h, and returns an extended history h_s by adding a *snapshot point* to h for each transaction in h. Given a transaction T_i, the snapshot point of T_i in h_s, denoted s_i, precedes every operation of transaction T_i in h_s. A history h is in SI if, and only if, there exists a function $\mathcal{S}$ such that $h_s = \mathcal{S}(h)$ and h_s satisfies the following rules:

D1 (Read Rule)

$$\begin{aligned}
&\forall r_i(x_{j \neq i}), w_{k \neq j}(x_k), c_k \in h_s : \\
&\quad c_j \in h_s && (D1.1)\\
&\quad \wedge\ c_j <_{h_s} s_i && (D1.2)\\
&\quad \wedge\ (c_k <_{h_s} c_j \vee s_i <_{h_s} c_k) && (D1.3)
\end{aligned}$$

D2 (Write Rule)

$$\begin{aligned}
&\forall c_i, c_j \in h_s : \\
&\quad ws(T_i) \cap ws(T_j) \neq \varnothing \\
&\quad \Rightarrow (c_i <_{h_s} s_j \vee c_j <_{h_s} s_i)
\end{aligned}$$

2.4 System

We consider a message-passing distributed system of n processes $\Pi = \{p_1, \ldots, p_n\}$. We shall define our synchrony assumptions later. Following Fischer et al. [13], an execution is a sequence of steps made by one or more processes. During

[3] Throughout the paper, read-only transactions are specified with an alphabet subscript, and update transactions are shown with numeric subscript.

an execution, processes may fail by crashing. A process that does not crash is said *correct*; otherwise it is *faulty.* We note $\mathfrak{F}$ the refinement mapping [14] from executions to histories, i.e., if ρ is an execution of the system, then $\mathfrak{F}(\rho)$ is the history produced by ρ. A history h is *acceptable* if there exists an execution ρ such that $h = \mathfrak{F}(\rho)$. We consider that given two sequences of steps U and V, if U precedes V in some execution ρ, then the operations implemented by U precedes (in the sense of $<_h$) the operations implemented by V in the history $\mathfrak{F}(\rho)$.[4]

2.5 Partial Replication

A data store $\mathcal{D}$ is a finite set of tuples (x, v, i) where x is an object (data item), v a value, and $i \in \mathcal{T}$ a version. Each process in Π holds a data store such that initially every object x has version x_0. For an object x, *Replicas*(x) denotes the set of processes, or *replicas*, that hold a copy of x. By extension for some set of objects X, *Replicas*(X) denotes the replicas of X; given a transaction T_i, *Replicas*(T_i) equals *Replicas*$(rs(T_i) \cup ws(T_i))$.

We make no assumption about how objects are replicated. The coordinator of T_i, denoted *coord*(T_i), is in charge of executing T_i on behalf of some client (not modeled). The coordinator does not know in advance the read set or the write set of T_i. To model this, we consider that every prefix of a transaction (followed by a terminating operation) is a transaction with the same id.

Genuine Partial Replication (GPR) aims to ensure that, when the workload is parallel, throughput scales linearly with the number of nodes [10]:

– **GPR.** For any transaction T_i, only processes that replicate objects accessed by T_i make steps to execute T_i.

2.6 Progress

The read rule of SI does not define what is the snapshot to be read. According to Adya [6], "transaction T_i's snapshot point needs not be chosen after the most recent commit when T_i started, but can be selected to be some (convenient) earlier point." To avoid that read-only transactions always observe outdated data, we add the following rule:

– **Non-trivial SI.** Consider an acceptable history h and a transaction T_i pending in h such that the next operation invoked by T_i is a read on some object x. Note x_j the latest committed version of x prior to the first operation of T_i in h. Let ρ be an execution satisfying $\mathfrak{F}(\rho) = h$. If $h.r_i(x_j)$ belongs to SI then *there exists* an execution ρ' extending ρ such that in history $\mathfrak{F}(\rho')$, transaction T_i reads at least (in the sense of $\ll_h$) version x_j of x.

In addition, we consider that the system provides the following progress guarantees on transactions:

[4] Notice that since steps to implement operations may interleave, $<_h$ is not necessarily a total order.

- **Obstruction-Free Updates.** For every update transaction T_i, if $coord(T_i)$ is correct then T_i eventually terminates. Moreover, if T_i does not write-conflict with some concurrent transaction, then T_i eventually commits.

- **Wait-Free Queries.** If $coord(T_i)$ is correct and T_i is a read-only transaction, then transaction T_i eventually commits.

3 Decomposing SI

This section defines four properties, whose conjunction is necessary and sufficient to attain SI. We later use these properties in Section 4 to derive our impossibility result.

3.1 Cascading Aborts

Intuitively, a read-only transaction must abort if it observes the effects of an uncommitted transaction that later aborts. By guaranteeing that every version read by a transaction is committed, rules D1.1 and D1.2 of SI prevent such a situation to occur. In other words, these rules *avoid cascading aborts.* We formalize this property below:

Definition 1 (Avoiding Cascading aborts). *History h avoids cascading aborts, if for every read $r_i(x_j)$ in h, c_j precedes $r_i(x_j)$ in h. ACA denotes the set of histories that avoid cascading aborts.*

3.2 Consistent and Strictly Consistent Snapshots

Consistent and strictly consistent snapshots are defined by refining causality into a dependency relation as follows:

Definition 2 (Dependency). *Consider a history h and two transactions T_i and T_j. We note $T_i \rhd T_j$ when $r_i(x_j)$ is in h. Transaction T_i depends on transaction T_j when $T_i \rhd^* T_j$ holds.*[5] *Transaction T_i and T_j are independent if neither $T_i \rhd^* T_j$, nor $T_j \rhd^* T_i$ hold.*

This means that a transaction T_i depends on a transaction T_j if T_i reads an object modified by T_j, or such a relation holds by transitive closure. To illustrate this definition, consider history $h_3 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).c_a.r_b(y_0).c_b$. In h_3, transaction T_a depends on T_1. Notice that, even if T_1 causally precedes T_b, T_b does not depend on T_1 in h_3.

We now define consistent snapshots with the above dependency relation. A transaction sees a consistent snapshot iff it observes the effects of all transactions it depends on [15]. For example, consider the history $h_4 = r_1(x_0).w_1(x_1).c_1.r_2(x_1)$ $.r_2(y_0).w_2(y_2).c_2.r_a(y_2).r_a(x_0).c_a$ In this history, transaction T_a does not see a

[5] We note $\mathcal{R}^*$ the transitive closure of some binary relation $\mathcal{R}$.

consistent snapshot: T_a depends on T_2, and T_2 also depends on T_1, but T_a does not observe the effect of T_1 (i.e., x_1). Formally, consistent snapshots are defined as follows:

Definition 3 (Consistent snapshot). *A transaction T_i in a history h observes a consistent snapshot iff, for every object x, if (i) T_i reads version x_j, (ii) T_k writes version x_k, and (iii) T_i depends on T_k, then version x_k is followed by version x_j in the version order induced by h ($x_k \ll_h x_j$). We write $h \in CONS$ when all transactions in h observe a consistent snapshot.*

SI requires that a transaction observes the committed state of the data at some *point* in the past. This requirement is stronger than consistent snapshot. For some transaction T_i, it implies that (SCONSa) there exists a snapshot point for T_i , and (SCONSb) if transaction T_i observes the effects of transaction T_j, it must also observe the effects of all transactions that precede T_j in time. A history is called strictly consistent if both SCONSa and SCONSb hold.

To illustrate this, consider the following history: $h_5 = r_1(x_0).w_1(x_1).c_1.r_a(x_1)$ $.r_2(y_0).w_2(y_2).c_2.r_a(y_2).c_a$. Because $r_a(x_1)$ precedes c_2 in h_5, y_2 cannot be observed when T_a takes its snapshot. As a consequence, the snapshot of transaction T_a is not strictly consistent. This issue is disallowed by SCONSa. Now, consider history $h_6 = r_1(x_0).w_1(x_1).c_1.r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2).c_a$. Since c_1 precedes c_2 in h_6 and transaction T_a observes the effect of T_2 (i.e., y_2), it should also observe the effect of T_1 (i.e., x_1). SCONSb prevents history h_6 to occur.

Definition 4 (Strictly consistent snapshot). *Snapshots in history h are strictly consistent, when for any committed transactions T_i, T_j, $T_{k \neq j}$ and T_l, the following two properties hold:*

- $\forall r_i(x_j), r_i(y_l) \in h : r_i(x_j) \not<_h c_l$ $\qquad$ (*SCONSa*)
- $\forall r_i(x_j), r_i(y_l), w_k(x_k) \in h :$
  $c_k <_h c_l \Rightarrow c_k <_h c_j$ $\qquad$ (*SCONSb*)

We note SCONS the set of strictly consistent histories.

3.3 Snapshot Monotonicity

In addition, SI requires what we call monotonic snapshots. For instance, although history h_7 below satisfies SCONS, this history does not belong to SI. Indeed, since T_a reads $\{x_0, y_2\}$, and T_b reads $\{x_1, y_0\}$, there is no extended history that would guarantee the read rule of SI.

$$\begin{tikzcd}
h_7 = r_a(x_0) \arrow[r] \arrow[drr] & r_1(x_0).w_1(x_1).c_1 \arrow[r] & r_b(x_1).c_b \\
r_b(y_0) \arrow[r] \arrow[urr] & r_2(y_0).w_2(y_2).c_2 \arrow[r] & r_a(y_2).c_a
\end{tikzcd}$$

SI requires monotonic snapshots. However, the underlying reason is intricate enough that some previous works [4, for instance] do not ensure this property, while claiming to be SI. Below, we introduce an ordering relation between snapshots to formalize snapshot monotonicity.

Definition 5 (Snapshot precedence). *Consider a history h and two distinct transactions T_i and T_j. The snapshot read by T_i precedes the snapshot read by T_j in history h, written $T_i \rightarrow T_j$, when $r_i(x_k)$ and $r_j(y_l)$ belong to h and either (i) $r_i(x_k) <_h c_l$ holds, or (ii) transaction T_l writes x and $c_k <_h c_l$ holds.*

For more illustration, consider histories $h_8 = r_1(x_0).w_1(x_1).c_1.r_2(y_0).w_2(y_2).r_a(x_1).c_2.r_b(y_2).c_a.c_b$ and $h_9 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).c_a.r_2(x_1).r_2(y_0).w_2(x_2).w_2(y_2).c_2.r_b(y_2).c_b$. In history h_8, $T_a \rightarrow T_b$ holds because $r_a(x_1)$ precedes c_2 and T_b reads y_2. In h_9, c_1 precedes c_2 and both T_1 and T_2 modify object x. Thus, $T_a \rightarrow T_b$ also holds. We define snapshot monotonicity using snapshot precedence as follows:

Definition 6 (Snapshot monotonicity). *Given some history h, if the relation $\rightarrow^*$ induced by h is a partial order, the snapshots in h are* monotonic*. We note MON the set of histories that satisfy this property.*

According to this definition, both $T_a \rightarrow T_b$ and $T_b \rightarrow T_a$ hold in history h_7. Thus, history h_7 does not belong to MON.

Non-monotonic snapshots are observed under update serializability [16], that is when queries observe consistent state, but only updates are serializable.

3.4 Write-Conflict Freedom

Rule D2 of SI forbids two concurrent write-conflicting transactions from both committing. Since in our model we assume that every write is preceeded by a corresponding read on the same object, every update transaction depends on a previous update transaction (or on the initial transaction T_0). Therefore, under SI, concurrent conflicting transactions must be independent:

Definition 7 (Write-Conflict Freedom). *A history h is write-conflict free if two independent transactions never write to the same object. We denote by WCF the histories that satisfy this property.*

3.5 The Decomposition

Theorem 1 below establishes that a history h is in SI iff (1) every transaction in h sees a committed state, (2) every transaction in h observes a strictly consistent snapshot, (3) snapshots are monotonic, and (4) h is write-conflict free. A detailed proof appears in our companion technical report [11].

Theorem 1. $SI = ACA \cap SCONS \cap MON \cap WCF$

To the best of our knowledge, this result is the first to prove that SI can be split into simpler properties. Theorem 1 also establishes that SI is definable on plain histories. This has two interesting consequences: (i) a transactional system does not have to explicitly implement snapshots to support SI, and (ii) one can compare SI to other consistency criterion without relying on a phenomena based characterization.[6]

[6] Contrary to, e.g., the work of Adya [6].

4 The Impossibility of SI with GPR

This section leverages our previous decomposition result to show that SI is inherently non-scalable. In more details, we prove that none of MON, SCONSa or SCONSb is attainable in some asynchronous failure-free GPR system Π when updates are obstruction-free and queries are wait-free. To prove these results, we first characterize below histories acceptable by Π.

Lemma 1. *Let $h = \mathfrak{F}(\rho)$ be an acceptable history by Π such that a transaction T_i is pending in h. Note X the set of objects accessed by T_i in h. Only processes in Replicas(X) make steps to execute T_i in ρ.*

Proof. (By contradiction.) Consider that a process $p \notin \mathit{Replicas}(X)$ makes steps to execute T_i in ρ. Since the prefix of a transaction is a transaction with the same id, we can consider an extension ρ' of ρ such that T_i does not execute any additional operation in ρ' and $\mathit{coord}(T_i)$ is correct in ρ'. The progress requirements satisfied by Π imply that T_i terminates in ρ'. However, process $p \notin \mathit{Replicas}(X)$ makes steps to execute T_i in ρ'. A contradiction to the fact that Π is GPR.

We now state that monotonic snapshots are not constructable in Π. Our proof holds because objects accessed by a transaction are not known in advance.

Theorem 2. *No asynchronous failure-free GPR system implements MON*

Proof. (By contradiction.) Let us consider (i) four objects x, y, z and u such that for any two objects in $\{x, y, z, u\}$, their replica sets do not intersect; (ii) four queries T_a, T_b, T_c and T_d accessing respectively $\{x, y\}$, $\{y, z\}$, $\{z, u\}$ and $\{u, x\}$; and (iii) four updates T_1, T_2, T_3 and T_4 modifying respectively x, y, z and u.

Obviously, history $r_b(y_0)$ is acceptable, and since updates are obstruction-free, $r_b(y_0).r_2(y_0).w_2(y_2).c_2$ is also acceptable. Applying that Π satisfies non-trivial SI, we obtain that history $r_b(y_0).r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2)$ is acceptable. Since T_a must be wait-free, $h = r_b(y_0).r_2(y_0).w_2(y_2).c_2.r_a(x_0).r_a(y_2).c_a$ is acceptable as well. Using a similar reasoning, history $h' = r_d(u_0).r_4(u_0).w_4(u_4).c_4$ $.r_c(z_0).r_c(u_4).c_c$ is also acceptable. We note ρ and ρ' respectively two sequences of steps such that $\mathfrak{F}(\rho) = h$ and $\mathfrak{F}(\rho') = h'$.

System Π is GPR. As a consequence, Lemma 1 tells us that only processes in $\mathit{Replicas}(x, y)$ make steps in ρ. Similarly, only processes in $\mathit{Replicas}(u, z)$ make steps in ρ'. By hypothesis, $\mathit{Replicas}(x, y)$ and $\mathit{Replicas}(u, z)$ are disjoint. Applying a classical indistinguishably argument [13, Lemma 1], both $\rho'.\rho$ and $\rho.\rho'$ are admissible by Π. Thus, histories $h'.h = \mathfrak{F}(\rho'.\rho)$ and $h.h' = \mathfrak{F}(\rho.\rho')$ are acceptable.

Since updates are obstruction-free, history $h'.h.r_3(z_0).w_3(z_3).c_3$ is acceptable. Note U the sequence of steps following $\rho'.\rho$ with $\mathfrak{F}(U) = r_3(z_0).w_3(z_3).c_3$. Observe that by Lemma 1 $\rho'.\rho.U$ is indistinguishable from $\rho'.U.\rho$. Then consider history $\mathfrak{F}(\rho'.U.\rho)$. In this history, T_b is pending and the latest version of object z is z_3, As a consequence, because Π satisfies non-trivial SI, there exists an extension of $\rho'.U.\rho$ in which transaction T_b reads z_3. From the fact that queries are wait-free and since $\rho'.\rho.U$ is indistinguishable from $\rho'.U.\rho$, we obtain that history $h_1 = h'.h.r_3(z_0).w_3(z_3).c_3.r_b(z_3).c_b$ is acceptable. We note U_1 the sequence of steps following $\rho'.\rho$ such that $\mathfrak{F}(U_1)$ equals $r_3(z_0).w_3(z_3).c_3.r_b(z_3).c_b$.

With a similar reasoning, history $h_2 = h'.h.r_1(x_0).w_1(x_1).c_1.r_d(x_1).c_d$ is acceptable. Note U_2 the sequence satisfying $\mathfrak{F}(U_2) = r_1(x_0).w_1(x_1).c_1.r_d(x_1).c_d$.

Executions $\rho'.\rho.U_1$ and $\rho'.\rho.U_2$ are both admissible. Because Π is GPR, only processes in *Replicas*(y, z) (resp. *Replicas*(x, u)) make steps in U_1 (resp. U_2). By hypothesis, these two replica sets are disjoint. Applying again an indistinguishably argument, $\rho'.\rho.U_1.U_2$ is an execution of Π. Therefore, the history $\hat{h} = \mathfrak{F}(\rho'.\rho.U_1.U_2)$ is acceptable. In this history, relation $T_a \rightarrow T_b \rightarrow T_c \rightarrow T_d \rightarrow T_a$ holds. Thus, $\hat{h}$ does not belong to MON. Contradiction.

Our next theorem states that SCONSb is not attainable. Similarly to Attiya et al. [17], our proof builds an infinite execution in which a query T_a on two objects never terminates. We first define a finite execution during which we interleave between any two consecutive steps to execute T_a, a transaction updating one of the objects read by T_a. We show that during such an execution, transaction T_a does not terminate successfully. Then, we prove that asynchrony allows us to continuously extend such an execution, contradicting the fact that queries are wait-free.

Definition 8 (Flippable execution). *Consider two distinct objects x and y, a query T_a over both objects, and a set of updates $T_{j\in[\![1,m]\!]}$ accessing x if j is odd, and y otherwise. An execution $\rho = U_1V_2U_2 \ldots V_mU_m$ where,*

- *transaction T_a reads in history $h = \mathfrak{F}(\rho)$ at least version x_1 of x,*
- *for any j in $[\![1, m]\!]$, U_j is the execution of transaction T_j by processes Q_j,*
- *for any j in $[\![2, m]\!]$, V_j are steps to execute T_a by processes P_j, and*
- *both $(Q_j \cap P_j = \varnothing) \oplus (P_j \cap Q_{j+1} = \varnothing)$ and $Q_j \cap Q_{j+1} = \varnothing$ hold,*

is called flippable.

Lemma 2. *Let ρ be an execution admissible by Π. If ρ is flippable and histories accepted by Π satisfy SCONSb, query T_a does not terminate.*

Proof. Let h be the history $\mathfrak{F}(\rho)$. In history h transaction T_j precedes transaction T_{j+1}, it follows that h is of the form $h = w_1(x_1).c_1.*.w_2(y_2).c_2.*\ldots$, where each symbol $*$ corresponds to either no operation, or to some read operation by T_a on object x or y.

Because ρ is flippable, transaction T_a reads at least version x_1 of object x in h. For some odd natural $j \geq 1$, let x_j denote the version of object x read by T_a. Similarly, for some even natural l, let y_l be the version of y read by T_a. Assume that $j < l$ holds. Therefore, h is of the form $h = \ldots w_j(x_j) \ldots w_l(y_l) \ldots$.

Note k the value $l+1$, and consider the sequence of steps V_k made by P_k right after U_l to execute T_a. Applying the definition of a flippable execution, we know that (F1) $(Q_l \cap P_k = \varnothing) \oplus (P_k \cap Q_k = \varnothing)$, and (F2) $Q_l \cap Q_k = \varnothing$. Consider now the following cases:

(Case $Q_l \cap P_k = \varnothing$.) It follows that ρ is indistinguishable from the execution $\rho'' = \ldots U_j \ldots V_k U_l U_k \ldots$. Then from fact F2, ρ is indistinguishable from execution $\rho' = \ldots U_j \ldots V_k U_k U_l \ldots$.

(Case $P_k \cap Q_k = \varnothing$) With a similar reasoning, we obtain that ρ is indistinguishable from $\rho' = \ldots U_j \ldots U_k U_l V_k \ldots$.

(Case $P_k \cap (Q_l \cup Q_k) = \varnothing$.) This case reduces to any of the two above cases.

Note h' the history $\mathfrak{F}(\rho')$. Observe that since ρ' is indistinguishable from ρ, history h' is acceptable. In history h', $c_k <_{h'} c_l$ holds. Moreover, $c_j <_{h'} c_k$ holds by the assumption $j < l$ and the fact that k equals $l + 1$. Besides, operations $r_i(x_j)$, $r_i(y_l)$ and $w_k(x_k)$ all belong to h'. According to the definition of SCONSb, transaction T_a does not commit in h'. (The case $j > l$ follows a symmetrical reasoning to the case $l > j$ we considered previously.)

Theorem 3. *No asynchronous failure-free GPR system implements SCONSb.*

Proof. (By contradiction.) Consider two objects x and y such that *Replicas*(x) and *Replicas*(y) are disjoint. Assume a read-only transaction T_a that reads successively x then y. Below, we exhibit an execution admissible by Π during which transaction T_a never terminates. We build this execution as follows:

[Construction.] Consider some empty execution ρ. Repeat for all $i >= 1$: Let T_i be an update of x, if i is odd, and y otherwise. Start the execution of transaction T_i. Since no concurrent transaction is write-conflicting with T_i in ρ and updates are obstruction-free, there must exist an extension $\rho.U_i$ of ρ during which T_i commits. Assign to ρ the value of $\rho.U_i$. Execution ρ is flippable. Hence, Lemma 2 tells us that transaction T_a does not terminate in this execution. Consider the two following cases: (Case $i = 1$) Because Π satisfies non-trivial SI, there exists an extension ρ' of ρ in which transaction T_a reads at least version x_1 of object x. Notice that execution ρ' is of the form $U_1.V_2.s.\ldots$ where *(i)* all steps in V_2 are made by processes in *Replicas*(x), and *(ii)* s is the first step such that $\mathfrak{F}(U_1.V_2.s.) = r_1(x_0).w_1(x_1).c_1.r_a(x_1)$. Assign $U_1.V_2$ to ρ . (Case $i > 2$) Consider any step V_{i+1} to terminate T_a and append it to ρ.

Execution ρ is admissible by Π. Hence $\mathfrak{F}(\rho)$ is acceptable. However, in this history transaction T_a does not terminate. This contradicts the fact that queries are wait-free.

SCONSa disallows some real time orderings between operations accessing different objects. Our last theorem shows that this property cannot be maintained under GPR.

Theorem 4. *No asynchronous failure-free GPR system implements SCONSa.*

Proof. (By contradiction.) Consider two distinct objects x and y such that *Replicas*(x) and *Replicas*(y) are disjoint. Let T_1 be an update accessing y, and T_a be a query reading both objects.

Obviously, history $h = r_a(x_0)$ is acceptable. Note U_a a sequence of steps satisfying $U = \mathfrak{F}(r_a(x_0))$. Because Π supports obstruction-free updates, we know the existence of an extension $U_a.U_1$ of U_a such that $\mathfrak{F}(U_1) = r_1(y_0).w_1(y_1).c_1$. By Lemma 1, we observe that $U_a.U_1$ is indistinguishable from $U_1.U_a$. Then, since

Π satisfies non-trivial SI and read-only transactions are wait-free, there must exist an extension $U_1.U_a.V_a$ of $U_1.U_a$ admissible by Π and such that $\mathfrak{F}(V_a) = r_a(y_1).c_a$. Finally, since $U_a.U_1$ is indistinguishable from $U_1.U_a$ and $U_1.U_a.V_a$ is admissible, $U_a.U_1.V_a$ is admissible too. The history $\mathfrak{F}(U_a.U_1.V_a)$ is not in SCONSa. Contradiction.

As a consequence of the above, no asynchronous system, even if it is failure-free, can support both GPR and SI. In particular, even if the system is augmented with failure detectors [18], a common approach to model partial synchrony, SI cannot be implemented under GPR. This fact strongly hinders the usage of SI at large scale. In the following section, we further discuss implications of this result.

5 Discussion

A straightforward corollary of any of the theorems we proved in Section 4 is that neither strict serializability [19], nor opacity [20] is attainable under GPR. In the case of opacity, this answers negatively to a problem posed by Peluso et al. [21].

The classical (non-genuine) solution for building strictly consistent monotonic snapshots is to use total order broadcast (e.g., [2, 3]).

When a transaction declares objects it accesses in advance, a GPR system can install a snapshot just after the start of the transaction. As a consequence, such an assumption sidesteps our impossibility result.

A transactional system Π is *permissive* with respect to a consistency criterion $\mathcal{C}$ when every history $h \in \mathcal{C}$ is acceptable by Π. Permissiveness [22] measures the optimal amount of concurrency a system allows. If we consider again histories h_1 and h_2 in the proof of Theorem 2, we observe that both histories are serializable. Hence, every system permissive with respect to SER accepts both histories. By relying on the very same argument as the one we exhibit to close the proof of Theorem 2, we conclude that no transactional system is both GPR and permissive with respect to SER. For instance, none of the systems presented in [10, 23] accept history $h_{10} = r_1(x_0).w_1(x_1).c_1.r_2(x_0).r_2(y_0).w_2(y_2).c_2$.

Recent distributed transactional systems (e.g., [9, 24]) support weaker consistency criteria than SI or SER. In particular, Walter [9] supports Parallel Snapshot Isolation (PSI). PSI is weaker than SI, and allows snapshots to be non-monotonic. But, it still requires SCONSa to be ensured. Sovran et al. justify the use of PSI in Walter by the fact that SI is too expensive in a geographically distributed environment [9, page 4]. Our impossibility result establishes that, in order to scale, a transactional system needs supporting both non-monotonic *and* non-strictly consistent snapshots.

6 Conclusion

Partial replication and genuineness are two key factors of scalability in replicated systems. This paper shows that ensuring snapshot isolation (SI) in a genuine partial replication system is impossible. To state this impossibility result, we prove that SI is decomposable into a set of simpler properties. We show that two of these properties, namely snapshot monotonicity and strictly consistent snapshots cannot be ensured. As a corollary of our results, a GPR system cannot support neither strict serializability, nor opacity.

References

[1] Daudjee, K., et al.: Lazy database replication with snapshot isolation. In: VLDB, pp. 715–726 (September 2006)

[2] Serrano, D., et al.: Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation. In: PRDC, pp. 290–297 (December 2007)

[3] Armendáriz-Iñigo, J.E., et al.: SIPRe: a partial database replication protocol with SI replicas. In: SAC, pp. 2181–2185 (March 2008)

[4] Bieniusa, A., et al.: Consistency in hindsight: A fully decentralized stm algorithm. In: IPDPS, pp. 1–12 (April 2010)

[5] Riegel, T., et al.: Snapshot isolation for software transactional memory. In: TRANSACT (June 2006)

[6] Adya, A.: Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. Ph.D., MIT (March 1999)

[7] Elnikety, S., et al.: Database Replication Using Generalized Snapshot Isolation. In: SRDS, pp. 73–84 (October 2005)

[8] Berenson, H., et al.: A critique of ansi sql isolation levels. In: SIGMOD, pp. 1–10 (May 1995)

[9] Sovran, Y., et al.: Transactional storage for geo-replicated systems. In: SOSP, pp. 385–400 (October 2011)

[10] Schiper, N., et al.: P-store: Genuine partial replication in wide area networks. In: SRDS, pp. 214–224 (November 2010)

[11] Ardekani, M.S., et al.: Non-Monotonic Snapshot Isolation. INRIA, Tech. Rep. RR-7805 (October 2012)

[12] Bernstein, P., et al.: Concurrency Control and Recovery in Database Systems. Addison Wesley Publishing Company (1987)

[13] Fischer, M.J., et al.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)

[14] Abadi, M., et al.: The existence of refinement mappings. Theory Computer Science 82, 253–284 (1991)

[15] Chan, A., et al.: Implementing Distributed Read-Only Transactions. IEEE Transactions on Software Engineering SE-11(2), 205–212 (1985)

[16] Garcia-Molina, H., et al.: Read-only transactions in a distributed database. ACM Trans. Database Syst. 7(2), 209–234 (1982)

[17] Attiya, H., et al.: Inherent limitations on disjoint-access parallel implementations of transactional memory. In: SPAA, pp. 69–78 (2009)

[18] Chandra, T.D., et al.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)

[19] Papadimitriou, C.H.: The serializability of concurrent database updates. Journal of the ACM 26(4), 631–653 (1979)

[20] Guerraoui, R., et al.: On the correctness of transactional memory. In: PPoPP, pp. 175–184 (2008)

[21] Peluso, S., et al.: Genuine replication, opacity and wait-free read transactions: can a stm get them all? In: WTTM, Madeira, Portugal (July 2012)
[22] Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 305–319. Springer, Heidelberg (2008)
[23] Sciascia, D., et al.: Scalable deferred update replication. In: DSN, pp. 1–12 (June 2012)
[24] Peluso, S., et al.: When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In: ICDCS, pp. 455–465 (June 2012)

Efficient Event Prewarning for Sensor Networks with Multi Microenvironments*

Yinglong Li[1,2,3], Hong Chen[1,2,**], Suyun Zhao[1], and Shangfeng Mo[1,2,3]

[1] Key Laboratory of Data Engineering and Knowledge Engineering, MOE, China
[2] School of Information, Renmin University of China, Beijing, China
[3] Hunan University of Science and Technology, Xiangtan, China
{liyinglong,chong,zhaosuyun,moshangfengxy}@ruc.edu.cn

Abstract. Early detecting the approaching events is the primary way of minimizing their damages in the sensor-based systems. The majority of existing approaches of event description and detection rely on using crisp raw sensory data, which requires large amount of data transmission as well as is memory-consuming, moreover, these approaches are only applicable to homogeneous sensor networks. This paper describes a novel efficient framework for event prewarning in sensor networks with multi microenvironments, which mainly includes a simple and practical data preprocessing method, Node-level Noteworthy Event (NNE) detection algorithm, event probability encodings of NNEs and two distributed Node-level Alert Event (NAE) detection algorithms. We demonstrate our algorithms by experimentally evaluating their performance in various scenarios using real and synthetic data. Our NAE detection algorithm by leveraging spatial correlation only requires a small amount of data transmission and can detect over 90% of NAEs with few false negatives.

Keywords: microenvironment, sensor network, event probability, node-level noteworthy event, node-level alert event.

1 Introduction

Sensor networks can be viewed as energy constrained distributed database systems, and their tasks are monitoring physical environments, processing sensed information, and forwarding results to base stations (sink). Since data transmission consumes most of the energy, a significant challenge for such kind of systems is to design reliable, energy-efficient data processing algorithms to maximize the lifetime of sensor networks. Moreover, the low amount of data transmission also contributes to quick response time and less signal interference in wireless communication.

Event detection is a common required service in sensor network based applications such as environmental monitoring [1,2] and object tracking [3,4], which

* This work was supported by the National Natural Science Foundation of China (grant no.: 61070056, 61033010 and 61202114).

** Corresponding author.

has attracted increasing research attention. Various studies on event description and detection approaches have been reported in the literature [1]-[8]. However, these approaches rely on using raw sensor measurements, which results in large data transmission inside the network and long response time. Moreover, the aforementioned approaches are only applicable to sensor networks with single microenvironment, while in many sensor network applications, the monitoring area consists of multi MicroEnvironments (MEs), where the characteristics of each ME are different. For example, different types of materials with different ignition points (e.g., burning point of methanol is less than 30°C and fire point of turpentine is under 65 °C) are placed in different areas of a warehouse, where each resulting segment (area) will have different tolerance for warehouse fire. Each of these segments of the warehouse, then, represents a ME as shown in Fig. 1. When users check the k most likely impending fire locations in such kind

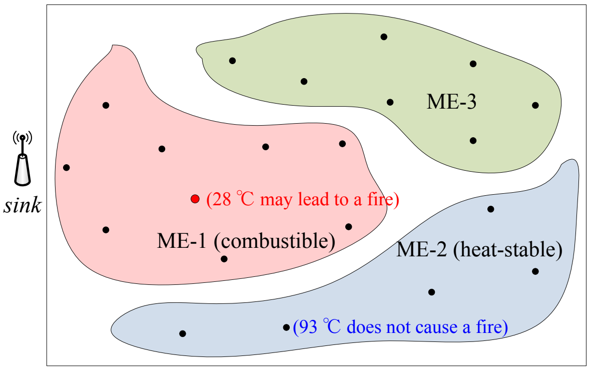


Fig. 1. An example of three MEs in a warehouse

of sensor networks, if the existing event detection methods (such as the weighted voting schemes in [7,8] are used, they will lead to erroneous results, since the small temperature values in the combustible MEs might be the top k results.

In this paper we develop an event prewarning framework for sensor networks with multi MEs in a scenario of relatively short sampling period, which has not been studied before. The contributions of this paper are summarized as follows:

- A conceptually simple, yet practically effective data preprocessing approach is given to eliminate erroneous sensory data.
- The definition of Node-level Noteworthy Event (NNE) and the detection algorithm of NNE are proposed.
- Two distributed node-level alert event detection algorithms are devised, and extensive simulations are performed to validate our motivation.

The remainder of the paper proceeds as follows. Our network model is explained in Section 2, and Section 3 describes the data preprocessing approach. Then, node-level noteworthy event algorithm is proposed in Section 4. Section 5 details two node-level alert event detection algorithms. Meanwhile, simulation results are presented in Section 6. Finally, Section 7 presents conclusions and future research direction.

2 Network Model

This section gives an overview of our network model. It also covers the almost reasonable assumptions.

In our geographical location based clustering networks, there are two types of nodes which are regular sensor node and Cluster Head (CH) sensor node respectively in the network. Each node (we use "node" to refer to a regular sensor node or CH sensor node) has a unique identification (*nodeID*). Tentative CH nodes are selected from regular nodes mainly based on their remaining energy via a non-probabilistic fashion [9]. In addition to collect its own sensory data, every CH node manages the topology information of all regular nodes within its cluster, as shown in Fig. 2(a).

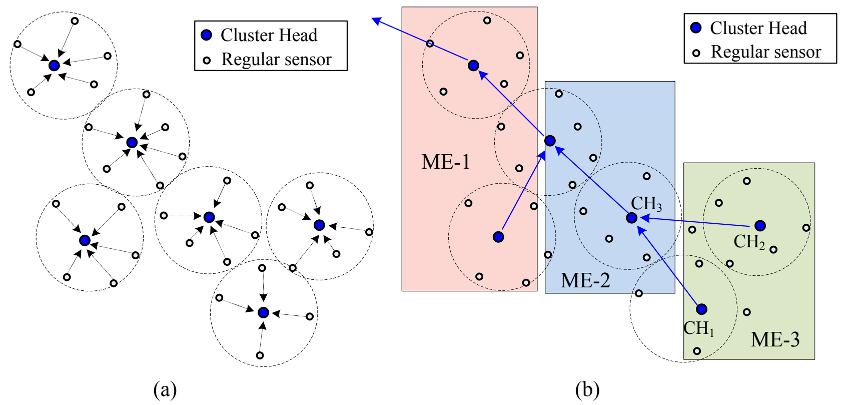


Fig. 2. (a) Cluster-based network. (b) CH-TAG network with three MEs.

CH nodes communicate with each other hop by hop. We assume that a data aggregation tree (i.e., TAG [10]) is constructed over these CH nodes, and eventually connected to base station, as is shown in Fig. 2 (b). We call the CH nodes based aggregation tree as CH-TAG. Our network also consists of multi MEs, and an example of three MEs in a CH-TAG network is shown in Fig. 2(b).

3 Data Preprocessing

Event detection techniques need to prevent erroneous data from influencing the detection reliability. Most of existing outlier (here mainly refers to errors and noise) detection [5] approaches relay on using the neighboring information, which leads to a large amount of data transmission and is not suitable for energy constrained sensor networks. Therefore, we give a node-level erroneous data detection approach only leveraging the *temporal correlation* of the local measurements within the recent sampling periods.

The erroneous sensory data we concentrated here mainly come from FAULTY (failed) nodes and noisy data in normal working nodes. Therefore, our goal of data preprocessing is to identify and prune faulty nodes and eliminate noisy data in functioning nodes.

Definition 1 (FAULTY Node Rule). Most of the data in the FAULTY node significantly deviate from the normal pattern of sensed data [11]. A simple and effective method is that we can know whether a sensor node is a FAULTY node by checking the standard deviation and temporal correlation of the sensor readings within recent time window. FAULTY nodes should be pruned as early as possible.

Definition 2 (NOISY Data Rule). In functioning node, *smoothing factor* [12] based approach (*improved smoothing factor*) is used to identify and eliminate the noisy data, where the *improved smoothing factor* can be defined as follows: Given:

• a set of sensor measurements *MS*

• a dissimilarity function $D : D(MS) \rightarrow R^+$, where R^+ means positive real number.

$$ISF(M_i) = D(MS) - D(MS - \{M_i\}). \tag{1}$$

The *improved smoothing factor* (*ISF*) indicates how much the dissimilarity can be reduce by removing an element (sensor measurement) from the set *MS*. Observe that *ISF* may be negative for some measurement M_i if the dissimilarity of $MS - \{M_i\}$ is higher than that of the original set *MS*.

Example. Let

• the set *MS* be the set of float values 32.6, 31.9, 11.7, 32.8, 93.5, 33.3;

• the dissimilarity function $D : D(MS) \rightarrow R^+$ be the variance of the numbers in the set, i.e., $\sum_{i=1}^{n}(x_i - \bar{x})^2/n$,then we get $D(MS) = 646.22$

By computing the *ISF* for each measurement M_i via (1), we get:

Table 1. *ISF* calculation for each measurement M_i

M_i	$MS - \{M_i\}$	$D(MS - \{M_i\})$	$ISF(M_i)$
32.6	{31.9, 11.7, 32.8, 93.5, 33.3}	764.69	-118.47
31.9	{32.6, 11.7, 32.8, 93.5, 33.3}	762.32	-116.1
11.7	{32.6, 31.9, 32.8, 93.5, 33.3}	592.64	53.58
32.8	{32.6, 31.9, 11.7, 93.5, 33.3}	803.93	-157.71
93.5	{32.6, 31.9, 11.7, 32.8, 33.3}	70.43	575.79
33.3	{32.6, 31.9, 11.7, 32.8, 93.5}	766.82	-120.6
32.6	{31.9, 11.7, 32.8, 93.5, 33.3}	764.69	-118.47

Thus, the measurement 93.5 has the highest probability to be the noisy data as its *ISF* is the largest, followed by the element 11.7, as shown in Table 1. The measurement whose *ISF* is over a preset threshold can be eliminated, and the *improved smoothing factor* provides tunable accuracy guarantees based on users requirement that how much noise data should be eliminated.

4 Node-Level Noteworthy Event

Definition 3 (Node-level Noteworthy Event (NNE)). node u is a node-level noteworthy event once us measurements make the *event probability* (ep) of u reach a preset threshold value *thrd* (usually 0.5).

4.1 NNE Detection Algorithm

After pruning FAULTY nodes and eliminating noisy data in functioning nodes, each node calculates its ep based on its measurements and ep function, where the ep function varies in different MEs, or the function parameters are different in each ME, which can be defined based on empirical or domain knowledge. In the warehouse fire monitoring system with multi MEs, if two properties (*temperature* (t) and *humidity* (h)) are mainly used for defining fire event, then an example of ep function can be defined as follows.

$$ep(t,h) = \begin{cases} 0, & t \leqslant t_{as}, h... \\ \omega \times avg(t)/(t_{ig} - t_{as}) + (1-\omega) f(avg(h)), & t_{as} < t < t_{ig}, h... \\ 1, & t \geqslant t_{ig}, h... \end{cases} \quad (2)$$

Where avg means getting the average value of measurements. t_{as} means the absolute safe temperature, which by no means causes a fire, and t_{ig} means the ignition temperature, which varies in different MEs. The first part of (2) demonstrates that the higher temperature leads to higher probability of fire, and the $f(h)$ in the second part of (2) is a function of how *humidity* affect the fire event, which *humidity* usually has a negative impact on fire event. ω is a tunable value, which determines how much *temperature* and *humidity* influence the fire event respectively. Due to the different $[t_{as}, t_{ig}]$ in each ME, (2) can describe the characteristics of fire occurrence in different MEs. Therefore, each node in different MEs can determine whether it is a NNE by the Algorithm 1.

In Algorithm 1, $ep(i)$ is the event probability of node i, and $pFlag(i)$ is a flag to determine whether node i can be pruned. If $pFlag(i)$ is "TRUE", then node i is not a noteworthy node and should be pruned as early as possible.

4.2 *ep* Encoding of NNEs

In order to reduce the data transmission, we discretize the ep values and give their encodings. We use linguistic characters to define the severity grades (*GRADE*) of NNEs and each *GRADE* corresponds to a non-uniform ep sub range $\Re(ep)$.

Algorithm 1. NNE Detection

```
1: for each node $i$ do
2:     $pFlag \leftarrow FALSE$;
3:     if $i$ is FAULTY node then
4:         $pFlag \leftarrow TRUE$;
5:     else
6:         eliminate the noisy data
           using NOISY Data Rule;
7:         calculate the $ep(i)$;
8:         if $ep(i) < thrd$ then
9:             $pFlag \leftarrow TRUE$;
10:        else
11:            $i$ is a NNE;
12:        end if
13:    end if
14: end for
```

When *GRADE* is closer to the event occurrence threshold, the interval of $\Re(ep)$ is smaller.

For example, as shown in Table 2, we define eight *GRADE*s (‘*a*’, ‘*b*’, ‘*c*’, ‘*d*’, ‘*e*’, ‘*f*’, ‘*g*’ and ‘*h*’) (the first column, not be fully presented due to the limited space) whose interpretations are shown in the second column with different $\Re(ep)$. The *ep* sub range interval of *GRADE* ‘*a*’ is the smallest as shown in the last column, and the threshold *thrd* is 0.5, which means every node whose *ep* is below 0.5 (below ‘*h*’) can be pruned. We also give 3-bits encoding for each *GRADE* to reduce the total data transmission, as is shown in the third column of Table 2.

Table 2. An Example of *ep* encoding based on non-uniform *ep* sub ranges

GRADE	Interpretation	Encoding	$\Re(ep)$
a	*Almost happened*	000	[0.98, 1.00)
b	*Very serious*	001	[0.95, 0.98)
c	*Serious*	010	[0.91, 0.95)
d	*Alert*	011	[0.86, 0.91)
...	...	...	...
h	*Noteworthy*	111	[0.50, 0.62)

The above non-uniform *ep* sub ranges $\Re(ep)$ can be obtained via mathematical models based on the actual situation of specified application. The accuracy of event prewarning might be higher if more *GRADE*s are used.

5 Node-Level Alert Event Detection

Definition 4 (Node-level Alert Event (NAE). A sensor node *u* belonging to NNEs becomes a node-level alert event if it has relatively large *ep* and has at least one neighboring NNE. Here, the relatively large *ep* means that the ep of *u* meets application-specific threshold, such as 0.86 (‘*d*’) used later in this paper.

NAEs are the node-level prewarning events that we focus in this paper. And we propose two following NAE detection approaches.

5.1 NAE Detection without Considering Spatial Correlation

NAE detection scheduling without considering spatial correlations, denoted as NAED-noSC, can be described as follows. Each NNE whose *GRADE* reaches a predefined threshold (‘*d*’ in this paper) is forwarded to the base station as NAE. The main data in each message packet is “*GRADE* (Encoding of *GRADE*)” and “*nodeID*”, denoted as “*GRADE+nodeID*”. The spatial correlation of these NAEs can be checked to further confirm the approaching events (or event regions) according to the global network topology information at the base station. The data transmission of NAED-noSC is very small, but there might be false positives and false negatives.

Accuracy analysis of NAED-noSC. In NAED-noSC, since every NNE with a “*Alert*” *GRADE* (‘*d*’) is detected as NAE, there might be isolated NAEs (without any neighboring NNEs) leading to false positives. While those NNEs whose *GRADE*s are little bit smaller than ‘*d*’ but have neighboring NNEs might not be detected as NAEs in NAED-noSC, which leads to false negatives.

5.2 NAE Detection by Leveraging Spatial Correlation

Existence of *spatial correlation* implies that the readings from sensor nodes geographically close to each other are expected to be largely correlated. We would be more confident that there is an actual fire if there are at least two neighboring nodes reporting high temperature and low humidity readings. There might be a false positive if there is just one NAE without any neighboring NNE. This is particularly reasonable in sensor networks where nodes are usually densely deployed.

The strategy here is that we update the *GRADE*s of spatial-correlated NNEs, and the Approach of *GRADE* Update of spatial-correlated NNEs (AoGU) can be described as follows. If there are two spatial-correlated NNEs, then both of their *GRADE*s are increased to a higher level *GRADE*s. For instance, there are three NNEs N_1, N_2 and N_3, and their *GRADE*s are ‘*h*’, ‘*f*’ and ‘*e*’ respectively. If N_1 and N_2 are spatial-correlated, and N_2 is N_3’s neighboring NNE, but there is no spatial correlation between N_1 and N_3, then their *GRADE*s after grade update are ‘*g*’, ‘*d*’ and ‘*d*’ respectively. The magnitude of *GRADE* update is tunable, which is specified depending on the application scenario.

NAE Detection by Leveraging Spatial Correlations (NEAD-bySC). Firstly, every CH node gets all the NNEs in its local cluster, and then each leaf CH node (CH-TAG based network) forwards all the NNEs in its cluster to its parent CH node. When a non-leaf CH node Q has obtained all the NNEs from its local cluster and its child clusters, it Checks the Spatial Correlation of these NNEs and Updates the *GRADE*s of spatial-correlated NNEs (CSC-UG) based on AoGU. Finally, all the non-isolated NAEs in Q’s child clusters are forwarded to base station as NAEs, and then Q forwards all the NNEs in its local cluster to its parent cluster, until all the non-leaf CHs finished the CSC-UG and NAE detection.More details of NAED-bySC are described in Algorithm 2.

Algorithm 2. NAED-bySC

```
Step 1. NNEs forwarding
1: for each NNE $i$ do
2:     $uFlag(i) \leftarrow FALSE$;
3:     $i$ be forwarded to its local CH node;
4: end for
5: for each leaf CH node $i$ do
6:     $i$ forwards all the NNEs to its parent CH node;
7: end for
Step 2. CSC-UG and NAE detection
1: for each non-leaf CH node $i$ do
2:     CSC-UG of $i$'s NNEs is done;
3:     if $GRADE$(NNE $j$) be updated then
4:         $uFlag(j) \leftarrow TRUE$;
5:     end if
6:     for each NNE $k$ in $i$ do
7:         if $GRADE(k) \geqslant$ '$d$'&&$uFlag(k) == TRUE$ then
8:             $k$ is detected as a NAE;
9:         end if
10:    end for
11:    $i$ forward all the NNEs to its parent CH node;
12: end for
```

Where $uFlag(i)$ is a flag to demonstrate whether the *GRADE* of NNE i be updated, which is used for identifying whether i is an isolated NNE. If $uFlag(i)$ is "FALSE", then i is a isolated NNE. In Algorithm 2, the main data structure of NNE and NAE message packet of line 3 in *step 1* and line 8 in *step 2* are both "*GRADE + nodeID*", and the main data structure of NNE message packet in line 11 of *step 2* is "*GRADE + nodeID + LOCATION*", where the "*LOCATION*" is the geographic coordinates which are used for spatial correlation detection.

Example. As shown in Fig. 3, there are three clusters with cluster head CH_1, CH_2 and CH_3. Firstly, all NNEs are forwarded to its local CH node, such as NNE S_1, S_3 and S_5 in CH_1 cluster are forwarded to CH_1. Then each leaf CH node forwards their NNEs to its parent CH node, such as CH_1 and CH_2 forward their NNEs to CH_3. Then CH_3 does CSC-UG based on the AoGU. S_{11} and S_3 are spatial-correlated, and S_6 is the neighboring node of S_5 and S_7, as shown the green dotted lines in Fig. 3. All the *GRADE*s of these NNEs are updated based on AoGU, as shown in Fig. 3. S_1 is not a NAE because it is an isolated NNE although it has a high *GRADE*, and all the non-isolated NAEs in CH_3's child clusters (cluster CH_1 and CH_2) are forwarded to base station as NAEs. Finally, CH_3 forwards all the NNEs in its local cluster to its parent cluster, until all the CHs finishes the NAE detection.

Accuracy Analysis of NAED-bySC. In NAED-bySC, some spatial-correlated NNEs become NAEs due to their *GRADE*s update, which reduces the false negative. For instance, in Fig. 3, the NNE S_{11}('e') in cluster CH_3 is spatial-correlated with S_3 in cluster CH_1, so the *GRADE* of S_{11} is updated to 'd' and S_{11} becomes a NAE, while S_{11} is a false negative in NAED-noSC. All the isolated NNEs will not be detected as NAEs in NAED-bySC, so there is no false positive in NAED-bySC. For example, the NNE S_1 with *GRADE* 'c' is not detected as NAE since

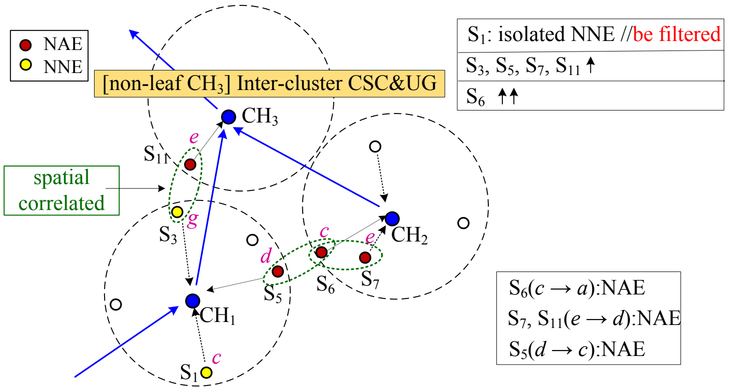


Fig. 3. Inter-cluster CSC-UG and NAE detection

it is an isolated NNE without any neighboring NNE. Therefore, the false rate of NAED-bySC is very low.

Tradeoff. Generally speaking, larger scope of inter-cluster CSC-UG contributes higher accuracy of NAE detection, but the data transmission will be lager. There is a tradeoff between data transmission and accuracy. In NAED-bySC, we only consider the spatial correlation of the nodes from parent-child clusters as well as the nodes from brother-brother clusters, without other spatial correlation. Since the spatial correlation we have considered accounts for the most of the all spatial correlation, NAED-bySC can guarantee high accuracy, while requires small amount of data transmission. More experimental analysis is given in later section 6.

6 Performance Evaluations

We use OMNET++ [13] to evaluate our algorithms in terms of data transmission and accuracy of NAE detection. We use part of real data (*Surface Temperature* and *Relative Humidity*) from LUCE [14] and part of synthetic data which is generated and injected with outliers via scenarios of other MEs due to lack of real data of sensor networks with multi MEs.

Owing to the lack of research work on event detection in multi MEs sensor networks in literature, we performed comparison among three following methods:

- *Optimized Conventional Method* (OCM)The idea of conventional method [4, 9, 10, etc.] is that every node uses the information from its neighboring nodes to check whether it is an event node (or there is an event region). For a fair comparison, event probabilities instead of raw sensor readings are used. Since all the spatial correlation is checked, the OCM can get the true NAEs accurately.
- NAED-bySC (Algorithm 2).
- NAED-noSC.

6.1 Comparison of Data Transmission

Since data transmission consumes most of the energy in sensor networks, We evaluate the data transmission of OCM, NAED-bySC and NAED-noSC in four different networks, where 40, 80, 120 and 160 nodes are randomly deployed in four regions of 300 × 300 m^2, 400 × 400 m^2, 500 × 500 m^2 and 600 × 600 m^2 respectively. The communication radius R is 80 m. *ep* is a *float* number, which requires 4 bytes space (due to the 32-bit simulation platform). $LOCATION$ consists of x-coordinate and y- coordinate (both *float* number), and $GRADE$ is 3 bits, and *nodeID* is *short int* that requires 16 bits space. We did not consider the data transmission of other information in message packet, e.g., HEAD.

Among the three methods mentioned above, the data transmission of OCM is the largest, and the one of NAED-noSC is the smallest in our four networks. The average data transmission of NAED-noSC is 1.3% of the one of OCM, and the average data transmission of NAED-bySC is 5.5 percent of the one of OCM. And the average data transmission of NAED-noSC is 23.89% of the one of NAED-bySC, as is shown in Fig. 4(a). So our algorithms can reduce the data transmission greatly compared with OCM. Although the data transmission of NAED-noSC is the smallest, its accuracy is not satisfactory (more details will be described in Subsection 6.2).

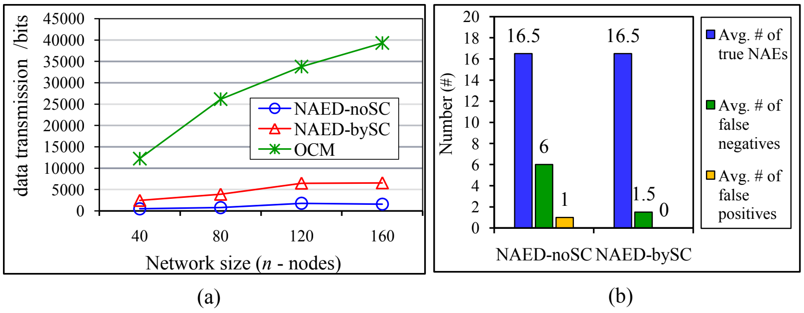


Fig. 4. (a) Comparison of data transmission. (b) Accuracy comparison.

6.2 Comparison of Accuracy

Besides good performance on data transmission, another important goal of event detection is accuracy. We study the accuracy performance of NAED-noSC, NAED-bySC and OCM based on our four networks mentioned in Subsection 6.1. Here accuracy is defined as:

$$1-(numberoffalsepositives+numberoffalsenegatives)/numberoftrueNAEs$$

As all the spatial correlation is examined for event detection in OCM, there is no false positive or false negative (accuracy is 100%). There are a few false positives in NAED-noSC, and the average (Avg.) number (#) of false positives in our four networks was 1 (the total number of true NAEs is 16.5), shown as in

Fig. 4(b). However there are relatively large number of false negatives in NAED-noSC, and the Avg. # of false negatives in the four networks was 6; There was no false positive in NAED-bySC, however there are a few false positives in NAED-bySC because that we do not consider all the spatial correlation. The Avg. # of false negatives of NAED-bySC in our four networks was 1.5, and other results are shown in Fig. 4(b). These experimental results verify the correctness of the accuracy analysis of NAED-noSC and NAED-bySC in Subsection 5.1 and Subsection 5.2.

The accuracies of NAED-bySC in the four networks were 90.9%, 93.75%, 90.91% and 88.89% respectively, and the average accuracy of NAED-bySC was more than 90% (91.11% actually). The accuracy of NAED-noSC was relatively lower than NAED-bySC, namely 54.55%, 62.5%, 59.09% and 55.56% respectively in our four networks, which are shown in Fig. 5(a).

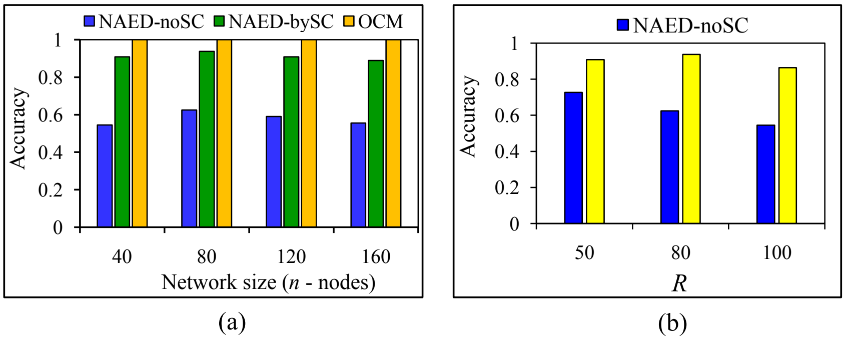


Fig. 5. (a) Accuracy comparison ($R = 80$). (b) Accuracy comparison (80 nodes).

With the increase of node density, the accuracy of NAED-noSC decreases. In our 80 nodes network, with the increase of R (equivalent to the increase of node density), the accuracy of NAED-noSC decreased, shown as the blue column chart in Fig. 5(b). This is because the increase of node density leads to the increase of spatial correlation, thus NAED-noSC will miss more true NAEs. The variation of node density (or the change of communication radius in the same network) almost has no effect on the accuracy of NAED-bySC. This is because that the increase of spatial correlation among nodes contributes to the increase of true NAEs, meanwhile, more NAEs will be detected in NAED-bySC. Therefore, there is almost no causal relationship between accuracy and node density, which is shown as the yellow column chart in Fig. 5(b).

7 Conclusions

This paper presents a novel efficient framework for event prewarning in sensor networks with multi MEs, which mainly includes a simple and practical data preprocessing method, NNE detection algorithm, event probability encodings of

NNEs and two distributed NAE detection algorithms (NAED-noSC and NAED-bySC). Experimental evaluation demonstrates our approach reduces the data transmission greatly compared with conventional approaches, and NAED-bySC guarantees good detection accuracy. When the network situation is bad, namely when the proportion of NNEs with relatively high *GRADE*s is large, NAED-noSC is a more suitable approach with few false negatives. Our on-going work is to enhance our approach to deal with the phenomenon in which the alert event enlarges or disappears with time elapsed.

References

1. Aslan, Y.E., Korpeoglu, I., Ulusoy, Ö.: A Framework for Use of Wireless Networks in Forest Fire Detection and Monitoring. Computers, Environment and Urban Systems 36, 614–625 (2012)
2. Kapitanova, K., Son, S.H., Kang, K.-D.: Using Fuzzy Logic for Robust Event Detection in Wireless Sensor Networks. Ad Hoc Netw. 10, 709–722 (2012)
3. Hubbell, N., Han, Q.: DRAGON: Detection and Tracking of Dynamic Amorphous Events in Wireless Sensor Networks. IEEE T. on Parall. and Distr. 23, 1193–1204 (2012)
4. Shih, K.-P., Wang, S.-S., Chen, H.-C., Yang, P.-H.: CollECT: Collaborative Event Detection and Tracking in Wireless Heterogeneous Sensor Networks. Comput. Commun. 31, 3124–3136 (2008)
5. Zhang, Y., Meratnia, N., Havinga, P.: Outlier Detection Techniques for Wireless Sensor Networks: A Survey. IEEE Commun. Surveys & Tutorials. 12, 159–170 (2010)
6. Premkumar, K., Prasanthi, V.K., Anurag, K.: Delay Optimal Event Detection on Ad Hoc Wireless Sensor Networks. ACM T. on Sensor Netw. 8, 1–39 (2012)
7. Luo, X., Dong, M., Huang, Y.: On Distributed Fault-tolerant Detection in Wireless Sensor Networks. IEEE Trans. on Comput. 55, 58–70 (2006)
8. Ould-Ahmed-Vall, E., Ferri, B.H., Riley, G.F.: Distributed Fault-tolerance for Event Detection Using Heterogeneous Wireless Sensor Networks. IEEE T. on Mobile Comput. 11, 1994–2007 (2012)
9. Taheri, H., Neamatollahi, P., Younis, O.M., Naghibzadeh, S., Yaghmaee, M.H.: An Energy-Aware Distributed Clustering Protocol in Wireless Sensor Networks Using Fuzzy Logic. Ad Hoc Netw. 10, 1469–1481 (2012)
10. Samuel, M., Michael, J.F., Joseph, H., Wei, H.: TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. In: Fifth Symp. Oper. Syst. Desi. and Impl., pp. 131–146. ACM (2002)
11. Sharma, A.B., Golubchik, L., Govindan, R.: Sensor Faults: Detection Methods and Prevalence in Real-world Datasets. ACM T. on Sensor Networks. 6, 1–34 (2010)
12. Arning, A., Agrawal, R., Raghavan, P.: A Linear Method for Deviation Detection in Large Database. In: ACM KDD, pp. 164–169. ACM (1996)
13. OMNET++ platform, `http://www.omnetpp.org`
14. LUCE, `http://sensorscope.epfl.ch/index.php/Environmental_Data`

Efficient Parallel Block-Max WAND Algorithm

Oscar Rojas[1], Veronica Gil-Costa[2,3], and Mauricio Marin[1,2]

[1] DIINF, University of Santiago, Chile
[2] Yahoo! Labs Santiago, Chile
[3] CONICET, University of San Luis, Argentina

Abstract. Large Web search engines are complex systems that solve thousands of user queries per second on clusters of dedicated distributed memory processors. Processing each query involves executing a number of operations to get the answer presented to the user. The most expensive operation in running time is the calculation of the top-k documents that best match each query. In this paper we propose the parallelization of a state of the art document ranking algorithm called Block-Max WAND. We propose a 2-steps parallelization of the WAND algorithm in order to reduce inter-processor communication and running time cost. Multi-threading tailored to Block-Max WAND is also proposed to exploit multi-core parallelism in each processor. The experimental results show that the proposed parallelization reduces execution time significantly as compared against current approaches used in search engines.

1 Introduction

Large-scale Web search engines are built as a collection of services hosted by the respective data center. Each service is deployed on a set of processing nodes (processors) of a high-performance cluster of computers. Services are software components such as (a) calculation of the top-k documents that best match a query; (b) routing queries to the appropriate services and blending of results coming from them; (c) construction of the result Web page for queries; (d) advertising related to query terms; (e) query suggestions, among many other operations. The service relevant to this paper is the top-k calculation service.

The top-k calculation nodes are assumed to perform document ranking for queries by using the WAND algorithm [4]. This algorithm is been currently used by a number of commercial vertical search engines. The concept is that the document collection is evenly distributed on P processing nodes or partitions, and for each partition an inverted index is constructed from the respective documents. The inverted indexes enable the fast determination of the documents that contain the terms of the query under processing. They contain additional static and dynamic data to enable the WAND algorithm to reduce the number of documents that are fully evaluated during the ranking process. The static value, called *upper-bound* value, is calculated for each index term at construction time whereas the dynamic value, called *threshold* value, starts in zero for each new query and is updated during the WAND computations across the inverted file.

To answer a query, a broker machine sends the query to all of the P partitions. Then for each query, the top-k nodes locally compute the most relevant documents and send them to a broker machine. Later, the broker merges those $P \times k$ document results to obtain the global top-k document results. Hence, our aim is to reduce the total number of full document evaluations (score calculations) performed during the document ranking process, which in turn leads to a reduction in the running time required by the WAND algorithm to finish the ranking process. Also our aim is to reduce the total number of documents communicated among the top-k calculation nodes and the broker machine. The big picture is that by doing this one can be able to reduce the total number of top-k calculation nodes deployed in production.

In this paper we propose to make the top-k calculation service efficient in the sense of significantly reducing the average amount of computation and communication per query executed by the processing nodes hosting the service (namely, the top-k calculation nodes). We propose a 2-steps algorithm which determines the number of document results that must be sent to the broker by each top-k calculation node. The algorithm does not lose precision of results. We also propose a multi-threading strategy to schedule query processing in each of the top-k calculation nodes by using the Block-Max WAND (BMW) algorithm proposed in [6]. We analyze the effect of running queries with both high and low computational costs over a cluster of processors supporting multi-threading.

The remaining of this paper is organized as follows. Section 2 reviews related work and Section 2.1 describes the WAND algorithm. Section 3 describes our proposal. Section 4 presents a performance evaluation study considering different metrics and Section 5 presents conclusions.

2 Background and Related Work

To speed up query processing, the top-k calculation service relies on the use of an inverted index or inverted file) which is a data structure used by all well-known Web Search Engines. This index enables the fast determination of the documents that contain the query terms and contains data to calculate document scores for ranking. The index is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of relevant terms found in the document collection. Each of these terms is associated with a posting list which contains the document identifiers where the term appears in the collection along with data used assign a score to the document. To solve a query, it is necessary to get from the posting lists the set of documents associated with the query terms and then to perform a ranking of these documents in order to select the top-k documents as the query answer.

One important bottleneck in query ranking is the length of the inverted file, which is usually kept in compressed format. To avoid processing the entire posting lists or reducing the amount of expensive computations performed in each posting list item like the WAND [4]. Several optimizations have been proposed in the technical literature for the WAND algorithm. The aim of [8] and [10] is to improve performance by computing tighter upper bound scores for each term. The work in [1] and [2] propose to maintain the posting lists ordered by score instead of document identifier (docID) in order to retrieve and evaluate first the documents with the highest scores. But in these cases, compression tends to be less effective. An optimization of the WAND algorithm named the Block-Max WAND [6] has been devised to avoid decompressing the entire posting lists.

A number of papers have been published on parallel query processing upon distributed inverted files (for recent work see [9]). Many different methods for distributing the inverted file onto P partitions and their respective query processing strategies have been proposed in the literature [4]. The different ways of doing this splitting are mainly variations of two basic dual approaches: document partition and term partition. In the former, documents are evenly distributed on P partitions and an independent inverted index is constructed for each of the P sets of documents. In the last one, a single inverted index is constructed from the whole document collection to then distribute evenly the terms with their respective posting lists onto the P partitions.

Also query throughput is further increased by using application caches (for recent work see [12]). But, to the best of our knowledge, the work presented in this paper is the first attempt to evaluate and determine the relevant features that have to be taken into consideration when evaluating the WAND algorithm in a distributed cluster of processors supporting multi-threading.

2.1 The WAND Query Evaluation Process

The method proposed by Broder et al. [4] assumes a single threaded processor containing an inverted index. As usual, each query is evaluated by looking for query terms in the inverted index and retrieving each posting list. Documents referenced from the intersection of the posting lists allow to answer conjunctive queries (AND bag of word query) and documents retrieved at least from one posting list allow to answer disjunctive queries (OR bag of word query).

The ranking is used to compute the similarity between documents and the query. Then this function returns the top-k documents. There are several ranking algorithms such as BM25 or the vector model [4]. Ranking algorithms should be able to quickly process large inverted lists. But the size of these lists tends to grow rapidly with the increasing size of the Web. Therefore, in practice, these algorithms use early termination techniques avoiding processing complete lists [3]. In some early termination techniques the posting lists are sorted so that most relevant documents are found first. Other ingenious techniques have been proposed when the posting lists are sorted by docIDs. They reduce running time avoiding computing the scores of all documents of the posting lists by skipping document score computations. This is the case of the WAND method proposed by [4].

The WAND approach uses a standard docID sorted index. It is a query processing method based on two levels. In the fist level, some potential documents are selected as results using an approximate evaluation. Then, in the second level those potential documents are fully evaluated to obtain their scores. This two-steps process uses a heap to keep the current top-k documents where in the root is located the document with least score. The root score provides a threshold value which is used to decide the full score evaluation of the remaining documents in the posting lists associated with the query terms. To this end the algorithm iterates through posting lists to evaluate them quickly using a pointer movement strategy based on pivoting. In other words, pivot terms and pivot documents are selected to move forward in the posting lists which allows skipping many documents that would have been evaluated by an exhaustive algorithm.

In Figure 1.(a) we show how the WAND algorithm works for a query with three terms "*tree, cat* and *house*". First, posting lists of the query terms are sorted by docIDs upper bounds (UBs) from top to bottom. Then we add the upper bounds of the terms until we get a value greater or equal to the threshold. In Figure 1.(a) by adding the UBs of the first two terms we get $(2+4 \geq 6)$. Thus *cat* is selected as the pivot term. We assume that the current document in this posting list is "503". Therefore, this document becomes the pivot document. If the first two posting lists do not contain the document 503, we proceed to select the next pivot. Otherwise we compute the score of the document. If the score is greater or equal to the threshold we update the heap by removing the root document and adding the new document. This iterative algorithm is repeated until there are no documents to process or until it is no longer possible for the sum of the upper bounds to exceed the current threshold.

The work presented in [6] proposes using compressed posting lists organized in blocks (see Figure 1.(b)). Each block stores the upper bound (Block max) for the documents inside that block in uncompressed form, thus enabling to skip large parts of the posting lists by skipping blocks. This drastically reduces the cost of the WAND algorithm but does not guarantee correctness because some relevant documents could be lost. To solve this problem, the authors propose a new algorithm that moves forward and backwards in the posting lists to ensure that no documents are missed. Independently, the same idea was presented in [5]. In this later work, authors presented an algorithm for disjunctive queries that first performs pre-processing to split blocks into intervals with aligned boundaries and to discard intervals that cannot contain any document capable of making into the top-k results. Multi-threading algorithms for ranking methods different to WAND have been studies in [11] and [7].

3 Two-Level Ranking on a Distributed Search Engine

In this work we consider a Web search engine in which there is one broker and P top-k calculation nodes, where P indicates the level of document partitioning considered in the distribution of the document collection. Each top-k calculation node has its own inverted index, where the posting lists refer to local documents. When a new query arrives, the broker sends the query for evaluation to each node. Then, the nodes work on their inverted indexes to produce query answers and pass the results back to the broker. Conventionally, search engines use the standard asynchronous multiple master/slave paradigm to process queries.

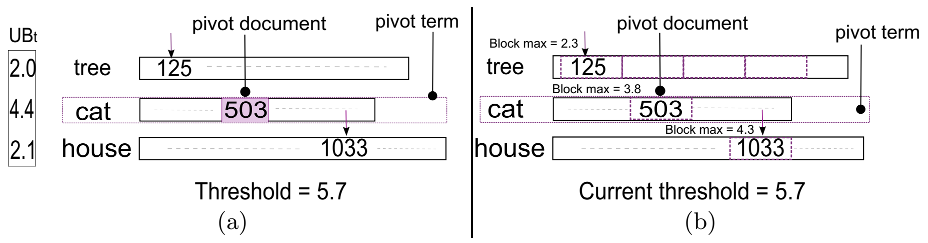


Fig. 1. (a) Executing WAND algorithm. (b) The Block-Max WAND (BMW): each block stores the upper bound for the documents inside the block.

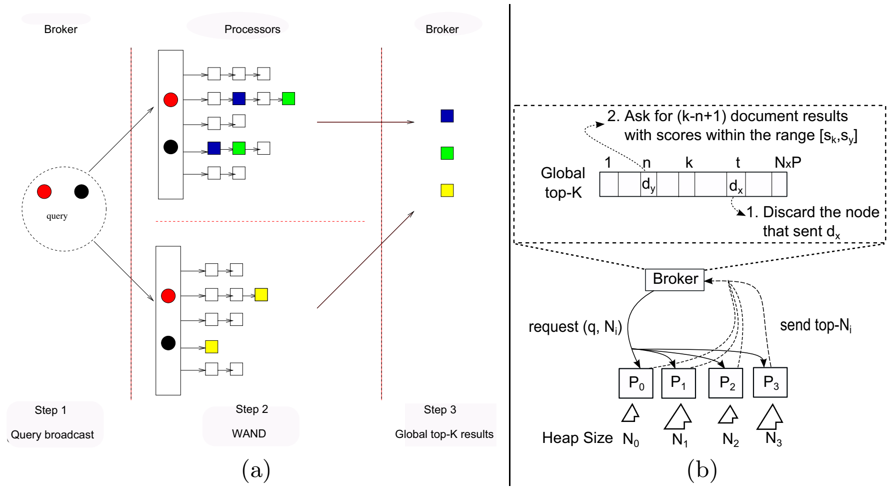


Fig. 2. (a) A distributed WAND query evaluation process. (b) Proposed algorithm.

A WAND-based search engine in this context considers parallel term iteration over each top-k calculation node. If a document satisfies the threshold condition, then it is inserted in the heap of the top-k document results. Notice that document identifiers are kept sorted in posting lists allowing iteration in linear time proportional to the posting list length. Finally, a full document score evaluation is executed over the set of candidate documents determined by the WAND algorithm, and the global top-k results are determined. Figure 2.(a) illustrates this process.

Let us shortly discuss some considerations for the query evaluation process. There exist two alternatives for WAND scores estimation. As the search engine is document-partitioned, upper bounds for each term can be calculated considering each sub-collection, i.e. each processor determines its own term upper bounds. In this case upper bounds are stored in local inverted indexes and eventually each term can register P different upper bounds. Another alternative is to calculate the upper bound for each term considering the full collection, thus upper bounds correspond to global scores. In this case it is also possible to store these values in the local inverted indexes, replicating upper bounds across the processors. We consider that the second alternative is more recommendable because it allows to skip more documents in the WAND iteration process.

3.1 Distributed 2-Steps Algorithm

In this section we detail our proposed algorithm which aims to obtain accurate results while reducing the number of operations (scores calculations) performed when executing the WAND algorithm, the memory allocated to the heap data structure and the communication among processors. The algorithm works as follows: (1) in the first step, the top-k calculation nodes send $N = (k/P + \alpha) < k$ document results to the broker machine, (2) the broker machine detects and discards the nodes that do not have more relevant documents, (3) the broker requests more documents to the remaining nodes. Our hypothesis is that k/P is the average number of document results retrieved from each top-k calculation node as the document collection tends to be evenly distributed among the P partitions. Thus we propose to emulate ranking as a distributed priority queue.

Formally, given a set of top-k nodes $P = \{p_1, p_2, \ldots, p_p\}$ and for each query q, each node $p_i \in P : 1 \leq i \leq p$, sends to the broker machine a set of document results $\{< d_1, sc_1 >< d_2, sc_2 >, \ldots, < d_N, sc_N >\}$, where d_j is the document identifier, sc_j is the score associated with the document and N is the heap size (i.e. the number of document results initially requested). Each top-k node computes a disjoint set of document results.

After the broker machine performs the merge of $N \times P$ document results, it gets a list of tuples: $[< d_{x,1}, sc_{x,1}, i >, \ldots, < d_{w,N\times P}, sc_{w,N\times P}, s >]$. The first component of each tuple represents the document identifier and the position of the document within the global document result set. For example, $d_{x,1}$ represents the most relevant document (i.e. the top-1 document) identified by x. The second component represents the score of the document using the same nomenclature. The third component represents the identifier of the node which sent the document. Then, the proposed algorithm determines whether to request more documents results to the top-k nodes as follows (see Figure 2.(b)):

1. If $\exists < d_{x,t}, sc_{x,t} > \in p_i$ with $sc_{x,t} > sc_{s,k}$ (i.e. the global position of document x is $t > k$) then remove p_i from the list. In other words, no more document results are requested to p_i.
2. If $\exists < d_{y,n}, sc_{y,n} > \in p_i$ with $sc_{y,n} < sc_{s,k}$ and $sc_{y,n}$ is the least relevant document sent by p_i, then request to p_i the next $k - n + 1$ document results. $sc_{s,k}$ is the score of the k-th document in the global set of results. The upper-bound for those requested documents is given by $sc_{y,n}$ and the lower-bound is given by $sc_{s,k}$. In other words, the broker machine requests to p_i documents with scores in the range of $[sc_{s,k}, sc_{y,n}]$. The number of requested documents is given by $k - n + 1$, because $k - n$ is the number of documents required to get the k-th position and $+1$ is used to support documents with the same score.

If we have a set of document results $\{< d_{a,1}, sc_{a,1}, 1 >, < d_{b,2}, sc_{b,2}, 1 >, < d_{c,3}, sc_{c,3}, 2 >, < d_{d,4}, sc_{d,4}, 3 >, < d_{e,5}, sc_{e,5}, 2 >, < d_{f,6}, sc_{f,6}, 3 >\}$ for $P = \{p_1, p_2, p_3\}$ where $|P| = 3$. In this example $k = 4$ and $\alpha = 1$, therefore each processor sends $N = 4/4 + 1 = 2$ documents results to the broker machine. In this case, p_2 and p_3 are discarded and no more documents are required from them because they have sent documents that are located after the k-th global position. On the contrary, it is necessary to ask to p_1 a total of $k-2+1=3$ more documents, with scores is in the range $[sc_{d,4}, sc_{b,2}]$. Recall that the formula uses $+1$ to include documents with the same score. As in the first step the broker sets $N=2$, the size of the heap used by the WAND algorithm to store the most relevant documents at the nodes side is set to $N=2$. But in the next step, the broker machine asks to p_1 a total of $N=3$ documents results. Therefore the size of the heap is set to 3. To avoid re-computing all document results from the beginning, the broker also sends the range $[sc_{d,4}, sc_{b,2}]$ to p_1 to discard documents with scores outside this range.

The α parameter used to request document results in the first step is dynamically set for each query and for each top-k node as follows. For a period of time Δ_i we warm up the system by running an oracle algorithm which predicts the optimum value of α. In other words, queries are solved in the first step of the proposed algorithm. Also, every time a query is processed, we store the query terms (t_i), the posting list size of each term (L_i) and the optimal α parameter for each top-k node $(q_a =< t_1, t_2, L_1, L_2, \alpha_{p_1}, \ldots \alpha_{p_P} >)$. The optimal α parameter determines the additional number of documents each node has to send to the broker machine. This information is kept in memory for just one interval of time. Then, in the following intervals of times $[\Delta_{i+1}, \ldots \Delta_{j-1}, \Delta_j]$ where Δ_j is the current period of time, we estimate the α_{p_i} values for a query q_a by applying the rules:

1. If q_a was processed in Δ_{j-1} we use the value α_{p_i} stored in the previous interval for $i = 1 \ldots P$.
2. We define the set $X = q_a.\text{terms} \cap q_b.\text{terms}$,$\forall q_b$ processed in Δ_{j-1}. Then if $X <> \emptyset$, we select the query q_b which maximize $|X|$ (i.e. the query with the greatest amount of terms in X). If more than one query has the same maximum amount of terms in X, then we select the one which contains the term with the largest posting list.
3. Otherwise, for each top-k node p_i we use the average α_{p_i} value registered in Δ_{j-1}.

3.2 Multi-threading Algorithms

In this section we describe our multi-threading algorithms. Our first algorithm uses a local heap (**LH**) data structure for each thread. The posting lists are evenly distributed among threads. In other words, each thread holds a portion of the inverted file. Then each thread process incoming queries with its own local inverted file and using a local heap of size k, where k is the top-k documents retrieved to the user. At the end, the documents identifiers stored in the local heaps have to be merged. A synchronization barrier is implemented before computing the merge operation.

Our second approach, named shared heap (**SH**) inverted files are distributed among threads as in the LH approach. But query processing is performed using a global heap of size k. Then, no merge operation is performed at the end of the query process, but additional *locks* have to be implemented to guarantee exclusive access to the threads when updating the heap contents. Our hypothesis is that for large collections where queries are more expensive to compute, it is better to use a single global heap, because the threshold of the heap can be quickly updated. Achieving the optimal threshold value quickly has several advantages: (1) we can reduce the number of scores computations and (2) fewer heap update operations are preformed (reducing the number of locks).

4 Evaluation

4.1 Data Preparation and Experiment Settings

We experiment with a query log of 16,900,873 queries submitted to the AOL Search service between March 1 and May 31, 2006. We set the term weights according to the frequency of the term in the query log. Then, we applied these queries to a sample (1.5 TB) of the UK Web obtained in 2005 by Yahoo!, over a collection compounded by 26,000,000-terms and a 56,153,990-document inverted index. We consider a Web search engine computing top-100 and top-1000 document results. In the case of the top-100 results we considered a sample of 1,000,000 queries.

We divide the following experiments into two groups. First, we evaluate our proposed 2-steps algorithm in a distributed cluster of 16 processors. We measure the number of scores computations, number of heap updates, running time and communication cost. Second, we compute the number of decompressed blocks performed by the Block-Max WAND (BMW) [6] using the multi-threading LH and SH approaches. The Block-Max WAND algorithm groups the posting lists into blocks of fixed size. Each block stores 100 documents in compress form. The optimal threshold value is quickly reached because each block has its own UB_t. We also measure the speed-up achieved by both multi-threading algorithms. The results were obtained on a cluster of 16 processors with 32-core AMD Opteron 2.1GHz Magny Cours processors, sharing 32 GB of memory. All experiments were run using an inverted file of 27Gb in main memory.

4.2 Distributed Algorithm Evaluation

We compare the performance achieved by our proposal against the art baseline algorithm which always request k document results per query to each node partition. Figure 3 shows normalized running times and Figures 4 shows communication cost. We show results normalized to 1 in order to better illustrate the comparative performance. To this end, we divide all quantities by the observed maximum in each case.

For a small top-k, our proposal algorithm significantly improves the baseline performance by 40%. This means that the proposal can reduce the total number of scores computation performed by the WAND algorithm at the top-k nodes side and the number of document results communicated to the broker machine. This claim is confirmed in Figure 4.(a) which shows that the proposal reduces communication cost logarithmically. Also, Figure 5.(a) shows that the proposal reduces the average number of score computation by 50% with 16 processors

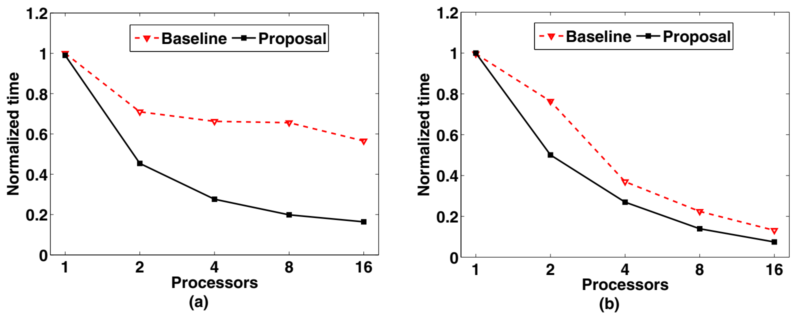


Fig. 3. Running time reported for (a) top-100 and (b) top-1000 document results

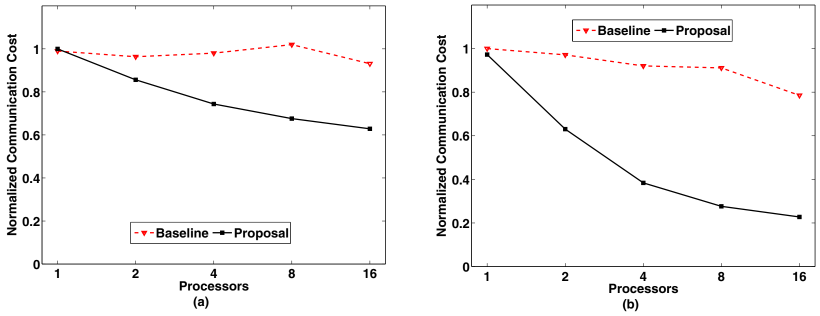


Fig. 4. Communication cost for (a) top-100 and (b) top-1000 document results

and Figure 5.(c) shows that for the same number of processors the number of heap updates is reduced by almost 60%.

With a larger number of document results (Figure 3.(b)) the gain achieved by the proposal algorithm (in terms of running time) is reduced to 10% in average. Again, these results are validated in Figure 5.(b) and Figure 5.(d) for the number of score computation and the number of heap updates respectively. In this case, the proposal algorithm reports 3% less number of score computations than the baseline and 25% less heap updates for a total of 16 processors. However, communication cost is improved by the proposal due to less than k document results are requested per query per top-k node.

To understand the effect of using a larger value of k we have to remember that posting lists follow the Zip law [4]. In other words, there are few documents with high score and many documents with low score. Moreover, posting lists are in compress form organized in blocks and each block has an upper bound named block max for the documents store inside the block. Therefore, with a larger k the threshold value tends to be small and more score computations are preformed due to less blocks are discarded when comparing the block max and the threshold.

4.3 Multi-threading Algorithms Evaluation

The total cost of solving a query is determined by three main stages: (1) the load of parameters, (2) the search algorithm and (3) the merge and sort of

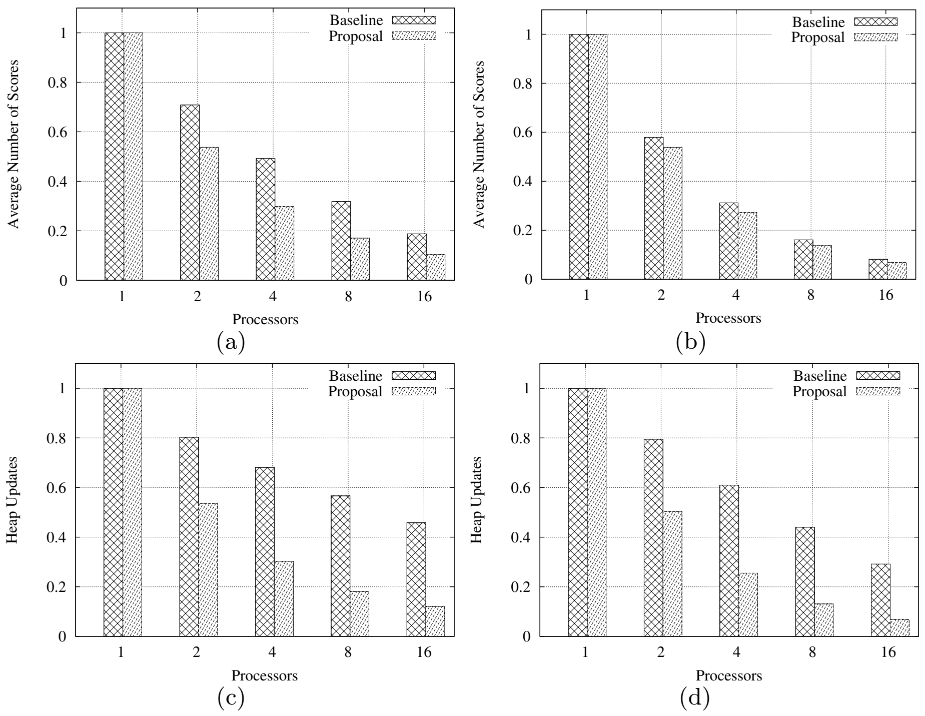


Fig. 5. Score computation and heap update operations reported for (a)(c) top-100 and (b)(d) top-1000 document results

results. The first stage involves the computation of the upper bounds UB for each block of the posting lists and also the multiplication of each term weight with the UB. This product is used at running time. In the second stage we apply the WAND to find the top-k documents for the query. In the last stage we perform the merge of local heaps and sort the final results. Usually the higher computational cost is performed by the last two stages. The algorithm that computes the top-k documents and merge tasks and sort algorithms. Therefore, we focus on the number of scores, the number of heap updates and number of blocks decompressed using the Max-Block WAND.

Figure 6 shows the results regarding the number of decompressed blocks. First we show the average number of blocks required by query per thread. Then we show the average number of decompressed blocks using both multi-threading approaches and its relation to the number of heap updates.With a small k value, the algorithms decompress only the blocks containing the documents that will be added to the heap. These results show that the SH approach drastically reduces the number of decompressed blocks which is an essential issue in search engines where fast access to the information and resource usage optimization is essential.

This because queries are evaluated in parallel and threshold values are accessed and updated jointly by all threads. The threshold value is the decisive factor in the speed of the WAND algorithm. It is used to decide whether to

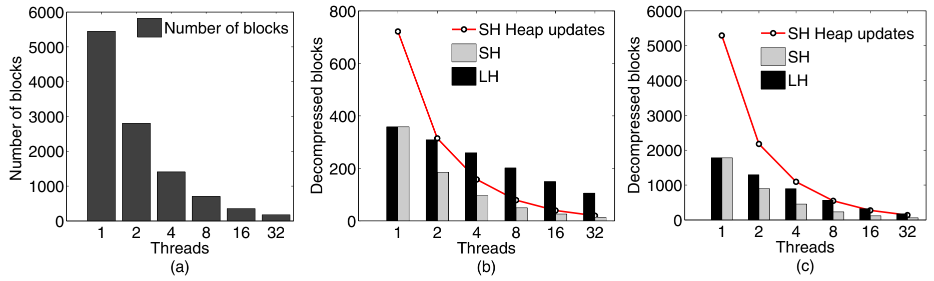


Fig. 6. Number of decompressed blocks achieved by both the LH and SH approaches

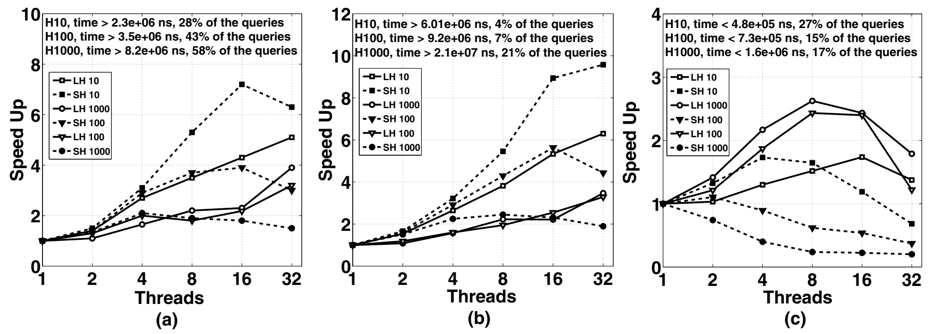


Fig. 7. (a)Speed-up for expensive queries requiring at least a running time of {2ms,3ms,8ms}) for a heap size of $k = \{10, 100, 1000\}$ (b)Speed-up for expensive queries requiring at least a running time of {6ms,19ms,20ms}) for a heap size of $k = \{10, 100, 1000\}$ (c) Speed-up for queries that have a low computational cost requiring a running time less than {0.4ms,0.7ms,1.6ms} for a heap size of $k = \{10, 100, 1000\}$

compute a score and whether to access a compress block of the posting list. Also, regardless of multi-threaded version, this figure shows that the WAND algorithm is very efficient when unzipping the posting lists. By using a heap of size $k = 100$, the worst decompression level is reached with a single thread.

Figures 7.(a) and 7.(b) show the speed-up obtained for queries that are expensive to compute. Typically, these queries tend to compute a larger number of scores and make a greater number of updates on the heap. Figure 7.(a) shows results for a heap of size $k = 10$ and executing 28% of the queries log. The sequential average time to run these queries is $2.3e + 06ns$. For a heap of size $k = 100$ we executed 43% of the query log and for a heap size of $k = 100$ we executed 58% of the query log. Figure 7.(b) shows the same experiment but for queries more expensive to process, requiring at least a sequential processing time of $6.01e + 06ns$ for a heap size of $k = 10$, $9.2e + 06$ for a heap size of $k = 100$ and $2.1e + 07$ for a heap size of $k = 1000$. With this experiment we show that the performance of the SH approach begins to increase when computing queries with higher computational cost. Finally, Figure 7.(c) shows results obtained for queries requiring a low computational cost. In other words, queries that are quickly solved with a low number of scores calculations and few heap updates. The LH approach presents the best performance.

5 Conclusions

We have presented a 2-steps algorithm which aims to reduce computation and communication cost between top-k nodes and the broker machine. In the first step, it uses an adjustment parameter which is dynamically set for each query to request results from each processor. Our experiments show that the proposed algorithm significantly outperforms the baseline strategy.

We also evaluated the WAND algorithm using two multi-threading strategies. We performed experiments using queries with both high and low computational costs. The shared heap (SH) multi-threading approach is less efficient than the local heaps (LH) approach for low cost queries. In general, SH reduces the number of decompressed blocks, the number of heap updates and the number of scores computed to retrieve the top-k documents. However, the gain in these metrics is not reflected in the execution time for low cost queries. Thus a combination of both approaches should be used based on a prediction of the running time cost of queries. In practice, the running time tends to be proportional to the size of the involved posting lists.

References

1. Anh, V.N., Moffat, A.: Pruned query evaluation using pre-computed impacts. In: SIGIR, pp. 372–379 (2006)
2. Bast, H., Majumdar, D., Schenkel, R., Theobald, M., Weikum, G.: Io-top-k: Index-access optimized top-k query processing. In: VLDB, pp. 475–486 (2006)
3. Blanco, R., Barreiro, A.: Probabilistic static pruning of inverted files. ACM Trans. Inf. Syst. 28(1) (2010)
4. Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.Y.: Efficient query evaluation using a two-level retrieval process. In: CIKM, pp. 426–434 (2003)
5. Chakrabarti, K., Chaudhuri, S., Ganti, V.: Interval-based pruning for top-k processing over compressed lists. In: ICDE, pp. 709–720 (2011)
6. Ding, S., Suel, T.: Faster top-k document retrieval using block-max indexes. In: SIGIR, pp. 993–1002 (2011)
7. Jonassen, S., Bratsberg, S.E.: Intra-query Concurrent Pipelined Processing for Distributed Full-Text Retrieval. In: Baeza-Yates, R., de Vries, A.P., Zaragoza, H., Cambazoglu, B.B., Murdock, V., Lempel, R., Silvestri, F. (eds.) ECIR 2012. LNCS, vol. 7224, pp. 413–425. Springer, Heidelberg (2012)
8. Long, X., Suel, T.: Optimized query execution in large search engines with global page ordering. In: VLDB, pp. 129–140 (2003)
9. Marin, M., Costa, V.G.: Sync/async parallel search for the efficient design and construction of web search engines. PARCO 36(4), 153–168 (2010)
10. Strohman, T., Turtle, H.R., Croft, W.B.: Optimization strategies for complex queries. In: SIGIR, pp. 219–225 (2005)
11. Tatikonda, S., Cambazoglu, B., Junqueira, F.: Posting list intersection on multicore architectures. In: SIGIR (2011)
12. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: WWW, pp. 401–410 (2009)

Gunther: Search-Based Auto-Tuning of MapReduce

Guangdeng Liao, Kushal Datta, and Theodore L.Willke

Intel Labs, Hillsboro, Oregon, USA
{guangdeng.liao,kushal.datta,theodore.l.willke}@intel.com

Abstract. MapReduce has emerged as a very popular programming model for large-scale data analytics. Despite its industry-wide acceptance, the open source Apache™ Hadoop™ framework for MapReduce remains difficult to optimize, particularly in large-scale production environments. The vast search space defined by the hundreds of MapReduce configuration parameters and the complex interactions between them makes it time consuming to rely on *manual tuning*. Hence something more is needed. In this paper we evaluate approaches to the automatic tuning of Hadoop MapReduce including ones based on cost-based and machine learning models. We determine that they are inadequate and instead propose a search-based approach called Gunther for Hadoop MapReduce optimization. Gunther uses a Genetic Algorithm which is specially designed to aggressively identify parameter settings that result in near-optimal job execution time. We evaluate Gunther on two types of clusters with different resource characteristics. Our experiments demonstrate that Gunther can obtain near-optimal performance within a small number of trials (<30), outperforming existing auto-tuning solutions and industry recommended configurations. We also describe a methodology for reducing the dimensionality of the auto-tuning problem, further improving search efficiency without sacrificing performance improvement.

Keywords: Hadoop, Genetic Algorithm, Parameter Optimization, Auto-tuning.

1 Introduction

MapReduce is a distributed programming model used to process large datasets across thousands of machines. It has gained much popularity due to its simple yet expressive interface, scalability and fine-grained fault tolerance [5, 9]. Apache™ Hadoop™ [9] is an open-source implementation of the MapReduce model and is widely used for data mining, log processing and machine learning [7, 16, 17, 18, 22]. Hadoop exposes 200+ parameters providing users the flexibility to customize it according to their need [26]. Some parameters have significant performance impact. The major challenge lies in quickly identifying the best parameter settings for a particular application on a given cluster [1, 4, 25].

The common practice is to tune up Hadoop using rule-of-thumb settings published by industry leaders, such as Cloudera and MapR [4, 9, 25], but these recommendations are too general and fail to capture the specific requirements of a given application and resource constraints (i.e., amount of CPU, network and storage) of a given cluster. Additionally, the large parameter space, with its complex inter-dependencies, and the sheer scale of many clusters increases the complexity of manual tuning, in which a person repeatedly runs jobs in an attempt to identify the best parameter settings using trial-and-error. An *efficient*, *effective* and *automated* approach to parameter optimization is the only viable solution.

Cost-based auto-tuning is used in database systems [3]. Motivated by this, researchers proposed a cost-based approach for Hadoop MapReduce optimization [12, 13]. However, it is extremely difficult for simple cost-based models to accurately predict the performance of a wide range of Hadoop applications over a wide range of clusters provisioned with different CPU, storage, memory and network technologies. Furthermore, cost-based models are strictly bound to a particular version of a framework and do not evolve with the framework. Machine learning models are another popular approach [3, 15, 24]. In contrast to cost-based models they rely on training sets to "learn" model coefficients and hence are more adaptive and flexible. Unfortunately, our studies show that it is difficult to construct an accurate machine learning model without a large training set involving hundreds or potentially thousands of trials.

In this paper we propose Gunther, a search-based auto-tuner for Hadoop MapReduce that addresses these challenges. It employs a search algorithm that iteratively evaluates the variation in performance of a MapReduce application for different configuration settings and often attains a near-optimal solution. Our method extends to any version of the Hadoop framework and different types of clusters and thus is flexible and adaptive.

We evaluate auto-tuning approaches using the two metrics – (i) *efficiency* or how fast the search can find a good configuration, and (ii) *effectiveness* which measures the performance improvement achieved. We study the performance of Hadoop on a number of clusters and discover that it is a nonlinear and multimodal function of Hadoop's configuration settings. We evaluate search algorithms that are popular for finding global optima on multimodal surfaces and select Genetic Algorithm (GA) as our search strategy [23]. We then optimize GA for the Hadoop auto-tuning problem to strike the right balance between search efficiency and effectiveness. We evaluate the result on two clusters with different resource characteristics for several applications. Our experiments demonstrate that Gunther obtains near-optimal configurations within 30 trials in both types of clusters, and yields better performance improvement than configurations recommended by a cost-based approach and industry rule-of-thumb settings. Studies of workload characteristics show less than 10% of the jobs have runtimes of 5 hours or more [31]. Hence for majority of Hadoop users 30 trials is a small price. For larger jobs tuning time is ameliorated when many users keep running their applications for years.

While Gunther is very effective at identifying near-optimal configurations, its efficiency can be further improved by reducing the dimensionality of the search space by ignoring parameters that have little performance effect. We propose a methodology that uses job counters to classify applications into groups that are sensitive to the same or nearly the same subset of parameters. Once applications are classified we limit the search to the subset and achieve near-optimal performance while reducing the search time.

2 Background and Motivation

2.1 Hadoop and MapReduce

In MapReduce, users need only implement map and reduce functions and the rest is handled by the framework. The Hadoop MapReduce framework takes care of task scheduling, parallelization, inter-process communication, load balancing, and fault tolerance. Large datasets are partitioned into small blocks by Hadoop Distributed File System (HDFS). Execution proceeds in three phases: map, shuffle, and reduce. Instances of the user-defined map function, or map tasks, process key-value pairs from one data block in HDFS to emit a list of intermediate key-values. These pairs are then operated on by instances of reduce tasks which aggregate them and store them back to HDFS. After the completion of map tasks the intermediate files are copied to the reduce nodes. This comprises the network-intensive shuffle phase. Reduce tasks fetch the key-value pairs from map output files and then sort and merge them before processing them. After processing reduce tasks write the final key-value pairs back to HDFS.

2.2 Motivation of Parameter Tuning

Hadoop is parameterized to permit users to manage the dataflow in different phases. Hadoop versions beyond 0.20 expose 200+ tunable parameters and 10+ parameters affect its performance [4, 9]. Figure 1 shows how the total number of reduce tasks affect sort performance on an 8-node Intel® Xeon® processor E3 cluster. As the number of tasks is increased from the default of 8 to 80, the job execution time improves by a factor of 1.9X. This is because increasing the number of reduce tasks reduces the amount of data per reduce task and alleviates storage contention during shuffling. A strong impact is also observed for memory buffer size for sort (i.e., *io.sort.mb*) in Figure 1. The performance difference between the best and worst settings is 15%. *io.sort.mb* determines when to sort and spill data to storage. It affects both CPU sorting time and storage write time. All these demonstrate that the performance of Hadoop MapReduce applications can be substantially improved by tuning these parameters.

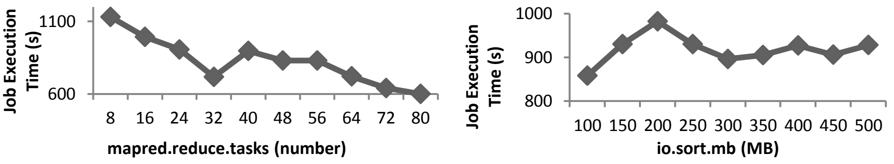
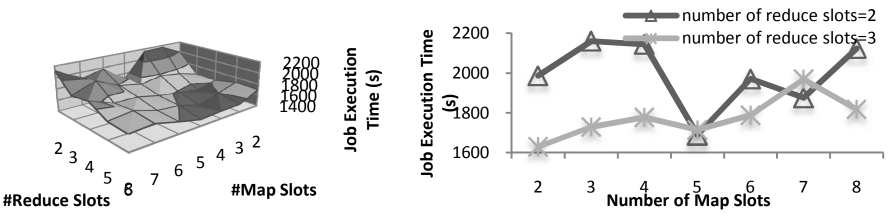


Fig. 1. The performance impact of *mapred.reduce.tasks* and *io.sort.mb*



(a) (b)

Fig. 2. Sort performance for (a) different map and reduce slots; (b) for reduce slots = 2 and 3

2.3 Issues with Manual Tuning

Hadoop parameter optimization requires domain-specific knowledge of the application, the Hadoop framework, and the cluster systems architecture. And *manual tuning* is extremely challenging given explicit and implicit dependencies between different configuration parameters and their effect on the different aspects of the system. Figure 2(a) illustrates a scenario where joint exploration of two parameters, namely the number of map and reduce slots, results in a complicated non-linear surface representing the runtimes of a Sort job. This problem is exacerbated as more parameters are added to the exploration. Figure 2(b) presents the performance as a function of the number of map slots and reduce slots 2 and 3. Clearly, the optimal number of map slots depends on the number of reduce slots. Hence, the parameters need to be examined jointly to locate the global optimal configuration. The manual evaluation of all possible combinations of all performance-related configuration parameters may take months, rendering it impractical. These issues motivate auto-tuning approaches.

3 Approaches to Auto-Tuning

Performance models are used to automatically tune databases and other complex systems [3, 8, 15, 21, 24]. The common approaches involve cost-based or machine learning models. Cost-based models are constructed a priori and calibrated by evaluating the costs of various operations. Machine learning models are used similarly but derived by learning from training sets.

3.1 Cost-Based Models

Cost-based models are built using domain-specific knowledge. In the context of MapReduce, researchers at Duke University recently proposed a Hadoop auto-tuner called Starfish [1, 11, 12, 13]. In Starfish, cost is measured in terms of CPU cycles of different execution phases, such as CPU cost of processing a key-value pair in the map function. To the best of our knowledge, Starfish is the first attempt to address the Hadoop auto-tuning problem. In this subsection, we use it as a case study and reveal its limitations due to model inaccuracy.

Starfish's model uses the average CPU and I/O cost of reads and writes to estimate average map execution time [11]. These costs are measured by profiling a job with a single configuration and the model predicts the same map time as the number of reduce tasks is swept from 64 to 256, as shown in Figure 3, even though the actual time changes considerably.

Additionally, we present task execution information for three configurations in Table 1. The map times are highly variable, with standard deviations of up to ~90%. This complicates task scheduling and could skew the job execution time. The model does not consider these effects, thus introducing errors. The above problems occur because the contention for hardware resources between map and reduce tasks changes with configurations and this contention largely affects task time. While a more complex cost-based model (e.g., one that sensitizes costs to these effects) may address these issues, it is challenging to build such a model.

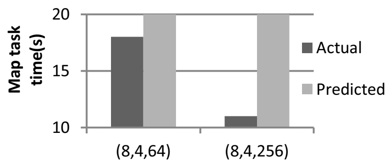


Fig. 3. Map time for sort with 8 map, 4 reduce slots, and 64 or 256 reduce tasks

Table 1. Map task times for three configurations

(map slots, reduce slots, reduce tasks)	Job time(s)	Average map time(s)	Standard deviation of map time(s)
(8, 4, 64)	1847	18	16
(8, 4, 256)	1600	11	8
(6, 6, 384)	1417	9.8	6

The other limitation of cost-based modeling is its inability to adapt as frameworks evolve. Hadoop MapReduce is a relatively new framework and is evolving at a very fast pace. For instance, the next generation framework called YARN [27] has a significantly different architecture. Hence existing cost-based models may not work and need to be redesigned. The same issue arises as clusters evolve with new processors, memory, and storage technologies.

3.2 Machine Learning Models

Machine Learning (ML) models have been proposed in many fields to estimate the performance of complex systems [8, 15, 24]. However, they are impractical for MapReduce auto-tuning as they require large training sets in order to build an accurate model. Our exploration of this method using models like artificial neural network, support vector regression, multiple linear regression and M5 decision tree revealed that more than 200 evaluations are needed to obtain an accuracy of ~90% with five configuration parameters. Since most MapReduce applications involve batch processing with long execution times (tens of minutes to hours), collection of training sets is slow. Although we can use logs as training sets, they typically capture a small number of configurations. This leads to data under-fitting.

4 Gunther: A Search-Based Auto-Tuner

To overcome the inadequacies of cost-based and ML models, we propose a search-based auto-tuner, called Gunther, to optimize configuration settings in Hadoop MapReduce. We perceive auto-tuning as a black-box optimization problem and use search algorithms to solve it. The objective of the search is to evaluate candidate solutions with the stimulus of different parameters to minimize an objective function. The minimization problem is expressed below:

$$f(X^*) = \arg\min_{X_i \in range(i)} f(X_1, X_2, ..., X_D), 1 \le i \le D \quad (1)$$

where $f: X \rightarrow Y$ is the response function, X_i denotes the i^{th} parameter, D is the number of parameter dimensions, $range(i)$ is range of the i^{th} parameter X_i, and X^* is the optimal configuration for which $f(X)$ attains minimum value. In our case, the Hadoop MapReduce configuration parameters define the parameter space and each functional evaluation involves measuring the execution time of a Hadoop job. This approach offers three benefits. First, we overcome modeling inaccuracy since we use job execution times during search. Second, our approach can be used to optimize future MapReduce frameworks. Hence it is more adaptive. Third, our approach does not require the large training set, hence, is more practical.

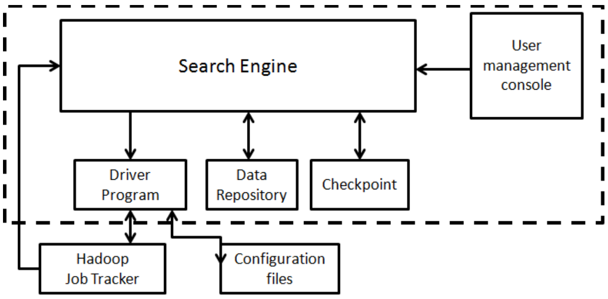


Fig. 4. Gunther Architecture

4.1 Gunther Overview

Gunther's architecture is illustrated in Figure 4. The search algorithm is implemented in the search engine (SE). SE generates a new configuration and asks the driver program to run the application through the JobTracker with the new configuration. After the run is complete, SE writes log files to the repository and analyzes them to obtain execution times. The search terminates after the algorithm meets the convergence criteria or reaches a specified number of trials, and we have designed management console to facilitate progress monitoring.

4.2 Evaluation of Search Algorithm

The effectiveness and efficiency of the search algorithm is critical. To select the right search algorithm, we evaluated both local and global search algorithms on Hadoop performance surfaces using three parameters: map slots, reduce slots and number of reduce tasks. For these, we exhaustively evaluated job execution time. Figure 2 illustrates the response surface of Hadoop sort for two dimensions. We do not show data for other applications but they exhibit similar behavior. From experiments, we observe that Hadoop surfaces are non-linear and multimodal, with many local minima. Local search techniques widely used for unimodal surfaces, such as *Nelder Mead* and *Powell* Search, are inadequate for multimodal surfaces because they easily get stuck at local minima. We found that *Nelder Mead* and *Powell* Search easily settle in local minima on our response surfaces, which are 15% worse than the optimal solution in most cases. This indicates that global search techniques may be more effective.

Global search algorithms are classified into gradient-based search, stochastic search and evolutionary search. Due to the lack of gradient information, we rule out the gradient based techniques. We identify four stochastic and evolutionary search methods for evaluation: simulated annealing (SA) [20], genetic algorithm (GA) [23], particle swarm optimization (PSO) [19] and recursive random search (RRS) [28]. We used Rastrigin function [28] to perform evaluation. This function generates a multimodal surface and is challenging due to its large search space and number of local minima. We searched the surface using SA, RRS, GA, and PSO on 10-dimensional surfaces. We observed that SA exhibits the worst performance and GA is the best, followed by PSO. Due to page limits, we do not present detailed results here.

4.3 Applying Genetic Algorithms to the Auto-Tuning of MapReduce

The studies in the previous section motivated us to apply GA to our problem. Algorithm 1 describes a typical GA cycle. The algorithm encodes the potential solutions to a problem as candidates. The solutions are evaluated with a fitness (or objective) function that is tailored to the problem. Candidates with higher fitness scores are deemed better solutions.

Algorithm 1. Traditional GA

```
Input: $P_0$ : A randomly selected initial population of size $M$
Output: $C_{Best}$
1.  $P = P_0$
2.  For all $C_i$ in $P$ do evaluate $\boldsymbol{fitness}(C_i)$
3.  $C_{Best} = maximum(\boldsymbol{fitness}(C_i))$
4.  For $N$ generations (or while search converges) do
5.      For $i = 1,2,\dots,\frac{M}{2}$ do
6.          $\boldsymbol{Select}$ parents from $P$
7.          With probability $p_c$ crossover $parent_i$ and $parent_{i+1}$ to create candidate $child_i$ and
            $child_{i+1}$
8.          With probability $p_m$ mutate $child_i$ and $child_{i+1}$
9.          Evaluate $\boldsymbol{fitness}(child_i)$ and $\boldsymbol{fitness}(child_{i+1})$
10.         Update $P$
11.         Recalculate $C_{Best}$
```

GA begins with an initial population of randomly generated candidates. It evolves the population during each generation by using the genetic operators select, crossover, mutate, and update. A popular selection operator is the *Roulette Wheel* mechanism. In this method, if $\mathrm{fitness}(C_i)$is the fitness of candidate C_i in the population, its probability of being selected is $P_i = \frac{\mathrm{fitness}(C_i)}{\sum_{i=1}^{M}\mathrm{fitness}(C_i)}$, where M is population size. This allows candidates with good fitness values to have a higher probability of being selected as parents. A crossover function is called with a given probability. It is used to cut the sequence of elements from two chosen parents/candidates and swap them to produce two children/candidates. The mutation function aims to avoid local optima by randomly mutating an element with a given probability. Both crossover and mutation probabilities are input parameters of GA. At the end of each generation, a new population replaces the current population.

In our application, each element g_i represents a Hadoop parameter. A candidate C_i consisting of all parameters, denoted as $C_i = g_1 g_2 .. g_D$, represents a Hadoop job configuration, where D is the number of parameters. The fitness of a candidate is calculated using $\text{fitness}(C_i) = \frac{1}{\text{Job completion time}_i}$. We choose a population of size 2D. GA is a generic search strategy and its operators need to be implemented in an application-specific manner. We define the operators for our problem as follows.

Select Operator

```
Input:    P
Output:   $\text{parent}_i$ and $\text{parent}_{i+1}$
1.   List L = Sort P in decreasing order of $\text{fitness}(C_i)$
2.   $\text{Avg}_P = \frac{1}{M}\sum_{j=1}^{M} \text{fitness}(C_j)$
3.   if $\text{fitness}(L_k) > \text{Avg}_P$ then $\text{parent}_i = L_k$
4.   if $\text{fitness}(L_{k+1}) > \text{Avg}_P$ then $\text{parent}_{i+1} = L_{k+1}$
```

Select Operator: From experimentation we observe that it is unlikely that two low fitness candidates will produce an offspring with high fitness. This is because, in real clusters, bad performance is often caused by the improper configuration of a few key parameters and these bad settings continue to be inherited. For instance, for an application that is both CPU and shuffle-intensive in a cluster with excessive I/O bandwidth and limited CPU resources, enabling compression of map outputs would stress the CPU and degrade application performance, regardless of others. The selection method should eliminate this configuration quickly. We also observe that good candidates are more likely to produce good offspring/candidates.

The popular Roulette-Wheel selection mechanism has a higher probability of selecting good candidates to be parents than bad ones, but this approach still results in too many job evaluations. Therefore, our selection procedure is more aggressive and deterministically selects good candidates to be parents. The idea is to quickly eliminate poor candidates from the population. To do this, we calculate the mean fitness of the population for each generation and only select parents with fitness scores that exceed the mean.

Update Operator

```
Input: $\text{child}_i$, $\text{child}_{i+1}$, L
Output: L
1. $k = \text{sizeof}(L)$
2. If $\text{fitness}(\text{child}_i) > L_k$ then $L_k = \text{child}_i$
3. If $\text{fitness}(\text{child}_{i+1}) > L_{k-1}$ then $L_{k-1} = \text{child}_{i+1}$
```

Update Operator: In GA, the update operator directly replaces parents with their offspring, even if their offspring have lower fitness values. Thus the algorithm does not always retain better solutions, which slows convergence. We modify the update procedure so that child_i and child_{i+1} only replace poorer solutions, i.e., parents whose fitness values are lower than the created candidates in the population. If the offspring are less fit, they are discarded and the parents survive. This directs the search more efficiently toward better solutions.

Mutate: The mutated value of a parameter is randomly chosen from its range. Since our select aggressively prunes poor regions, we can use an atypically high mutation rate (e.g., p_m=0.1) without impacting convergence. The value of p_m is empirically determined.

Crossover: We use a one-point crossover. A cut point is randomly chosen in each parent's candidate/job configuration and all parameters beyond that point are swapped between the two parents to produce two children. We empirically set crossover probability p_c to be 0.7.

Non-redundancy: Classical GA does not remember prior search and is likely to evaluate some regions more than once. We enhanced our GA to remember search to avoid duplication.

5 Performance Evaluation

5.1 Experimental Setup

We deployed Gunther with Hadoop 0.20.3 on Cluster1 and Cluster2. Each cluster has one master node and 16 slaves. Each node is configured with 16GB memory and one Quad Core Intel Xeon processor E3 with HT enabled. The nodes are interconnected through a 1GbE switch. Cluster1 was designed to be storage bottlenecked and used 3 1TB HDDs for HDFS and intermediate data and a separate 1TB HDD for OS, and Cluster2 was designed to be network bottlenecked, with 3 240GB Intel SSD (520 Series) [14] for HDFS and intermediate data. We selected Sort, Nutch, Kmeans and Terasort from the HiBench suite as our applications. We use rule-of-thumb (RoT) configurations as our baseline. We used Starfish 3.0 in the comparison.

Table 2. Hadoop parameters considered

Parameter Name	Range	Rule-of-Thumb	Description
mapred.tasktracker.map.tasks.maximum	2:12::1	8	Maximum number of map tasks for a node
mapred.tasktracker.reduce.tasks.maximum	2:12::1	4	Maximum number of reduce slots for a node
mapred.reduce.tasks	4N:16N:4N	4N	# reduce tasks in a job. N is the number of nodes
io.sort.mb	100:500::50	100	Size (MB) of buffer to use while sorting map output
mapred.output.compress	True/False	False	Compress the output of the job
mapred.compress.map.output	True/False	False	Compress the output of each map task

Table 2 shows the ranges and recommended values of the six parameters we tuned. Our motivation for exploring these parameters is two-fold. First, these parameters affect the utilization of different resources, such as CPU, memory, storage, and network. By tuning them, we believe we can achieve better balance among these resources and improve performance. Second, these parameters impact both task- and cluster-level performance. *io.sort.mb* affects task-level performance and the rest of parameters change the distributed system's data flow and have cluster-level performance effects. The second column in the table describes the parameter bounds and step sizes explored (e.g., map slots vary between 2 and 12 in steps of 1).

5.2 Search Effectiveness and Efficiency

In Figure 5a, we compare the best execution time found by Gunther and Starfish with the RoT and best performance for Cluster1. We randomly sampled the space by running a large number of experiments offline for each application. The best performance found is considered best. The figure demonstrates that Gunther achieves near-optimal performance and is more effective than Starfish. Compared to RoT, Gunther yields a 25% performance improvement on average across all workloads. The maximum improvement of 30% is achieved for Terasort. Correspondingly, Starfish achieves an 11% improvement on average, with a maximum improvement of 29%. Figure 5b presents results from Cluster2. Starfish shows no improvement compared to RoT. This is because Starfish assumes sufficient network bandwidth is available, which leads to inaccurate estimation of shuffle time in Cluster2. However, Gunther is able to capture the network bottleneck. As a result, Gunther improves performance by up to 33%, which is close to best. Note that in this cluster RoT is indeed the best configuration for Terasort and there is no opportunity to improve performance. Table 3 enumerates the number of trials it took Gunther to converge. It was able to converge within 30 trials in all cases.

Fig. 5. a) Comparison on Cluster1 b) Comparison on Cluster2

Table 3. Number of trials to reach the best performance

	Sort	Terasort	Nut	Kmeans
Cluster1	20	15	14	12
Cluster2	24	10	21	10

5.3 Comparison with Other Algorithms

Figure 6 shows comparison of Gunther's modified GA with PSO and RRS on Cluster1 and Cluster2 for all four applications. The figure shows the best execution times achieved for budgets of 20, 30, and 40 trials, normalized to RoT (lower is better). Overall, Gunther achieves higher performance than PSO and RRS and also converges in ~ 20 trials, whereas PSO and RRS often take more than 40 trials. Our results demonstrate that Gunther is more effective and efficient than PSO and RRS.

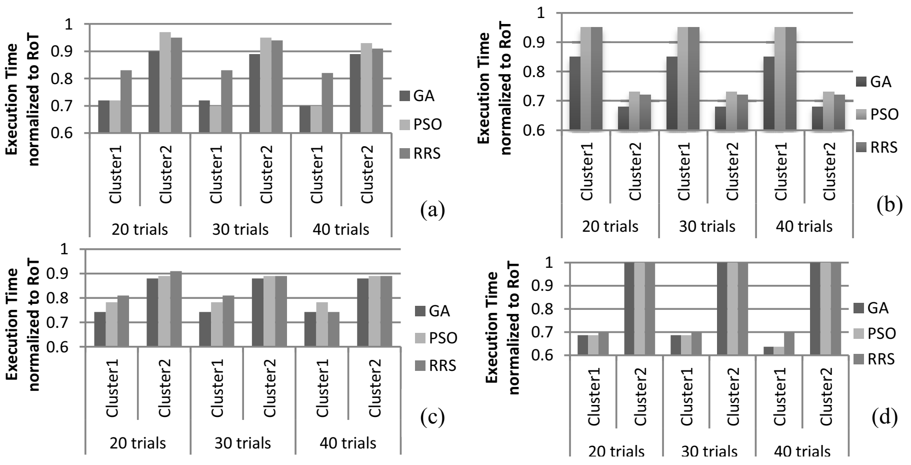


Fig. 6. Search algorithms on clusters for (a) Sort, (b) Nutch, (c) Kmeans, and (d) Terasort benchmarks

6 Using Classification to Improve Search Efficiency

The efficiency of search can be further improved with a minor impact on effectiveness by selectively reducing the dimensionality of the search space. This is possible because some parameters affect performance more than others and the impact depends on what resources are bottlenecked. By profiling an application once with RoT settings, we can rule out parameters that affect resources that are not bottlenecked.

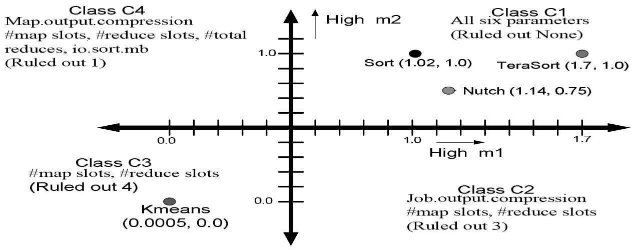


Fig. 7. Classification of applications based on *m1* and *m2*

In this section, we present an application classification methodology based on two metrics that are calculated from five Hadoop job counters. The metric *m1* is defined as the ratio between the count of spilled records and the sum of counts of map input records and reduce input records. Spill records are the number of key-value pairs written from memory to storage during the shuffling phase. This metric represents shuffle intensity. We can rule out *io.sort.mb*, *mapred.compress.map.output*, *mapred.reduce.tasks* when *m1* values are low because they primarily impact shuffle performance. The metric *m2* is defined as the ratio of the count of *hdfs_bytes_written* to *hdfs_bytes_read*. These count the amount of data read and written by map and reduce tasks, respectively. Low *m2* indicates the compression controlled via *mapred.output.compress* will have small performance effects and can be ruled out.

We classify applications into four classes (C1, C2, C3, and C4) and results are shown in Figure 7. The threshold of 0.5 is derived empirically and in the future we intend to automatically determine its value using a clustering algorithm. Sort, Terasort and Nutch are placed into Class C1 with high *m1* and *m2*, indicating that we cannot rule out any parameters. On the other hand, Kmeans is placed into Class C3 with low *m1* and *m2* because it is compute-intensive, with low I/O utilization. This means we can rule out 4 parameters prior to search. Experimental results in Table 4 are supportive. Limiting the Kmeans (Class C3) search to 2 dimensions had a negligible effect on search effectiveness but cut the search time in half. But, when we ruled out the same parameters for Class C1 applications, both the effectiveness and efficiency were impacted. For example, Sort on Cluster1 improved only 3.9% when 4 parameters were rule out compared to 29.5% when no parameters were ruled out. The improvement on Cluster2 was also impacted. We observe similar patterns for Nutch and Terasort.

Interestingly, applications in C1 are more strongly affected by dimensionality reduction on Cluster1 than Cluster2 even though the metrics *m1* and *m2* are the same. This is because our metrics do not reflect differences in dynamic cluster runtime information (e.g., resource utilization). We believe that developing new metrics considering dynamic information will improve the selectivity of classification. Moreover, our classification only rules out parameters and cannot rule them in. A more powerful method would do both. Finally, the current metrics were selected based on domain expertise. In future, we plan to use machine learning-based classifiers (e.g. principal component analysis, etc.) to automate the selection process.

Table 4. Results of dimensionality reduction

Cluster	Workload	2 Dimensions		6 Dimensions	
		Trials	Improvement (%)	Trials	Improvement (%)
Cluster1	Sort	14	3.90	20	29.48
	Kmeans	**7**	**24.77**	**12**	**25**
	TeraSort	14	0	15	30
	Nutch	4	6.77	14	15
Cluster2	Sort	10	7	24	11
	Kmeans	**5**	**11.5**	**12**	**12**
	TeraSort	5	0	10	0
	Nutch	14	25	21	33

7 Other Related Work

7.1 Hadoop Optimization

Hadoop tuning has been studied [1, 4, 11, 12, 13, 22]. The conventional practice is to rely on rules-of-thumb to find good configurations for applications [4]. In contrast to rule-based approaches, Starfish [11, 12, 13] leverages a cost-based model to tune Hadoop applications. Some studies look beyond Hadoop tuning to library extensions and runtime improvements. Manimal [16] performs static analysis of Hadoop programs and deploys optimizations to avoid reads of unneeded data. Panacea [22] proposes a compiler that performs transformations for Hadoop applications to reduce overheads of iterative applications. Twister [7] proposes a new in-memory library to improve the performance of iterative MapReduce applications.

7.2 Auto-Tuning Other Systems

Cost-based, ML and search-based models are used to auto-tune complex systems [3, 6, 8, 15, 21, 24, 28, 29]. Most of the work in database systems uses cost-based models to find the optimal configuration. For instance, IBM DB2 [21] provides an advisor for setting default values for a large number of parameters, which relies on built-in cost models. Similarly, machine learning models are used for auto-tuning many systems. Ganapathi et al. [8] proposed KCCA to derive the relationship between configurations and performance. The search algorithms evaluated in this paper have been used to identify near-optimal configurations for other complex systems. For example, Ye et al. [28] used RRS to tackle network configuration. In addition, Zheng et al. [29] constructed a parameter dependency graph and applied a simplex search method to find good configurations for web services. Duan et al. [6] tuned database parameters by developing adaptive sampling based on a Gaussian process. Zhu et al. [30] used an online learning algorithm to adjust the parameters of applications and optimize performance.

8 Conclusion and Future Work

In this paper, we assessed model-based approaches for Hadoop MapReduce optimization and identified major limitations. Our findings motivated us to propose and implement Gunther, a search-based auto-tuner. We studied several global search algorithms and selected GA as our search strategy. We modified GA to strike the right balance between search efficiency and effectiveness and evaluated the resulting search algorithm on two clusters. Experimental results demonstrate that Gunther achieves near-optimal performance in a small number of trials (<30) and yields better performance improvement than rule-of-thumb settings and a cost-based auto-tuner. We also proposed an application classification method that further improves the search efficiency of Gunther by ruling out less important parameters. Our preliminary results are very encouraging, demonstrating that the number of trials can be reduced by half without sacrificing performance. In the future, we intend to extend our classification method to include both static application characteristics and cluster runtime information.

References

1. Babu, S.: Towards Automatic Optimization of MapReduce Programs. In: SOCC, pp. 137–142 (2010)
2. Beck, A.: A Fast Iterative Shrinkage-Threshold Algorithm for Linear Inverse Problems. In: SIAM (2009)
3. Chaudhuri, S., Narasayya, V.: Self-tuning database systems: a decade of progress. In: VLDB 2007 (2007)
4. Cloudera: 7 tips for Improving MapReduce Performance
5. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI (2004)
6. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with iTuned. In: VLDB 2009 (2009)
7. Ekanayake, J., et al.: Twister: a runtime for iterative mapreduce. In: HPDC (2010)
8. Ganapathi, A., et al.: A case for machine learning to optimize multicore performance. In: HotPar (2009)
9. Hadoop mapreduce, `http://hadoop.apache.org`
10. HiBench, `https://github.com/hibench/HiBench-2`
11. Herodotou, H.: Hadoop Performance Models. Technical report, Duke Univ. (2010)
12. Herodotou, H., et al.: What-if Analysis, and Cost-based Optimization of MapReduce Programs. In: PVLDB (2011)
13. Herodoto, H., et al.: Starfish: A Self-tuning System for Big Data Analytics. In: CIDR (2011)
14. Intel SSD, `http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-ssd.html`
15. Ipek, E., de Supinski, B.R., Schulz, M., McKee, S.A.: An approach to performance prediction for parallel applications. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 196–205. Springer, Heidelberg (2005)
16. Jahani, E., et al.: Automatic Optimization of MapReduce Programs. In: PVLDB (2011)
17. Jiang, D., et al.: The Performance of MapReduce: An In-depth Study. In: PVLDB (2010)
18. Kambatla, K., et al.: Towards optimizing hadoop provisioning in the cloud. In: HotCloud (2009)
19. Kennedy, J., et al.: Particle Swarm Optimization. IEEE ICNN (1995)
20. Kirkpatrick, S., Gelatt, D.C., Vechhi, M.P.: Optimization by simulated annealing. Science (1983)
21. Kwan, S., et al.: Automatic Configuration of IBM DB2 Universal Database. IBM TR (2002)
22. Liu, J., et al.: Panacea: Towards Holistic Optimization of MapReduce Applications. In: CGO 2012 (2012)
23. Mitchell, M.: An Introduction to Genetic Algorithms. The MIT Press (1996)
24. Singer, J., et al.: Garbage collection auto-tuning for java mapreduce on multi-cores. In: ISMM (2011)
25. `http://Vaidya.hadoop.apache.org/mapreduce/docs/r0.21.0/vaidya.html`
26. White, T.: Hadoop: The Definitive Guide. Yahoo Press (2010)
27. YARN, `http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html`
28. Ye, T., Kalyanaraman, S.: A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration. In: SIGMETRICS, pp. 196–205 (2003)
29. Zheng, W., Bianchini, R., Nguyen, T.D.: Automatic Configuration of Internet Services. In: Eurosys 2007 (2007)
30. Zhu, Q., et al.: Automatic tuning of interactive perception applications. UAI (2010)
31. Gridmix3 - Emulating Production Workload for Apache Hadoop: `http://developer.yahoo.com/blogs/hadoop/gridmix3-emulating-production-workload-apache-hadoop-450.html`

Multi-criteria Checkpointing Strategies: Response-Time versus Resource Utilization

Aurelien Bouteiller[1], Franck Cappello[2], Jack Dongarra[1], Amina Guermouche[3], Thomas Hérault[1], and Yves Robert[1,4]

[1] University of Tennessee Knoxville, USA
[2] University of Illinois at Urbana Champaign, USA & INRIA, France
[3] Univ. Versailles St Quentin, France
[4] Ecole Normale Supérieure de Lyon, France
{bouteill,dongarra,herault,yrobert1}@eecs.utk.edu,
cappello@illinois.edu, amina.guermouche@uvsq.fr

Abstract Failures are increasingly threatening the efficiency of HPC systems, and current projections of Exascale platforms indicate that rollback recovery, the most convenient method for providing fault tolerance to general-purpose applications, reaches its own limits at such scales. One of the reasons explaining this unnerving situation comes from the focus that has been given to per-application completion time, rather than to platform efficiency. In this paper, we discuss the case of uncoordinated rollback recovery where the idle time spent waiting recovering processors is used to progress a different, independent application from the system batch queue. We then propose an extended model of uncoordinated checkpointing that can discriminate between idle time and wasted computation. We instantiate this model in a simulator to demonstrate that, with this strategy, uncoordinated checkpointing per application completion time is unchanged, while it delivers near-perfect platform efficiency.

1 Introduction

The progress of many fields of research, in chemistry, biology, medicine, aerospace and general engineering, is heavily dependent on the availability of ever increasing computational capabilities. The High Performance Computing (HPC) community strives to fulfill these expectations, and for several decades, has embraced parallel systems to increase computational capabilities. Although there is no alternative technology in sight, the core logic of delivering more performance through ever larger systems bears its own issues, and most notably declining reliability. In the projections issued by the International Exascale Software Project (IESP) [1], even if individual components are expected to enjoy significant improvements in reliability, their number alone will drive the system Mean Time Between Failures (MTBF) to plummet, entering a regime where failures are not uncommon events, but a normal part of applications execution [2].

Coordinated rollback recovery, based on periodic, complete application checkpoint, and complete restart upon failure, has been the most successful and usually deployed failure mitigation strategy. Unfortunately, it appears that coordinated checkpoint/restart will suffer from unacceptable I/O overhead at the scale envisioned for future systems, leading to poor overall efficiency barely competing with replication [3]. In recent years, an alternative automatic rollback recovery technique, namely uncoordinated checkpointing with message logging [4], has received a lot of attention [5,6]. The key idea of this approach is to avoid the rollback of processes that have not been struck by failures, thereby reducing the amount of lost computation that has to be re-executed, and possibly permitting overlap between recovery and regular application progress. Unfortunately, under the reasonable hypothesis of tightly coupled applications (the most common type, whose complexity often compels automatic fault tolerance), processes that do not undergo rollback have to wait for restarted processes to catch up before they can resume their own progression, thereby spending as much time idling than they would have spent re-executing work in a coordinated approach.

In this paper, we propose to consider the realistic case of an HPC system with a queue of independent parallel jobs (from a single workflow, or even submitted by different users). Instead of solely focusing on per-application completion time, which is strongly challenged by numerous failures, the goal of such a system is to complete as many useful computations as possible (while still retaining reasonable per-application completion time). The proposed application deployment scheme addressed in this paper makes use of automatic, uncoordinated checkpoint/restart. It overlaps idling time suffered by recovering applications, by progress made on another application. This second application is loaded on available resources, meanwhile uncoordinated rollback recovery is taking place on the limited subset of the resources that needs to re-execute work after a failure. Based on this strategy, we extend the model proposed in [7] to make a distinction between wasted computation and processor idle time. The waste incurred by the individual application, and the total waste of the platform, are both expressed with the model, and we investigate the trade-offs between optimizing for application efficiency or for platform efficiency.

The rest of this paper is organized as follows: Section 2 gives an informal statement of the problem. The strategy that targets platform efficiency is described in Section 3. Section 4 presents the model and the scenarios used to analyze the behavior of the application-centric and platform-centric scenarios. Section 5 is devoted to a comprehensive set of simulations for relevant platform and application case studies. Section 6 provides an overview of related work. Finally we give some concluding remarks and hints for future work in Section 7.

2 Background and Problem Statement

Many approaches have been proposed to resolve the formidable threat that process failures pose to both productivity and efficiency of HPC applications. Algorithm Based Fault Tolerance [8], or Naturally Fault Tolerant Methods [9] promise to deliver exceptional performance despite failures, by tailoring recovery actions for each particular application. However, the use of intrinsic algorithmic properties is an application-specific solution that often entails excruciating software engineering efforts, which makes it difficult to apply to production codes.

In a sharp contrast, checkpoint/restart rollback recovery strategies can be provided automatically, without modifications to the supported application. Although the classical coordinated checkpoint approach seems to be reaching its limits [3], recent optimizations and experimental studies outline that compelling performance can be obtained from uncoordinated checkpointing [5,6]. Rollback recovery protocols employ checkpoints to periodically save the state of a parallel application, so that when a failure strikes some process, the application can be restored into one of its former states. In a parallel application, the recovery line is the state of the entire application after some processes have been restarted from a checkpoint. Unfortunately, not all recovery lines are consistent (*i.e.* result in a correct execution); in particular, recovery lines that separate the emission and matching reception event of a message are problematic. The two main families of rollback recovery protocols differ mostly in the way they handle these messages crossing the recovery line [4]. In the coordinated checkpoint approach, a collection of checkpoints is constructed so that consistency threatening messages do not exist between checkpoints of the collection (using a coordination algorithm). Since the checkpoint collection forms the only recovery line that is guaranteed to be correct, all processes have to roll back simultaneously, even if they are not faulty. As a result, the bulk amount of lost work is increased, and the strategy is not optimal for a given number of failures. The non-coordinated checkpoint approach avoids duplicating the work completed by non-faulty processes. Checkpoints are taken independently, and only failed processes endure rollback. Obviously, the resulting recovery line is not guaranteed to be correct without the addition of supplementary state elements to resolve the issues posed by crossing messages. Typically, message logging and event logging [4] store the necessary state elements during the execution of the application. When a process has to roll back to a checkpoint, it undergoes a managed, isolated re-execution, where all non-deterministic event outcomes are forced according to the event log, and messages from the past are served from the message log without rollback of original senders.

Problem Statement: For typical HPC applications, which are often tightly coupled, the ability of restarting only faulty processes (hence limiting duplicate computation to a minimum) does not translate into great improvements of the application completion time. This is illustrated in the instantiations of the model that we recently proposed [7], which captures the intricacies of advanced uncoordinated recovery techniques. Despite being spared the overhead of executing duplicate work, surviving processes quickly reach a synchronization point where further progress depends on input from rollback processes. Since the recovered processes have a significant amount of duplicate work to re-execute before they can catch up with the general progress of the application, surviving processes spend a significant amount of time idling; altogether, the overall application completion time is only marginally improved. Yet, it is clear that, given the availability of several independent jobs, idle time can be used to perform other useful computations, thereby diminishing the wasted time experienced by the platform as a whole.

3 Strategy to Improve Platform Efficiency

In this paper, we propose a scheduling strategy that complements uncoordinated rollback recovery, in order to decrease the waste of computing resources during recovery periods. When a failure occurs (represented as a lightning bolt in Figure 1), a set of spare processes is used to execute the duplicate work of processes that have to roll back to a checkpoint ($R + ReExec$). However, unlike regular uncoordinated checkpoint, instead of remaining active and idling, the remainder of the application is stopped, and flushed from memory to disk. The resulting free resources are used to progress an independent application $App2$. When the recovering processes have completed sufficient duplicate work, the supplementary application can be stopped (and its progress saved with a checkpoint); the initial application can then be reloaded and its execution resumes normally. In the next section, we propose an analytical model for this strategy, that permits to compute the supplementary execution time for the initial application, together with the total waste of computing resources endured by the platform. This model extends on previous work [7], which considered only the impact on application efficiency, and therefore let one of the key advantages of uncoordinated recovery unaccounted for, in the (reasonable) hypothesis of tightly coupled applications. We then use the model to investigate the appropriate checkpoint period, and to predict adequate strategies that deliver low platform waste while preserving application completion time.

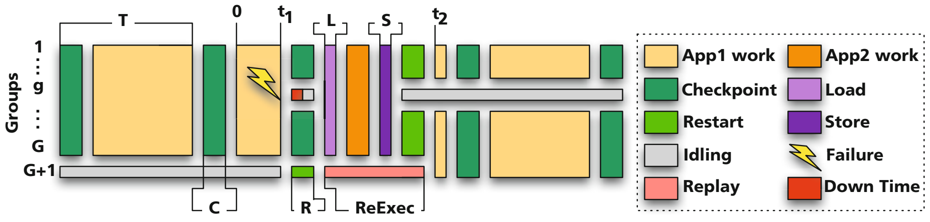


Fig. 1. The S_{Plat} scenario: a deployment strategy that improves the efficiency of resource usage in the presence of failures

4 Model

In this section, we introduce the model used to investigate the performance behavior of the proposed deployment scheme. We detail two execution scenarios: a regular uncoordinated checkpoint deployment that uses the whole platform for a single application; and the aforementioned approach that sacrifices a group of processors as a spare dedicated to recovery of failed processors, but can use the remainder of the platform to progress another application during a recovery. The goal is to compare the *waste* – the fraction of time where resources are not used to perform useful work. The minimum waste will be achieved for some optimal checkpointing period, which will likely differ for each scenario.

Table 1. Key model parameters

μ_p	Platform MTBF
G or $G+1$	Number of groups
T	Length of period
W	Work done every period
C	Checkpoint time
D	Downtime
R	Restart (from checkpoint) time
α	Slow-down execution factor when checkpointing
λ	Slow-down execution factor due to message logging
β	Increase rate of checkpoint size per work unit

Model Parameters and Notations. The model parameters and notations are summarized in Table 1.

– The system employs a hierarchical rollback recovery protocol with message-logging for protection against failures. The platform is therefore partitioned into $G+1$ processor[1] groupsthat can recover independently. In the S_{Plat} scenario, one of these groups is used as a spare, while all $G+1$ participate to the execution in the S_{App} scenario. We let $\text{WASTE}_{app}(T)$ denote the waste induced by the S_{App} scenario with a checkpointing period of T, and T_{app}^{opt} the value of T that minimizes it. Similarly, We define $\text{WASTE}_{plat}(T)$, T_{plat}^{opt} for the S_{Plat} scenario.
– Checkpoints are taken periodically, every T seconds. Hence, every period of length T, we perform some useful work W and take a checkpoint of duration C. Without loss of generality, we express W and T with the same unit: a unit of work executed at full speed takes one second. However, there are two factors that slow-down execution:
  - During checkpointing, which lasts C seconds, we account for a slow-down due to I/O operations, and only αC units of work are executed, where $0 \leq \alpha \leq 1$. The case $\alpha = 0$ corresponds to a fully blocking checkpoint, while $\alpha = 1$ corresponds to a fully overlapped checkpoint, and all intermediate situations can be represented;
  - Throughout the period, we account for a slow-down factor λ due to the communication overhead induced by message logging. A typical value is $\lambda = 0.98$ [5,6];
  - Altogether, the amount of work W that is executed every period of length T is

$$W = \lambda((T - C) + \alpha C) = \lambda(T - (1 - \alpha)C) \tag{1}$$

– We use D for the downtime and R for the time to restart from a checkpoint, after a failure has struck. We assume that $D \leq C$ to avoid clumsy expressions, and because it is always the case in practice. However, one can easily extend the analysis to the case where $D > C$.
– Message logging has both a positive and a negative impact on performance:

[1] Our approach is agnostic of the granularity of the processor, which can be either a single CPU, or a multi-core processor, or any relevant computing entity.

- During the recovery, inter-group messages entail no communication as they are available from the message log, in local memory. This results in a speed-up of the re-execution (up to twice as fast for some applications [10]), which is captured in the model by the ρ factor.
- Every inter-group message that has been logged since the last checkpoint must be included in the current checkpoint. Consequently, the size of the checkpoint increases with the amount of work per unit. To account for this increase, we write the equation

$$C = C_0(1 + \beta W) \tag{2}$$

The parameter C_0 is the time needed to write this application footprint onto stable storage, without message-logging. The parameter β quantifies the increase in the checkpoint time resulting from the increase of the log size per work unit (which is itself strongly tied to the communication to computation ratio of the application).

– Combining Equations (1) and (2), we derive the final value of the checkpoint time

$$C = \frac{C_0(1 + \beta\lambda T)}{1 + C_0\beta\lambda(1 - \alpha)} \tag{3}$$

We point out that the same application is deployed on G groups instead of $G + 1$ in the S_{Plat} scenario. As a consequence, when processor local storage is available, C_0 is increased by $\frac{G+1}{G}$ in S_{Plat}, compared to the S_{App} case.

Computing the Waste. The major objective of this paper is to compare the minimum waste resulting from each scenario. Intuitively, the period T^{opt}_{app} (single application) will be smaller than the period T^{opt}_{plat} (platform-oriented) because the loss due to a failure is higher in the former scenario. In the latter scenario, we lose a constant amount of time (due to switching applications) instead of losing an average of half the checkpointing period in the first scenario. We then aim at comparing the four values $\text{WASTE}_{app}(T^{opt}_{app})$, $\text{WASTE}_{app}(T^{opt}_{plat})$, $\text{WASTE}_{plat}(T^{opt}_{plat})$, and $\text{WASTE}_{plat}(T^{opt}_{app})$, the later two values characterizing the trade-off when using the optimal period of a scenario for the other one.

Let T_{base} be the parallel execution time without any overhead (no checkpoint, failure-free execution). The first source of overhead comes the rollback-and-recovery protocol. Every period of length T, we perform some useful work W (whose value is given by Equation (1)) and take a checkpoint. Checkpointing induces an overhead, even if there is no failure, because not all the time is spent computing: the fraction of *useful* time is $\frac{W}{T} \leq 1$. The failure-free execution time T_{ff} is thus given by the equation $\frac{W}{T}T_{ff} = T_{base}$, which we rewrite as

$$(1 - \text{WASTE}_{ff})T_{ff} = T_{base}, \text{ where } \text{WASTE}_{ff} = \frac{T - W}{T} \tag{4}$$

Here WASTE_{ff} denotes the waste due to checkpointing and message logging in a failure-free environment. Now, we compute the overhead due to failures.

Failures strike every μ_p units of time in average, and for each of them, we lose an amount of time t_{lost}. The final execution time T_{final} is thus given by the equation $(1 - \frac{\mathrm{t}_{lost}}{\mu_p})T_{final} = T_{ff}$ which we rewrite as

$$(1 - \textsc{Waste}_{fail})T_{final} = T_{ff}, \textit{ where } \textsc{Waste}_{fail} = \frac{\mathrm{t}_{lost}}{\mu_p} \tag{5}$$

Here $\textsc{Waste}_{fail}$ denotes the waste due to failures. Combining Equations (4) and (5), we derive that

$$(1 - \textsc{Waste}_{final})T_{final} = T_{base} \tag{6}$$

$$\textsc{Waste}_{final} = \textsc{Waste}_{ff} + \textsc{Waste}_{fail} - \textsc{Waste}_{ff}\textsc{Waste}_{fail} \tag{7}$$

Here $\textsc{Waste}_{final}$ denotes the total waste during the execution, which we aim at minimizing by finding the optimal value of the checkpointing period T. In the following, we compute the values of $\textsc{Waste}_{final}$ for each scenario. The analysis restricts to (at most) a single failure per checkpointing period. Simulation results in Section 5 discuss the impact of this hypothesis (which, to that best of our knowledge, is mandatory for a tractable mathematical analysis).

Scenario S_{App}. We have $\textsc{Waste}_{ff} = \frac{T-W}{T} = \frac{T-\lambda(T-(1-\alpha)C)}{T}$ for both scenarios but recall that we enroll $G+1$ groups in scenario S_{App} and only G groups in in scenario S_{Plat}, so that the value of C is not the same in Equation (3). Next, we compute the value of $\textsc{Waste}_{fail}$ for the S_{App} scenario.

$$\textsc{Waste}_{fail} = \frac{1}{\mu_p}\left[D + R + \frac{T-C}{T} \times \mathrm{ReExec}_1 + \frac{C}{T} \times \mathrm{ReExec}_2\right] \tag{8}$$

where $\mathrm{ReExec}_1 = \frac{1}{\rho}\left(\alpha C + \frac{T-C}{2}\right)$, $\mathrm{ReExec}_2 = \frac{1}{\rho}\left(\alpha C + T - C + \frac{C}{2}\right)$

First, $D+R$ is the duration of the downtime and restart. Then we add the time needed to re-execute the work that had already completed during the period, and that has been lost due to the failure. The time spent re-executing lost work is split into two terms, depending whether the failure strikes when a checkpoint is taking place or not. ReExec_1 is the term when no checkpoint was taking place at the time of the failure; it is therefore weighted by $(T-C)/T$, the probability of the failure striking within such a $T-C$ timeframe. ReExec_1 first includes the re-execution of the work done in parallel with the last checkpoint (of initial duration C), but no checkpoint activity happens during re-execution, so it takes only αC time units. Then we re-execute the work done in the work-only area. On average, the failure happens in the middle of the interval of length $T-C$, hence the time lost has an expected value of $\frac{T-C}{2}$. We finally account for the communication speedup during re-execution by introducing the ρ factor. We derive the value of ReExec_2 with a similar reasoning, and weight it by the probability C/T of the failure striking within a checkpoint interval. After simplification, we derive

$$\textsc{Waste}_{fail} = \frac{1}{\mu_p}\left(D + R + \frac{1}{\rho}\left(\frac{T}{2} + \alpha C\right)\right) \tag{9}$$

Scenario S_{Plat}. In this scenario, the first G groups are computing for the current application and are called *regular* groups. The last group is the *spare* group. As already pointed out, this leads to modifying the value of C, and hence the value of WASTE_{ff}. In addition, we also have to modify the value of T_{base}, which becomes $\frac{G+1}{G}T_{base}$, to account for the fact that it takes more time to produce the same work with fewer processors. We need to recompute WASTE_{final} accordingly so that Equation (6) still holds and we derive:

$$(1 - \text{WASTE}_{final})T_{final} = \frac{G+1}{G}T_{base} \tag{10}$$

$$\text{WASTE}_{final} = \frac{1}{G+1} + \frac{G}{G+1}\left(\text{WASTE}_{ff} + \text{WASTE}_{fail} - \text{WASTE}_{ff}\text{WASTE}_{fail}\right) \tag{11}$$

We now proceed to the computation of WASTE_{fail}, which is intricate. See Figure 1 for an illustration:

– Assume that a fault occurs within group g. Let t_1 be the time elapsed since the completion of the last checkpoint. At that point, the amount of work that is lost and should be re-executed is $W_1 = \alpha C + t_1$. Then:
  1. The faulty group (number g) is down during D seconds;
  2. The spare group (number $G + 1$) takes over for the faulty group and does the recovery from the previous checkpoint at time t_1. It starts re-executing the work until time $t_2 = t_1 + R + \text{ReExec}$, when it has reached the point of execution where the fault took place. Here ReExec denotes the time needed to re-execute the work, and we have $\text{ReExec} = \frac{W_1}{\rho}$;
  3. The remaining $G - 1$ groups checkpoint their current state while the faulty group g takes its downtime (recall that $D \leq C$);
  4. At time $t_1 + C$, the now free G groups load another application from its last checkpoint, which takes L seconds, perform some computations for this second application, and store their state to stable storage, which takes S seconds. The amount of work for the second application is computed so that the store operation completes exactly at time $t_2 - R$. Note that it is possible to perform useful work for the second application only if $t_2 - t_1 = R + \text{ReExec} \geq C + L + S + R$. Note that we did not assume that $L = C$, nor that $S = R$, because the amount of data written and read to stable storage may well vary from one application to another;
  5. At time $t_2 - R$, the G groups excluding the faulty group start the recovery for the first application, and at time t_2 they are ready to resume the execution of this first application together with the spare group: there remains $W - W_1$ units of work to execute to finish up the period. From time t_2 on, the faulty group becomes the spare group.

To simplify notations, let $X = C + L + S + R$ and $Y = X - R$. We rewrite the condition $t_2 - t_1 = R + \text{ReExec} \geq X$ as $\text{ReExec} \geq Y$, i.e., $\frac{\alpha C + t_1}{\rho} \geq Y$. This is equivalent to $t_1 \geq Z$, where $Z = \rho Y - \alpha C$. So if $t_1 \geq Z$, the first G groups lose X seconds, and otherwise they lose $R + \text{ReExec}$ seconds. Since t_1 is uniformly distributed over the period T, the first case happens with probability $\frac{T-Z}{T}$ and the second case with probability $\frac{Z}{T}$. As for the second case, the expectation of t_1 conditioned to $t_1 \leq Z$ is $E[t_1|t_1 \leq Z] = \frac{Z}{2}$, hence the expectation of the time lost is $E[R + \text{ReExec}|t_1 \leq Z] = R + \frac{Y}{2} + \frac{\alpha C}{2\rho}$. Altogether the formula for $\text{W}\textsc{aste}_{fail}$ is:

$$\text{W}\textsc{aste}_{fail} = \frac{1}{\mu_p}\left(\frac{T-Z}{T} \times X + \frac{Z}{T} \times (R + \frac{Y}{2} + \frac{\alpha C}{2\rho})\right) \tag{12}$$

– if the failure strikes during the first Z units of the period, which happens with probability $\frac{Z}{T}$, there is not enough time to load the second application, and the regular groups all waste $E[R + \text{ReExec}|t_1 \leq Z]$ seconds in average
– if the failure strikes during the last $T - Z$ units of the period, which happens with probability $\frac{T-Z}{T}$, then the regular groups all waste X units of time, and they perform some useful computation for the second application in the remaining time that they have before the spare group catches up.

5 Results

We instantiated the proposed waste model with different scenarios. Due to lack of space, we present here only two representative scenarios that illustrate the proposed approach. Parameters are set in accordance to target system specifications, and using experimentally observed values for message logging cost. Details for all parameters, as well as supplementary scenarios, consistent with the examples presented here, are available in the companion technical report [11]. The first scenario shown in Fig. 2, instantiates the model with features of the K-Computer ([12]); the checkpoint growth rate (β) is set according to a matrix-matrix multiply operation. The results present the waste from the platform perspective (green lines) and from the application perspective (red lines). The optimal checkpoint

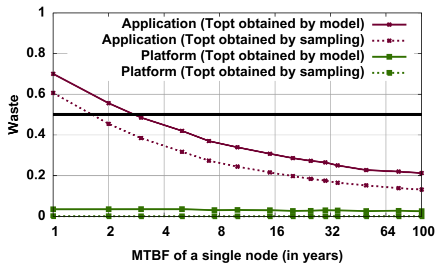


Fig. 2. Waste as function of the compute node MTBF, considering a matrix multiplication, on the K-Computer model

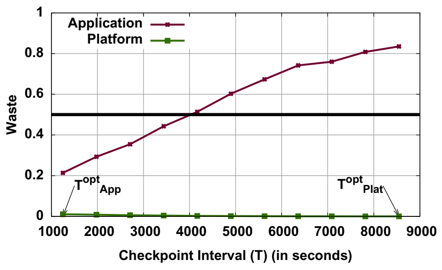


Fig. 3. Waste as function of the checkpoint period, considering a 2D-stencil operation, on the Fat Exascale Computer model (20 years individual processor MTBF)

period is computed by minimizing the model-computed waste. Because it is impossible to account for simultaneous failures in a closed-form formula, we then simulate an execution according to the model, except that simultaneous failures are possible; we verify that the model prediction is accurate nonetheless by comparing it with the best performing period obtained by sampling input periods in the simulator (dashed lines). Comparing the waste obtained when using the model-predicted optimal checkpoint period versus the brute force obtained period in the simulator reveals that the model slightly overestimate the waste, due to optimizing for the case where no simultaneous failures happen. However, general trends are respected, and the difference is under 7%.

When comparing the waste incurred on the application versus the waste of platform resources, this figure demonstrates the huge benefit of introducing a spare node (and loading a second application) on platform efficiency. Indeed, while the application waste, due to I/O congestion at checkpoint time, starts from a relatively high level when the component MTBF is very low (and thus when the machine usability is low), the platform waste itself is almost negligible.

Figure 3 fixes the MTBF of a single component to 20 years, and study the impact of choosing the optimal checkpoint interval so as to target either platform efficiency, or application efficiency. To do so, we varied the checkpoint period between the Application optimal value, and the Platform optimal value, as given by the model. To illustrate the diversity of experiments we conducted, the modeled system is one of the envisioned machines for Exascale systems [1] (the "Fat" version, featuring heavy multicore nodes), and the modeled application is a 2D-stencil application that fills up the system memory. Platform-optimal checkpoint periods are much longer than application-optimal checkpoint periods on the same machine, and both experiments exhibit a waste that increases when using a checkpoint period far away from their optimal. However, because the spare node is so much more beneficial to the general efficiency of the platform than to the efficiency of the application, it is extremely beneficial to select the optimal application checkpoint interval: the performance of the platform remains close to an efficiency of 1, while the waste of the application can be reduced significantly.

6 Related Work

An alternative approach to accelerate uncoordinated rollback recovery, in the Charm++ language, is to split and rebalance the re-execution [13]. For production MPI codes, written in Fortran/C, accounting for the different data and computation distribution during recovery periods entails an in depth rewrite (which may be partially automated by compilation techniques [14]). Even when such splitting is practical, the recovery workload is a small section of the application that is stretched on all resources, which, in accordance with Gustafson law [15], typically results in lower parallel efficiency at scale.

Overlapping downtime of programs blocked on I/O or memory accesses is achieved by a wide range of hardware and software techniques that improve computational throughput (Hyper-threads [16], massive oversubscription in task based systems [17], etc.) Interestingly, co-scheduling [18] can leverage checkpoint-restart to improve communication/computation overlap. However, these have seldom been considered to overcome the cost of rollback recovery itself. Furthermore, checkpoint-restart modeling tools to assess the effectiveness of compensation techniques have not been available yet; the work proposed here supersedes previous models [19,7] in characterizing the difference in terms of platform efficiency when multiple, independent applications must be completed.

7 Conclusion

We have proposed a deployment strategy that permits to overlap the idle time created by recovery periods in uncoordinated rollback recovery with useful work from another application. This opportunity is unique to uncoordinated rollback recovery, since coordinated checkpointing requires the rollback of all processors, hence generates a similar re-execution time, but without idle time. We designed an accurate analytical model that captures the waste resulting from failures and protection actions, both in terms of application runtime and resource usage. The model results are compatible with experimentally observed behavior, and simplifications to express the model as a closed formula introduce only a minimal imprecision, that we have quantified through simulations.

The model has been used to investigate the effective benefit of the uncoordinated checkpointing strategy to improve platform efficiency, even in the most stringent assumptions of tightly coupled applications. Indeed, the efficiency of the platform can be greatly improved, even when using the checkpointing period that is the most amenable to minimizing application runtime. Finally, although replication (with a top efficiency of 50%) sometime delivers better per-application efficiency, we point out that a hierarchical checkpointing technique with dedicated spare nodes, as the one proposed in this paper, is the only approach that can provide a global platform waste close to zero.

For future works, we notice that the spare group is left unused outside of recovery period. Since the next activity on this group is a restart, it could be employed to aggressively prefetch checkpoints. It could also execute compute intensive yet accurate failure prediction models, to adapt checkpoint frequency and prefetching according to sensed runtime hardware conditions.

Acknowledgments. Y. Robert is with the Institut Universitaire de France. This work was supported in part by NSF, DOE, and the ANR Rescue project.

References

1. Dongarra, J., Beckman, P., Aerts, P., Cappello, F., Lippert, T., Matsuoka, S., Messina, P., Moore, T., Stevens, R., Trefethen, A., Valero, M.: The international exascale software project: a call to cooperative action by the global high-performance community. IJHPCA 23(4), 309–322 (2009)

2. Gibson, G.: Failure tolerance in petascale computers. Journal of Physics: Conference Series 78, 012022 (2007)
3. Ferreira, K., Stearley, J., Laros, J.H.I., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D.: Evaluating the Viability of Process Replication Reliability for Exascale Systems. In: Proc. of SC 2011. ACM/IEEE (2011)
4. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Survey 34, 375–408 (2002)
5. Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.J.: Correlated set coordination in fault tolerant message logging protocols. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 51–64. Springer, Heidelberg (2011)
6. Guermouche, A., Ropars, T., Snir, M., Cappello, F.: HydEE: Failure containment without event logging for large scale send-deterministic MPI applications. In: Proc. 26th IPDPS, pp. 1216–1227. IEEE (May 2012)
7. Bosilca, G., Bouteiller, A., Brunet, E., Cappello, F., Dongarra, J., Guermouche, A., Herault, T., Robert, Y., Vivien, F., Zaidouni, D.: Unified model for assessing checkpointing protocols at extreme-scale. Research report RR-7950, INRIA (2012)
8. Huang, K., Abraham, J.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers 100(6), 518–528 (1984)
9. Chen, Z., Fagg, G.E., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., Dongarra, J.: Fault tolerant high performance computing by a coding approach. In: Proc. 10th ACM SIGPLAN PPoPP, pp. 213–223. ACM (2005)
10. Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P., Cappello, F.: MPICH-V: a multiprotocol fault tolerant MPI. IJHPCA 20(3), 319–333 (2006)
11. Bouteiller, A., Cappello, F., Dongarra, J., Guermouche, A., Herault, T., Robert, Y.: Multi-criteria checkpointing strategies: Optimizing response-time versus resource utilization. Research report ICL-UT-1301, University of Tennessee (February 2013)
12. Miyazaki, H., Kusano, Y., Okano, H., Nakada, T., Seki, K., Shimizu, T., Shinjo, N., Shoji, F., Uno, A., Kurokawa, M.: K computer: 8.162 petaflops massively parallel scalar supercomputer built with over 548k cores. In: ISSCC, pp. 192–194. IEEE (2012)
13. Chakravorty, S., Kale, L.: A fault tolerance protocol with fast fault recovery. In: Proc. 21st IPDPS, pp. 1–10. IEEE (March 2007)
14. Yang, X., Du, Y., Wang, P., Fu, H., Jia, J.: FTPA: Supporting fault-tolerant parallel computing through parallel recomputing. IEEE Transactions on Parallel and Distributed Systems 20(10), 1471–1486 (2009)
15. Gustafson, J.L.: Reevaluating Amdahl's law. Communications of the ACM 31, 532–533 (1988)
16. Thekkath, R., Eggers, S.J.: The effectiveness of multiple hardware contexts. In: Proc. of the 6th ASPLOS, pp. 328–337. ACM (1994)
17. Huang, C., Zheng, G., Kalé, L., Kumar, S.: Performance evaluation of Adaptive MPI. In: Proc. 11th ACM SIGPLAN PPoPP, pp. 12–21. ACM (2006)
18. Bouteiller, A., Bouziane, H.L., Herault, T., Lemarinier, P., Cappello, F.: Hybrid preemptive scheduling of message passing interface applications on grids. IJHPCA 20(1), 77–90 (2006)
19. Daly, J.T.: A higher order estimate of the optimum checkpoint interval for restart dumps. FGCS 22(3), 303–312 (2004)

Topic 9: Parallel and Distributed Programming (Introduction)

José Cunha, Michael Philippsen, Domenico Talia, and Ana Lucia Varbanescu

Topic Committee

This topic provides a forum for presentation of new results and practical experience in the development of parallel and distributed programs. The development of high-performance, correct, portable, and scalable parallel programs is a hard task, requiring advanced algorithms, realistic modeling, adequate programming abstractions and models, efficient design tools, high performance languages and libraries, and experimental evaluation. Current challenges in this topic are concerned with improved solutions for conciliating the transparency and expressiveness of the programming abstractions and models, with new issues arising in modern applications with increasing problem size and complexity, and in heterogeneous computing infrastructures with varying performance, scalability, failure and dynamic behaviors. This motivates for example, abstractions for handling concurrency, parallelism and distribution, and support for predictable performance, self-adaptation, fault-tolerance, and large-scale deployment.

This year, a diversity of papers was submitted to this topic, proposing relevant and valuable research contributions. As a result of the reviewing process, 4 papers were accepted for publication. Globally, the accepted papers discuss the gaps between high-level programming abstractions and the domain experts and application developers, and present reports of their implementation and evaluation via concrete applications and performance benchmarks.

One of the papers (by Nanz, West, and Silveira) presents a comparative study of four approaches with distinct parallel programming abstractions and communication paradigms, which are then evaluated via a suite of benchmark programs. Original and revised versions by experts are compared with respect to source code size, coding time, execution time, and speedup. Two of the papers discuss structured models of parallelism as ways of easing the parallel programming tasks. One paper (by Legaux, Hu, Loulergue, Matsuzaki, and Tesson) explores algorithmic skeletons in conjunction with a notion of list homomorphism in the context of the bulk synchronous parallelism (BSP) model and reports on its use for parallel algorithm development in two applications, with an implementation of BSP homomorphism in a data-parallel algorithmic skeletons library. The other paper (by Tasci and Demirbas) also explores the BSP model for parallel processing of large-scale graph applications. The paper discusses modifications to the BSP model and how these were supported with improved performance, on top of Giraph, an open-source clone of Pregel, Google's scalable infrastructure for graph-based mining. Another accepted paper (by Sharp and Morgan) discusses a critical concern in transactional memory systems, regarding the occurrence of conflicts that lead to transactions aborting and retrying their execution. The paper describes an implementation for determining the scheduling of aborted transactions, and discusses the effectiveness of the proposal via performance benchmarks.

We would like to thank all the authors who submitted papers to this topic, and the external reviewers, for their contribution to the success of the conference. We also thank the overall coordination and valuable support that was provided by the conference chairs.

Examining the Expert Gap in Parallel Programming

Sebastian Nanz[1], Scott West[1], and Kaue Soares da Silveira[2,⋆]

[1] ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch
[2] Google Inc., Zurich, Switzerland
kaue@google.com

Abstract. Parallel programming is often regarded as one of the hardest programming disciplines. On the one hand, parallel programs are notoriously prone to concurrency errors; and, while trying to avoid such errors, achieving program performance becomes a significant challenge. As a result of the multicore revolution, parallel programming has however ceased to be a task for domain experts only. And for this reason, a large variety of languages and libraries have been proposed that promise to ease this task. This paper presents a study to investigate whether such approaches succeed in closing the gap between domain experts and mainstream developers. Four approaches are studied: Chapel, Cilk, Go, and Threading Building Blocks (TBB). Each approach is used to implement a suite of benchmark programs, which are then reviewed by notable experts in the language. By comparing original and revised versions with respect to source code size, coding time, execution time, and speedup, we gain insights into the importance of expert knowledge when using modern parallel programming approaches.

1 Introduction

The belief that "parallel programming is hard, and best left to experts" has long become a developers' mantra. Indeed, concurrency makes parallel programs prone to errors such as atomicity violations, data races, and deadlocks, which are hard to detect because of their nondeterministic nature. Furthermore, achieving performance is a significant challenge, as scheduling and communication overheads or lock contention may lead to adverse effects, such as parallel slow down.

In spite of these facts, the comfort of leaving parallel programming to domain experts is fading away: the industry-wide shift to multicore processors has made parallelism relevant for mainstream developers. To support their efforts, a variety of advanced programming languages and libraries have been designed, promising an improved development experience over traditional multithreaded programming. The effectiveness of these approaches in practice hinges on their ability to allow developers to easily achieve good results constructing parallel

⋆ The ideas and opinions presented here are not necessarily shared by my employer.

programs. However, how can one quantify "good results", and "easily"? This will clearly depend on what improvements to a parallel program are still left to be made, and how much effort they require to be implemented.

In this paper, we propose to study these effects by examining the expert gap in parallel programming. The expert gap is the distance in expertise between an expert and a competent (though inexperienced in the expert's domain) developer of a parallel program. The gap is quantified by the difference in lines of code used, absolute performance, scalability, and the correction cost (in coding time) to bring the novice code up to the standards of the expert. This is expressed by the following research questions:

Q1: *To what extent do expert comments reduce code size?*
Q2: *To what extent do expert comments reduce execution time?*
Q3: *To what extent do expert comments increase speedup?*
Q4: *What is the overhead of implementing the experts' corrections?*

To address the research questions, we performed a study with four popular parallel programming approaches: Chapel [5], Cilk [2], Go [8], and Threading Building Blocks (TBB) [14]. In the study, we asked notable experts in the respective approaches to review a suite of six parallel benchmark programs [18] implemented by an experienced developer, who had however no previous expertise in the approaches. After implementation of their comments, the experts performed a second review to check that their comments had been addressed appropriately. We recruited high-profile experts, namely either leaders or prominent members of the respective compiler development teams:

– Brad Chamberlain, Principal Engineer at Cray Inc., technical lead on Chapel
– Jim Sukha, Software Engineer at Intel Corp., Cilk Plus development team
– Luuk van Dijk, Software Engineer at Google Inc., Go development team
– Arch D. Robison, Sr. Principal Engineer at Intel Corp., TBB chief architect

This process led to a solution pool of 48 programs, i.e. six problems in four approaches, each before and after expert review. The data also allows the approaches to be compared with each other, which is an extensive study in itself [12].

The remainder of this paper is structured as follows. Section 2 provides background on the four approaches and the benchmark problems used in the study. Section 3 presents the results of the study for each of the four metrics. Section 4 discusses threats to validity. Section 5 presents related work and Section 6 concludes with an outlook on future work.

2 Background

This section provides background on the parallel programming approaches and the benchmark problems used in this study.

2.1 Overview of the Approaches

Table 1 summarizes the characteristics of Chapel, Cilk, Go, and TBB, together with year of appearance, and the corporation currently supporting further development. The four approaches were selected in the following way: to ensure the availability of suitable experts, we required that the approaches are under active development and have gained popularity; amongst the remaining approaches, we preferred those that would add to the variety of programming paradigms, communication paradigms, and/or programming abstractions considered. Well established approaches such as OpenMP and MPI were not considered, as we wanted to focus on cutting-edge approaches.

Chapel [5] describes parallelism in terms of independent computation implemented using threads, but specified through higher-level abstractions. It provides a variety of parallel constructs such as parallel-for (**forall**), **reduce**, and **scan**, leading to very concise parallel code. Its programming model targets both high-performance computers as well as clusters and desktop multicore systems.

Cilk [2] exposes parallelism through high-level primitives that are implemented by the runtime system, which takes care of load balancing using dynamic scheduling through work stealing. The keyword **cilk_spawn** marks the concurrent variant of the function call statement, which starts the (possibly) concurrent execution of a function. The synchronization statement **cilk_sync** waits for the end of the execution of all the functions spawned in the body of the current function; there is an implicit **cilk_sync** statement at the end of all procedures. Lastly, there is an additional **cilk_for** construct. This construct is a limited parallel variant of the normal **for** statement, handling only simple loops.

Go [8] is a general-purpose programming language targeted towards systems programming. Parallelism is expressed using an approach based on Communicating Sequential Processes (CSP) [9]. The statement **go** starts the execution of a function call as an independent concurrent thread of control, or *goroutine*, within the same address space. *Channels* (indicated by the **chan** type) provide a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type; channels can be synchronous or asynchronous. Few parallel constructs are readily available in Go, resulting in more verbose code. For example, to construct a parallel-for loop the work gets dispatched to a channel from one go routine, while a number of goroutines fetch work from this channel and process it.

Threading Building Blocks (TBB) [14] is a parallel programming template library for the C++ language. Parallelism is expressed using Algorithmic Skeletons [6], and the runtime system takes care of scheduling and load balancing using work stealing. Similar to Chapel, a variety of parallel constructs are available, such as `parallel_for`, `parallel_reduce`, and `parallel_scan`.

2.2 Benchmark Problems

We chose the problems suggested in [18] as benchmarks, which comprehend a wide range of parallel programming patterns. Reusing a tried and tested set

Table 1. Main language characteristics

Name	Programming abstraction	Communication paradigm	Programming paradigm	Year	Corporation
Chapel	Partitioned Global Address Space (PGAS)	message passing / shared memory	object-oriented	2006	Cray Inc.
Cilk	Structured Fork-Join	shared memory	imperative / object-oriented	1994	Intel Corp.
Go	Communicating Sequential Processes (CSP)	message passing / shared memory	imperative	2009	Google Inc.
TBB	Algorithmic Skeletons	shared memory	C++ library	2006	Intel Corp.

has the benefit that estimates for the implementation complexity exist and that problem selection bias can be avoided by the experimenter. We chose these particular benchmark problems to keep the amount of time spent with each problem reasonably small (experts could devote only a limited amount of time to the review). The problems have been designed for this purpose [18]; in order to be more representative of large applications, they can also be chained together.

Again to keep the number of implementations manageable, we selected the following six problems from [18]:

– Random matrix generation (randmat)
– Histogram thresholding (thresh)
– Weighted point selection (winnow)
– Outer product (outer)
– Matrix-vector product (product)
– Chaining of problems (chain)

Note that the last problem, chain, corresponds to a chaining together of the inputs and outputs of the other five.

3 Results

This section presents and discusses the data collected in the study, which is also made available in an online repository[1]. Table 2 provides absolute numbers for all versions of the code, before and after expert review. The figures in this section display the relative differences between the expert and non-expert versions.

3.1 Source Code Size

The differences between the non-expert and expert versions with respect to lines of code are given in Figure 1. Note that the suggestions by the experts were implemented again by the non-expert programmer, such that there is no influence of different coding styles on the code size. Addressing research question **Q1**, it

[1] `https://bitbucket.org/nanzs/multicore-languages`

Table 2. Measurements for all metrics, before and after expert comments

	Problem	randmat		thresh		winnow		outer		product		chain	
	Version[1]	nv	ex	nv	ex	nv	ex	nv	ex	nv	ex	nv	ex
Source code size	Chapel	33	32	58	61	72	74	55	58	34	36	145	159
	Cilk	48	40	119	95	146	139	83	72	65	58	320	251
	Go	52	71	141	118	144	191	103	98	89	86	345	330
	TBB	52	53	110	98	142	137	83	81	63	62	302	302
Coding time (min)	Chapel	76	100	121	156	134	155	55	64	43	45	76	137
	Cilk	101	154	251	294	112	121	26	39	12	15	77	118
	Go	45	76	132	163	92	163	24	31	18	21	56	91
	TBB	35	37	196	207	41	43	32	43	23	23	24	26
Execution time (sec)[2]	Chapel	18.7	3.1	7.8	13.1	21.4	21.3	1.6	1.6	1.4	1.4	36.0	36.0
	Cilk	0.5	0.4	0.9	0.8	0.8	0.7	0.3	0.2	0.3	0.2	2.4	1.7
	Go	2.9	0.5	2.1	1.6	2.0	1.3	1.5	2.4	1.1	0.3	177.7	38.4
	TBB	0.3	0.2	1.2	0.6	1.0	1.0	0.3	0.3	0.2	0.2	2.8	2.8
Speedup[2]	Chapel	1.2	2.8	2.8	2.8	2.3	2.1	3.4	3.5	1.7	1.7	2.0	2.1
	Cilk	13.6	16.8	13.4	14.9	19.1	20.2	8.1	8.1	4.2	5.8	17.3	20.2
	Go	4.1	21.2	8.9	8.1	8.0	11.5	10.4	4.7	1.9	7.5	0.6	1.9
	TBB	20.7	21.2	8.1	14.8	9.4	9.5	7.4	7.4	7.2	7.3	12.5	12.6

[1] nv: novice; ex: expert [2] average times and speedups are given

is apparent from the figure that suggested changes decreased the source code size only moderately, and increased it in several cases. On average the code size decreased, over all languages, by only 1.6% (*SD* (standard deviation) = 13.9%).

There are differences between the individual languages though. The increases in size for Chapel, on average by about 4.3% (*SD* = 4.2%), can be explained mainly by one requested change from the expert: domain definitions were consistently hoisted outside of parallel regions, which saves them being recomputed.

Cilk solutions decreased in size on average by 14.5% (*SD* = 6.2%). This change can be traced back to one of the expert comments to replace **cilk_spawn** /**cilk_sync** style code with **cilk_for**; according to the expert, **cilk_for** simplifies the code while doing the same recursive divide-and-conquer underneath, and should therefore be preferred.

Increases in code size on average by 6.7% (*SD* = 22.1%) are visible in Go. The outliers are randmat and winnow with increases of 36.5% and 32.6% on average. For randmat, this can be explained by a suggested change of data structure; since the randmat program is small to begin with, this relatively small change amounts to a seemingly large increase percentage-wise. For winnow, the increase in performance results from the suggestion of the expert to add parallel merge sort, which is not part of Go's standard library; the original sort didn't parallelize well, resulting in a performance hit.

TBB code size decreased on average by a very moderate 2.8% (*SD* = 4.4%).

In summary, while there is a moderate decrease in code size on average, program restructuring can also lead to moderately increased code sizes (Cilk),

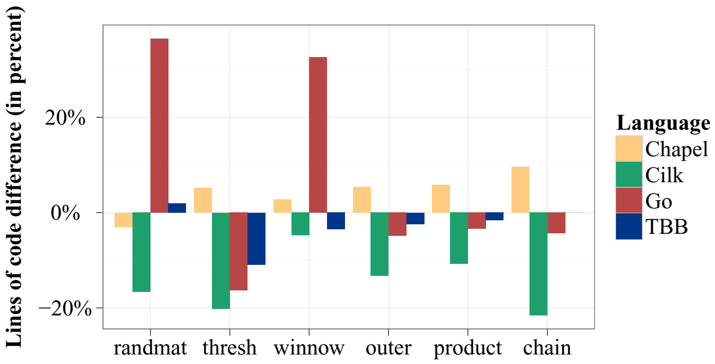


Fig. 1. Source code size (LoC) difference

and performance considerations may give reason to increase code size by about one third (Go). Large code size increases tend to indicate an algorithmic change, as in the case of Go. Large code size decreases indicate that functionality was duplicated needlessly and can be removed, as in the case of Cilk. Small changes indicate tweaking, where the code was overall fine, but could use refinement.

3.2 Execution Time

The performance tests were run on a 4 × Intel Xeon Processor E7-4830 (2.13 GHz, 8 cores; total 32 physical cores) server with 256 GB of RAM, running Red Hat Enterprise Linux Server release 6.3. Language and compiler versions used were: chapel-1.6.0 with gcc-4.4.6, for Chapel; Intel C++ Compiler XE 13.0 for Linux, for both Cilk and TBB; go-1.0.3, for Go.

Each performance test was repeated 30 times, and the mean of the results was taken. All tests use the same inputs, the size-dominant of which is a $4 \cdot 10^4 \times 4 \cdot 10^4$ matrix (about 12 GB of RAM). This size, which is the largest input size all languages could handle, was chosen to test scalability. The language Go provided the tightest constraint, while the other languages would have been able to scale to even larger sizes. An important factor in the measurement is that for all problems the I/O time is significant, since they involve reading/writing matrices to/from the disk. In order for the measurements to not be dominated by I/O, all performance tests were run with input and output code removed (input matrices were generated on-the-fly instead).

In Figure 2 the differences in execution time are displayed. Addressing research question **Q2**, on average expert comments reduced execution time by 18.1% (*SD* = 38.3%).

Again, results for the individual approaches show a number of differences. On average, execution time was reduced by 2.5% (*SD* = 48.0%) for Chapel. There is one outlier: in the problem thresh, the expert execution time increases by about 67.8%. The expert gave comments on a version that was compiled with

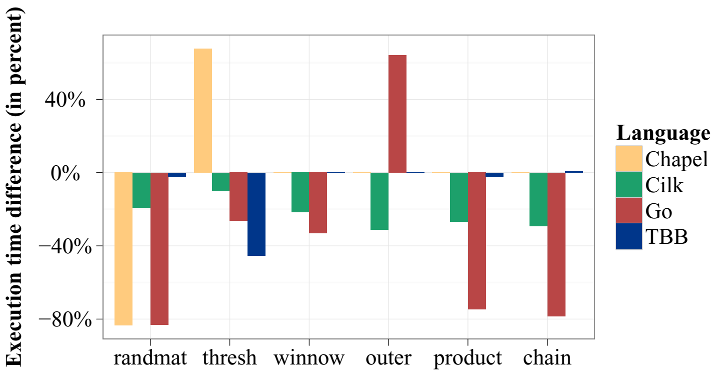


Fig. 2. Execution time difference

version 1.5 of Chapel. After changing to version 1.6 (as suggested by the expert) for the final measurements, the *non-expert* version experienced a significant reduction in execution time, while the expert version remained the same; this illustrates the often fragile nature of optimization.

Cilk's execution times were reduced by 23.0% on average ($SD = 7.8\%$).

Go's execution times were reduced by 38.6% on average ($SD = 55.8\%$). These improvements can be attributed to one important change in the way parallelism was achieved. In the non-expert versions, a divide-and-conquer pattern was frequently used. Instead, the expert recommended a distribute-work-synchronize pattern. While the divide-and-conquer approach creates one goroutine per task, the distribute-work-synchronize creates one for each processor core; for fine-grained task sizes, the overhead of the excessive creation of goroutines then causes a performance hit. Again, there was an outlier. In the problem outer, the Go expert had suggested to change the data structure from a one-dimensional to a two-dimensional array for clarity, without apparent performance differences on smaller problem sizes on a desktop machine. In the final measurement, it is however the cause of a 64% increase in execution time in the expert version; this highlights the fact that program optimizations have to take both the target machine and the target problem size into account.

TBB's execution times were reduced by 8.3% on average ($SD = 18.2\%$).

In summary, expert comments reduced execution time by a moderate amount. Also, there were outliers that increased the execution times, highlighting the fact that performance profiling is important in addition to expert knowledge.

3.3 Speedup

Figure 3 shows the changes in speedup on 32 cores; the speedup is measured relative to an execution on a single thread. Addressing research question **Q3**, across all languages/problems an average speedup of 1.5 is achieved ($SD = 1.1$).

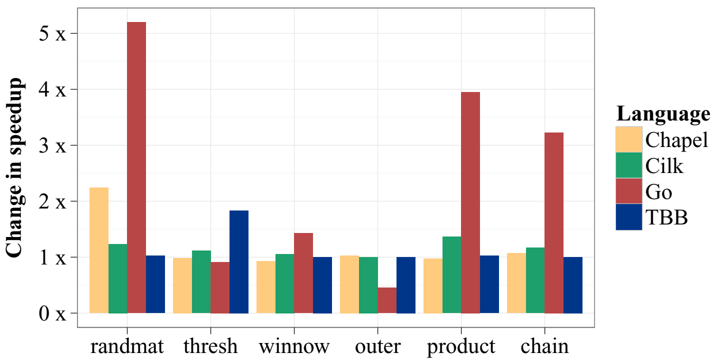


Fig. 3. Change in speedup

Except for Go, speedup seems to have been influenced little by the expert comments; most of the time no further speedup (i.e. 1 × speedup) is visible. Chapel shows on average a speedup of 1.2 (*SD* = 0.5), Cilk 1.2 (*SD* = 0.1), and TBB 1.1 (*SD* = 0.3).

In Go a more substantial average speedup of 2.5 (*SD* = 1.9) is visible, which is due to strong improvements in the case of randmat, product, and chain. This is most likely caused by the change in concurrency pattern used, as discussed in Section 3.2. It emphasizes the fact that it is critical in Go to know about idiomatic patterns to make full use of the performance offered by the language. A slowdown is visible for the outer problem in Go, which corresponds to the discussed issue in outer for the execution time difference.

In summary, speedup improvements due to expert comments are moderate in general. Only in the case of Go, the knowledge of an idiomatic pattern brought about significant improvements. Go highlights the fact that expertise is more valuable in approaches where there are fewer prepackaged solutions (such as parallel-for constructs).

3.4 Correction Time

Figure 4 displays which fraction of the original coding time was spent on implementing the corrections suggested by the experts. Addressing research question **Q4**, to implement the expert comments took about 29.9% of the original time spent on average (*SD* = 24.6%). Over all languages and problems, a maximum of 79.4% of the original coding time was spent.

This moderate overhead is reflected consistently by Chapel (29.3%, *SD* = 26.4%), Cilk (34.8%, *SD* = 20.2%), and Go (46.1%, *SD* = 26.0%). Only TBB has a significantly lower coding time overhead of 9.4% (*SD* = 12.8%).

In summary, none of the original problems needed to be rewritten completely; the changes were incremental. This is in accordance with the noted changes in source code size. Also, in combination with the observations about speedup, it

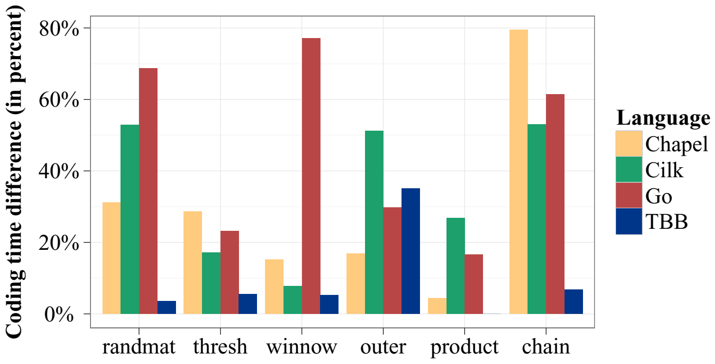


Fig. 4. Correction time

is clear that the time spent correcting the Go code translates into a notable speedup, making the availability of an expert an effective way to increase scalability and performance of the code.

4 Threats to Validity

As a threat to external validity, the study results are not necessarily generalizable to other languages and libraries. We have simply chosen four popular approaches, and it is interesting to see that the study results tend towards the same direction for all of them. But the results do not readily transfer to other approaches with entirely different programming abstractions and potentially different implementation quality.

Furthermore, it is arguable whether the study results transfer to large applications, due to the size of the programs used. The modest problem size is intrinsic to the study: the use of top experts is crucial to reliably answer the research questions and, unfortunately, this also means that the program size has to remain reasonable to fit within the review time the experts were able to donate. However, a recent study [13] confirms that the amount code dedicated to parallel constructs for 10K and 100K LOC programs is between 12 and 60 lines of code on average; this makes our study programs representative of the parallel portions of larger programs.

We use one developer (with six years of development experience; working at Google Inc.), and one expert per language (as listed in Section 1). Each expert was high-profile, i.e. using another expert or a group of experts would most likely lead to worse suggestions for improvement. However, the high-profile experts may not all have devoted the same effort to the task, leading to a suggestions of different quality; this effect could be mitigated by having groups of experts. Similarly to the point made above, using a group of developers, while preferable (e.g. addressing concerns regarding the influence of a learning effect when solving the same problem in different languages), would make the expert review step impossible (too many programs to review). To instead compare many novice programs with an ideal program is possible, but a markedly different study: an expert is required to filter out harmless differences between programs. Making all changes to turn one program into another is not representative of the effort required to bring the program up to an acceptable level.

Problem selection bias, a threat to internal validity, is avoided in part by using an existing problem set, instead of creating a new one. The threat that specific problems could be better suited to some languages than others remains, as it could already be present in the existing problem set. As a positive indication, none of the experts criticized the choice of problems.

5 Related Work

Although the claim that parallel programming requires domain experts is often repeated, few studies investigate the validity of this claim in the context of modern parallel programming approaches. However, a number of studies on *comparing* approaches to parallel programming influenced our work.

Szafron and Schaeffer [17] assess the usability of two parallel programming systems (a message passing library and a high-level parallel programming system) using a population of 15 students, and a single problem (transitive closure). Six metrics were evaluated: number of work hours, lines of code, number of sessions, number of compilations, number of runs, and execution time. They conclude that the high-level system is more usable overall, although the library is superior in some of the metrics; this highlights the difficulty in reconciling the results of different metrics.

Hochstein et al. [10] provide a case study of the parallel programmer productivity of novice parallel programmers. The authors consider two problems (game of life and grid of resistors) and two programming models (MPI and OpenMP). They investigate speedup, code expansion factor, time to completion, and cost per line of code, concluding that MPI requires more effort than OpenMP overall in terms of time and lines of code. Rossbach et al. [15] conducted a study with 237 undergraduate students implementing the same program with locks, monitors, and transactions. While the students felt on average that programming with locks was easier than programming with transactions, the transactional memory implementations had the fewest errors. Ebcioglu et al. [7] measure the productivity of three parallel programming languages (MPI, UPC, and X10), using 27 students, and a single problem (Smith-Waterman local sequence matching). For each of the languages, about a third of the students could not achieve any speedup.

Nanz et al. [11] present an empirical study with 67 students to compare the ease of use (program understanding, debugging, and writing) of two concurrency programming approaches (SCOOP and multi-threaded Java). They use self-study to avoid teaching bias and standard evaluation techniques to avoid subjectivity in the evaluation of the answers. They conclude that SCOOP is easier to use than multi-threaded Java regarding program understanding and debugging, and equivalent regarding program writing. Burkhart et al. [3] compare

Chapel against non-PGAS models (Java Concurrency, OpenMP, MPI, CUDA, PATUS) in a classroom setting both in terms of productivity (working hours, parallel overhead, lines of code, learning curve) and performance. Results for Chapel were favorable on the productivity metrics, but lagged behind other languages on the performance side. Cantonnet et al. [4] analyze the productivity of two languages (UPC and MPI), using the metrics of lines of code and conceptual complexity (number of function calls, parameters, etc.), obtaining results in favor of UPC. Bal [1] is a practical study based on actual programming experience with five languages (SR, Emerald, Parlog, Linda and Orca) and two problems (traveling salesman, all pairs shortest paths). It reports the authors' experience while implementing the solutions.

Skillicorn and Talia [16] is an assessment of the suitability for realistic portable parallel programming of parallel programming models and languages, using six criteria (ease of programming, presence of a software development methodology, architecture-independence, ease of understanding, guarantee of performance, and estimation of cost). It includes a classification of models of parallel computation.

6 Conclusion

In order to make developers embrace parallel programming, the reputation of parallelism as an arcane art has to be dispelled. Designers of parallel programming approaches work towards this goal, but results supporting their claims of improved performance and usability are scarce. While it is easy to check that one approach can offer performance improvements over another, it is entirely unclear whether a non-expert would ever achieve this performance in practice.

In this paper we presented, to the best of our knowledge, the first study that explores the alleged gap between expert and non-expert parallel programmers. The results positively confirm the effectiveness of the design of Chapel, Cilk, Go, and TBB: all the approaches "work" in the sense that, on average, a top expert can only to a moderate degree improve programs written by a non-expert; the study confirms this across four program metrics, namely code size, execution time, speedup, and coding time.

More studies are needed to investigate the difference between expert and non-expert usage, and we hope that our study incites more work in this direction. In particular, our study can be applied to a greater variety of languages, e.g. to include widely used approaches such as OpenMP and MPI. The most expensive resources in our study were the experts: the time they were able to spend on reviewing programs strongly limited the number of programs in the study. In future work, it would be interesting to replace expert review by comparisons of expert-written, "ideal" programs with non-expert ones. While such an approach would not be able to replicate the detailed insights gained by the review, it would make it easier to obtain more data for the analysis.

Acknowledgments. We are grateful to Brad Chamberlain, Arch D. Robison, Jim Sukha, and Luuk van Dijk for devoting their time, effort, and expertise to this study. This work was partially supported by the European Research Council under the European Union's Seventh Framework Programme (ERC Grant agreement no. 291389), FAPERGS/CNPq 008/2009 Research Project: "GREEN-GRID: Sustainable High Performance Computing", the Hasler Foundation, and ETH (ETHIIRA).

References

1. Bal, H.E.: A comparative study of five parallel programming languages. Future Generation Computer Systems 8(1-3), 121–135 (1992)
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: PPoPP 1995, pp. 207–216. ACM (1995)
3. Burkhart, H., Sathe, M., Christen, M., Rietmann, M., Schenk, O.: Run, stencil, run: HPC productivity studies in the classroom. In: PGAS 2012 (2012)
4. Cantonnet, F., Yao, Y., Zahran, M.M., El-Ghazawi, T.A.: Productivity analysis of the UPC language. In: IPDPS 2004. IEEE Computer Society (2004)
5. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. 21(3), 291–312 (2007)
6. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press (1991)
7. Ebcioglu, K., Sarkar, V., El-Ghazawi, T., Urbanic, J.: An experiment in measuring the productivity of three parallel programming languages. In: P-PHEC 2006, pp. 30–37 (2006)
8. Go programming language, `http://golang.org/`
9. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
10. Hochstein, L., Carver, J., Shull, F., Asgari, S., Basili, V.: Parallel programmer productivity: A case study of novice parallel programmers. In: SC 2005. IEEE Computer Society (2005)
11. Nanz, S., Torshizi, F., Pedroni, M., Meyer, B.: Design of an empirical study for comparing the usability of concurrent programming languages. In: ESEM 2011, pp. 325–334. IEEE Computer Society (2011)
12. Nanz, S., West, S., Soares da Silveira, K., Meyer, B.: Benchmarking usability and performance of multicore languages. In: ESEM 2013. IEEE Computer Society (2013)
13. Okur, S., Dig, D.: How do developers use parallel libraries? In: FSE 2012, pp. 54:1–54:11. ACM (2012)
14. Reinders, J.: Intel threading building blocks – outfitting C++ for multi-core processor parallelism. O'Reilly (2007)
15. Rossbach, C.J., Hofmann, O.S., Witchel, E.: Is transactional programming actually easier? In: PPoPP 2010, pp. 47–56. ACM (2010)
16. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. ACM Comput. Surv. 30(2), 123–169 (1998)
17. Szafron, D., Schaeffer, J.: An experiment to measure the usability of parallel programming systems. Concurrency: Pract. Exper. 8(2), 147–166 (1996)
18. Wilson, G.V., Irvin, R.B.: Assessing and comparing the usability of parallel programming systems. Tech. Rep. CSRI-321, University of Toronto (1995)

Programming with BSP Homomorphisms

Joeffrey Legaux[2], Zhenjiang Hu[1], Frédéric Loulergue[2], Kiminori Matsuzaki[3], and Julien Tesson[4]

[1] National Institute of Informatics, Tokyo, Japan
`Hu@nii.ac.jp`
[2] LIFO, Université d'Orléans, France
{`Joeffrey.Legaux`,`Frederic.Loulergue`}`@univ-orleans.fr`
[3] Kochi University of Technology, Kochi, Japan
`matsuzaki.kiminori@kochi-tech.ac.jp`
[4] Université Paris Est, LACL, UPEC, France
`Julien.Tesson@lacl.fr`

Abstract. Algorithmic skeletons in conjunction with list homomorphisms play an important role in formal development of parallel algorithms. We have designed a notion of homomorphism dedicated to bulk synchronous parallelism. In this paper we derive two application using this theory: sparse matrix vector multiplication and the all nearest smaller values problem. We implement a support for BSP homomorphism in the Orléans Skeleton Library and experiment it with these two applications.

Keywords: Algorithmic skeletons, Constructive algorithms, Bulk synchronous parallelism, All nearest smaller values, Sparse linear algebra.

1 Introduction

Parallel programming needs to be as widespread as parallel machines that now range from smartphones to supercomputers. Structured models of parallelism such as algorithmic skeletons [2] or bulk synchronous parallelism [21], ease the writing and reasoning on parallel programs. Algorithmic skeletons are, or can be seen as, higher-order functions that capture usual parallel patterns but that have a semantics identical or close to usual higher-order functions on collections, in particular lists. The most famous ones are the map and reduce skeletons. Bulk synchronous parallelism offers an abstract and simple model of parallelism yet allowing to take realistically into account the communication costs of parallel algorithms. It has been used in many application domains.

The theory of lists [1] is a powerful tool to systematically develop correct functional programs. From a specification, or naive implementation of a program, it allows to derive step-by-step, a more efficient version. Algorithmic skeletons in conjunction with list homomorphisms (or homomorphisms for short) play an important role in formal development of parallel algorithms [3, 7, 14].

We have defined a notion of homomorphism dedicated to bulk synchronous parallelism, and explored its theory [5, 17] in the context of the Coq proof assistant [18]. Our SDPP [19] framework allows to derive step-by-step correct parallel programs in Coq and then to extract functional parallel programs for the

OCaml [10] language and the BSML library [11] that can be compiled and run in parallel. If our long term goal is to provide sufficient automation to use the Coq proof assistant to ease the development of efficient parallel programs, our framework still lacks automation and the purely functional programs we can extract cannot compete yet with high-level C++ hand-written code. Therefore on a practical side it would be interesting to have a support for BSP homomorphisms in an efficient library of algorithmic skeletons such as OSL : the C++ Orléans Skeleton Library [8]. The work presented in this paper provides such a support and we illustrate its use through the derivation of non-trivial applications.

The main technical contributions of this paper can be summarised as follows.

- We derive two applications in a systematic way using the theory of BSP homomorphisms: a sparse-matrix vector multiplication and the all nearest smaller values algorithm;
- We implement support for the execution of BSP homomorphisms in the Orléans Skeleton Library;
- We experiment with these applications implemented with OSL on parallel machines.

The organisation of this paper is as follows. We start by reviewing the basic concepts of homomorphism and recall the definition of the BSP homomorphisms and their theory (section 2). We then show how to derive BSP homomorphisms from specifications in section 3. Section 4 is devoted to the Orléans Skeleton Library, in particular support for BSP homomorphisms with the `bh` skeleton. We experiment with the derived applications in section 5. We discuss the related work in Section 6 and conclude the paper in section 7.

2 BSP Homomorphisms

Our notations are basically based on the programming language Haskell [15]. Functional application is denoted by a space and an argument may be written without brackets. Thus $f\ a$ means $f(a)$. Functions are curried, i.e. functions take one argument and return a function or a value, and the function application associates to the left. Thus $f\ a\ b$ means $(f\ a)\ b$. Infix binary operators will often be denoted by $\oplus$, $\otimes$, $\odot$. Functional application binds stronger than any other operators, so $f\ a \oplus b$ means $(f\ a) \oplus b$, but not $f(a \oplus b)$. Lists are finite sequences of values of the same type. A list is either the empty list, a singleton or a concatenation of two lists. We denote $[]$ for the empty list, $[a]$ for a singleton list with element a, and $x +\!\!+\, y$ for a concatenation of two lists x and y. The concatenation operator is associative. Lists are destructed by pattern matching.

Definition 1 (Homomorphism). Function h is said to be a *homomorphism*, if it is defined recursively in the form of

$$h\ [\,] = id_{\odot} \qquad h\ [a] = f\ a \qquad h\ (x +\!\!+\, y) = h(x) \odot h(y)$$

where $id_{\odot}$ denotes the identity unit of $\odot$. Since h is uniquely determined by f and $\odot$, we will write $h = (\![\odot, f]\!)$.

Definition 2 (BSP Homomorphism (BH)). Given a function k, and two homomorphisms $g_1 = (\!|\oplus, f_1|\!)$, $g_2 = (\!|\otimes, f_2|\!)$[1], h is said to be a *BH*, if it is defined in the following way.

$$\begin{cases} h\ []\ l\ r = [] \\ h\ [a]\ l\ r = [k\ a\ l\ r] \\ h\ (x +\!\!+\, y)\ l\ r = h\ x\ l\ (g_1\ y \oplus r)\ +\!\!+\ h\ y\ (l \otimes g_2\ x)\ r \end{cases}$$

The above h defined with functions k, g_1, g_2, and associative operators $\oplus$ and $\otimes$ is denoted as $h = BH(k, (\!|\oplus, f_1|\!), (\!|\otimes, f_2|\!))$.

Function h is a *higher-order* homomorphism, which computes on a list and returns a new list of the same length. In addition to the input list, h has two additional parameters, l and r, which append necessary information to perform computation on the list. The information of l and r, as defined in the third equation, is propagated from left and right with functions $g_2, \otimes$ and $g_1, \oplus$ respectively. By definition, a BH can be computed in parallel since it is a composition of local computations and of homomorphisms which can be easily parallelised [3].

Rather than checking directly that a function is a *BH* we use an indirect way using the *mapAround* function. *mapAround*, compared to *map*, captures more interesting independent computations on each element of lists. Intuitively, *mapAround* maps a function to each element (of a list) but is allowed to use information of the sublists on the left and right of the element, e.g.,

$$\begin{aligned} & mapAround\ f\ [x_1, x_2, \ldots, x_n] \\ = & [f\ ([], x_1, [x_2, \ldots, x_n]), f\ ([x_1], x_2, [x_3, \ldots, x_n]), \ldots, f\ ([x_1, \ldots, x_{n-1}], x_n, [])]. \end{aligned}$$

Theorem 1 (Parallelization *mapAround* with *BH*). For a function $h = mapAround\ f$, if we can decompose f as $f\ (ls, x, rs) = k\ (g_1\ ls, x, g_2\ rs)$ where g_i is a composition of a function p_i with a homomorphism, $g_i\ x = p_i((\!|\oplus_i, k_i|\!)\ x)$, then

$$h\ xs = BH(k', (\!|\oplus_1, k_1|\!), (\!|\oplus_2, k_2|\!))\ xs\ \iota_{\oplus_1}\ \iota_{\oplus_2}$$

where $k'\ (l, x, r) = k(p_1\ l, x, p_2\ r)$ holds, where $\iota_{\oplus_1}$ is the (left) unit of $\oplus_1$ and $\iota_{\oplus_2}$ is the (right) unit of $\oplus_2$.

Proof. This can be proved by induction on the input list of h. The detailed proof in Coq is discussed in [5, 17].

Theorem 1 is general and powerful in the sense that it can parallelize not only *mapAround* but also other collective functions, such as *scan*, to *BH* [5, 17].

3 Program Derivation Using BSP Homomorphisms

In this section, we demonstrate with two nontrivial examples how to derive applications using the BH theory. One is the all nearest smaller values problem and the other is the sparse matrix-vector multiplication.

[1] See [17] for a discussion about weaker conditions for the definition of BSP homomorphism.

3.1 All Nearest Smaller Values

The All Nearest Smaller Values (ANSV) problem is as follows:

> Let $as = [a_1, a_2, \ldots, a_n]$ be an array of elements from a totally ordered domain. For each a_j, find the nearest element to the left of a_j and the nearest element to the right of a_j that are less than a_j. If there is no such an element, we put $-\infty$ instead.

An example of the input and the output for the function *ansv* that solves this problem is as follows.

$$\begin{aligned}&\mathit{ansv}\ [3,1,4,1,5,9,2,6,5]\\&= [(-\infty,1),(-\infty,-\infty),(1,1),(-\infty,-\infty),(1,2),(5,2),(1,-\infty),(2,5),(2,-\infty)]\end{aligned}$$

A direct specification of the ANSV algorithm is as follows:

```
$ansv\ as = mapAround\ nsv\ as$
            $\textbf{where}\ nsv\ (ls, x, rs) = (nsvL\ x\ ls, nsvR\ x\ rs)$
                  $nsvL\ x\ [\,] = -\infty$
                  $nsvL\ x\ (ls \mathbin{+\!\!+} [l]) = \textbf{if}\ l < x\ \textbf{then}\ l\ \textbf{else}\ nsvL\ x\ ls$
                  $nsvR\ x\ [\,] = -\infty$
                  $nsvR\ x\ ([r] \mathbin{+\!\!+} rs) = \textbf{if}\ r < x\ \textbf{then}\ r\ \textbf{else}\ nsvR\ x\ rs$
```

where we simply use *mapAround* to compute on each element and its surround (left and right arrays) with the function *nsv*. In the definition of *nsv*, *nsvL x ls* is to compute the rightmost element in *ls* that is less than *x*, while *nsvR x rs* computes the leftmost element in *rs* that is less than *x*.

However, to use the Theorem 1, the computations on the left and right arrays need to be expressed as homomorphisms independent from the center element (this precondition derives from the function shape requirement in the theorem). We can give such a definition where we first select the candidates from the left and right arrays and then choose a correct one from them. Since the computation for the left and right is symmetrical, we here discuss the right one.

We are looking for the nearest smaller values, thus we can discard values that are equal or superior to the previous elements and only retain a sample of decreasing candidates as shown in Figure 1. Therefore, we can decompose the definition of *nsvR* as follows into a homomorphism that removes unnecessary elements from an array and a function that picks up the nearest smaller value. Since the result of the homomorphism is a list in which elements are in decreasing order, the binary operator of the homomorphism just removes elements from the right list that are larger than the rightmost element.

```
$nsvR\ v\ rs = pickupR\ v\ ((\![\oplus, id]\!)\ rs)$
    $\textbf{where}\ (ls \mathbin{+\!\!+} [l]) \oplus rs = ls \mathbin{+\!\!+} [l] \mathbin{+\!\!+} dropWhile\ (\lambda x.x > l)\ rs$
          $pickupR\ x\ [\,] = -\infty$
          $pickupR\ x\ ([r] \mathbin{+\!\!+} rs) = \textbf{if}\ r < x\ \textbf{then}\ r\ \textbf{else}\ pickupR\ x\ rs$
```

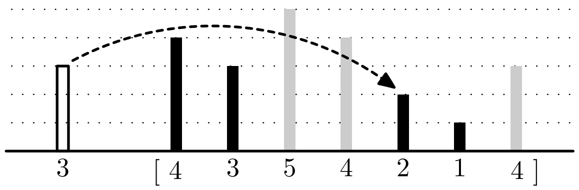


Fig. 1. The candidates in a right array. The values that keep a decreasing order are kept as candidates (in black), while the others are discarded (in gray). Here the value 2 will be kept as the final solution since it is the closest candidate that is inferior to the center value 3 (in white).

We have decomposed *nsvR* into a function *pickupR* and a homomorphism $(\![\oplus, id]\!)$; the function *nsvL* can be similarly decomposed into *pickupL* and $(\![\otimes, id]\!)$. Thus we can rewrite *nsv* as follows :

$$\begin{aligned} nsv\ (ls, x, rs) &= k\ ((\![\otimes, id]\!)\ ls, x, (\![\oplus, id]\!)\ rs) \\ \textbf{where}\ k\ (l, x, r)\ &=\ (pickupL\ x\ l,\ pickupR\ x\ r) \end{aligned}$$

This form matches the one needed to apply the Theorem 1 in order to derive the ANSV into a *BH*.

3.2 Sparse Matrix-Vector Multiplication

Sparse matrices are often compressed into array representations. We develop a parallel program to compute the multiplication of a sparse matrix and a vector.

Here we consider an array representation that consists of triples (y, x, a):

- y: the row-index of the nonzero element,
- x: the column-index of the nonzero element, and
- a: the value of the nonzero element.

We assume that elements are sorted with respect to the indices y and x. For example, the following matrix A is represented by the array *as* with five triples.

$$A = \begin{pmatrix} 1.1 & 2.2 & 0 \\ 0 & 1.3 & 1.4 \\ 0 & 0 & 3.5 \end{pmatrix} \qquad \begin{aligned} as = [&(0,0,1.1),\ (0,1,2.2), \\ &(1,1,1.3),\ (1,2,1.4),\ (2,2,3.5)] \end{aligned}$$

In the matrix-vector multiplication, there is a result element for each row. Let us put the result on the first element in the row, and clear the other values with a dummy value denoted as $\Box$. For example, multiplying a vector $[3.0, 4.0, 1.0]$ to the array representation *as* yields

$$mult\ as\ [3.0, 4.0, 1.0] = [(0,0,12.1),\ (0,1,\Box),\ (1,1,6.6),\ (1,2,\Box),\ (2,2,3.5)]\ .$$

Note that we can apply the array packing [5] to compact the result into the result vector $[12.1, 6.6, 3.5]$.

Now we develop the specification of this problem using the *mapAround* function. The first and important step is to determine which kind of values are needed from the left or from the right. To check whether an element is the first one in the row, we simply compare the row-index of the element with that of the left element. When we compute the result value, we need the partial sum of the rightward values in the row, multiplied by the vector. Therefore, the values passed from the right are the row-index of the right element and the partial sum in the row (of right element). Based on these insights, we can develop a specification with the *mapAround* function. In the following program, $v\langle i\rangle$ denotes the ith element of the vector v.

```
$mult\ as\ v = mapAround\ (f\ v)\ as$
$\mathbf{where}\ f\ v\ (ls, (y,x,a), rs) = \mathbf{let}\ y_l = g_l\ ls; (y_r, s_r) = g_r\ v\ rs$
                              $\mathbf{in\ if}\ (y_l == y)\ \mathbf{then}\ (y, x, \Box)$
                                $\mathbf{elseif}\ (y_r == y)\ \mathbf{then}\ (y, x, v\langle x\rangle * a + s_r)$
                                $\mathbf{else}\ (y, x, v\langle x\rangle * a)$
```

Now we give the definition of the auxiliary functions and check that they are homomorphisms. The function g_l just takes the row-index of the last element in a list. It is a homomorphism

$$g_l = (\![\gg, \lambda(x,y,a).y]\!) \quad \mathbf{where}\ \ a \gg b = b\ ,$$

and any value (here we use -1) is a left unit of the operator $\gg$. The function $g_r\ v$ is a bit more complicated and is defined as follows.

```
$g_r\ v\ [(y,x,a)] = (y, a * v\langle x\rangle)$
$g_r\ v\ [as +\!\!+ (y,x,a)] = \mathbf{let}\ (y', s) = g_r\ v\ as$
                          $\mathbf{in}\ (y', \mathbf{if}\ y' == y\ \mathbf{then}\ s + a * v[x]\ \mathit{else}\ s)$
```

This function is indeed a homomorphism as follows.

```
$g_r\ v\ [(y,x,a)] = (y, a * v\langle x\rangle)$
$g_r\ v\ (ls +\!\!+ rs) = g_r\ v\ ls \odot g_r\ v\ rs$
    $\mathbf{where}\ (y_l, s_l) \odot (y_r, s_r) = \mathbf{if}\ y_l == y_r\ \mathbf{then}\ (y_l, s_l + s_r)\ \mathbf{else}\ (y_l, s_l)$
```

A right unit of the operator $\odot$ is $(-1, 0)$.

Now we can apply the Theorem 1 to the specification above and obtain the following BH.

```
$mult\ as\ v = BH(k\ v, (\![\odot, \lambda(y,x,a).(y, a * v\langle x\rangle)]\!), (\![\gg, \lambda(x,y,a).y]\!))\ as$
  $\mathbf{where}\ k\ v\ (y_l, (y,x,a), (y_r, s)) = \mathbf{if}\ y == y_l\ \mathbf{then}\ (y, x, \Box)$
                                 $\mathbf{elseif}\ y == y_r\ \mathbf{then}\ (y, x, a * v\langle x\rangle + s)$
                                 $\mathbf{else}\ (y, x, a * v\langle x\rangle)$
        $a \gg b = b$
        $(y_l, s_l) \odot (y_r, s_r) = \mathbf{if}\ y_l == y_r\ \mathbf{then}\ (y_l, s_l + s_r)\ \mathbf{else}\ (y_l, s_l)$
```

4 BH in the Orléans Skeleton Library

4.1 An Overview of Orléans Skeleton Library

Orléans Skeleton Library is a C++ library of data-parallel algorithmic skeletons. It is implemented on top of MPI and takes advantage of the expression templates optimisation techniques [22] to be very efficient yet allowing programming in a functional style. Programming with OSL is very similar to programming in sequential as OSL offers a global view of parallel programs [4]. OSL programs operate on *distributed arrays* that are one dimensional arrays such that, at the time of the creation of the array, data is distributed among the processors. Distributed arrays are implemented as a template class `DArray<T>`. A distributed array consists of `bsp_p` partitions, where `bsp_p` is the number of processing elements of the parallel (BSP) machine. Each partition is an array of elements of type `T`.

To give a quick, yet precise, overview of OSL, Fig. 2 presents an informal semantics for the main OSL skeletons together with their signatures. In this figure, `bsp_p` is noted p. A distributed array of type `DArray<T>` can be seen "sequentially" as an array $[t_0, \ldots, t_{t.size-1}]$ where $t.size$ is the (global) size of the (distributed) array t (and we use the same notation if t is a C++ vector). But as with the `getPartition` skeleton, the user can expose the distribution of the distributed array, this informal semantics should also indicates how the array is distributed. We write the distribution as a subscript D of the distributed array. D is a function from $\{0, \ldots, \texttt{bsp\_p} - 1\}$ to $\mathbb{N}$.

The first skeleton, `map` (and variants such as `zip`, `mapIndex`, *etc.*) is the usual combinator used to apply a function to each element of a distributed array (or two for `zip`). The first argument of both `map` and `zip` could be a C++ functor either extending `std::unary_function` or `std::binary_function`, respectively.

Parallel reduction and parallel prefix computation with a binary *associative* operator $\oplus$ are performed using respectively the `reduce` and `scan` skeletons. Communications are needed to execute both skeletons.

`permute` and `shift` are communication skeletons. The next skeleton only modifies the distribution of the distributed array, not its content: `redistribute` distributes the content of the distributed array according to a vector of integers representing the target distribution. All the skeletons up to `redistribute` preserve the distribution. It means that if they are applied to evenly distributed arrays, the result will be an evenly distributed array. The `redistribute` skeleton may thus seems useless. However, some algorithms such as BSP regular sampling sort, require intermediate and final results that are not evenly distributed. To implement such algorithms, two additional skeletons are needed: `getPartition` and `flatten`. The `getPartition` skeleton exposes how a distributed array is distributed among the processors, while `flatten` is the inverse operation.

As a very short OSL example program, we can compute the variance $\sum_{i=0}^{n-1}(x_i - \frac{\sum_{j=0}^{n-1} x_j}{n})$ of a sequence of random variables x_i:

Skeleton	**Signature**
	Informal semantics
`map`	`DArray<W> map(W f(T), DArray<T> t)`
	`map`$(f,\ [t_0,\ldots,t_{t.size-1}]_D) = [f(t_0),\ldots,f(t_{t.size-1})]_D$
`reduce`	`<T> reduce(T⊕(T,T), DArray<T> t)`
	`reduce`$(\oplus,\ [t_0,\ldots,t_{t.size-1}]_D) = t_0 \oplus t_1 \oplus \ldots \oplus t_{t.size-1}$
`scan`	`DArray<T> scan(T⊕(T,T), DArray<T> t)`
	`scan`$(\oplus,\ [t_0,\ldots,t_{t.size-1}]_D) = [\oplus_{i=0}^{0} t_i;\ldots;\oplus_{i=0}^{t.size-1} t_i]_D$
`permute`	`DArray<T> permute(int f(int), DArray<T> t)`
	`permute`$(f,\ [t_0,\ldots,t_{t.size-1}]_D) = [t_{f^{-1}(0)},\ldots,t_{f^{-1}(t.size-1)}]_D$
`shift`	`DArray<T> shift(int o, T f(T), DArray<T> t)`
	`shift`$(o,\ f,\ [t_0,\ldots,t_{t.size-1}]_D) = [f(0),\ldots,f(o-1),t_0,\ldots,t_{t.size-1-o}]_D$
`redistribute`	`DArray<T> redistribute(Vector<int> dist, DArray<T> t)`
	`redistribute`$(\mathtt{dist},[t_0,\ldots,t_{t.size-1}]_D) = [t_0,\ldots,t_{t.size-1}]_{\mathtt{dist}}$
`getPartition`	`DArray< Vector<T> > getPartition(DArray<T> t)`
	`getPartition`$([t_0,\ldots,t_{t.size-1}]_D)$ $= \big[[t_0,\ldots,t_{D(0)-1}],\ldots,[t_{j_i},\ldots,t_{j_i+D(i)-1}],\ldots,[t_{j_{p-1}},\ldots,t_{t.size-1}]\big]_{E_p}$ where $E_p(i) = 1$ and $j_i = \sum_{k=0}^{k=i-1} D(k)$
`flatten`	`DArray<T> flatten(DArray< Vector<T> > t)`
	`flatten`$([a_0,\ldots,a_{a.size-1}]_D)$ $= \big[a_0[0],\ldots,a_0[a_0.size-1],a_1[0],\ldots,a_{a.size-1}[a_{a.size-1}.size-1]\big]_{D'}$ where $D'(i) = \sum_{j_i \le k < j_i+D(i)} a_k.size$ and $j_i = \sum_{k=0}^{k=i-1} D(k)$

Fig. 2. OSL Skeletons

```
double avg = osl::reduce(std::plus<double>(), x) / x.getGlobalSize();
double variance = osl::reduce(std::plus<double>(),
                  osl::map(boost::bind(std::minus<double>(),avg, _2), x));
```

4.2 Using the BH Skeleton

The signature of the `bh` skeleton is:

```
DArray<typename K::result_type>
  bh(K k, Homomorphism<T, L> * hl, Homomorphism<T, R> * hr,
    L l, R r, const DArray<T>& temp)
```

According to Definition 2, a BH is defined by a function `k` and two homomorphisms `g1` and `g2`, which are applied on a list (in the form of a distributed array `temp`) with two boundary elements `L` and `R`.

`k` can be easily implemented as a usual functor whose `()` operator takes three arguments: the left summary (which will be the result of the application of `g1` on the left part of the list), the current element and the right summary. For `g1` and `g2`, we define a generic virtual base class `Homomorphism` which defines the needed function f, operator $\odot$ and its unit $id_\odot$ (Definition 1). The user can then implement its own homomorphism by creating a derived class that provides concrete implementations of those 3 items.

`k`, `g1` and `g2` are the first three parameters of our generic BH skeleton. To apply it to actual data, we need to provide three last arguments: the boundary elements `L` and `R`, and the list in the form of a `DArray`. The return value will be a list of the same size, whose type of elements will be the result type of `k`.

Implementing for example the computation of the prefix-sum on an array of integers can be easily done. First, we need the left homomorphism that subsequently adds all the values:

```
class HAdd: public Homomorphism<int, int> {
    public:
        HAdd() { neutral = 0;}
        inline int f(const int& i) {return i;}
        inline int o(const int& i1, const int& i2) {return i1+i2;}
};
```

We do not have any computation to conduct on the right side. However we still need to provide an homomorphism to the `bh` skeleton, so we can define one that always returns the same value. This homomorphism, named `HConst`, is defined in a similar way than `HAdd` but with each operator returning `0`.

We now only have to define the `k` function which will simply add the computed sum of the left sub-array with the current element:

```
struct AddLeft {
  typedef int result_type;
  inline int operator()(int l, int i, int r) const { return l+i; }
};
```

We can now apply the skeleton to compute the prefix sum on any distributed array `d`, using zeros for the boundary values:

```
DArray<int> result = osl::bh(AddLeft(),new HAdd(), new HConst(), 0, 0, d);
```

4.3 Implementation of the BH Skeleton

The `bh` skeleton is implemented with the usual expression template mechanism of our library, so it can be integrated seamlessly in any OSL expression and trigger the fusion optimisation when it is relevant. The recursive definition of homomorphisms provides room for a major optimisation. If we apply the definition to an array of elements, we can write the third recursive rule as such:

$$h\ [x_1, \ldots, x_n] = h\ [x_1, \ldots, x_n - 1] \odot h\ [x_n] = h\ [x_1, \ldots, x_n - 1] \odot f\ (x_n).$$

This allows us to pre-compute locally the application of the homomorphism to each sub-array in a linear time as we only have to apply f and $\odot$ once per element. Without this optimisation, we would have to conduct these operations i times for each of the n x_i elements, thus resulting in a square complexity. Thanks to the associativity of homomorphisms, we can symmetrically implement the same optimisation for the right homomorphism that applies on the end of the array.

A disadvantage is that in order to achieve this purpose we have to consider the local part of the array on each node in its entirety: This forces us to break the loop fusion mechanism, which is based on the fact that each element of the array is treated separately. However fusion can still occur on the expression (if there actually exists one) that produces the input array `temp`.

5 Experiments

We implemented programs computing the ANSV and sparse-matrix vector multiplication using our implementation of the BH skeleton in the OSL library. We then measured the scalabity of those programs when parallelised over several cores on two architectures : a shared-memory computer containing 4 processors with 12 computer cores each (thus a total of 48 cores), and a distributed-memory cluster of 8 machines each containing 2 processors of 4 cores (for a total of 64 cores). More experiments are currently undergoing on a larger cluster containing several hundreds cores. Those measures were conducted using a statistical evaluation protocol [20] in order to ensure stability and reproducibility of the results. ANSV was solved on a 10^7 elements array. Sparse matrix-vector multiplication was conducted on a 10^9 elements matrix with 10% of non-zero elements, leading to an actual 10^8 elements of data.

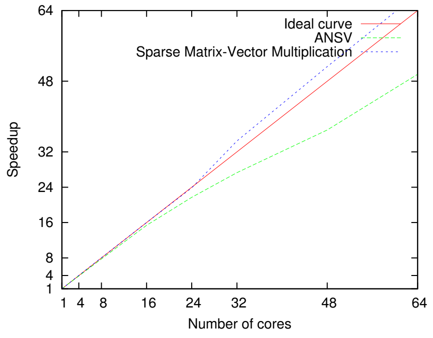


Fig. 3. Distributed memory

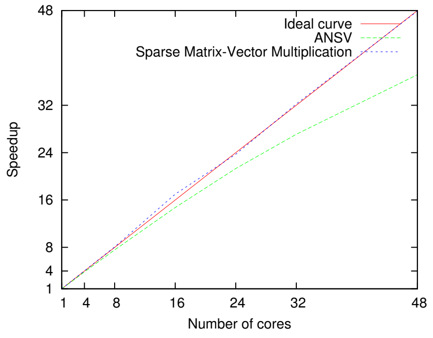


Fig. 4. Shared memory

The ANSV problem scales well although sub-linearly, we may expect its performance to peak at a greater number of cores. This could be explained by the fact that each processor has to communicate its local array of candidate elements to every other core. Those arrays can reach a consequent size on big problems, and the cost of this communication operation may rapidly overcome the parallelisation gains on larger numbers of cores.

On the other hand, the sparse matrix-vector multiplication is perfectly linear. As in this problem the processors only have to exchange a pair of numbers, the communication cost is probably too small to impact the scaling of the algorithm at this level. We also get super-linear speedups on the distributed architecture with a large number of cores, which seems to indicate that this particular computation is limited by the memory bandwidth on the shared memory architecture.

6 Related Work

There are many algorithmic skeletons libraries, for various host languages: [6] is a recent survey of such libraries. Depending on the supported data structures, these libraries could be used to implement programs obtained by systematic developement based on the theory of lists [3, 14], trees [13] or arrays. However none support BSP homomorphisms. Compared to BSP implementations of skeletons [23] together with usual theories, our theoretical framework and OSL library allow to derive and implement efficient programs such as the all nearest smaller values program.

Several researchers worked on formal semantics for BSP computations, for example [9,16]. But to our knowledge none of these semantics was used to generate programs as the last step of a systematic development. LOGS is a semantics of BSP programs and was used to generate C programs [24]. The main difference with our approach is that it starts from a local and imperative view of parts of the program to build a larger one, and we start from a global and functional view and refine it.

7 Conclusion and Future Work

The theory of bulk synchronous parallel homomorphism allows to derive non-trivial applications. The support of BSP homomorphism in the Orléans Skeleton Library through the *BH* skeleton can be used to implement such applications. In the SkeTo and OSL libraries, fusion [12] is done by the expression templates technique. More global optimisations could be done, in particular using the Proto framework for C++: This is planned. However we still need to investigate the theory of fusion for BSP homomorphisms before incorporing *BH* fusion in OSL.

Acknowledgements. This work is partly supported by ANR (France) and JST (Japan) (project PaPDAS ANR-2010-INTB-0205-02 and JST 10102704). Joeffrey Legaux is supported by a PhD grant from the *Conseil Général du Loiret*.

References

1. Bird, R.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design, pp. 5–42. Springer (1987)
2. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989), `http://homepages.inf.ed.ac.uk/mic/Pubs`
3. Cole, M.: Parallel Programming with List Homomorphisms. Parallel Processing Letters 5(2), 191–203 (1995)
4. Deitz, S.J., Callahan, D., Chamberlain, B.L., Snyder, L.: Global-view abstractions for user-defined reductions and scans. In: PPoPP, pp. 40–47. ACM, New York (2006)
5. Gesbert, L., Hu, Z., Loulergue, F., Matsuzaki, K., Tesson, J.: Systematic Development of Correct Bulk Synchronous Parallel Programs. In: International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 334–340. IEEE (2010)
6. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Software, Practrice & Experience 40(12), 1135–1160 (2010)

7. Gorlatch, S., Bischof, H.: Formal Derivation of Divide-and-Conquer Programs: A Case Study in the Multidimensional FFT's. In: Mery, D. (ed.) Formal Methods for Parallel Programming: Theory and Applications, pp. 80–94 (1997)
8. Javed, N., Loulergue, F.: Parallel Programming and Performance Predictability with Orléans Skeleton Library. In: International Conference on High Performance Computing and Simulation (HPCS), pp. 257–263. IEEE (2011)
9. Jifeng, H., Miller, Q., Chen, L.: Algebraic laws for BSP programming. In: Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y. (eds.) Euro-Par 1996, Part II. LNCS, vol. 1124, pp. 359–368. Springer, Heidelberg (1996)
10. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml System release 4.00.0 (2012), `http://caml.inria.fr`
11. Loulergue, F.: Parallel Juxtaposition for Bulk Synchronous Parallel ML. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 781–788. Springer, Heidelberg (2003)
12. Matsuzaki, K., Emoto, K.: Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 72–89. Springer, Heidelberg (2010)
13. Matsuzaki, K., Hu, Z., Takeichi, M.: Parallelization with tree skeletons. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 789–798. Springer, Heidelberg (2003)
14. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic Inversion Generates Divide-and-Conquer Parallel Programs. In: Conference on Programming Language Design and Implementation (PLDI), pp. 146–155. ACM Press (2007)
15. O'Sullivan, B., Stewart, D., Goerzen, J.: Real World Haskell. O'Reilly (2008)
16. Stewart, A., Clint, M., Gabarró, J.: Barrier synchronisation: Axiomatisation and relaxation. Formal Aspects of Computing 16(1), 36–50 (2004)
17. Tesson, J.: Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels. Ph.D. thesis, LIFO, University of Orléans (November 2011), `http://hal.archives-ouvertes.fr/tel-00660554/en/`
18. The Coq Development Team: The Coq Proof Assistant, `http://coq.inria.fr`
19. The SDPP Development Team: Systematic Development of Parallel Programs, `http://traclifo.univ-orleans.fr/SDPP`
20. Touati, S.A.A., Worms, J., Briais, S.: The Speedup Test. Tech. Rep. inria-00443839, INRIA Saclay - Ile de France (2010), `http://hal.inria.fr/inria-00443839`
21. Valiant, L.G.: A bridging model for parallel computation. Comm. of the ACM 33(8), 103 (1990)
22. Veldhuizen, T.: Techniques for Scientific C++. Computer science technical report 542, Indiana University (2000)
23. Zavanella, A.: The skel-BSP global optimizer: Enhancing performance portability in parallel programming. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 658–667. Springer, Heidelberg (2000)
24. Zhou, J., Chen, Y.: Generating C code from LOGS specifications. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 195–210. Springer, Heidelberg (2005)

Giraphx: Parallel Yet Serializable Large-Scale Graph Processing

Serafettin Tasci and Murat Demirbas

Computer Science & Engineering Department
University at Buffalo, SUNY

Abstract. Bulk Synchronous Parallelism (BSP) provides a good model for parallel processing of many large-scale graph applications, however it is unsuitable/inefficient for graph applications that require coordination, such as graph-coloring, subcoloring, and clustering. To address this problem, we present an efficient modification to the BSP model to implement serializability (sequential consistency) without reducing the highly-parallel nature of BSP. Our modification bypasses the message queues in BSP and reads directly from the worker's memory for the internal vertex executions. To ensure serializability, coordination is performed— implemented via dining philosophers or token ring— only for border vertices partitioned across workers. We implement our modifications to BSP on Giraph, an open-source clone of Google's Pregel. We show through a graph-coloring application that our modified framework, *Giraphx*, provides much better performance than implementing the application using dining-philosophers over Giraph. In fact, Giraphx outperforms Giraph even for embarrassingly parallel applications that do not require coordination, e.g., PageRank.

1 Introduction

Large-scale graph processing finds several applications in machine-learning [1], distributed simulations [2], web-search [3], and social-network analysis [4]. The significance of these applications led to the development of several graph processing frameworks recently. Due to the large size of the graphs considered, these frameworks employ a distributed/parallel execution model; Most adopt the Bulk Synchronous Parallel (BSP) [8] approach to this end. A popular example is the Pregel [3] framework from Google. Pregel inspired several open-source projects, including Apache Giraph [5], Hama [6], and Golden Orb [7], all of which use the BSP model. Although asynchronous graph processing frameworks such as GraphLab [1] and PowerGraph [16] were proposed recently, the BSP model is still used the most due to its simplicity, flexibility, and ease of use.

In the BSP approach to graph processing, the large input graph is partitioned to the worker machines, and each worker becomes responsible for execution of the vertices that are assigned to it. Then BSP's superstep concept is used for coordinating the parallel execution of the workers. A superstep consists of three parts. *Concurrent computation:* Concurrently every participating worker executes computations for the vertices they are responsible for. *Communication:*

The workers send messages on behalf of the vertices they are responsible for to their neighboring vertices. The neighboring vertices may or may not be in the same worker. *Barrier synchronization:* When a worker reaches this point (the barrier), it waits until all other workers have finished their communication actions, before the system as a whole can move to the next superstep. A computation involves many supersteps executed one after the other in this manner. So, in a superstep, the worker uses values communicated via messages from the previous superstep, instead of most recent values.

BSP provides good parallelism and yield. However, the *max-parallel* execution model used in BSP is not suitable for writing applications that require coordination between vertices. Consider the graph coloring problem, where the aim is to find a minimal coloring of the vertices such that no two adjacent vertices share the same color. A simple program is that, at each superstep a vertex picks the smallest color not used by any of its neighbors and adopts it as its color. If executed in a max-parallel manner, this program does not converge: If two neighboring vertices have the same color at any superstep, they loop on back and forth changing their colors to be the same. This max-parallel style concurrency violations can occur even when the worker has a single thread of control[1], because vertices in a worker communicate via message-passing in queues, and as such they operate on each other's previous round states. So, in effect, all vertices in a worker are executing concurrently for a superstep, even though in reality the vertices execute in a serial manner since the worker has a single thread.

For developing applications that require coordination, the *serializability* semantics [9] (also known as *interleaving* or *sequential consistency*) is better. Serializability ensures that for every parallel execution, there exists a sequential execution that produces an equivalent result. This model provides "in effect" the guarantee that any vertex computation is executed atomically (or in isolation) with respect to the rest of the system and this gives a cleaner abstraction to write graph programs.

The problem with serializability, however, is that it may throttle the parallelism/performance of the system, so how serializability is implemented matters. The straightforward implementation (executing all vertex computations sequentially in the graph and disallowing any parallel execution across workers) is, of course, very inefficient. It is easy to observe that if two vertices are not neighbors they can be executed in parallel, and since they do not share state, their executions do not conflict with each other, and they can be serialized (pretending as if one occurs before the other). One can implement this restriction using a dining philosopher program [10] to regulate that no two neighboring nodes execute at the same superstep. Running the application on top of dining philosophers in Giraph accomplishes serializability, but with a steep cost (as we show in our experimental results in Section 4).

Our Contributions. We present a simple extension to achieve serializability in BSP-based frameworks while keeping the highly-parallel and bulk-efficient

[1] This is the case in Giraph. Even when the worker is executed on a multicore machine, the worker executes as a single thread to keep concurrency issues manageable.

nature of BSP executions. We implement our extension on the open-source Apache Giraph framework, and call the extended framework *Giraphx*. To provide interworker serializability, we augmented Giraphx with two alternative coordination mechanisms: dining philosophers and token ring.

We give experimental results from Amazon Web Services (AWS) Elastic Compute Cloud (EC2) with upto 32 workers comparing the performance of Giraphx with that of Giraph. We show through a graph-coloring application that Giraphx consistently provides better performance than implementing the application using dining-philosophers over Giraph. Our experiments use mesh graphs and Google Web graphs, and show the effects of edge-locality in improved performance (in some cases upto an order of magnitude improvement) of Giraphx compared to Giraph. The results reveal that while dining-philosopher-based Giraphx performs better for large worker numbers, the token-ring-based Giraphx is superior for smaller clusters and low edge-locality situations.

Our experiments also show that Giraphx provides better performance than Giraph even for applications that are embarassingly parallel and do not require coordination. We show this through running a PageRank [11] application on both Giraph and Giraphx. The improved performance in Giraphx is due to the faster convergence it achieves by providing the vertices the ability to read the most recent data of other vertices in the serializability model.

Overview of our Method. In Giraphx, we categorize vertices as border vertices and internal vertices. If all the neighbors of a vertex are in the same worker as that vertex, then it is an *internal vertex*; else it is called a *border vertex*. In order to provide serializability, we modify Giraph to bypass the message queue and read directly from worker's memory for the internal vertex executions (we can have direct memory reads because the vertices are in the same worker). Since vertices read current values of neighbors (instead of using previous round values from the messages), interleaving execution, and hence atomic execution is achieved. In the above example, this modification solves the graph coloring problem easily and efficiently (without being hampered by running dining philosophers on vertices and slowing execution). When border vertices, partitioned across workers, are involved, additional synchronization is needed. For this, we use dining-philosopher or a worker-based token ring algorithm for synchronizing execution. Giraphx is much cheaper than running dining philosophers over Giraph because dining philosophers is run only on cross-worker edges of border vertices (which is generally a small fraction of all the vertices) in Giraphx, so the overhead comes only on this fraction not on the entire graph as in Giraph.

Outline of the Rest of the Paper. We discuss Giraph and dining philosophers implementation on Giraph in Section 2. In Section 3, we present Giraphx, and introduce our dining-philosopher-based Giraphx (d-Giraphx) and token-ring-based Giraphx (t-Giraphx). We present our experiment results from EC2 deployments in Section 4, and related work in Section 5.

2 Giraph

Giraph leverages Hadoop's MapReduce framework [12]. The master and all workers in Giraph are started as MapReduce worker tasks. [2] Hadoop's master-worker setup readily solves the monitoring/handling reliability of the workers, optimizing performance for communication, deploying distributed code, distributing data to the workers, and load-balancing.

Writing a Giraph program involves subclassing the BasicVertex class and overriding the Compute() method. Each worker goes over the graph vertices in its assigned partitions and runs Compute() for each active vertex once in every superstep. At each Compute operation, the vertex can read the messages in its incoming queue, perform computations to update its value, and send messages to its outgoing edges for evaluation in the next superstep. A Giraph program terminates when there are no messages in transit and all vertices vote to halt.

2.1 d-Giraph

While Giraph fits the bill for many graph-processing applications, it fails to provide a mechanism for applications where neighboring vertices need to coordinate their executions. Also, while the synchronous execution model in Giraph is easy to use, the inability to read the most recent data may lead to slow convergence. Consider the graph coloring problem. If two neighboring vertices have the same color at any superstep, they loop on back and forth changing their colors to be the same.

To solve this problem, we need to schedule the computation of vertices such that no conflicting vertices operate at the same superstep. For this purpose, we developed a serializable Giraph implementation called *d-Giraph*, that ensures that in each neighborhood only one vertex can compute at a superstep while others have to wait for their turn. d-Giraph uses the hygienic dining philosophers algorithm for vertex coordination [10]. The basic d-Giraph operation consists of the following steps:

1. At superstep 0, every vertex sends a message containing its id to all outgoing edges so that at superstep 1 vertices will also learn their incoming edges.
2. At superstep 1, every vertex sends its randomly generated initial fork acquisition priority to all edges together with its vertex value for initial distribution of forks in a deadlock-free manner.
3. Computation starts at superstep 2 in which every vertex gathers its initial forks and learns initial values of neighbors.
4. Then each vertex checks if it has all its forks. If so, it performs vertex computation, otherwise it executes a skip (state is not changed).
5. Each vertex replies incoming fork request messages, and then sends request messages for its missing forks. New vertex value is sent only if it is updated.

[2] Giraph does not have a reduce phase: It uses only the map phase of MapReduce and this single map phase runs until all supersteps are completed.

Despite achieving serializability, d-Giraph hurts parallelism significantly by allowing only a small fraction of all vertices to operate at each superstep; a steep price to pay for serializability.

3 Giraphx

In order to provide efficient serializability, in Giraphx we modify Giraph to bypass the message queue and read directly from worker's memory for the internal vertex executions (we can have direct memory reads because the vertices are in the same worker). Since vertices read current values of other vertices' variables (instead of using previous round values from the messages), interleaving execution, and hence atomic execution is achieved. When border vertices, partitioned across workers, are involved, additional synchronization is needed. A border vertex cannot make direct memory reads for its interworker edges and blocking remote reads are costly. In this case, we revert to the BSP model and use messaging in tandem with a coordination mechanism for these border edges. We propose two such mechanisms: a dining philosopher based solution as in Section 2.1 called *d-Giraphx* and a simpler token-based solution called *t-Giraphx*.

The only side effect of these modifications to Giraph semantics is the increase in update frequency of internal vertices compared to border vertices. However this difference causes no complications in most graph problems.

To ease migration from Giraph, Giraphx closely follows the Giraph API except small modifications in handling of intra-worker communications. To implement Giraphx, we added approximately 800 lines of Java code to Giraph (including the coordination mechanisms). While T-Giraphx has no memory overhead over Giraph, d-Giraphx uses $\sim 30\%$ more memory primarily for synchronization messages (i.e. fork exchange) in dining philosophers.

3.1 d-Giraphx

d-Giraphx uses dining philosophers for establishing coordination of the interworker edges in Giraphx. To implement d-Giraphx, d-Giraph is modified as follows:

1. Each vertex prepares a list that denotes the locality information about the neighbors. If all neighbors are local then the vertex marks itself as *internal*, else as *border*.
2. If a vertex is internal, it operates at each superstep. If it is a border vertex, it checks whether it has gathered all forks or not. If yes, it first iterates over local neighbor list and reads their values and then iterates over its incoming messages to learn the values of its nonlocal neighbors.
3. Border vertices are also responsible for sending and replying fork exchange messages while internal vertices skip it.

Since the amount of messaging in d-Giraphx is proportional to the number of interworker edges and border vertices, partitioning plays an important role in improving the performance of the system. In a smartly-partitioned graph, since the border nodes is a small fraction of the internal nodes, d-Giraphx performs upto an order of magnitude better than d-Giraph as we show in Section 4.

3.2 t-Giraphx

Despite d-Giraphx is an obvious improvement over d-Giraph, it suffers from large coordination overhead when the number of border nodes is large (i.e., the edge-locality is low). To address these low edge-locality situations for which any smart-partitioning of the input graph does not provide much benefit, a solution which avoids coordination messages would be preferred. For this purpose, we implement a token-based version of Giraphx called t-Giraphx.

In t-Graphx, coordination is done at the worker level instead of at the vertex level. At each superstep one of the workers has the token for computation. When a worker acquires the token, it runs Compute() on all its vertices whether they are border or internal. Similar to d-Giraphx, vertices use messaging for inter-worker neighbors and direct memory reads for same-worker neighbors. When a worker does not have the token, it can only operate on internal vertices. In a non-token worker, the border vertices skip computation in this superstep, but they still broadcast their state to neighboring vertices in other workers by sending a message. Since t-Giraphx does not have any fork exchange overhead, it uses less messaging and thus converges faster than d-Giraphx, whenever the number of workers is sufficiently small.

t-Giraphx does not scale as the number of workers increase. Consider a graph problem where convergence is acquired after k iterations at each vertex. In a graph with average vertex degree w, d-Giraphx processes all vertices once in approximately every w supersteps independent of worker number N, and thus converges in at most $w * k$ supersteps independent of N. On the other hand, t-Giraphx will need $N * k$ supersteps, resulting in longer completion times when $N >> w$, and shorter completion times when $N << w$.

Some problems, such as graph subcoloring[3], require multi-hop coordination. While dining philosophers achieves 1-hop neighborhood coordination, it can be extended to provide multi-hop neighborhood coordination by defining the multi-hop neighbors as "virtual" neighbors. For example, for 2-hop coordination, this can be done by adding another superstep in d-Giraphx after superstep 0 in which vertices send a message containing the ids of all neighbors to every neighbor. This way, in the following superstep all vertices will have a list of vertices in its 2-hop neighborhood and then can run dining philosophers on this list. The primary drawback of these multihop coordination extensions is the significant increase in the number of edges per vertex, and the resulting increase in the total running time of the protocol. In applications that require multihop coordination, since w will grow exponentially, t-Giraphx becomes a better alternative to d-Giraphx.

[3] In graph subcoloring, the aim is to assign colors to a graph's vertices such that each color class induces a vertex disjoint union of cliques. This requires 2-hop neighborhood coordination.

4 Experiments

To evaluate the performance of Giraphx, we conducted experiments on EC2 using medium Linux instances, where each instance has two EC2 compute units and 3.75 GB of RAM. In our experiments, we used up to 33 instances, where one instance is designated as the master, and the remaining 32 instances are workers. We used two network topologies for our experiments. First is a planar mesh network where each node is connected to its 4 neighbors in the mesh (right, left, top, bottom) plus one of the diagonal neighbors (e.g. right-top). The second is Google's web graph dataset [14] in which vertices represent web pages and directed edges represent hyperlinks between them. The Google web graph dataset consists of approximately 900,000 vertices and 5 million edges. This dataset is quite challenging since it has a highly skewed degree distribution where some vertices have degrees up to 6500.

We used three partitioners in our experiments: a hash partitioner (which assigns vertices to workers pseudo-randomly), a mesh partitioner (which assigns each worker a neighborhood in the mesh), and the metis partitioner (which smartly-partitions a graph to provide high edge-locality) [13]. For Giraphx, the cost of learning internal versus border vertices are included in the provided experiment results. In all experiments, the partitioning run times are included in the results.

4.1 Graph-Coloring Experiments on Mesh Graph

We first fix the number of workers as 16, and perform experiments with increasing the number of vertices in a mesh graph. We use a mesh-partitioner so the percentage of local vertices in these experiments stays between 94%–99%. Figure 1 demonstrates that as the size of the graph increases the running time of d-Giraph increases at a much faster pace than Giraphx-based methods. The basic reason is the lack of d-Giraph's ability to make use of the locality in the graph. Every vertex needs to coordinate with all neighbors causing large delays. While the local nodes in Giraphx can compute at every superstep, in d-Giraph vertices have to wait until all forks are acquired to make computation. In contrast, d-Giraphx avoids a significant fraction of the messaging in d-Giraph by incorporating local memory reads for internal vertices.

Figure 1(b)) shows that increase in the graph size does not necessarily increase the number of supersteps for the three frameworks compared. Since the graph topology does not change, average vertex degree and hence the number of supersteps is stable as the graph size changes in d-Giraph and d-Giraphx. Since worker number does not change, superstep number in t-Giraphx is also stable. We also see that t-Giraphx takes more supersteps than the other methods, since it takes 16 supersteps for the token to return to a worker. However, t-Giraphx compensates this disadvantage and converges in less time than d-Giraph by avoiding the coordination delays that dining-philosophers induce for d-Giraph.

Next in Figure 2 we fix the number of vertices as 1 million, and increase the number of workers to observe the effects of parallelism on the runtime. As long

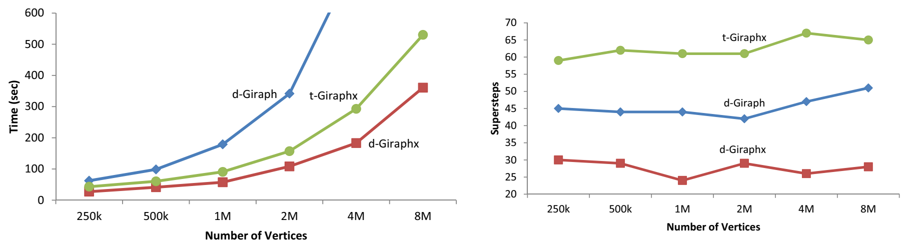


Fig. 1. Change in time and superstep number as the graph size increases for 16 workers

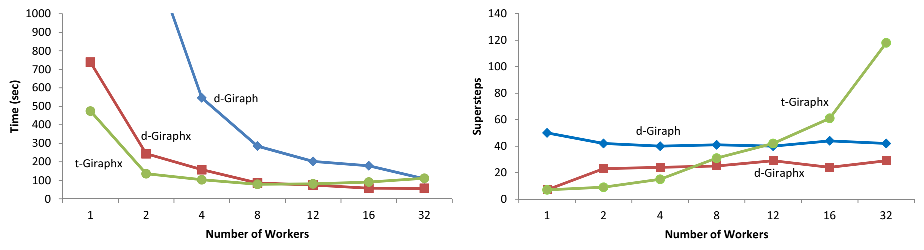


Fig. 2. Change in time and superstep number as the amount of worker machines increases for 1 million vertices

as the edge-locality is high, d-Giraph and d-Giraphx take advantage of every worker added since the load per worker decreases accordingly. t-Giraphx also benefits from the increase in computation power. But after 8 workers, adding more workers hurts t-Giraphx due to the increase in the superstep numbers proportional to the number of workers.

Finally, in Figure 3, we ran an experiment where we use the hash partitioner on the mesh network to show the effects of partitioning. While the mesh partitioner provides an optimal partitioning in a mesh network, using a hash partitioner causes vertices to lose their locality and most of them become border vertices. As a result, d-Giraphx loses the performance improvement it gains from internal vertices, and performs only slightly better than d-Giraph. The performance of t-Giraphx is not affected too much, since it does not suffer from the increased coordination cost on border vertices.

4.2 Graph-Coloring Experiments on Google Web Graph

To reveal the performance of the proposed frameworks on a real-world graph, we ran another set of experiments on the challenging Google web graph. This graph has a highly skewed degree distribution and it is hard to find a partitioning that will provide good edge-locality. These high-degree vertices cause communication and storage imbalance on vertices as well as imperfect partitioning. In this graph, using the hash partitioner 99% of all vertices become border

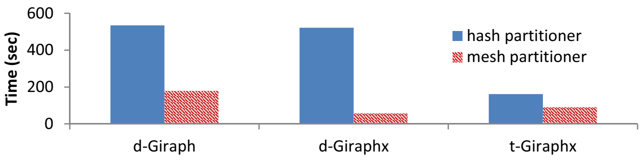


Fig. 3. Computation times with hash versus mesh partitioners for 16 workers and 1M vertices

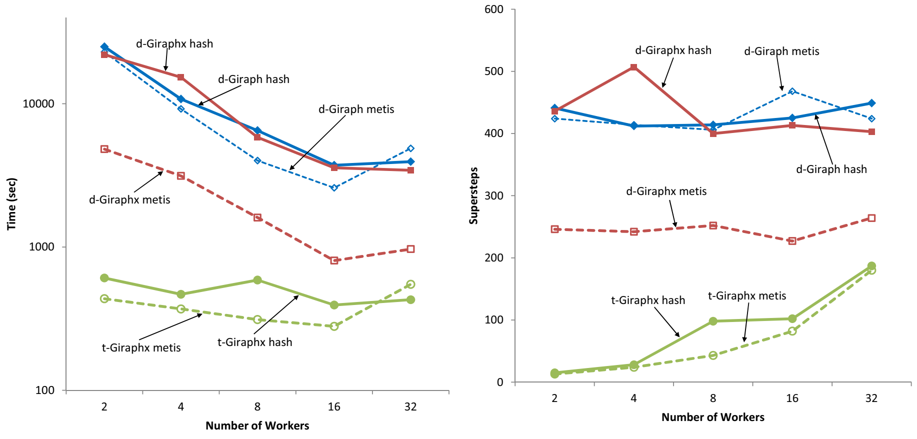


Fig. 4. Results with hash versus metis partitioner on the web graph. Solid lines indicate hash partitioner, and dashed lines metis partitioner. Time is given in log scale.

vertices, and this hurts parallelism immensely. To prevent the loss of parallelism and improve the locality, we preprocessed the graph using the metis partitioner. When metis is used, ratio of border vertices drops to around 6%. However these vertices are mostly high-degree hubs which have a lot of interworker neighbors. Therefore even with metis, webgraph still performs worse than the same-sized mesh-partitioned mesh network.

Figure 4(a) shows a comparison of the three frameworks on the Google web graph as the number of workers increase. Regardless of whether metis is used or not, t-Giraphx always performs better than other methods. This is because t-Giraphx has a predictable number of supersteps independent from the degree distribution while the number of high-degree vertices adversely affect the number of supersteps in d-Giraph and d-Giraphx (see Figure 4(b)).[4] The skewed degree distribution leads to a star topology centered on high-degree vertices resulting in larger fork re-acquisition intervals for d-Giraph and also for d-Giraphx to a

[4] In the mesh graph where the maximum degree is 5, the number of supersteps required for all vertices to compute at least once is around 10. In comparison, in the web graph where the maximum degree is 6000 the number of supersteps jumps to 50 for metis partitioning and 70 for hash partitioning.

degree. d-Giraphx improves with better partitioning in terms of both time and supersteps, because even the highest degree vertices now have a much higher number of local neighbors decreasing the size of the subgraph on which coordination through fork exchange is required.

An interesting result is the increase in runtime as the number of workers exceeds 16 in metis partitioning. In t-Giraphx, the increase in the number of supersteps is the dominant reason for this. In case of d-Giraph and d-Giraphx, after 16 workers the computation power gained by adding new workers is overshadowed by the loss of locality.

4.3 Pagerank Experiments on Google Web Graph

Giraphx also provides performance improvements for graph applications that do not require coordination among vertices. To demonstrate this, we modified the PageRank implementation in the Giraph framework, and ran it on Giraphx to compare their performances. PageRank computes the importance of the vertices/webpages iteratively using a weighted sum of neighbor values, and does not require serializability among vertex computations.

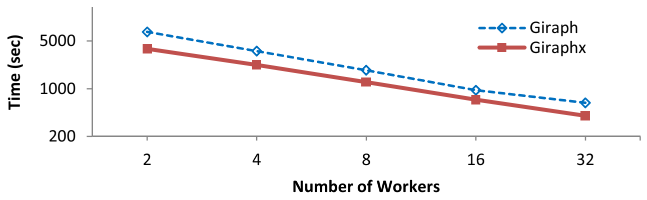


Fig. 5. Comparison of Giraph and Giraphx on Pagerank application (log scale)

In this experiment we used the Google web graph with metis partitioning. In our implementation, vertices vote for termination when $\Delta < 10^{-9}$ where Δ is the total change in vertex values from the previous superstep. Figure 5 shows that Giraphx finishes computation %35 faster than Giraph, for all worker numbers. The improved performance in Giraphx is due to the faster convergence it achieves. The experiment logs reveal that convergence takes just 64 supersteps in Giraphx compared to 117 supersteps in Giraph.

In Giraphx, the internal vertices take advantage of the direct memory reading feature, and read the most recent data of other vertices, which leads to faster convergence times. In fact, the same argument is also cited as the main advantage of asynchronous systems (e.g. GraphLab [1]) over the BSP model as we discuss in Section 5. Giraphx can provide faster convergence as in asynchronous systems without sacrificing the convenience of the BSP model for application development and debugging.

5 Related Work

The alternative approach to synchronous BSP-based systems is to use an asynchronous system which updates the vertices immediately and therefore uses the most recent data at any point in computation. In addition, asynchronous systems avoid the barrier synchronization cost. However asynchrony brings along programming complexity and may require consideration of consistency issues.

Distributed GraphLab [1] is a well-known powerful and flexible asynchronous framework. Asynchrony helps faster convergence and vertex prioritization but it requires selection of a consistency model to maintain correct execution in different problems. In addition, unlike GiraphX, it does not allow dynamic modifications to graph structure. PowerGraph [16] proposes a unified abstraction called Gather-Apply-Scatter which can simulate both asynchronous and synchronous systems by factoring vertex-programs over edges. This factorization helps reduction of communication cost and computation imbalance in power-law graphs. However, it has the same shortcomings as GraphLab. GRACE [15] also provides a parallel execution engine which allows usage of both execution policies.

Since partitioning plays an important role in efficient placement of graph data over cluster nodes, some studies focus on partitioning the graph data. A recent work [18] shows that SPARQL queries can be processed up to 1000 times faster on a Hadoop cluster by using a clever partitioning, custom data replication and an efficient data store optimized for graph data. A bandwidth aware graph partitioning framework to minimize the network traffic in partitioning and processing is proposed in [19]. Finally, another recent work [20] shows that using simple partitioning heuristics can bring a significant performance improvement that surpasses the widely-used offline metis partitioner.

6 Conclusion

In this paper, we proposed efficient methods to bring serialization to the BSP model without changing its highly-parallel nature and clean semantics. We showed how dining philosophers and token ring can be used for achieving coordination between cross-worker vertices. We alleviated the cost of these coordination mechanisms by enabling direct memory reads for intraworker vertex communication.

We implemented Giraphx on top of Apache Giraph and evaluated it on two applications using real and synthetic graphs. We showed through a greedy graph coloring application that Giraphx achieves serialized execution in BSP model with consistently better performances than Giraph. Our experiment results showed that while d-Giraphx performs better for large worker numbers, t-Giraphx performs better for small worker numbers and low edge-locality situations. Finally we showed that, due to the faster convergence it provides, Giraphx outperforms Giraph even for embarrassingly parallel applications that do not require coordination, such as PageRank.

References

1. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. 5(8), 716–727 (2012)
2. Braun, S.A.: A cloud-resolving simulation of hurricane bob (1991): Storm structure and eyewall buoyancy. Mon. Wea. Rev. 130(6), 1573–1592 (2002)
3. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 135–146. ACM, New York (2010)
4. `http://www.facebook.com/about/graphsearch/`
5. `http://incubator.apache.org/giraph/`
6. `http://hama.apache.org/`
7. `http://goldenorbos.org/`
8. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM 33(8), 103–111 (1990)
9. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers (1993)
10. Chandy, K.M., Misra, J.: The drinking philosopher's problem. ACM Trans. Program. Lang. Syst. 6(4), 632–646 (1984)
11. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the Seventh International Conference on World Wide Web 7, WWW7, pp. 107–117 (1998)
12. `http://hadoop.apache.org/`
13. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20(1), 359 (1999)
14. `http://snap.stanford.edu/data/web-Google.html/`
15. Wang, G., Xie, W., Demers, A., Gehrke, J.: Asynchronous large-scale graph processing made easy. In: Proceedings of CIDR 2013 (2013)
16. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood (October 2012)
17. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood (October 2012)
18. Huang, J., Abadi, D.J., Ren, K.: Scalable sparql querying of large rdf graphs. PVLDB 4(11), 1123–1134 (2011)
19. Chen, R., Yang, M., Weng, X., Choi, B., He, B., Li, X.: Improving large graph processing on partitioned graphs in the cloud. In: Proceedings of the Third ACM Symposium on Cloud Computing, SoCC 2012, pp. 3:1–3:13. ACM, New York (2012)
20. Stanton, I., Kliot, G.: Streaming graph partitioning for large distributed graphs. In: Yang, Q., Agarwal, D., Pei, J. (eds.) KDD, pp. 1222–1230. ACM (2012)

Hugh: A Semantically Aware Universal Construction for Transactional Memory Systems

Craig Sharp and Graham Morgan

School of Computing Science, Newcastle University
{craig.sharp,graham.morgan}@ncl.ac.uk

Abstract. In this paper we describe an implementation for exploring the scheduling of aborted transactions within transactional memory systems. We consider application semantics to be just as important as guaranteeing linearizability in arriving at an appropriate execution strategy. Our approach exploits parallelism to simultaneously create different execution orderings for rescheduled aborted transactions and chooses the most beneficial for application progression. The overall solution guarantees a lock-free universal construction if there exists at least one transaction that can commit. The appropriateness of our approach is demonstrated via micro-benchmark performance figures.

Keywords: Transactional Memory, Contention Management, Shared Memory, Concurrency Control, STM.

1 Introduction

Given the current barriers to processor frequency scaling, processor manufacturers have focused developments on parallel scaling of processing, in what has become known as the parallel revolution [1]. Unfortunately, writing concurrent software and solving problems in parallel is notoriously difficult for a wide class of problems (particularly in the area of system design). The key to exploiting parallelism effectively lies in the concurrency control mechanism used to provide correctness and progress guarantees to the concurrent program.

Transactional Memory has offered programmers a technique which simplifies the implementation of concurrency control. Atomic blocks allow sections of concurrent code to be composed, in a manner which is trivial in comparison to locking-based approaches. The problem with Transactional Memory lies in the occurrence of conflicts which require transactions to abort and retry their execution. If conflicts occur regularly and persistently, the Transaction Manager requires a Contention Management Policy (CMP) to mitigate the degradation of performance caused by aborted transactions.

From the programmer's perspective, conflicts fall into two categories: concurrent conflicts and semantic conflicts. A concurrent conflict occurs when the reads and writes of a transaction encounter an inconsistent state of shared memory and many contention managers combat these types of conflicts. A transaction may execute without interference however and still need to re-execute because semantically, the application cannot progress. For example, a transaction may need to consume an item from a shared buffer but finds it empty, or a bank account may have insufficient funds to permit a withdrawal.

Typically, a 'semantic conflict' can be dealt with in the application by (i) letting the transaction commit and re-execute in the future, (ii) or by using primitives (*retry*, *orElse* etc) as provided by Harris et al [2] which essentially allow ad-hoc coordination of transaction execution. We believe that the former approach is detrimental for application progression, raising the possibility of needless future conflicts when transactions re-execute. Meanwhile, the use of primitives places a burden on the application developer that must be addressed with an ad hoc solution, (re-introducing a fundamental problem of coordination with pessimistic concurrency control, which Software Transactional Memory originally sought to address).

In this paper we present an implementation of a Universal Construction approach to Contention Management called *Hugh*, that tackles both concurrent and semantic conflicts in an atomic object based STM model. We describe a speculative technique which serializes conflicting transactions to resolve concurrent conflicts and a parallel exploration which tackles semantic conflicts. Within the scope of this paper we consider a semantic conflict as simply the intentional abortion of a transaction by its own thread, and assume such conflicts can be avoided by executing the transaction in an alternative schedule.

The remainder of the paper is organized as follows: Section 2 describes the implementation. Section 3 describes the related work and Section 4 provides an evaluation of results obtained from an implementation of our technique. Section 5 concludes the paper and describes possible avenues for future work.

2 Implementation

The concept of the Universal Construction (hereafter UC) was first proposed by Herlihy [3] and allows any sequential data structure to be transformed into a linearizable representation that can be accessed and updated by n threads. There are three phases of UC operation: (i) threads prepare and announce a proposed input to add to the UC, (ii) each announcing thread performs consensus to decide which input will be added and (iii) a log of inputs is updated by the winning thread to reflect its input. We begin with an overview of how we use the UC technique and then provide greater detail in the remainder of this section.

2.1 Overview

We use the UC technique to provide conflict resolution and therefore the threads which use our implementation consist of the threads of aborted transactions. *Hugh* accepts as input a permutation of one or more sequentially executed transactions and decides which permutation will be added to the log.

When some $thread_a$ encounters a conflict it prepares its input to the UC by first adding its aborted transaction to a global *Transaction Table*; after this

Registration Phase the parameters of $thread_a$'s permutation are set. The thread then enters a *Speculative Phase* where it re-executes aborted transactions that have been added to the *Transaction Table*. We provide $thread_a$ with a private cache to hold copies of modified atomic objects, but no transaction is committed. Transactions are executed sequentially to prevent concurrent interference, but application semantics may still cause a transaction to abort explicitly (i.e. a semantic conflict).

During the *Speculative Phase* of $thread_a$, other threads may execute their own speculative transactions in parallel with $thread_a$. Once the *Speculative Phase* ends, each participating thread then enters a *Commit Phase* to decide which single thread's cache of modified atomic objects will be committed using a consensus algorithm. Threads whose transactions are committed return to normal execution, while those that remain aborted commence another *Registration Phase*.

Figure 1 contrasts our approach with a serializing CMP (like [4] for example). Two hypothetical scenarios, both containing a *depositor* and *withdrawer* transaction access a shared object. In scenario 1, the CMP reorders transactions to avoid concurrent conflicts. Although the *withdrawer* transaction can commit, it may need to re-execute in future (if deposits must precede withdrawals for example). In scenario 2, our approach is illustrated where a semantic abort occurs and each thread re-executes a different permutation of the aborted transactions.

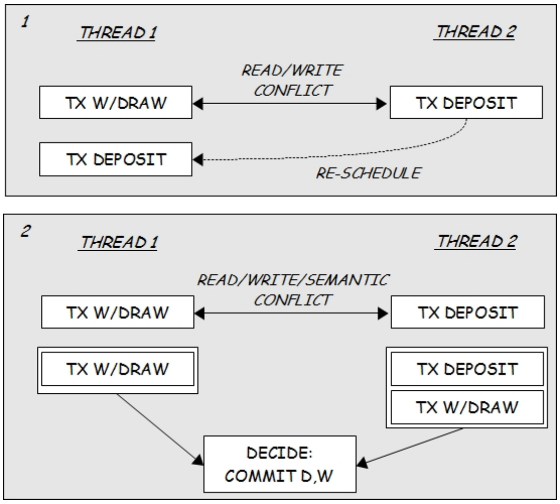


Fig. 1. In scenario 1 a read/write conflict has occurred between two transactions called withdraw (w/draw) and deposit. The depositor is aborted and rescheduled to execute after the withdrawer has committed. In scenario 2, Thread 1 aborts the depositor but then also aborts because of a semantic conflict caused by attempting to execute a withdrawal before a deposit. The conflict is resolved by the execution of an ordering which allows both to commit (deposit then withdraw).

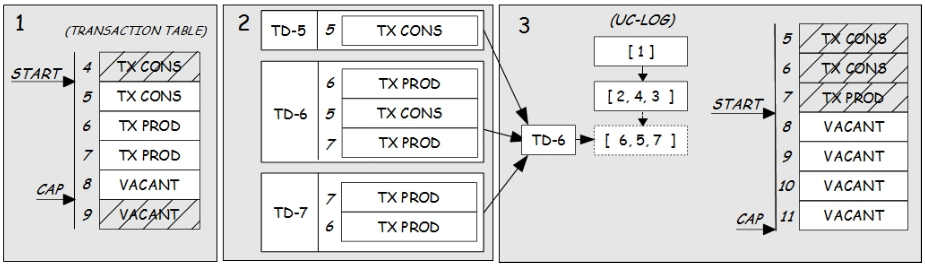


Fig. 2. In phase 1, threads add their transactions to the Transaction Table. In phase 2, a thread executes permutations of transactions within the window of the Transaction Table. In phase 3, transactions perform consensus to decide which permutation will be committed, and the result is added to the log of the Universal Construction and the Transaction Table window is advanced.

2.2 Aims and Contribution

Serializing aborted transactions to avoid concurrent conflicts has already been explored [5,6,4] given that under high contention, serialization can produce better throughput than a parallel approach. To our knowledge however, *Hugh* is the first to use additional threads to provide multiple serialized executions of aborted transactions in parallel and does not require the overhead of a thread-pool.

We combine both *direct-update* and *deferred-update* approaches in our implementation; until a transaction aborts, threads modify atomic objects directly but when transactions are retried, thread-private caches hold updates to accessed objects (as in the deferred-update model). While the use of direct-updates (sometimes called encounter time locking) is not a requirement of our approach, this technique was preferred having been shown to reduce the degree of wasted transaction execution [7]. Although the use of thread-private caching increases memory usage, it is hoped that this can reduce the occurrence of 'cache bouncing' (this approach was found to be particularly effective in Remote Core Locking [8]).

While the serial execution of aborted transactions tackles the problem of concurrent interference, we address semantic conflict by requiring that each thread execute different permutations of aborted transactions:

1. The possibility of having to re-execute a transaction because of a semantic conflict can be minimized given that there is a greater chance some permutation will avoid the semantic conflict;
2. If more transactions are aborted and conflicts are high, then this results in a greater number of threads available to explore more permutations of transaction execution. This in turn increases the possibility of finding an optimal transaction ordering so that more aborted transactions can be committed;
3. If parallel processing resources are increased, then a greater number of permutations can be explored in parallel, theoretically increasing the exploratory capacity of our approach within a shorter time-frame. Concurrency control is essentially transformed into a parallel state-space exploration problem.

List point 3 suggests our approach would benefit from a policy controlling the operating system scheduling of threads to processors and a platform with a plentiful supply of cores. This is beyond the scope of this paper and we confine our discussion to a purely 'user-level' implementation. We now describe each of the three phases of conflict resolution in detail.

2.3 Registration Phase

In order to re-execute its transaction, a thread is first required to register its transaction within a global table which we call the *Transaction Table*. When the transaction is added to this table it then belongs to the set of transactions from which permutations may be generated during the *Speculative Phase*. To avoid concurrency errors, threads use the synchronization primitive *compare-and-swap* to atomically increment an integer variable (called *current*) and thus gain a unique entry into the table.

Figure 2(1) shows the layout of the *Transaction Table*. In addition to *current*, the *Transaction Table* also maintains two integer variables called *start* and *cap*, which provide a 'window' (of size *cap* minus *start*) within which a thread's transactions must lie before it may execute its *Speculative Phase*. The window is the maximum length of the permutation the thread submits to the consensus algorithm. The first index of the permutation is equal to the threads index in the *Transaction Table* and subsequent entries are indices in the *Transaction Table* within the range of the window. For example, suppose some $thread_6$ registers and takes the $6th$ entry into the *Transaction Table* and $start = 5$ while $cap = 9$, then a valid permutation for $thread_5$ is $\{6, 5, 7, 8\}$ (see Figure 2(2)).

During the commit phase, the window is 'advanced' to allow new threads to begin their *Speculative Phase* (Figure 2(3)). Note that increasing the size of the window increases the maximum number of transactions that may commit in a single commit phase (throughput) but also incurs extra computational overhead, including the computation of consensus (the maximum number of participants in the consensus algorithm is equal to the window size). While our own experiments found that a maximum window size of 16 produced the best performance, an attractive avenue for future work would be to expand and contract the size of the window at runtime, based on the level of contention.

2.4 Speculative Phase

Once a thread has registered its aborted transaction, it commences its *Speculative Phase* (for brevity, we shall hereafter refer to these threads as *speculators*). The speculator executes transactions held in the *Transaction Table* with the aim of executing as many transactions to completion as possible. While the speculator is executing, new speculators may register (causing newly aborted transactions to appear in the *Transaction Table*) and begin their own speculative execution. All speculators must ensure two conditions are met: (i) exclusivity of atomic objects to ensure any speculative execution is sequentially *consistent* and (ii) the *Speculative Phase* must *terminate*.

Consistency. While speculators modify private copies of atomic objects, they must ensure that no active (non-aborted) thread modifies the original, otherwise their execution would be inconsistent and could not commit. Speculators must therefore have exclusive access to any atomic object they update, (as they do not update the objects directly, they do not require exclusivity from other speculators). We require that each atomic object has an owner field and that active transactions have to install themselves as owner of any atomic object they wish to modify. To support exclusivity, each atomic object also possesses an integer field denoting its version (*version*) and a reference to a global clock (*clock*). In addition, we provide a global transaction (*spec*) to denote that an object is currently owned by a speculator. The procedures for accessing atomic objects are:

– The first time an atomic object is accessed by a speculator, it checks whether the object is owned by another speculator ($owner = spec$). If true, the thread caches a copy of the object and continues its transaction (subsequent accesses modify the copy);
– If the object is not owned by a speculator and ($version <= clock$), it sets ($version = clock+1$), aborts the current owner of the object, and installs the *spec* transaction as the new owner. Setting the value of *version* eliminates the possibility that another thread can repeatedly prevent the speculator from changing the owner of an atomic object;
– Once consensus has been reached and the winning transactions have been committed, *clock* is atomically incremented so that ($version \leq clock$) is true, and any thread may once again own the object.

Before a thread executing an active transaction tries to install itself as the owner of any atomic object, it first checks whether $version \leq clock$. If this evaluates to false, then the thread knows it must abort because the object is currently being modified by a speculator. The thread will now register and become a speculator itself.

Terminating Speculation. Transactions are executed according to the indices of the speculator's permutation until either: (i) a transaction aborts, (ii) all transaction at the indices in the permutation have been executed, (iii) or the next index in the *Transaction Table* does not contain a transaction. The maximum number of transactions any speculator may execute during a single *Speculative Phase* is equal to the the size of the window. As each speculator executes a unique permutation, then for n speculators this means that a maximum of n permutations may be executed during a single session. Each speculator records the indices of the transactions it has executed successfully. Once a speculator has finished its *Speculative Phase*, it moves onto the *Commit Phase*.

2.5 Commit Phase

Once each speculator has completed its *Speculative Phase*, the log of the UC must be updated with the permutation of transactions the speculator has executed. To accomplish this:

1. Each speculator submits its permutation of executed transactions to the *decide* method of a consensus protocol;
2. The winner is decided by the permutation with the greatest number of executed transactions. The winning speculator commits the changes to the atomic objects in its cache and appends the permutation to the log (a linked list) provided by the UC.

The log provides each speculator with the necessary information to determine whether its own transaction has been successfully committed. Each speculator searches for its allocated index into the *Transaction Table* within the winning permutation appended to the log. If the permutation contains a speculator's index, that speculator's transaction has been committed. Once the winning speculator has committed the atomic objects in its cache, it atomically increments the global clock (such that $version \leq clock$ is true), indicating that any thread may once again own any atomic object. The window in the *Transaction Table* is then advanced by the winning speculator.

3 Related Work

Implementing an optimum Contention Management Policy (CMP) itself is a non-trivial task and numerous approaches have been developed. The first CMP techniques can be categorized by their employment of a wait-based criteria [9] (such as *Greedy*, *Karma*, *Polka* etc). These approaches were relatively trivial to integrate with existing STMs, requiring no involvement from the scheduling mechanism of the platform. Heber et al [10] observed an inefficiency with wait-based approaches given the difficulty in ascertaining the duration that an aborted transaction should wait before re-executing; too short could cause a repeat conflict and too long would be inefficient.

Serializing Contention Managers attempt to improve the inefficiency of wait-based CMPs by rescheduling aborted transactions to execute after a conflicting transaction and *Hugh* loosely follows this approach. Bai et al [5] introduced an approach which used 'keys' to predict the likelihood of conflicts between transactions; such transactions could then be scheduled to execute in sequence to avoid the possibility of concurrent conflict. Dolev et al introduced CAR-STM [6] which like Bai's work predicts the likelihood of conflicts between transactions and executes those transactions serially. Ansari et al developed an approach called Steal on Abort [4] where transactions could be 'stolen away' from threads with high workloads and work sharing between transaction executing threads was facilitated. *Hugh* differs from these serialising techniques by considering the effects of semantic conflicts and executing multiple transaction orderings in parallel.

Adaptive CMPs have also been developed, Yoo and Lee for example, introduced ATS [11]; a serializing CMP which used a threshold value to dynamically determine when aborted transactions should be serialized, based on a measure called the *contention intensity* and Heber et al provided *CBench*, a useful benchmark for evaluating serializing CMPs [10]. Heber et al identified a phenomenon they call mode oscillations, (where performance is hurt because the Contention Manager repeatedly switches between serialization and parallel execution) and implemented a stabilization algorithm to address the problem. Although adaptation has not been explored in our approach, potential future work may involve varying the window size with respect to contention levels.

Like *Hugh*, several contributions have approached transaction memory in the context of building a UC. Wamhoff [12] and Chuong [13] demonstrated how transactions could be used with a UC to handle failure. More recently Crain et al [14] developed a UC which in theory could remove the need for programmers to observe aborts. Unlike previous UC approaches, *Hugh* uses the UC for contention management only and submits multiple transactions as input to the consensus algorithm.

4 Evaluation

In this section we present results from a set of micro-benchmarks performed on an implementation of our system. The tests were executed on a Dell Alienware desktop PC featuring 4 x dual-core 3.40GHz Intel(R) processors (i7-2600) with 16GB of RAM, running Windows 7. The Transactional Memory software was executed in Visual Studio 2010 with a C Sharp implementation of the Java DSTM2 benchmark suite [15] (using the obstruction free factory with visible reads). Each experiment is carried out using an increasing number of threads (from 2 to 12) and executed 10 times with the average results provided. The Polka Contention Management Policy [16] has been cited as providing the best performance of wait-based Contention Managers, and so this was used to provide a comparison with our implementation (using the default parameters with respect to back-off time).

Two benchmarks were used to test the performance of our implementation: a linked list and a hash table. In both benchmarks, threads are divided into 'producers' and 'consumers' in equal number. Producers and consumers take a random value and attempt to *insert* this into the data structure in the case of the producer, or *remove* it in the case of the consumer. The highest frequency of read/write conflicts is expected in the linked list benchmark compared to the hash table which distributes items in an array of linked lists based on hashes generated from each item.

Performance results under increasing levels of semantic conflicts are provided. When there are no semantic conflicts (labelled S-L0), then threads only abort transactions if there is a read/write conflict. With Level 1 semantic conflicts (S-L1), consumer threads explicitly abort their transaction if they attempt

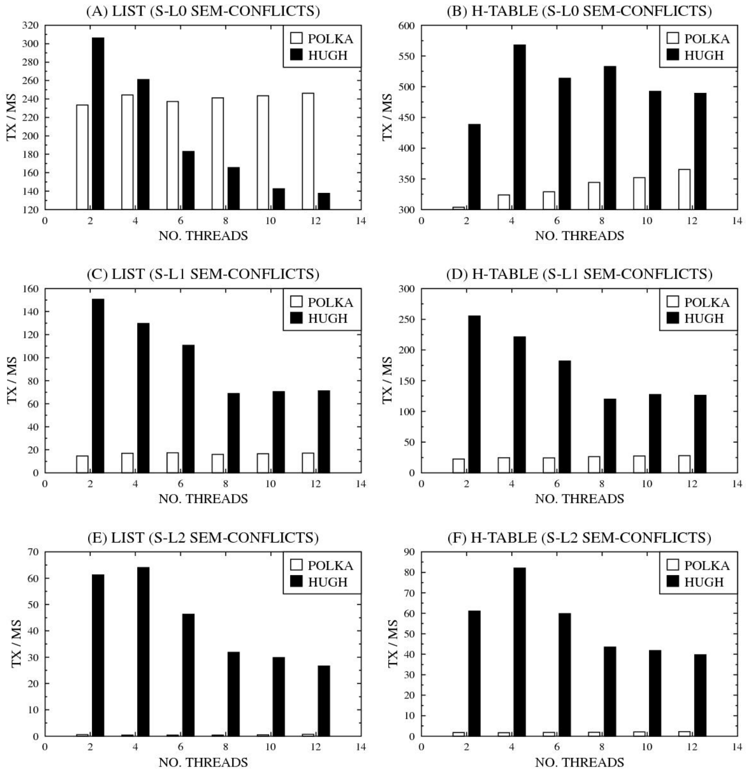


Fig. 3. Transaction Throughput

to remove an item which is not already present in the data-structure. Using Level 2 semantic conflicts (S-L2) also causes producers to abort their transactions if they attempt to add an item to a data-structure which is already present.

4.1 Transaction Throughput

Figure 3 illustrates the results for transaction throughput. The Y-axis denotes the number of transactions committed per millisecond and X-axis shows the number of threads present. In Graph A, using the list benchmark with S-L0 semantic conflicts we can see that the Polka manager performs better than *Hugh* once the number of threads increases beyond 6 due to the increase in read/write conflicts. One possible explanation is that the serialization of aborted transactions used by *Hugh* is less effective in this situation than the Polka policy and the lack of semantic conflicts means there is little benefit from the execution of permutations. In Graph B where the hash table reduces the number of read/write conflicts we see better performance under both policies, though the greatest increase in throughput is witnessed with *Hugh*.

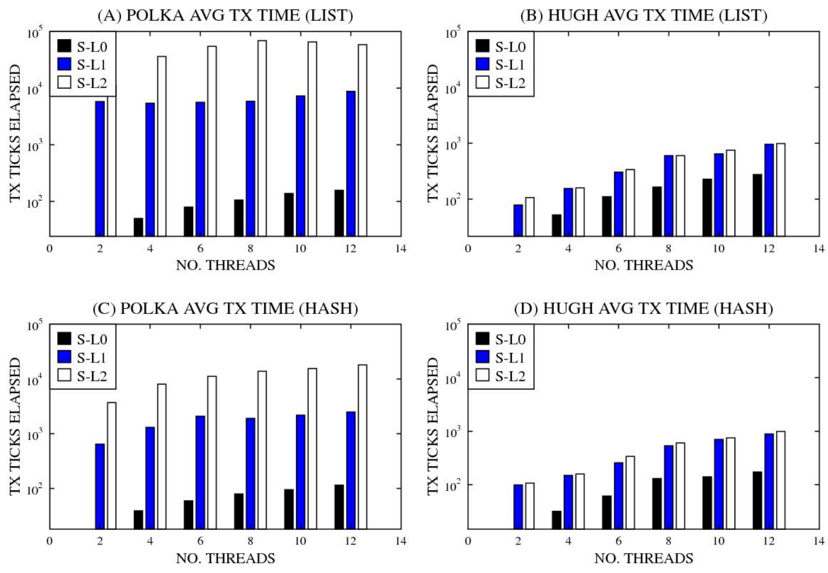


Fig. 4. Transaction Timing (Ticks)

Once semantic conflicts are introduced, *Hugh* performs markedly better than Polka under both benchmarks. With S-L1 semantic conflicts, *Hugh* shows a minimum improvement in throughput over Polka by a factor of approximately 4.3 and 4.5 for the list (Graph C) and hash table (Graph D) respectively. With S-L2 semantic conflicts, *Hugh* shows a minimum improvement by a factor of approximately 40 and 18, for the list (Graph E) and hash tables (Graph F) respectively.

Observe that with the Polka manager, as semantic conflicts are introduced the type of data structure used has less of an effect on mitigating the presence of aborts. It seems reasonable to assume that strategies for mitigating conflicts in transactional memory which rely on more 'concurrent' data-structures are of little benefit if one takes into account the kinds of semantic conflicts generated in these experiments.

4.2 Average Transaction Execution Time

In Figure 4 the average transaction execution time (ATET) is shown. In each graph, the Y-axis measures the ATET but note that the scale used is logarithmic for greater clarity and the maximum value is 10^5 ticks for all graphs. Each graph provides the results for a particular contention manager with a particular benchmark, and each bar shows the performance under a different semantic conflict level. The time is measured in elapsed ticks, (the fastest unit of time that can be measured on the platform) and denotes the average time spent executing a transaction by all threads.

One would expect that greater throughput generally corresponds to less average time spent executing a transaction (this is not guaranteed however, as unlike execution time, throughput also includes time spent outside of transaction execution). Given that *Hugh* resolves both concurrent and semantic conflicts, there should be less time required to execute a transaction when semantic conflicts are introduced, whereas with the Polka manager, transaction time should increase if repeated conflicts cause threads to back off (which involves calling the sleep function).

The performance of the Polka manager is shown in graphs A and C. One may observe that the ATET increases substantially as the level of semantic conflicts is increased. Conversely, the performance of *Hugh* (graphs B and D) does not exhibit the same degree of increase in ATET as the number of semantic conflicts is increased. This seems to suggest that the overhead of executing our policy does not increase substantially as semantic conflicts increase, unlike the Polka manager.

5 Conclusion and Future Work

This paper presents *Hugh*, a UC where threads conduct speculative execution of aborted transactions and 'commit by consensus', to mitigate both concurrent conflicts, and semantic conflicts; where some logical condition in the application ultimately prevents the progress of threads. We have described how conflicts can be resolved by a parallel exploration of transaction permutations and provided initial results which demonstrate increased throughput over a published contention manager.

The evaluation section presented some encouraging results via micro benchmarks in a custom scenario. Future work will require further testing with more sophisticated benchmarks. One issue with existing benchmarks, however, is that they evaluate performance with respect to concurrent conflicts, rather than the progress of the application (although this is not surprising given the immense scope of what can be defined as a semantic conflict).

We believe the most significant contribution made by our approach is the treatment of transaction conflict resolution as a state space exploration problem and in future we plan to conduct experiments with transactions of greater complexity, (nested transactions for instance). We anticipate that far from being a hindrance, semantic conflicts are useful as they will allow the state space of aborted transactions to be 'pruned' in favour of permutations which actually provide greater progress to the application.

References

1. Herlihy, M., Luchangco, V.: Distributed computing and the multicore revolution. ACM SIGACT News 39(1), 62–72 (2008)
2. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60. ACM (2005)

3. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(1), 124–149 (1991)
4. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 4–18. Springer, Heidelberg (2009)
5. Bai, T., Shen, X., Zhang, C., Scherer, W., Ding, C., Scott, M.: A key-based adaptive transactional memory executor. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–8. IEEE (2007)
6. Dolev, S., Hendler, D., Suissa, A.: Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, pp. 125–134. ACM (2008)
7. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 237–246. ACM (2008)
8. Lozi, J., David, F., Thomas, G., Lawall, J., Muller, G.: Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In: Work in Progress in the Symposium on Operating Systems Principles, SOSP, vol. 11 (2011)
9. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 258–264. ACM (2005)
10. Heber, T., Hendler, D., Suissa, A.: On the impact of serializing contention management on stm performance. Journal of Parallel and Distributed Computing (2012)
11. Yoo, R., Lee, H.: Adaptive transaction scheduling for transactional memory systems. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, pp. 169–178. ACM (2008)
12. Wamhoff, J., Fetzer, C.: The universal transactional memory construction. Technical report, 12 pages, University of Dresden, Germany (2010)
13. Chuong, P., Ellen, F., Ramachandran, V.: A universal construction for wait-free transaction friendly data structures. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, pp. 335–344. ACM (2010)
14. Crain, T., Imbs, D., Raynal, M.: Towards a universal construction for transaction-based multiprocess programs. In: Bononi, L., Datta, A.K., Devismes, S., Misra, A. (eds.) ICDCN 2012. LNCS, vol. 7129, pp. 61–75. Springer, Heidelberg (2012)
15. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. ACM SIGPLAN Notices 41, 253–262 (2006)
16. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 240–248. ACM (2005)

Topic 10: Parallel Numerical Algorithms (Introduction)

Julien Langou, Matthias Bolten, Laura Grigori, and Marian Vajteršic

Topic Committee

The solution of large-scale problems in Computational Science and Engineering relies on the availability of accurate, robust and efficient numerical algorithms and software that are able to exploit the power offered by modern computer architectures. Such algorithms and software provide building blocks for prototyping and developing novel applications, and for improving existing ones, by relieving the developers from details concerning numerical methods as well as their implementation in new computing environments.

The topic includes many different aspects, ranging from fundamental algorithmic concepts, to their efficient implementation on modern parallel architectures, such as multicore and multi-GPU systems, to their application in design and prototyping scientific simulation software, as well as to performance analysis.

From the papers submitted to this year's Euro-Par, the topic of Parallel Numerical Algorithms involving these themes attracted submissions from various continents. Each paper received at least four reviews and finally five were selected for presentation following extensive discussions between members of Euro-Par's Program Committee.

Kuzmin, Luisier and Schenk describe a parallelization based on techniques used for sparse direct solvers to compute selected entries of the inverse of a sparse matrix. The technique is successfully applied in the context of quantum transport calculation.

Agullo, Buttari, Guermouche and Lopez presents an implementation of the multifrontal QR factorization based on the StarPU runtime. The parallelism related to the factorization of each frontal matrix as well as the parallelism available among different frontal matrices and exposed by the separator tree is exploited. The authors show that a runtime system as StarPU can be successfully used to implement sparse/irregular matrix computations. The paper confirms the interest of runtime systems, even for sparse computations.

Schindewolf, Rocker, Karl and Heuveline compare different ways of implementing the Conjugate Graduate method (CG) on multi-core CPUs. The authors apply transactional memory technique and show that a "pipeline" CG method enables to speedup execution time by reducing communication and synchronization costs.

Lotz, Naumann, Sagebaum, and Schanen explain how to compute the discrete adjoint, with the Algorithmic Differentiation (AD) tool "dco", within the PETSc framework. Technical work described includes management by AD of the BLAS and LAPACK used in PETSc, and differentiation of the MPI communications involved (with a focus on persistent communication). This strategy to obtain discrete adjoints targets difficult situations, involving libraries and tools at various functionality levels (PETSc, BLAS/LAPACK), as well as MPI parallelism. These situations will occur typically in large, real-life applications. This strategy makes full use of the versatility of overloading-based AD.

Schreiber, Weinzierl, and Bungartz focus on solvers for partial differential equations and consider dynamically adaptive grids arising from spacetrees. The authors use the fact that such grids have an underlying tree formalism and use it to decompose such grids into clusters on-the-fly. The authors also describe an approach for dynamically adaptive cluster reordering and skipping. The algorithms are implemented using OpenMP tasks and TBB based on a depth-first traversal of trees.

These five papers provide a selected overview of recent developments in the design and implementation of numerical methods on modern parallel architectures.

It is appropriate, at this time, to thank the authors who submitted papers to our topic and to congratulate those whose papers were accepted. We are especially grateful to the referees who provided us with carefully written and informative reviews. Finally, we thank the conference organizers for providing the opportunity to the participants to present and discuss the state-of-the-art in Parallel Processing in the beautiful city of Aachen.

Cluster Optimization and Parallelization of Simulations with Dynamically Adaptive Grids

Martin Schreiber, Tobias Weinzierl, and Hans-Joachim Bungartz

Technische Universität München,
Boltzmannstrasse 3, 85748 Garching
{martin.schreiber,weinzier,bungartz}@in.tum.de

Abstract. The present paper studies solvers for partial differential equations that work on dynamically adaptive grids stemming from spacetrees. Due to the underlying tree formalism, such grids efficiently can be decomposed into connected grid regions (clusters) on-the-fly. A graph on those clusters classified according to their grid invariancy, workload, multi-core affinity, and further meta data represents the inter-cluster communication. While stationary clusters already can be handled more efficiently than their dynamic counterparts, we propose to treat them as atomic grid entities and introduce a skip mechanism that allows the grid traversal to omit those regions completely. The communication graph ensures that the cluster data nevertheless are kept consistent, and several shared memory parallelization strategies are feasible. A hyperbolic benchmark that has to remesh selected mesh regions iteratively to preserve conforming tessellations acts as benchmark for the present work. We discuss runtime improvements resulting from the skip mechanism and the implications on shared memory performance and load balancing.

Keywords: dynamic adaptivity, cluster skipping, shared memory load balancing, space-filling curve.

1 Introduction

Mesh-based solvers for partial differential equations (PDEs) that rely on the combination of recursive spatial sub-refinement with space-filling curves (SFCs) are popular in multiple application fields [1,6,13,15,21]. They embed the computational domain into a geometric primitive or a strip of primitives, and subdivide the primitives locally and recursively into smaller primitives. Those primitives are ordered along the SFC. Such a spacetree formalism facilitates dynamically adaptive grids and parallel mesh processing, as the curve prescribes a unique total mesh element order that can be cut into equally sized partitions for parallelization. In particular matrix-free solvers with heterogeneous solution smoothness such as explicit schemes for hyperbolic conservation laws resolving shock fronts benefit from the dynamic adaptivity [11]. They then usually sweep the grid once per time step and update (partially) the solution in each grid cell [7,20].

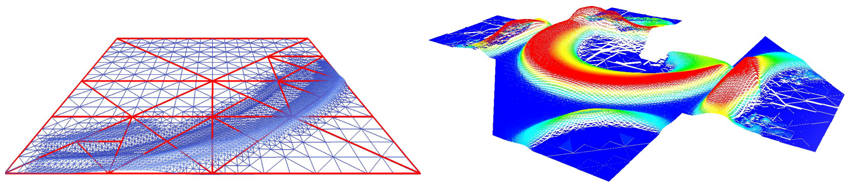


Fig. 1. Shock wave runs through domain while the grid changes dynamically and decomposes into clusters. Some of them were removed from the right illustration.

An efficient single compute node grid traversal here is essential. Multiple spacetree codes report on memory-efficient encodings, while the space-filling curve implies high memory access locality due to hash tables [9] or stack-based grid data schemes [1,21,22], e.g. Both ingredients tackle the memory bandwidth challenge which is expected to be one of the most crucial challenges in upcoming architectures [4]. We refer to [21,22] and remarks therein for measurements. Skipping coarser levels of the tree and to traverse only its leaves are further techniques reducing the total workload [2,6] if the geometric multi-scale structure is not required. Tree cuts developed for geometric multigrid solvers in turn allow to skip grid regions under-running a given mesh size threshold [21]. To the best of our knowledge, there is however neither a formalism nor an analysis of a technique that allows to skip whole grid regions independent of their resolution.

The present paper discusses an approach where multi-scale grid regions are skipped throughout the subsequent traversal. This speeds up algorithmic phases where either only spatial subregions are of interest or a holistic mesh processing does not justify the effort. Examples are meshing traversals reconstructing a proper 2:1 balancing [17] or local time stepping where regions lagging behind in time have to be updated prior to other mesh elements [7,20]. An example for the latter are solvers for linear equation systems that update preferentially sets of unknowns with significant residuals [16]. While the present work focuses on triangle-based meshing in combination with the Sierpiński space-filling curve [3], all paradigms can directly be applied to other SFC-based codes.

Obviously, an on-the-fly choice of spatial subsets handled by the traversal interplays with the traversal's concurrency and the parallelization—in particular if massive numbers of tightly coupled cores have to be handled that are sensitive to NUMA effects, ill-balancing, latency, and tasking overhead [4,18]. We introduce a shared memory parallelization that does not deteriorate due to the skipping and compare it to straightforward task-based parallelization. Reduced memory access and improved data affinity here compensate the reduced concurrency level.

The remainder is organized as follows: We first briefly describe the mesh paradigm and define the term cluster (Sect. 2). In Sect. 3, we then pick up this formalism to introduce the cluster skip mechanism. Implications of this mechanism on the shared memory load balancing and communication behavior are subject of the subsequent section, where we also introduce our affinity-aware implementation. Some results for a benchmark exhibit promising performance properties, before a brief conclusion and outlook in Sect. 7 close the discussion.

2 Grid Construction and Clustering

Our grid follows the spacetree/-forest formalism [1,6,21,22]: The computational domain is embedded into a triangle or a strip of triangles. For each triangle, we autonomously decide whether this triangle shall be split once. Such a scheme yields a binary tree or forest where triangles obtained due to a split are children of their preimage. Both the splitting rule and the order of the two children follow the construction scheme of the Sierpiński space-filling curve (SFC), i.e. the curve prescribes which triangle faces may be split, and the curve induces an order on the children [1,2,18]. The SFC in combination with depth-first defines a total order on all triangles of all levels. Let the *level* of a triangle be the minimum number of refinement steps required to construct the triangle. The initial triangle or the initial triangle strip, respectively, have level zero. Unrefined triangles are *leaves*. Unknowns are assigned to leaves only. We reiterate from [21,22] that a depth-first traversal of the spacetree induces an element-wise traversal of the leaves. Such a depth-first traversal can be formalized and realized as stack automaton [14,21] triggering in turn a matrix-free solver.

Starting from the notion of a binary triangle tree yielding the adaptive grid or a binary forest, respectively, we introduce the following notions: A *cluster* is a subtree of the binary mesh tree. It is identified by a unique tree node (empty circle in Fig. 2). If a triangle belongs to a cluster, all its successors, i.e. all finer triangles covered by it, belong to the same cluster, too. As the SFC defines a depth-first total order on all triangles, it induces an order on the clusters.

Let $\mathcal{C}$ be the set of clusters, and let each cluster have a list of neighbor clusters, i.e. clusters whose triangles shared at least one face or a part of it with a triangle from the respective cluster. The following algorithm clusters the spacetree:

– Assign each leaf a weight $W = 1$ and a marker $R = 0$.
– Let each cluster in $\mathcal{C}$ hold exactly one unrefined triangle and vice versa. Each cluster has at most three neighbors. The cluster cardinality equals the number of leaves.
– Run through the spacetree bottom-up:

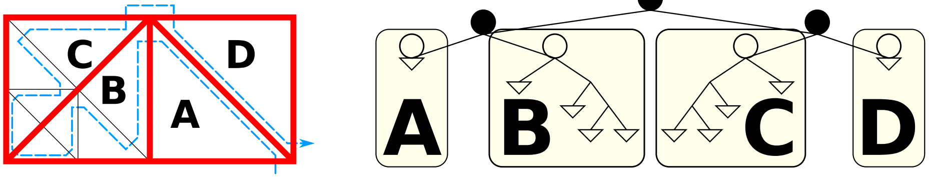


Fig. 2. Domain triangulation with clusters marked with thick borders (left). Representation of same grid by a binary tree constructed with the Sierpiński SFC (right).

    - Set a refined node's weight to the sum of the left and the right child, i.e. $W = W_{lhs} + W_{rhs}$.
    - If $W \leq W_{join}$ being a given threshold, merge the two clusters of the right and the left child. Replace these two cluster triangles in $\mathcal{C}$ by their parent, i.e. reduce $\mathcal{C}$'s cardinality by one. Also merge the neighbor lists of the two children. This is an operation with linear complexity.
- Run through the spacetree top-down:
    - If a node is the left child of a parent triangle, set $R = R_{parent}$.
    - If a node is a right child of a parent triangle, set $R = R_{parent} + W_{lhs}$.

The algorithm assigns each leaf to one cluster (Fig. 2). The clusters' size is controlled via W_{join}. Each cluster has a distinguished coarsest triangle holding its W and R value, and each cluster knows all of its neighbor clusters. Once such a clustering is found, we easily can adopt it whenever the grid changes. For this, it also does make sense to introduce a split weight W_{split} as counterpart of W_{join}. Whenever a triangle is refined, its two children inherit the cluster affiliation. If a cluster exceeds the threshold W_{split}, it decomposes into two clusters—each one represented by the two triangles on the 1st recursion level. There is no need to construct clusters from scratch several times.

The clusters define a graph on the mesh where each graph node is a cluster. Two nodes are connected if their clusters shared a common face. This graph is small compared to the connectivity graph of the original mesh.

Our algorithm refers to a binary tree. An extension to a binary forest is straightforward. Furthermore, a bottom-up construction of the clusters starting from the whole tree or forest in practice is not an optimal choice. Instead, it does make sense to create the binary tree up to given level. The triangles of this initial level then prescribe an *initial clustering*. Starting from the initial clustering, the grid is refined further and the clustering is adopted.

Our cluster analysis and mesh traversal fits to a recursive realization. Though straightforward, in practice it might make sense to reduce recursion overhead. One approach is to replace it by an iterative scheme to avoid call-stack overhead [2,8]. Formally, such a transformation equals recursion unrolling. If recursion unrolling is applied within clusters only and if clusters hold exclusively totally balanced subtrees, i.e. all leaves within one particular cluster have the same level, clustering and non-recursive realization mirror the optimization from [8]. Such an approach however relies on invariant grid regions and has to be used carefully if the grid changes frequently. Related work furthermore stores the leaves of the clusters only and reconstructs the coarser levels of the tree bottom-up [2,15]. Our implementation holds the whole tree and sticks with a recursive realization, but case distinctions within the recursive code are eliminated aggressively: Automaton states together with their possible transitions are rewritten by a code-generator into specialized functions with a minimal set of case distinctions [19] severely reducing branching mis-predictions. Argument-controlled context profiling validates that this pays off [10]. Furthermore, PDE-specific operations are invoked on the leaves only.

3 Dynamically Adaptive Cluster Reordering and Skipping

Let f be a marker operation on triangles that identifies those to be refined or coarsened next. Our element-wise traversal runs through the grid or tree, respectively, and evaluates f on each triangle. The subsequent mesh sweep then refines or coarsens which might yield non-conforming grids, i.e. grids with hanging nodes. As we rely on conforming tessellations, the non-conform refinement or coarsening mark further triangles. We have to traverse the grid multiple times until the global grid becomes conform again (Fig. 3), i.e. grid modifications might trigger a cascade of grid traversals propagating the grid updates.

Let each triangle hold a state $S \in \{0, 7\}$ and one marker per face encoded in a 3-tuple $T_f = (000)$. If f modifies a triangle, it updates its state as well as the face meta information. The number of possible refinements and meta information updates is fixed (Fig. 4). From the face meta data adjacent cells can derive how they have to adopt to make the grid conform. Along the SFC this information propagation resembles Gauß-Seidel. Otherwise it is a Jacobi-like information spreading which motivates the fact that multiple sweeps are typically necessary to make the grid conforming.

The state encodes the triangle's local state, the incoming marker adjacent refinement/coarsening information. The following table gives the new triangle state as well as the marker forwarded to adjacent triangles:

		incoming edge marker							
	state	000	001	010	011	100	101	110	111
no request	**0**	000,**0**	100,**5**	100,**6**	100,**7**	000,**4**	000,**5**	000,**6**	000,**7**
INVALID	**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**
local coarsening request	**2**	000,**2**	100,**5**	100,**6**	100,**7**	000,**4**	000,**5**	000,**6**	000,**7**
local refine request	**3**	100,**4**	100,**5**	100,**6**	100,**7**	000,**4**	000,**5**	000,**6**	000,**7**
refined: hyp	**4**	000,**4**	000,**5**	000,**6**	000,**7**	000,**4**	000,**5**	000,**6**	000,**7**
refined: hyp, left	**5**	000,**5**	000,**5**	000,**7**	000,**7**	000,**5**	000,**5**	000,**7**	000,**7**
refined: hyp, right	**6**	000,**6**	000,**7**	000,**6**	000,**7**	000,**6**	000,**7**	000,**6**	000,**7**
refined: hyp, right, left	**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**

We consider clusters to be atomic entities coupled to other clusters via their one-dimensional boundary sub-manifold. Let an *active* cluster be a cluster where f has marked elements. Obviously, this property can be reduced throughout a traversal of the cluster tree. If all triangles hold $T_f = (000)$ after the traversal, the cluster is not active. A cluster's *active* flag is an or-combination of all triangle meta data.

The marker propagation is a face data exchange. It directly implies a marker semantics on clusters. A cluster without local refinement becomes active if and only if a neighboring cluster has set a face marker in the iteration along a common face. As we have the neighborhood relation at hand, this leads to a publish-poll pattern:

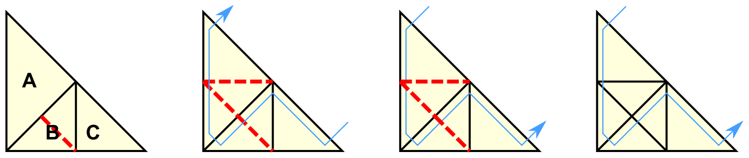


Fig. 3. From left to right: grid with three triangles A,B, and C. As B is refined, A has to be refined twice to preserve a conform tessellation.

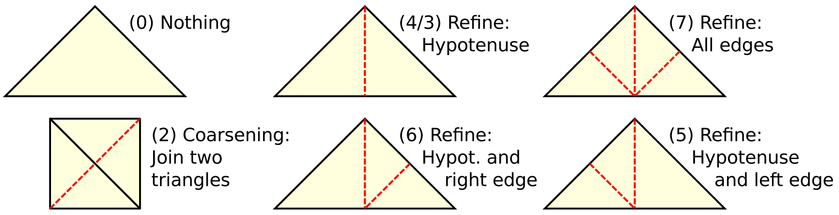

(0) Nothing
(4/3) Refine: Hypotenuse
(7) Refine: All edges
(2) Coarsening: Join two triangles
(6) Refine: Hypot. and right edge
(5) Refine: Hypotenuse and left edge

Fig. 4. Different states from the automaton to form a conforming grid. The dashed lines are new edges of the refined triangle.

– If a cluster is *active*, the traversal automaton has to run through its grid. Marker information from the boundary faces is polled. This might set the cluster's state to active again. It also might set markers along the cluster's boundary, i.e. publish new markers.
– If a cluster is *not active*, the traversal automaton runs over the boundary markers published by the neighbors. If all of them are unset, no grid updates within the cluster are necessary. If one marker is set, the cluster is active and is traversed.

With the second case distinction at hand, we are able to *skip clusters* throughout the traversal when we know a priori, i.e. when the automaton enters the cluster's coarsest triangle, that they are not active. In this case, we continue with grid elements of another not yet processed cluster. As each sweep polls the marker information and updates only subregions of the grid, the overall data flow pattern resembles a petri net on the cluster graph.

For the realization of data exchange along clusters representing fractions of the SFC, we refer to [1,2,18] and [21,22] for $d > 2$ with hypercubes instead of triangles. The SFC distinguishes right from left neighbors uniquely, i.e. the published markers' cardinality is bounded by the surface. As the triangles are lined up like pearls on a string along the Sierpiński SFC, the cluster partitions are connected and exhibit a quasi-optimal surface-volume ratio [5]. The SFC furthermore linearizes the left and right boundary uniquely whereas the boundary fractions of relevance for one particular neighbour are connected, continuous, and already published in the read order [5,18,21].

Though it can happen that clusters are set active multiple times due to grid conformity sweeps, we do not observe such a behavior often. Furthermore, if we assume f to be idempotent on the triangle, the proof is straightforward that the marker update mechanism does not induce cycles. While the order for the refinement and coarsening here is given implicitly, reordering for different purposes (local time stepping, e.g.) might yield additional benefit.

4 Cluster-Based Parallelization

Without considering parallelization so far, we observe that leaves are coupled due to their faces to neighbouring triangles. For our present applications, leaves write data to their common faces in one sweep. At the end of the traversal, both triangles adjacent to any face have accumulated their data on the face. These data typically are input parameters to flux computations. In the next iteration, the data is read within the triangle and the triangle's values are updated. Such a data flow mirrors the marker pattern of the previous section.

With cluster based parallelization and by considering clusters to be atomic grid entities, we can split the data exchange into two phases. First, clusters write data to their interface faces: faces shared with other cluster. If grid cluster interface faces are duplicated per cluster, this write process is thread-safe. With a stack- and stream-based approach, the data for each adjacent cluster is consecutively stored in memory. We can run length encode which face data are sent to which neighbour and exchange data belonging to a particular neighbor en block and efficiently. Then, we can merge the duplicated interfaces explicitly prior to the next grid traversal. By running the merge operation with consideration of the order of clusters along the SFC, duplicate flux computations can be avoided at the cluster interface. Due to the sub-manifold and the minimal surface property [5], these merge operations are cheap compared to the overall updates and the memory overhead is small.

Depth-first traversals of trees are a classic demonstrator for task-based parallelism where a shared data structure's disjoint subsets are handled by different threads. No data overhead besides cluster surface communication buffers is induced. We pick up this property to derive two different parallelization schemes. In combination with skipping, minimal cluster sizes, software-based affinities as supported by TBB, and the comparison of TBB and OpenMP tasks, this yields a multitude of different parallel algorithmic flavors.

For our *massive tree split* approach, we make the traversal automaton traverse the grid top-down. In each node, it spawns a right and a left task. If a child identifies a cluster, this subtask is not split up further. As the number of clusters typically exceeds the number of tasks, the grid traversal floods the system with tasks, delegates the distribution of tasks to threads to the runtime system, and relies on work stealing to achieve well-balanced workload. Subdomains handled by one thread might be discontinuous. Due to nondeterministic work stealing, the assignment to threads even might change from grid traversal to grid traversal. We expect data affinity penalties from this property that has to be compensated by a high concurrency level, and point out that TBB supports a manual choice of task affinities.

For our *owner-computes* approach, we analyze the tree attributes: The property W on the spacetree's root determines the total workload on the grid. Given

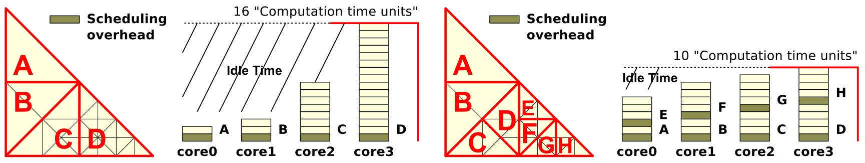


Fig. 5. Left image: A cluster size which leads to as many cluster as there are cores available on the system would lead to a workload imbalance. Right image: Creating more clusters than there are cores available leads to a better balanced problem even by considering the tasking overhead marked with dark-yellow boxes.

p threads, a thread i knows that each cluster with $R \in [iW/p, (i+1)W/p[$ is to be handled by this thread for a rather balanced work decomposition. Consequently, each thread can run through the tree processing only triangles or clusters, respectively, fitting into its work interval. This is a concurrent read due to publish-poll. Such a behavior mirrors the logical merge of multiple subsequent clusters along the Sierpiński curve for one task.

Traditional SFC parallelization [5,6,9,15,17,18,21] cuts the curve into equally sized chunks and distributes these chunks among the cores. With threshold based cluster splitting, such an equal balancing is not possible anymore, and it is obvious that an owner-computes scheme suffers from ill-balancing whereas flood filling might compensate ill-balancing due to work stealing. We point out that clusters of size one with an owner-computes scheme mirror a traditional SFC-based parallelization where the workload is cut into equally sized pieces along the curve. Our clustering either has to compensate ill-balancing due to an efficiency gain, or clustering has to tackle potential ill-balancing explicitly.

For the latter approach, we use *scan clustering* decomposing clusters on-the-fly into their tree if a cluster overlaps with an optimal SFC partitioning. This mechanism weakens the clusters' atomic property, but allows for a fine granular a priori load balancing. An additional parallization degree of freedom arises if any cluster exceeding a given size is decomposed automatically, while the big clusters are preserved for the skip mechanism. We then again rely on work stealing to tackle ill-balancing (Fig. 5).

5 Benchmark Scenario

A setup based on the shallow water equations (SWE) computing a radial breaking damn in a basin acts as benchmark for the present paper. The rectangular basin has side length of $5000m$. It is filled with fluid of a depth of $10m$ (sea level). The initial condition is a radial breaking dam with radius $500m$ around the point $(-2000m, 2500m)^T$ relative to the origin at $(0,0)^T$. Its height relative to the sea level is $1m$. We apply non-reflecting boundary conditions (Fig. 1).

The system is discretized using discontinuous Galerkin method with 1st order cell basis functions and 3rd order Gaussian quadrature on each face. An explicit

Euler time-stepping scheme with Rusanov fluxes acts as time stepping. Throughout the simulation, we refine each triangle with a water surface displacement relative to the normal sea level exceeding the threshold $0.1m$ and allow coarsening for triangles with a threshold above $0.01m$. The refinement is bounded by the maximal triangle level 8. Different algorithmic phases realized by grid traversals do exist: Setup and visualization, time stepping, and consistency traversals recovering the mesh conformity. The latter typically modify only few triangles.

6 Results

All experiments were conducted on an Intel Westmere with 4 Intel Xeon CPUs (E7-4850@2.00GHz) and 256 GB memory totally available on the platform. This gives 4×10 physical cores plus hyper-threading. For the parallelization, we have a TBB and OpenMP realization due to Intel Composer XE (ver. 2013.1.117). We used affinity bit-masks to hard-limit the number of threads and avoid TBB's automatic worker task creation. Thread affinities are set such that they map the first ten threads to the ten physical cores on the CPU on the first socket. Thread numbers 41–80 are mapped onto hyper-threading cores.

Prior to algorithmic studies, we first determined for a good cluster size for both OpenMP and TBB (Fig. 6). It results from extensive search. While TBB outperforms OpenMP for most settings, 8192 is a natural choice for the cluster size threshold. This value is used from hereon. We also stick with TBBs.

Next, we studied cluster skipping distinguishing adaptivity traversals, computation traversals, as well as cluster construction (Fig. 7). The construction time is negligible, the simulation time itself is independent of the skipping. The skipping however reduces the time spent to make the grid conforming when it has changed before. We observe that this improvement is the better the fewer threads are used which is a natural result from the inhomogeneity of the workload due to splitting, i.e. work balancing gains impact but also introduces overhead. However, the splitting optimization is robust. A normalization of the run-times without skipping reveals that the normalized speedup degradation is marginal.

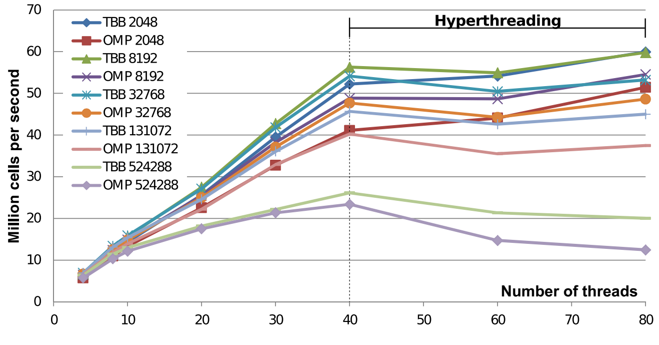


Fig. 6. OpenMP vs. TBB tasking comparing different cluster sizes

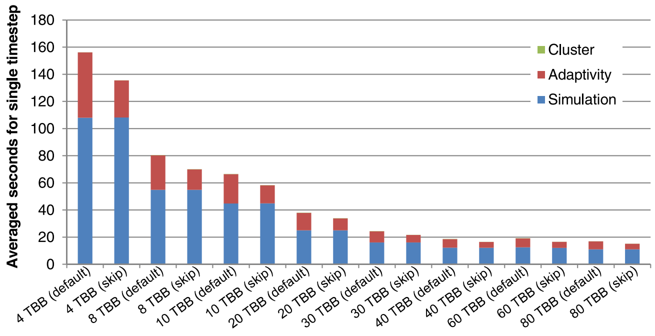


Fig. 7. Detailed timings for each simulation phase. Time taken for clustering phase is invisible small.

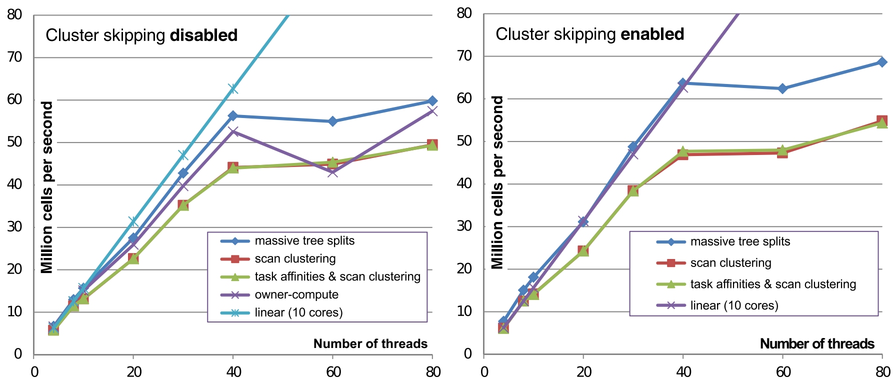


Fig. 8. Comparison of different parallelization strategies with an without skipping for short simulation time with few adaptivity traversals and few skips

For one hundred time steps, we compared different combinations of skipping and parallelization strategies (Fig. 8). The skipping again pays off and allows us to obtain linear speedup in some cases, i.e. the algorithmic optimization helps to close the gap between optimal and observed scaling—however only compared to non-skipping algorithms. Massive tree splits outperform the other parallel approaches as long as the cluster size is chosen reasonable (Fig. 6) and does not hinder the algorithm to exploit all cores. The owner-computes scheme cannot compete even though we use scan clustering obtaining theoretically almost perfectly balanced work decompositions. However, owner-computes with its manual data affinity yields better performance than TBB's task affinity feature. For this experiment, flooding the runtime system with tasks and cluster skipping are the methods of choice.

With first simulation results for a short run at hand, we studied the same setup for 15,000 time steps (Fig. 9). Longer observation intervals imply more grid changes. Cluster splitting still is improved by task affinities, but not significantly.

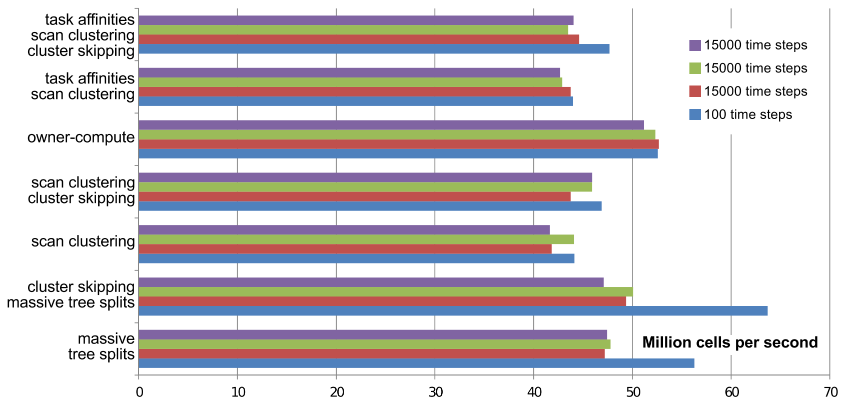


Fig. 9. Runtime per time step averaged over all algorithm phases. One measurement with 100 time steps, three samples with 15,000 time steps.

Though we assume the massive tree splits to yield good balancing due to work stealing, it is the only strategy that is not robust with respect to simulation time. We ascribe this to NUMA effects in combination with a touch-first data policy. In contrast, the owner-computes scheme outperforms the other strategies.

Our experiments reveal that for our setups, a equally balanced workload due to task stealing, e.g., is essential for the first grid traversals. When the grid changes significantly and the per-cluster workload as well as cluster distribution become inhomogeneous as well, affinity effects gain importance. A scheme that fixes the affinities due to a lack of task concurrency then outperforms task affinities assigned manually. The better work distribution with task stealing or equally cutting the SFC cannot compensate this.

7 Outlook

Future work comprises the development of an appropriate cost model anticipating skipping, workload inhomogeneity, and affinity issues. Furthermore, the interplay of the skip mechanism with a distributed memory parallelization is interesting as skips reduce cluster communication. Finally, we expect a better support of user-controlled affinity in programming languages and libraries. We are looking forward to use this or to contribute to this ourselves. Methodologically, an important locality-aware aspect for Invasive Computing [12] was created with forced affinities providing better performance for long simulation runs. On the application side, the present algorithms have to proof of value for implicit schemes where clusters and skips interplay with equation system solvers.

Acknowledgements. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing (SFB/TR 89). It is partially based on work supported by Award No. UK-c0020, made by the King Abdullah University of Science and Technology (KAUST). All software is freely available at *http://www5.in.tum.de/sierpinski.*

References

1. Bader, M., Böck, C., Schwaiger, J., Vigh, C.A.: Dynamically Adaptive Simulations with Minimal Memory Requirement - Solving the Shallow Water Equations Using Sierpinski Curves. SISC 32(1) (2010)
2. Bader, M., Rahnema, K., Vigh, C.: Memory-Efficient Sierpinski-Order Traversals on Dynamically Adaptive, Recursively Structured Triangular Grids. In: Jónasson, K. (ed.) PARA 2010, Part II. LNCS, vol. 7134, pp. 302–312. Springer, Heidelberg (2012)
3. Bartholdi, J.J., Goldsman, P.: Vertex-labeling algorithms for the hilbert spacefilling curve. Software: Practice and Experience 31(5), 395–408 (2001)
4. Borkar, S., Chien, A.A.: The future of microproc. Commun. ACM 54 (2011)
5. Bungartz, H.-J., Mehl, M., Weinzierl, T.: A Parallel Adaptive Cartesian PDE Solver Using Space–Filling Curves. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1064–1074. Springer, Heidelberg (2006)
6. Burstedde, C., Wilcox, L.C., Ghattas, O.: p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. SISC (3) (2011)
7. Dumbser, M., Käser, M., Toro, E.: An Arbitrary High Order Discontinuous Galerkin Method for Elastic Waves on Unstructured Meshes V: Local Time Stepping and p-Adaptivity. Geophysical Journal Int. 171(2), 695–717 (2007)
8. Eckhardt, W., Weinzierl, T.: A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 567–575. Springer, Heidelberg (2010)
9. Griebel, M., Zumbusch, G.: Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. Parallel Comp. 25(7), 827–843 (1999)
10. Küstner, T., Weidendorfer, J., Weinzierl, T.: Argument controlled profiling. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009 Workshops. LNCS, vol. 6043, pp. 177–184. Springer, Heidelberg (2010)
11. LeVeque, R.J., George, D.L., Berger, M.J.: Tsunami modelling with adaptively refined finite volume methods. Acta Numerica 20, 211–289 (2011)
12. Bader, M., Bungartz, H.-J., Schreiber, M.: Invasive computing on high performance shared memory systems. In: Keller, R., Kramer, D., Weiss, J.-P. (eds.) Facing the Multicore-Challenge III 2012. LNCS, vol. 7686, pp. 1–12. Springer, Heidelberg (2013)
13. March, W.B., et al.: Optimizing the comp. of n-point correlations on large-scale astronomical data. In: Proc. of the Int. Conf. on High Perf. Comp., Netw., Stor. and Analysis, SC 2012. IEEE Computer Society Press (2012)
14. Meister, O., Rahnema, K., Bader, M.: A Software Concept for Cache-Efficient Simulation on Dynamically Adaptive Structured Triangular Grids. In: De Boschhere, K., D'Hollander, E.H., Joubert, G.R., Padua, D., Peters, F. (eds.) Applications, Tools and Techniques on the Road to Exascale Computing. Advances in Parallel Computing, ParCo 2012, Gent, vol. 22, pp. 251–260. IOS Press (May 2012) ISSN: 0927-5452
15. Rahimian, A., Lashuk, I., Veerapaneni, S., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampath, R., Shringarpure, A., Vetter, J., Vuduc, R., Zorin, D., Biros, G.: Petascale direct numerical simulation of blood flow on 200k cores and heterog. arch. In: Proc. of the 2010 ACM/IEEE Int. Conf. for HPC, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society (2010)
16. Rüde, U.: Mathematical and computational techniques for multilevel adaptive methods. Frontiers in Applied Mathematics, vol. 13. SIAM (1993)

17. Sampath, R.S., Biros, G.: A parallel geometric multigrid method for finite elements on octree meshes. SISC 32(3), 1361–1392 (2010)
18. Schreiber, M., Bungartz, H.-J., Bader, M.: Shared memory parallelization of fully-adaptive simulations using a dynamic tree-split and -join approach. In: IEEE Int. Conf. on High Performance Comp., HiPC (2012)
19. Schreiber, M., et al.: Generation of parameter-optimised algorithms for recursive mesh traversal algorithms (to be published, 2013)
20. Unterweger, K., Weinzierl, T., Ketcheson, D., Ahmadia, A.: Peanoclaw—a functionally-decomposed approach to adaptive mesh refinement with local time stepping for hyperb. conservation law solvers. Technical report, Technische Universität München (2013)
21. Weinzierl, T.: A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids. Verlag Dr. Hut (2009)
22. Weinzierl, T., Mehl, M.: Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. SIAM Journal on Scientific Comp. 33(5), 2732–2760 (2011)

Discrete Adjoints of PETSc through `dco/c++` and Adjoint MPI

Johannes Lotz, Uwe Naumann, Max Sagebaum, and Michel Schanen[⋆]

LuFG Informatik 12: Software and Tools for Computational Engineering
RWTH Aachen University, Germany
{lotz,naumann,schanen}@stce.rwth-aachen.de
http://www.stce.rwth-aachen.de
Department of Mathematics and Center for Computational Engineering Science
RWTH Aachen University, Germany
sagebaum@mathcces.rwth-aachen.de
http://www.mathcces.rwth-aachen.de

Abstract. PETSc's [1] robustness, scalability and portability makes it the foundation of various parallel implementations of numerical simulation codes[1]. We formulate a least squares problem using a PETSc implementation as the model function and rely on adjoint mode Algorithmic Differentiation (AD) [2] for the accumulation of the derivative information. Various AD tools exist that apply the adjoint model to a given C/C++ code, while none is able to differentiate MPI [3] enabled code. We solved this by combining `dco/c++` and the Adjoint MPI library, leading to a fully discrete adjoint implementation of PETSc. We want to underline that this work differs from accumulating derivative information through AD for PETSc algorithms (see e.g. [4]). We compute derivative information of PETSc itself opening up the possibility of an enclosing optimization problem (as needed, e.g., by [5]).

1 Motivation

Our case study is the two-dimensional Bratu equation,

$$\nabla^2 \mathbf{u} = -\lambda \exp(\mathbf{u}), \tag{1}$$

describing a solid fuel ignition with the parameter $0 < \lambda < 6$ and boundary conditions

$$u = b_i \text{ for } x = 0, x = 1, y = 0, y = 1$$

at the borders of the two dimensional square. For a 4x4 grid, $\mathbf{b}$ is of size 12 while there are only 4 inner points. The Bratu equation is part of the MINPACK-2 test suite [6] as well as an example code of the non-linear solver *SNES* in PETSc. It serves as a code base for our least squares problem.

⋆ This work was supported by the Fond National de la Recherche of Luxembourg under grant PHD-09-145.

1 See Applications/Publications [1].

The differential equation in (1) is solved by discretization using finite difference on a two-dimensional grid. We do not take a detailed look at the actual implementation in PETSc, but rather take a black box perspective of the code. We now formulate an optimization problem over the original solution to the Bratu equation.

We distinguish the following grid points:

- the inner computed grid points $\mathbf{u}$
- the n boundary grid points $\mathbf{b}$ used in the boundary condition
- and a subset $\mathbf{u}^s$ of m observation points of the inner grid points $\mathbf{u}$, for which additional observed values $\mathbf{u}^{ob}$ are assumed to be provided.

The computed values $\mathbf{u}^s$ are a subset of the inner points $\mathbf{u}$ dependent on the boundary conditions. Additionally, we have observed values $\mathbf{u}^{ob}$ that allow us to rate the correctness of our model. This fact may be formulated as a least squares problem where we want to minimize the difference between the computed and observed values by adapting the approximated or guessed boundary conditions.

$$S = \frac{1}{2}\sum_{i=1}^{m}(u_i^s - u_i^{ob})^2,$$

The computation of the cost functional S depending on the boundary conditions

$$S = F(\mathbf{b}) : \mathbb{R}^n \to \mathbb{R}$$

is implemented in PETSc using its non-linear solver SNES for the computation of $\mathbf{u}$. We used the code found in example 5 of the SNES tutorials. The additional implementation of the cost functional S for the least squares problem is straightforward.

We now describe step by step how we generated an implementation of the gradient ∇F of PETSc that enables us to feed a gradient based solution method.

In Sect. 2 we present Algorithmic Differentiation (AD) as our method of choice for the gradient computation. Additionally, we provide a brief overview of PETSc's code structure and where challenges arise. In Sect. 3 we provide a technical description of our AD overloading tool `dco/c++`. It is used to generate the adjoint code of PETSc. However, PETSc relies on the BLAS [2] and LAPACK [3] library for the sequential computation. We provide a methodical description of how we achieved adjoints of these library. Sect. 4 covers the adjoining of the MPI communication using our in-house developed AMPI library.

2 Background

We resort to a Steepest Descent or Gradient Descent algorithm as a proof of concept in order to minimize the residual S. As the name hints, it relies on the

[2] `http://www.netlib.org/blas/`
[3] `http://www.netlib.org/lapack/`

gradient to iteratively compute a better fit of the computed observation point values $\mathbf{u}^s$ to the actual observations $\mathbf{u}^{ob}$ by adapting the boundary conditions according to

$$\mathbf{b}_{n+1} = \mathbf{b}_n + \alpha \nabla F(\mathbf{b_n}),$$

where ∇F is the gradient of the aforementioned residual S with respect to the boundary conditions $\mathbf{b}$. In order to acquire first-order gradient information by finite difference, one would have to perturb each of the n inputs b_i, $0 < i \leq n$. This requires $n + 1$ runs of F. Once for each perturbation and once for the original computation. For $n \gg 1$ the gradient accumulation potentially becomes computationally infeasible.

2.1 Algorithmic Differentiation

AD is the chain rule of differential calculus applied symbolically to each statement of a given code. This can be done automatically by resorting to a compiler or by overloading the mathematical operations in a source code.

For a multivariate scalar function (e.g. calculation of cost function S) $y = F(\mathbf{x})$, $\mathbb{R}^n \rightarrow \mathbb{R}$ the chain rule may be used in two ways. First, the straightforward application leading to the tangent-linear model $\dot{y} = \nabla F(\mathbf{x}) \cdot \dot{\mathbf{x}}$, where $\dot{\mathbf{y}}$ is the directional derivative of the output y with respect to inputs $\mathbf{x}$ in direction $\dot{\mathbf{x}}$. Notice that in order to accumulate the entire gradient we have to iteratively set $\dot{\mathbf{x}}$ to each of the n Cartesian basis vector in $\mathbb{R}^n$ leading to a runtime cost of $\mathcal{O}(n) \cdot cost(F)$.

Second, the adjoint model based on the associativity of the chain rule:

$$\bar{\mathbf{x}} = \bar{\mathbf{x}} + \nabla F\left(\mathbf{x}\right)^{\mathsf{T}} \cdot \bar{y}. \tag{2}$$

$\bar{\mathbf{x}}$ are called the adjoints of the inputs $\mathbf{x}$, whereas $\bar{y}$ is the adjoint of the output y. Notice that the computation of the adjoints is in reverse order of the computation of the values, thus requiring a complete data flow reversal of a program. The *forward section* consists of the computation of the values, whereas the *reverse section* computes the adjoints while using the values saved in the forward section. Furthermore, it is essential to understand that with one output, the gradient accumulation is achieved by one adjoint computation of the corresponding adjoint code reducing the runtime complexity to $\mathcal{O}(1) \cdot cost(F)$. Of course, the constant factor may still be considerable. However, given the independence of the runtime from the input size n, the adjoint model may end up as the only feasible solution. All the other options for a gradient accumulation, finite difference and tangent-linear model, will become computationally too expensive at some input size $n \gg 1$.

The discrete approach with the tangent-linear and adjoint model may be applied through handwritten code, where the original code is transformed statement-wise into the derivative code. This work is very tedious and error prone. AD tools do this mechanical work semi-automatically. Source transformation tools are similar to the handwritten transformation, whereas operator overloading tools achieve the same goal by overloading every intrinsic operation.

Compared to the numerical method of finite difference, there are two advantages. One, we are able to extract exact derivatives with up to machine precision, whereas the precision of finite difference is largely dependent on the spacing factor h. Second, a similar method to the adjoint model is not available while using finite difference. It shifts the complexity from $\mathcal{O}(n) \cdot cost(F)$ to $\mathcal{O}(m) \cdot cost(F)$, with n and m being the inputs and outputs, respectively. This unique feature makes the adjoint model crucial for optimizations where the cost function is scalar, while being influenced by a large number of parameters.

2.2 PETSc

As mentioned above, we use a tutorial example as a test case in PETSc. We chose tutorial 5 of the SNES solver. It implements a solution of the Bratu differential equation (1). We had to make a few amendments to the code which will be explained later in this paper. The source code is available on our website [4].

The parameter λ is set to 6. We used a value of 1.0 in order to have a more stable system. The other changes are only related to `dco/c++` and will be explained in the next section.

We compiled PETSc with the default options, although we provide a custom BLAS and LAPACK library described in Sect. 3.1 and 3.2. The MPI calls in PETSc involving data of type PetscReal and PetscScalar are all replaced by adjoint MPI calls. The aforementioned MPI calls may be separated in two types. For one there are several collective invocations of Allreduce. The other one is the persistent MPI communication in src/vec/vec/utils/vpscat.c. Both will be dealt with in Sect. 4.

3 Adjoint Model Generation Using `dco/c++`

`dco/c++` is implementing AD by overloading in C++. The range of capabilities covered by `dco/c++` is driven by various applications and research subjects. Current projects are in the area of financial engineering, atmospheric physics, or fluid mechanics. `dco/c++` is also used in research on the generation of discrete adjoints using parallel environments, in particular OpenMP and MPI as, e.g., used in PETSc.

The objective is to provide an efficient and robust tool for the computation of projections of derivatives of arbitrary order of a function given as an implementation in C/C++, while focusing on the adjoint mode. Additionally, the capability of coupling the robust overloading technique with optimized computer generated or hand-written external computations of adjoint projections is provided. This is used extensively for the adjoining of BLAS.

During various collaborative research and development projects, we were able to compute fast adjoints for real world applications. In some cases [7] we achieved a factor of roughly 3.5 for the ratio

$$R = \frac{\text{Run time of one adjoint computation}}{\text{Run time of one function evaluation}} .$$

[4] `https://www.stce.rwth-aachen.de/trac/petsc`

For being able to achieve such a factor, we make heavy use of the C++ template engine and we exploit algorithmical and mathematical insight, e.g., statement-level preaccumulation.

A standard run for computing adjoints using `dco/c++` consists of a so called *forward run* being the forward section and generating a *tape* followed by exactly one *reverse run* being the reverse section in case of a scalar cost function. This structure follows from the requirement of making all computed values available in reverse order (data flow reversal). The forward run saves all required information in the tape used during the reverse run. This structure is also applied to the part of PETSc, which is to be differentiated. As PETSc is using BLAS as well as LAPACK library routines for numerical methods, we have to deal with those libraries, too. The treatment is sketched in the following subsections.

3.1 BLAS

BLAS (Basic Linear Algebra Subprograms) [8] is used by PETSc for non-parallel tasks and is a set of basic routines operating on scalars, vectors and matrices of type 'double' including, e.g., scaling of a vector by a scalar, or matrix vector products. The implementation of those basic operations is typically optimized for performance by manufacturers for the specific hardware that is running the computation. It is therefore desirable to stick to the supported BLAS implementation – at least during the forward run of the overloaded program execution. We therefore do not overload the original routines to avoid killing cache performance. Due to the reasonable amount of different routines, we provide hand-written adjoint routines of the BLAS routines used in PETSc, which rely on the original BLAS implementation whenever possible. This includes the computation of the function values during the forward run. We expect this to produce at least binary identical results to the non-overloaded function run, which is desirable for verification purposes. Additionally we expect a better runtime compared to a reimplementation of the BLAS routines.

3.2 LAPACK

LAPACK (Linear Algebra PACKage) [9] is also used by PETSc for non-parallel tasks. In contrast to BLAS this library implements algorithms for general linear algebra problems like linear systems (called, e.g., by PETSc's SNES) or eigenvalue problems. The implementation of LAPACK includes calls to BLAS routines for all basic vector and matrix operations. Because LAPACK comes with a large number of different specialized functions (in total ca. 1600 routines) we chose this time to differentiate those routines in a black box way; change the datatypes in LAPACK to our `dco/c++` datatype. Again, hand-written and optimized derivative code would yield better performing code. We therefore aim to provide those optimized adjoint routines of LAPACK step by step in a project-driven fashion. As in BLAS, those hand-written routines should of course reuse the original LAPACK ones. This is useful for efficiency reasons as well as for keeping the code up-to-date with the current LAPACK implementation. This will include also the possibility of switching to GPU-enabled LAPACK (see e.g. MAGMA [10]) or other LAPACK implementations.

4 Adjoint MPI and PETSc

The reversal of MPI [11] communication in AD is directly related to the implied data flow reversal. The adjoint MPI library is specifically designed to reverse the MPI communication. It is separated from the actual AD tool involved (here `dco/c++`) and therefore only traces the communication itself. The actual data (values and adjoints) is stored through an AMPI interface in the data structures of the AD tool.

Tracing communications amounts to tracing data dependence in between the processes' address space. As a formalism, we apply the PGAS (Partitioned Global Address Space) notation to the MPI communication.

Table 1. Adjoint communication of an MPI_Send and MPI_Recv between two processes P1 and P2

MPI	MPI_Send(x,P2), MPI_Recv(x,P1)
PGAS	P2.x=P1.x
Adjoint PGAS	P1.$\bar{x}$+=P2.$\bar{x}$; P2.$\bar{x}$=0
Adjoint MPI	MPI_Recv($\bar{x}$,P2), MPI_Send($\bar{x}$,P1)

Sending and receiving data are actual assignments in PGAS that need to be adjoined according to the incremental adjoint model (2). Every communication is transformed into an *incremental communication*. The adjoint MPI library itself as well as the consequences of the incremental communication and the data flow reversal have been subject to various papers covering blocking, non-blocking, collective [12] and one-sided communication [13]. The MPI calls in our PETSc implementation were tracked using a custom MPI header in order to pinpoint the MPI calls that need to be adjoined. In summary, there were two types of MPI communication. First, persistent communication consisting of an MPI_Send/Recv_init, MPI_Start, MPI_Wait and MPI_Request_free. And second collective communication in the shape of a MPI_Allreduce.

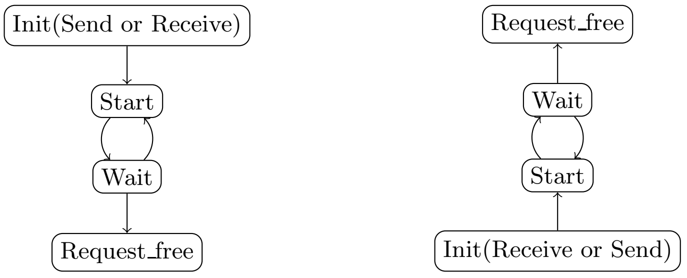
4.1 Persistent Communication

The reversal of non-blocking communication has already been subject of several publications [14,15]. They cover the reversal of Isend/Wait and Irecv/Wait pairs. In a nutshell, the reversal logic of the blocking send and receive is preserved. The send becomes a receive and the receive becomes a send. However, the ordering of the communication pairs Isend/Wait, Irecv/Wait is reversed. In the reverse section the Wait becomes either an Isend or Irecv whereas the Isend and Irecv both become a Wait. There are other effects which will not be discussed in this paper.

In our PETSc code we do not have pairs of non-blocking Isend/Wait and Irecv/Wait, but triplets of persistent Recv_init/Start/Wait and Send_init/Start/Wait calls. Persistent communication is similar in logic to the non-blocking pairs. As stated by the MPI standard the purpose of persistent communication is:

> Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a persistent communication request once and, then, repeatedly using the request to initiate and complete messages.

Thus, an MPI_Start may be called several times upon the same initialization using MPI_Send/Recv_init. MPI predicts a potential reduction of the communication overhead due to the persistent request. The requests need to be deallocated explicitly using MPI_Request_free as opposed to an implicit deallocation at the MPI_Wait using non-blocking communication. That logic should be preserved in adjoint MPI. In particular, we want to correctly and efficiently adjoin the multiple MPI_Start calls.



(a) Forward Communication (b) Reverse Communication

Fig. 1. Adjoint communication of a Init, Start, Wait and Request_free

As mentioned in the previous section, the core principle of AD by overloading is that we tape each step of the forward section in order to generate a correct adjoint section. The same holds for adjoint MPI. We trace each of the involved routines in the forward section along with its arguments. We now go step by step through the reverse section for each routine and look at what has to be stored in the forward section:

Request_free. The deallocation of the requests marks the end of the persistent communication instance. Therefore, in the adjoint section this marks the beginning of the adjoint persistent communication. We allocate the adjoint buffer here and call MPI_Send/Recv_init depending on the opcode (Send or Recv) that was saved during the forward section.

Wait. The Wait marked the end of a particular communication where the buffer was either received or sent. That way we will start here the adjoint communication with interchanged source and target. All this information was saved in the forward section, conveyed to the MPI_Wait through additional information stored in the request (see non-blocking communication) at the MPI_Start.

Start. MPI_Start marked the start of actual data communication. In the adjoint case this amounts to a Wait. We have to make sure, that the adjoints have arrived at that point, since they may be read from now on.

Init. By symmetry, the MPI_Send/Recv_init marks the end of the the adjoint communication. We may release the requests with an MPI_Request_free.

4.2 Collective Communication

For the reduction of residuals and certain simulation parameters, several reduction operations are involved in PETSc. They are all of the Allreduce kind where the result is accessible on all the processes. The operations are limited to MPI_MAX, MPI_SUM.

Maximum Operation

Value	Adjoint
$(y, m) = \max_{i=1}^{p} x_i$, with $x_m = y$	$\bar{x}_m = \bar{y};\ \bar{y} = 0$

For the maximum operation we need to save the process rank m of the element that is the maximum. Only this element has a non zero partial derivative with respect to the output. Hence, in the reverse section the adjoint of the result on all processes is summed up and only sent to the process that had the maximum value.

Sum Operation

Value	Adjoint
$y = \sum_{i=1}^{p} x_i$	$\bar{x}_i{+} = \bar{y};\ \bar{y} = 0$

In case of a sum, the adjoint has to be propagated to all processes involved. The adjoint communication amounts to a broadcast.

5 Results

As has been mentioned before, this paper is meant as a proof of concept that fully discrete adjoints of PETSc are possible with the available tools. However, a conclusion on the performance of discrete adjoint must be dismissed at this time. To understand why, we provide the following benchmarks made on the RWTH Aachen University cluster. All tests were conducted on the MPI nodes of the Bull HPC-Cluster. They are 2-socket systems equipped with Intel Westmere EP processors with 24GB memory each.

	Memory Usage	Original RT (s)	Adjoint Mode RT (s)
1 Process	16GB	0.22	3.8
2 Processes	8GB	0.14	3.45
4 Processes	4GB	0.09	2.3

With a grid of only 128x128 we reach a memory usage of 16GB in adjoint mode. The time it takes to trace the forward section as well as the adjoint computation is 3.8s. The problem size is a hard limit. However, with a runtime of only 3.8s seconds we are far from any real world application. Increasing the number of processes for the 128x128 to more than 4 is not reasonable either, since the overhead due the communication takes its toll. Therefore we are quite limited in the possible benchmarks. In the end they should prove that there is some speedup and that our method is on the right track.

To remedy the situation, one has two choices. Either apply a checkpointing scheme [16] or replace the computationally most expensive part, namely the linear solver. Replacing the linear solver with a continuous computation of the adjoints similar to the BLAS routines in this paper will considerably reduce the memory hit. This is the next step outlined in the coming outlook

6 Summary

We proved that the combination of `dco/c++` and Adjoint MPI is robust enough to compute semi-automatic discrete adjoints of PETSc. Four distinct steps were necessary. First, `dco/c++` had to be applied to PETSc by overloading all variables of type PetscReal and PetscScalar. Second, BLAS had to be continuously adjoined by writing adjoint BLAS functions by hand. Third, the LAPACK routines were adjoined using again `dco/c++`. Finally, the adjoint MPI library was used to adjoin all the MPI communication.

7 Outlook

While relying on AD tools, discrete adjoints are the straightforward way of adjoining a given code. However, they do not always match the desired derivative information of the simulated phenomenon. They only represent a differentiated algorithm that models a given phenomenon. Hence, applying for example a continuously differentiated linear solver in PETSc might yield different results and runtime behaviour. However these results may better fit the actual simulated phenomenon. This requires formulating the adjoint linear system that computes the adjoints of the original linear system. Unfortunately, this requires considerable changes to the PETSc code base, but definitely worth further investigations. For the time being only the BLAS routines are adjoined continuously.

The current case study should be superseded by a real world application in the future. The driving application behind this project are discrete adjoints of PADGE, an adaptive discontinuous Galerkin solver for 3D turbulent flow developed by the German Aerospace Center (DLR).

References

1. Balay, S., Brown, J., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page (2012), `http://www.mcs.anl.gov/petsc`
2. Naumann, U.: The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation. Software, Environments, and Tools. SIAM, Philadelphia (2012)
3. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4, 2009), `http://www.mpi-forum.org` (December 2009)
4. Hovland, P.D., McInnes, L.C.: Parallel Simulation of Compressible Flow using Automatic Differentiation and PETSc. Parallel Computing 27(4), 503–519 (2001); Parallel Computing in Aerospace
5. Sagebaum, M., Gauger, N.R., Naumann, U., Lotz, J., Leppkes, K.: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries. Technical Report AIB-2013-04, RWTH Aachen (February 2013)
6. Averick, B.M., Carter, R.G., Mor, J.J.: The Minpack-2 Test Problem Collection (1991)
7. Ungermann, J., Blank, J., Lotz, J., Leppkes, K., Hoffmann, L., Guggenmoser, T., Kaufmann, M., Preusse, P., Naumann, U., Riese, M.: A 3-D Tomographic Retrieval Approach with Advection Compensation for the Air-Borne Limb-Imager GLORIA. Atmos. Meas. Tech. 4(11), 2509–2529 (2011)
8. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic Linear Algebra Subprograms for Fortran Usage. ACM Transactions on Mathematical Software (TOMS) 5(3), 308–323 (1979)
9. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S.: Lapack users' guide, release 2.0, vol. 326, p. 327. SIAM, Philadelphia (1995)
10. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA projects. Journal of Physics: Conference Series 180, 012037 (2009)
11. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press (1994)
12. Hovland, P., Bischof, C.: Automatic Differentiation for Message-Passing Parallel Programs. In: IPPS 1998: Proceedings of the 12th International Parallel Processing Symposium on International Parallel Processing Symposium. IEEE Computer Society, Washington, DC (1998)

13. Schanen, M., Naumann, U.: A Wish List for Efficient Adjoints of One-Sided MPI Communication. In: Träff, J.L., Benkner, S., Dongarra, J.J. (eds.) EuroMPI 2012. LNCS, vol. 7490, pp. 248–257. Springer, Heidelberg (2012)
14. Schanen, M., Naumann, U., Hascoët, L., Utke, J.: Interpretative Adjoints for Numerical Simulation Codes using MPI. Procedia Computer Science 1(1), 1819 –1827 (2010); ICCS 2010
15. Utke, J., Hascoët, L., Heimbach, P., Hill, C., Hovland, P., Naumann, U.: Toward Adjoinable MPI. In: Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium. IEEE Computer Society, Washington, DC (2009)
16. Griewank, A., Walther, A.: Algorithm 799: Revolve: An Implementation of Checkpoint for the Reverse or Adjoint Mode of Computational Differentiation. ACM Transactions on Mathematical Software 26(1), 19–45 (2000)

Evaluation of Two Formulations of the Conjugate Gradients Method with Transactional Memory

Martin Schindewolf[1], Björn Rocker[2], Wolfgang Karl[1], and Vincent Heuveline[3]

[1] Karlsruhe Institute of Technology (KIT), Chair for Computer Architecture and Parallel Processing, Haid-und-Neu-Straße 7, 76131 Karlsruhe, Germany
{schindewolf,karl}@kit.edu
[2] Robert Bosch GmbH, Corporate Sector Research and Advance Engineering, Robert-Bosch-Platz 1, 70839 Gerlingen-Schillerhöhe, Germany
bjoern.rocker@de.bosch.com
[3] Karlsruhe Institute of Technology (KIT), Engineering Mathematics and Computing Lab (EMCL), Fritz-Erler-Str. 23, 76133 Karlsruhe, Germany
vincent.heuveline@kit.edu

Abstract. Transactional Memory (TM) offers new possibilities for algorithmic design. This paper evaluates TM implementations of two algorithmic variations of the wide-spread conjugate gradients method (CG) regarding their performance on multi-core CPUs employing TM. Through applying tools for TM that visualize the TM application behavior, we show that the main bottleneck for both is the waiting times at barriers and illustrate the implementation of reduction operations with TM in a beneficial way. Performance monitoring through using the PAPI interface uncovers the quantity and type of instructions that each algorithms requires. This basic work is the key for environment-aware numerics as well as a hint for software developers who plan to use TM.

1 Motivation through Previous Work

Transactional Memory (TM) has been proposed to facilitate the synchronization of multiple threads in a parallel shared memory program and promises performance gains through optimistic concurrency. In previous work [8], we investigated the possibility to apply Software Transactional Memory to the method of Conjugate Gradients (CG) formulated according to Saad's algorithm [11] without preconditioning. The method of Conjugate Gradients is a solver for linear systems of equations that is frequently used for problems in the area of structural mechanics and computational fluid dynamics. Due to its relevance, we investigate optimization opportunities through an in-depth analysis of the TM application's behavior and explore methods for its optimization in this paper.

Previous experiences with hardware TM systems show that transactions that include more shared memory updates show a better performance in case the contention between transactions is low [12]. Since contention has been low in previous experiments, we searched for a formulation of the CG algorithm in the literature that enables larger transaction sizes to transfer the optimization strategy from HTM to STM and found pipelined CG [10,15].

In this paper we demonstrate the implementation of the pipelined CG algorithm with TM and other OpenMP-based synchronization primitives to evaluate and compare these variants and illustrate the run time behavior in a post-processing step. In order to achieve this, we apply the components of a framework for the Visualization and Optimization of TM Applications (VisOTMA) to the resulting pipelined CG variant and its previous version in order to compare its performance, convergence behavior, and utilization of the microarchitecture. Compared with related work in tools for Transactional Memory applications, our approach targets the `C` programming language and complements TM events with readings of performance counters through the use of the PAPI interface [16]. Through this additional information, we reveal the cause that restricts performance with TM and the pipelined CG variant whereas a comparison with the previous version of CG shows the main differences in utilization of the microarchitecture.

2 Pipelined Conjugate Gradient Solver with OpenMP

Inspired by Meurant's algorithm [10], Strzodka and Göddeke refine the pipelined Conjugate Gradient solver to enable mixed precision and pipelined algorithms that accurately solve partial differential equations with low precision components on FPGAs [15]. From these collection of proposed algorithmic variants of the conjugate gradient method, we select the basic pipelined CG variant with three reduction operations that should be combined in one transaction. The idea behind the pipelined CG is that all computations on vector elements should be done in parallel. With this rearrangement, it becomes feasible to stream a vector instead of having to store all elements of the vector. First, Strzodka and Göddeke reorder all vector operations so that these can be performed in parallel [15]:

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha_k \mathbf{p}_k, \tag{1}$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{q}_k, \tag{2}$$

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k, \tag{3}$$

$$\mathbf{q}_{k+1} = \mathrm{A}\mathbf{p}_{k+1}. \tag{4}$$

The main contribution of this algorithm is to eliminate the requirement to compute all elements of the vector $\mathbf{r}_{k+1}$ in order to compute $\mathbf{p}_{k+1}$. Strzodka and Göddeke lift this restriction through introducing $\sigma_k = \rho_{k+1}$ that does not require knowledge of $\mathbf{r}_{k+1}$: $\sigma_k = \alpha_k(\alpha_k \mathbf{q}_k \cdot \mathbf{q}_k - \mathbf{p}_k \cdot \mathbf{q}_k)$. Then the scalar products are computed after the reordered vector operations shown in Equation 1:

$$\rho_{k+1} = \mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}, \tag{5}$$

$$\alpha_{k+1} = \frac{\rho_{k+1}}{\mathbf{p}_{k+1} \cdot \mathbf{q}_{k+1}}, \tag{6}$$

$$\sigma_{k+1} = \alpha_{k+1}(\alpha_{k+1}\mathbf{q}_{k+1} \cdot \mathbf{q}_{k+1} - \mathbf{p}_{k+1} \cdot \mathbf{q}_{k+1}), \tag{7}$$

$$\beta_{k+1} = \frac{\sigma_{k+1}}{\rho_{k+1}}. \tag{8}$$

This variant is useful in the case of a sparse matrix A that enables to compute one step of the algorithm in a fully pipelined fashion. The pipelined CG variant is suited in case the matrix A is sparse and does not require global communication. Therefore, the pipelined CG is e.g., applicable for solving the stationary heat equation without heat source.

The main loop of the pipelined CG with OpenMP iterates until $|\mathbf{r_{k+1}}| <= \epsilon$ being the convergence criteria for the algorithm. In the following, we will discuss the mapping of the algorithm from the Equation 1 and Equation 5 to this implementation. First, $\mathbf{u}_{k+1}$ and $\mathbf{r}_{k+1}$ are both computed according to Equation 1 and Equation 2 in an OpenMP `for` loop. Then, we compute p_{k+1} as described in Equation 3 and reset the vector $\mathbf{q}$. The sparse matrix multiplication, involving A and $\mathbf{p}$, takes place according to Equation 4. The implementation resets the scalar variables and performs three reductions to compute the scalar products $\mathbf{r_{k+1}} \cdot \mathbf{r_{k+1}}$, $\mathbf{p_{k+1}} \cdot \mathbf{q_{k+1}}$, $\mathbf{q_{k+1}} \cdot \mathbf{q_{k+1}}$. In comparison with the CG according to Saad [11], that demanded two separate reductions, the computation with pipelined CG requires three reductions. The advantage is that one enlarged critical section or transaction embraces all three of them. These three reductions implement the vector operations of Equation 5, 6, and 7. Please note that all of the above steps except resetting the scalar variables are performed in parallel. Computing Equation 8 again requires to serialize the execution and compute the values of $\alpha_{k+1}, \sigma_{k+1}$, and β_{k+1}. Finally we increment the number of iterations and compute the norm of r_{k+1}. The result is compared with ϵ in the `while` statement. In case the norm of r_{k+1} already satisfies the condition, the implementation will output the result and also compute the error. Otherwise, the algorithm performs another iteration of the loop until reaching convergence.

Our implementation uses OpenMP pragmas to mark parallel for loops and the single directive to implement the algorithm as described before. Note that our approach does not take advantage of the fact that pipelined CG supports the streaming of a vector. Instead our approach aims to implement the reduction, that can now be made three times larger than before, assuming a constant vector size. This different reduction pattern enables us to use larger transactions (or critical sections) and to implement them in two different ways. These different ways of implementing the reduction pattern are compared and analyzed in the following. The *Reduction* case uses the OpenMP reduction concatenating three reductions in one pragma: `#pragma omp for reduction(+:rho) reduction(+:qq) reduction(+:pq) schedule(static)`. The three reductions are implemented in three ways: *Fast*, *Slow Long*, and *Slow Short*. *Fast* executes the accumulation with a thread-local variable over a thread private part of the vector. After finishing this calculation, each thread performs one update to add the thread-local variable (e.g., `pq_priv`) to the shared memory one (e.g., `pq`). Thus, contention between threads stems from a single update of the shared memory variable. In the following *th* represents the number of parallel threads and *dim* the dimension. Regardless of *dim* the *Fast* pattern leads to *th* updates of the shared variable which is a huge gain compared to the *Slow* which updates the shared variable directly with each computation. Of course, the complexity to implement the *Fast* pattern is slightly higher. The *Fast* version of pipelined CG updates all three shared memory variables in one transaction/critical section. This enlarges the size of the transaction because instead of one update with normal CG for each of the reduction, there are three updates in pipelined CG. *Slow Long* updates the three shared memory locations in one transaction or critical section and does not use a thread-local variable for storing intermediate results. *Slow Short* also does not use thread-local variables to store intermediate results and performs each update of a shared memory location in a dedicated transaction or critical section. Thus, both *Slow* variants require the same amount of updates of the shared variable and differ only in the granularity of the applied synchronization mechanism. *Slow* updates the shared variable *dim* times. If the work is distributed evenly among *th* threads, each threads performs $\frac{dim}{th}$ updates. For $dim \gg th$ this pattern creates high contention on the shared variable because each thread accesses it multiple times. In a multi-core system this will result in coherency traffic that will invalidate the datum in the other caches, leading to performance loss. With TM, this leads to an increasing number of conflicts and, hence, rollbacks. For OpenMP atomic, the *Fast* version uses thread-local variables whereas the *Slow* version does not. If possible `omp atomic` maps to a native atomic instruction that updates one memory location without being interrupted by other processors. This atomicity is limited to one memory location and can not be extended. Thus, the *Atomic Fast* uses the thread-local variables to update

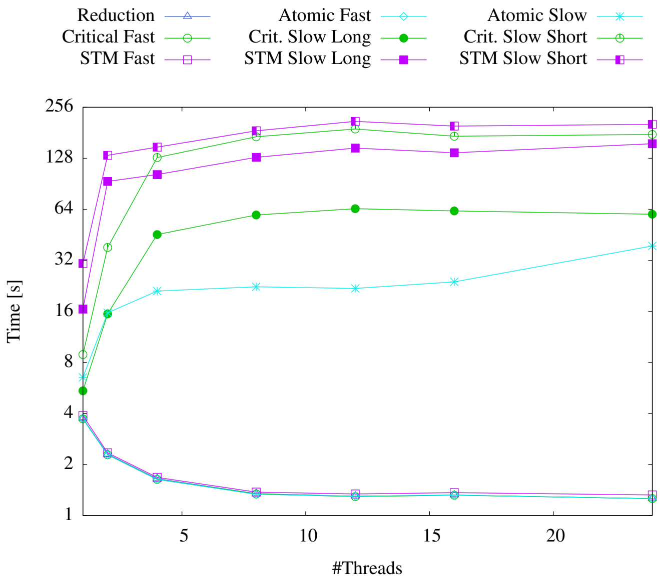


Fig. 1. Execution time of pipelined CG with 1 to 24 threads on W1. W1 is a two socket Westmere system with two Intel(R) Xeon(R) CPUs X5670 each with six cores, two way hyperthreading, and 2.93 GHz. Hence the total number of hardware threads is 24 and the upper limit for our exeriments.

a shared memory location and *Atomic Slow* updates a shared memory location for each new value. Since each value must be updated with a separate atomic instruction, there is no need to distinguish between long and short sections.

The execution time of the pipelined CG method is depicted in Figure 1. The y-axis holds the average execution time in seconds over 17 runs and the x-axis the number of threads. Again, the *Fast* versions of *Atomic*, *Critical*, and *Reduction* show a nearly identical behavior and are hard to distinguish. *Fast STM* also has a slightly higher execution time than e.g., *Reduction*. The ranking of the slow variants is as follows: *Atomic Slow Long*, *Critical Slow Long*, *STM Slow Long*, *Critical Slow Short*, *STM Slow Short*. Again, neither slow variant achieves the run time of the respective single thread. Thus, all slow variants show a slowdown for the execution with more than two threads.

The interesting insight is that neither of the short variants (STM or critical) performed as good as or better than a long variant. Thus, enlarging the granularity of critical sections under the given conditions results in a better performance, but the only way to achieve a speedup is the use of thread-local variables that avoid frequent updates of the shared memory and hereby reduce contention for shared locations.

2.1 Comparison of CG and Pipelined CG

In order to compare normal CG and pipelined CG we employ them to solve the one dimensional stationary heat equation without heat source which has been discretized by using finite differences with a 3-point stencil.

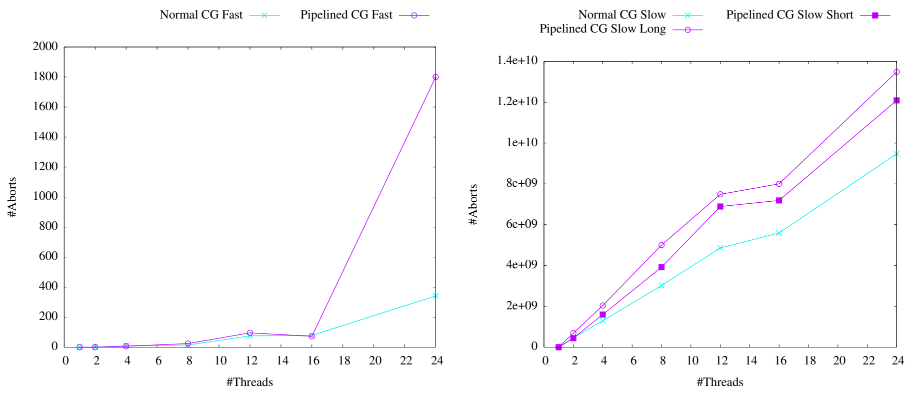
Table 1. Parameters for example problem solved with two implementations of the CG method

Dimension	Epsilon	Start vector	Solution
$5 * 10^6$	$1 * 10^{-13}$	$\mathbf{0}$	$\mathbf{1}$

The key parameters for the following experiments are shown in Table 1. Both CG variants execute a loop that iterates over the numerical algorithm. Each iteration refines the current solution and herewith reduces the error ($e_k = \|u_{sol} - u_k\|$). Usually, the error cannot be computed since the exact solution is unknown. In our experiments we have chosen the right hand side of the problem Au=b according to the formula $b_j = \sum_{i=1}^{n} a_{j,i}$ with $A \in \mathbb{R}^{n \times n}$, $b, u \in \mathbb{R}^n$ and $n \in \mathbb{N}$. As predicted by the theory, the error decreases monotonically. This is not sufficient to guarantee that both formulations of the CG work correct, but it is a strong indicator. Hence, we check the computed solution against the known solution to verify that both CG versions reach the correct result.

In order to compare a more realistic scenario, we have chosen an absolute stopping criteria for the residual in our experiments. The algorithm iterates as long as the residual $\|b - Au_k\|$ is greater than a given epsilon.

In practice the convergence may be perturbed through round-off errors that affect the numerical stability. Whether an algorithm is suited to find a solution to a given problem also depends on the algorithmic details as well as the implementation. Thus, a different formulation of the same algorithm may show a different convergence behavior. Moreover, even implementation details such as the order of elements when summing up a vector, may have an impact on the convergence behavior. Thus, the impact of new technologies, like TM, on the convergence behavior has to be researched thoroughly. For more experiments with CG see [8]. We implement both CG variants in the programming language `C` and parallelize them, as described earlier, with OpenMP and the described synchronization mechanisms. GCC in version 4.6.1 generates both executables with the compiler options `-fopenmp -O3 -g3` that affect performance.



(a) Aborts with *Fast* versions of normal and pipelined CG

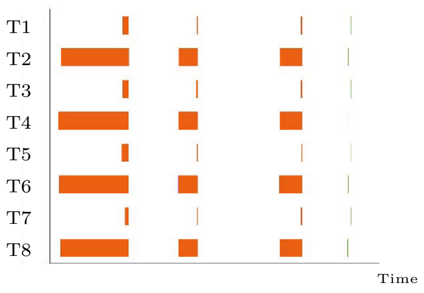
(b) Aborts in *Slow* versions of normal and pipelined CG

Fig. 2. Aborts with normal and pipelined CG with the number of threads ranging from 1 to 24 on W1

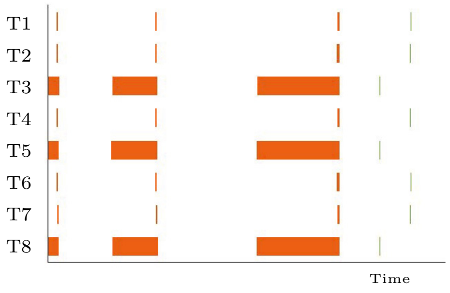
Software Transactional Memory Characteristics – The transactional characteristics of both CG variants are discussed first because these may dominate the utilization of architectural resources. For example a transaction that performs mainly integer operations and aborts frequently and, thus, repeats these operations multiple times contributes a larger share of integer operations than a transaction that successfully commits. Hence, large abort rates may change the utilization of the functional units. Figure 2 depicts the absolute number of aborted transactions of all threads for the *Fast* (left hand side) and the *Slow* variants (right hand side) of normal CG and pipelined CG. All of the presented numbers are averages over 17 runs.

For the *Fast* version, illustrated in Figure 2(a), the number of aborts is below 100 for up to 16 threads and normal CG and pipelined CG. With 24 threads, it rises to ≈ 340 for normal CG and ≈ 1800 for pipelined CG. Here, this is the only configuration for the *Fast* versions where normal CG has significantly less aborts than pipelined CG. This is remarkable because pipelined CG executes three times the number of loads and stores per transaction and herewith should have a higher probability of conflict. The fact that all of these transactions access the same three variables in the same order leads to a scenario where a transaction will conflict with another transaction if they both run at the exact same time. As pipelined CG requires more time to execute the longer transactions, this increases the conflict probability because a longer time inside a transaction although means a longer time in which a second thread may start a transaction and conflict with the former. This effect is dominating only at 24 threads because prior to that, both versions of CG perform equal. The relative abort rate for *Fast* with 8 threads is ≈ 3.5 % for normal CG and ≈ 6 % for pipelined CG.

Figure 2(b) demonstrates the reason for the missing performance with the *Slow* variants. For only 2 threads, normal CG already has $\approx 460*10^6$ aborts. For pipelined CG, the aborts for 2 threads are $\approx 440*10^6$ for the short and $\approx 687*10^6$ for the long variant. The reason for these high aborts are the transactions that update a single variable (or in the best case three variables) for each iteration of the loop. Due to these high abort numbers, the threads will not make progress and, hence have long execution times with the *Slow* variants.



(a) Visualization of normal CG

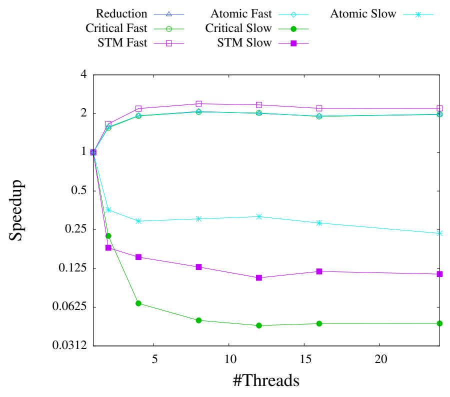


(b) Visualization of pipelined CG

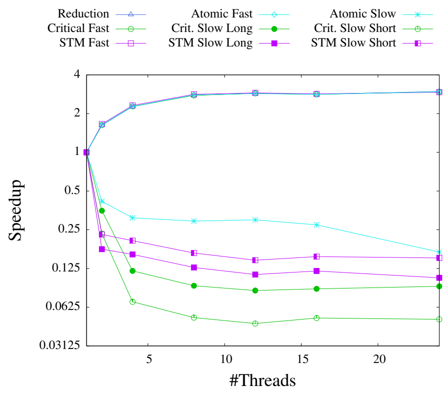
Fig. 3. Visualization of normal and pipelined CG with 8 threads on W1. Zoomed to relate transaction time (in green) with barrier wait time (in orange).

After studying the transactional characteristics in the previous paragraph, we would like to demonstrate the additional values and the flexibility of the VisOTMA framework by doing an in-depth analysis of the *Fast* versions of normal CG and pipelined CG. When first visualizing the TM application behavior with Paraver, we found many gaps between extremely small transactions. Thus, only a high zoom level would allow us to find the aborted transactions. After investigating these cases and finding that the overall TM performance for *Fast* is good (also cf. to previous paragraph), we decided to focus on the blank spots between the transactions. A code study reveals that, apart from computation, OpenMP constructs are most likely to consume the missing time in between the transactions. Both CG variants comprise 5 OpenMP `for` loops. By default these `for` loops come with an implicit barrier at the end of the execution. Thus, the fastest thread waits for the slowest one to complete its work and reach the barrier. OpenMP enables the programmer to specify the `nowait` clause to omit this barrier [4]. On the other hand, there is also the explicit `#pragma omp barrier` construct that produces a barrier. These manifold possibilities to generate barriers in OpenMP code and the importance for CG code convinced us to investigate the barrier wait times to relate these to the transaction execution times. We obtain the information about TM events through a tracing machinery [13] that features low intrusiveness and also access to hardware performance counters through PAPI [16]. The information is analyzed in a post-processing step and transformed into traces for the Paraver[1] tool for visualization. In our particular setup using the GCC compiler, `libgomp` is the OpenMP run time system and GCC does the expansion of OpenMP pragmas and the outlining of functions. In order to implement timed barriers, we needed to intercept the call to the original barrier function with one that would record the cycle counter of the processor on entry and exit of the barrier. These readings are then written to a thread-local trace file. Using the timed instead of the regular barriers is achieved through a simple replacement on assembly level. Through simply replacing the call to *GOMP_barrier* with a call to *ote_GOMP_barrier*, we achieved the desired functionality. Thus, *ote_GOMP_barrier* records the cycle counter before and after calling *GOMP_barrier*. Separate additional trace files for tracing these barriers are necessary because barriers are independent of executing transactions and the STM may not be initialized when calling a barrier. Thus, a post-processing step merges the barrier traces with the TM traces. Both trace files have the same time base and, hence, correlate in time. The visualization of the merged traces requires to register the new events at the various processing stages, but is straightforward. Figure 3 shows results of this effort for normal CG and pipelined. The picture is a timeline view of barriers and transactions executed by 8 threads. The threads, denoted with `T1` to `T8`, each occupy a slot on the y-axis. The x-axis shows the progress of time. The orange bars demonstrate the wait time of a particular thread at a barrier. These orange bars dominate Figure 3(a). Green bars illustrate how much time the execution of a transaction with a commit takes. These bars are present on the right hand side of Figure 3(a) and are extremely small. Figure 3(b) illustrates the run time behavior of the pipelined CG variant that exercises a similar execution pattern. Again, transaction times are hardly visible although these transactions have three times the amount of loads and stores of those transactions in normal CG. Additionally, we discover that pipelined CG shows a small perturbation that influences the start time of the transactions. In Figure 3(a) with normal CG all threads start their transaction at almost exactly the same point in time, whereas Figure 3(b) reveals that three threads start executing the transaction before all other threads. This behavior of pipelined CG is likely the cause for a better conflict rate than expected. Surprisingly, both figures highlight that the wait times at the barriers (colored in orange) exceed the execution time of a transaction (shown in green). These findings not only motivate research to avoid or omit these barriers, but also show that in a very regular setting, such as with the CG algorithm, TM can not show its strong side because the effects of optimistic concurrency, which enable some threads to proceed faster than others, are potentially turned into wait times at the barriers, waiting for the slowest thread that has been aborted to enable the progress of the fast threads.

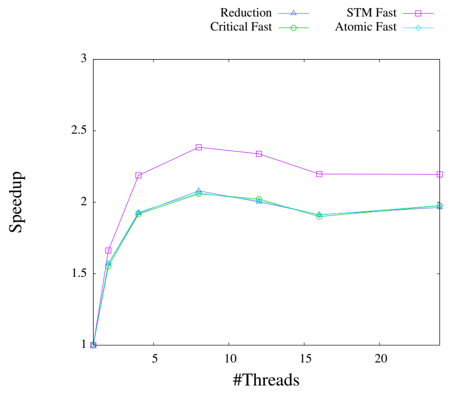
[1] Paraver Website `http://www.bsc.es/paraver`



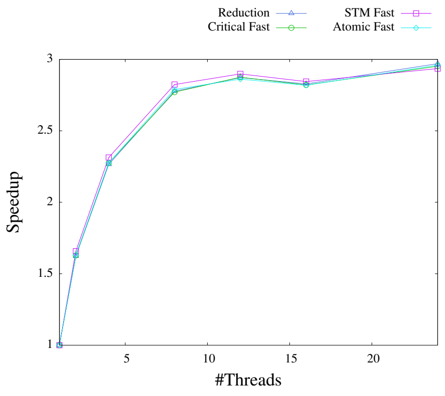
(a) Speedup of normal CG



(b) Speedup of pipelined CG



(c) Speedup of normal CG for fast versions only.

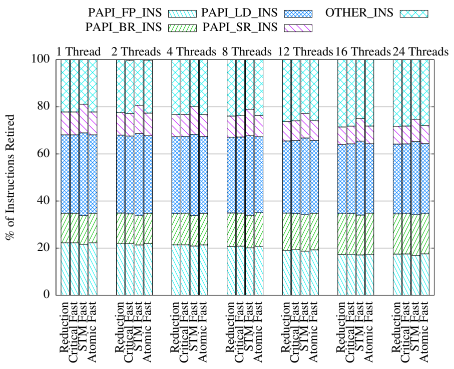


(d) Speedup of pipelined CG for fast versions only.

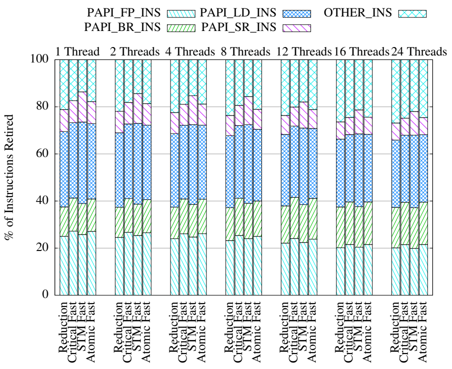
Fig. 4. Speedup with normal and pipelined CG with the number of threads ranging from 1 to 24 on W1

Speedup – We compare the achieved speedup of normal CG with that of pipelined CG. The speedup is computed according to $S(n) = \frac{T(1)}{T(n)}$, where $T(n)$ denotes the execution time with n threads and $T(1)$ is the respective single thread execution time (cf. to [7]). Often T_{seq} is used instead of $T(1)$ with T_{seq} being the serial reference implementation that does not incur the overheads of a threaded implementation. In these cases, often $T_{seq} < T(1)$ holds so that the resulting $S(n)$ would be smaller. Figure 4(a) depicts the speedup for normal CG whereas Figure 4(b) shows it for pipelined CG (both times on the y-axis). The x-axis holds the number of threads. Although the plots of the runtimes from previous sections contain the same information, this plot more evidently shows a slow down (speedup $<$ 1) for the slow variants and a speedup for the fast variants. Setting the scale of the y-axis is a compromise to fit all variants on one plot. This makes identifying the maximum achieved speedup difficult because of the low resolution in this segment. Therefore, a second plot focuses on displaying the results of the fast variants only. Figure 4(c) and Figure 4(d) show the speedup with a linear scale on the y-axis. This plot illustrates that the achievable speedup over the respective single thread performance is higher with pipelined CG, achieving the highest speedup of 2.97 with the regular reduction and 24 threads. For normal CG, *STM Fast* with 8 threads achieves the best speedup of 2.38. The overhead of the single thread execution has a large influence on the reported speedups because a larger overhead (e.g., with STM) leads to a slower execution time. If the speedup is computed relative to this single thread execution time, this yields a higher speedup because the baseline is worse. This effect could be avoided by having a fixed serial execution time for all benchmarked variants. Here, this effect leads to the situation that *STM Fast* has a higher speedup for e.g., pipelined CG with 8 threads, but a higher execution time than e.g., *Reduction*.

Convergence Behavior – This paragraph presents the results of examining the convergence behavior of normal CG and pipelined CG applied to a problem that solves the stationary heat equation without heat source. The parameters setting is identical with the one that has been shown in Table 1. Both variants of CG show a consistent convergence behavior across all tested thread counts and synchronization mechanisms. Normal CG converges after 25 iterations to a solution that satisfies the criteria. Pipelined CG finds a solution to the problem that satisfies the convergence criteria after 26 iterations. Therefore, for the numerical problem solved in this experiment, the choice of the algorithmic variant has an impact on the convergence behavior. Pipelined CG needs to perform one additional iteration which is equal to a relative increase of computational complexity of 4 % for the considered problem.



(a) Normal CG



(b) Pipelined CG

Fig. 5. Breaking down the total amount of instructions in CG on W1

Which instruction type contributes the largest share? Figure 5 shows a breakdown of instructions retired into the measured type of instructions. The available types are: floating point instructions (denoted as PAPI_FP_INS), branch instructions (labeled PAPI_BR_INS), load and store instructions (PAPI_LD_INS and PAPI_SR_INS respectively) and remaining instruction with the label OTHER_-INS. Other instructions have not been measured, but computed as the difference from the measured ones with the remainder of the retired instructions (PAPI_TOT_INS). The Figure has a normalized y-axis that shows 100 % of the retired instructions. Each of the instruction types has a box that represents its share of the retired instructions. These bars are grouped according to the thread count and each group shows the used synchronization mechanisms (*Reduction*, *Critical Fast*, *STM Fast*, and *Atomic Fast*). The number of threads for each group is also found below the legend. Figure 5(a) shows the breakdown for normal CG and Figure 5(b) shows pipelined CG. For both the following trends can be derived: the share of floating point instructions decreases as the number of threads increases although the actual number of floating point instruction is constant. This is due to the fact that the number of other instructions increases as the number of threads increases. These additional instructions stem from the spawning/coordinating more threads. For normal CG and *Reduction* the share of FP instructions decreases from 22 % for 1 thread down to 17 % for 16 threads. Pipelined CG and *Reduction* yields similar numbers: the FP rate decreases from 25 % for 1 thread to 20 % for 16 threads. To summarize Figure 5, we conclude that loads and stores contribute the highest share to the retired instructions ($\approx 40\,\%$), floating point instructions contribute $\approx 20\,\%$ and branch instructions $\approx 15\,\%$ while other instructions contribute $\approx 25\,\%$ and become increasingly important with larger thread counts. The actual number of events varies depending on the synchronization variants, thread count and algorithmic choice, but the dominant instructions in our experiments have been loads and stores.

3 Related Work

Different possibilities to realize the concept of Transactional Memory have been proposed for Software, Hardware, or both [6]. The publications on tool support for TM are rare. For Software Transactional Memory profiling solutions have been shown for the programming languages `C#` [17], Haskell [14], `Java` [1], and `C` [9,2] and for Hardware Transactional Memory for the TCC architecture [3].

In particular, none of these approaches attempted to optimize and evaluate a numerical algorithm through selecting, implementing, and evaluating a differently formulated variant of the algorithm that promised a higher TM performance. Hence, this work with its insights into the run time behavior and utilization of the microarchitecture through the two CG variants advances the state-of-the-art and may inspire other researchers that face the problem of optimizing a numerical algorithms that uses/with TM.

4 Findings with Variants of CG and Outlook

Our first finding is that the right way of organizing the reductions is key to performance. A reduction implemented with direct updates of the shared variable, as seen in the *Slow* synchronization variants, will not yield a speedup over execution with one thread regardless of the synchronization primitive. Instead thread-local variables that hold intermediate results, as demonstrated with *Fast* synchronization variants, are a requirement to achieve speedups. Moreover, the pipelined CG with larger transactions is a strong competitor for normal CG because the number of aborts is modest up to 16 threads. As a downside, pipelined CG required one more iteration to achieve convergence compared with normal CG for our example case. For both CG variants, the wait time at the barriers dominates the time for synchronization in the reduction operations of the *Fast* variants. This does not only limit the gains of parallel execution but also masks the effects of optimizing the TM performance. The regular problem structure of CG demands that barriers synchronize all threads after a step in the loop. Thus, a thread that executes a transaction and forces another thread to abort and execute again, simply waits longer at the next barrier for the remaining threads. This basic scenario still holds for longer transactions with pipelined CG and pipelined CG achieves a higher speedup than normal CG. As a result, the CG algorithm is not suited to demonstrate a performance gain with STM so that a practical and generic implementation should use the OpenMP reduction for now. On the other hand, the competitive execution time of pipelined CG with larger transactions and still moderate contention confirms the basic idea of optimizing the TM behavior through employing larger transactions. Moreover, the large difference in execution time for transactions and barriers suggests that future research should target more efficient barrier synchronization or techniques to elide barriers. Common to both CG variants, we found that higher thread counts lead to more L2 cache misses that hinder the scalability and that loads and stores contribute the largest amount to all kinds of instructions retired.

Future work should integrate the profiling of transactions with an existing profiler for OpenMP applications, e.g., `ompP` [5], in order to complement the time spent in transactions and barriers with other OpenMP constructs such as parallel sections and thread create or destroy. These measurements would complement the performance analysis and a programmer could relate the overheads associated with STM to those of OpenMP in general.

References

1. Ansari, M., Jarvis, K., Kotselidis, C., Luján, M., Kirkham, C., Watson, I.: Profiling Transactional Memory Applications. In: PDP 2009, pp. 11–20. IEEE Computer Society, Washington, DC (2009)
2. Castro, M., Georgiev, K., Marangozova-Martin, V., Mehaut, J.F., Fernandes, L.G., Santana, M.: Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures. In: PDP 2011, pp. 199–206. IEEE Computer Society, Washington, DC (2011)

3. Chafi, H., Minh, C.C., McDonald, A., Carlstrom, B.D., Chung, J., Hammond, L., Kozyrakis, C., Olukotun, K.: TAPE: A Transactional Application Profiling Environment. In: ICS 2005, pp. 199–208. ACM, New York (2005)
4. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering 05(1), 46–55 (1998)
5. Fuerlinger, K.: OpenMP Profiling with OmpP. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1371–1379. Springer, US (2011)
6. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Synthesis Lectures on Computer Architecture, vol. 5. Morgan & Claypool Publishers (June 2010)
7. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 5th edn. Morgan Kaufmann Publ. Inc., San Franc. (2011)
8. Janko, S., Rocker, B., Schindewolf, M., Heuveline, V., Karl, W.: Transactional Memory, OpenMP and Pthreads applied to the CG Algorithm for solving the stationary Heat Equation. In: VECPAR. Springer (July 2012)
9. Lourenço, J., Dias, R., Luís, J., Rebelo, M., Pessanha, V.: Understanding the Behavior of Transactional Memory Applications. In: PADTAD 2009, pp. 3:1–3:9. ACM, New York (2009)
10. Meurant, G.: Multitasking the conjugate gradient method on the cray x-mp/48. Parallel Computing 5(3), 267–280 (1987)
11. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia (2003)
12. Schindewolf, M., Bihari, B., Gyllenhaal, J., Schulz, M., Wang, A., Karl, W.: What Scientific Applications Can Benefit from Hardware Transactional Memory? In: SC 2012, pp. 90:1–90:11. IEEE Computer Society Press, Los Alamitos (2012)
13. Schindewolf, M., Karl, W.: Capturing Transactional Memory Application's Behavior – The Prerequisite for Performance Analysis. In: Pankratius, V., Philippsen, M. (eds.) MSEPT 2012. LNCS, vol. 7303, pp. 30–41. Springer, Heidelberg (2012)
14. Sonmez, N., Cristal, A., Unsal, O., Harris, T., Valero, M.: Profiling Transactional Memory Applications on an atomic block basis: A Haskell case study. In: MULTIPROG 2009 (January 2009)
15. Strzodka, R., Göddeke, D.: Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. In: FCCM 2006, pp. 259–268. IEEE Computer Society Press (May 2006), doi:10.1109/FCCM.2006.57
16. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting Performance Data with PAPI-C. In: Tools for High Performance Computing 2009, pp. 157–173. Springer, Heidelberg (2010)
17. Zyulkyarov, F., Stipic, S., Harris, T., Unsal, O.S., Cristal, A., Hur, I., Valero, M.: Discovering and Understanding Performance Bottlenecks in Transactional Applications. In: PACT 2010, pp. 285–294. ACM, New York (2010)

Multifrontal QR Factorization for Multicore Architectures over Runtime Systems

Emmanuel Agullo[1], Alfredo Buttari[2], Abdou Guermouche[3], and Florent Lopez[4]

[1] INRIA - LaBRI, Bordeaux
[2] CNRS - IRIT, Toulouse
[3] Université de Bordeaux - LaBRI, Bordeaux
[4] Université Paul Sabatier - IRIT, Toulouse

Abstract. To face the advent of multicore processors and the ever increasing complexity of hardware architectures, programming models based on DAG parallelism regained popularity in the high performance, scientific computing community. Modern runtime systems offer a programming interface that complies with this paradigm and powerful engines for scheduling the tasks into which the application is decomposed. These tools have already proved their effectiveness on a number of dense linear algebra applications. This paper evaluates the usability of runtime systems for complex applications, namely, sparse matrix multifrontal factorizations which constitute extremely irregular workloads, with tasks of different granularities and characteristics and with a variable memory consumption. Experimental results on real-life matrices show that it is possible to achieve the same efficiency as with an *ad hoc* scheduler which relies on the knowledge of the algorithm. A detailed analysis shows the performance behavior of the resulting code and possible ways of improving the effectiveness of runtime systems.

Keywords: sparse matrices, multifrontal method, QR factorization, runtime systems, heterogeneous architectures.

1 Introduction

The increasing degree of parallelism and complexity of hardware architectures requires the High Performance Computing (HPC) community to develop more and more complex software. To achieve high levels of optimization and fully benefit of their potential, not only the related codes are heavily tuned for the considered architecture, but the software is furthermore often designed as a single whole that aims to cope with both the algorithmic and architectural needs. If this approach may indeed lead to extremely high performance, it is at the price of a tremendous development effort and a very poor maintainability: At which price in terms of code refactoring can we extend a shared-memory software to handle distributed memory machines if it has been assumed some contiguity properties on data in memory at the algorithmic level? How to extend the same software to handle accelerators efficiently if the numerical algorithm itself has been designed to match a regular data distribution?

Alternatively, a modular approach can be employed. First, the numerical algorithm is written at a high level independently of the hardware architecture as a Directed Acyclic Graph (DAG) of tasks where a vertex represents a task and an edge represents a dependency between tasks. A second layer is in charge of the scheduling. Based on the scheduling decisions, a runtime system takes care of performing the actual execution of the tasks, both ensuring that dependencies are satisfied at execution time and maintaining data consistency. The fourth layer consists of the tasks code optimized for the underlying architectures. In most cases, the last three layers need not be written by the application developer. Indeed, it usually exists a very competitive state-of-the-art generic scheduling algorithm (such as work-stealing [4], Minimum Completion Time [19]) matching the algorithmic needs to efficiently exploit the targeted architecture (otherwise, a new scheduling algorithm may be designed, which will in turn be likely to apply to a whole class of algorithms). The runtime system only needs to be extended once for each new architecture. Finally, most of the time, the high-level algorithm can be cast in terms of standard operations (such as BLAS in dense linear algebra) for which vendors provide optimized codes. All in all, with such a modular approach, only the high-level algorithm has to be specifically designed, which ensures a high productivity. The maintainability is also guaranteed since the use of new hardware only requires (in principle) third party effort.

The dense linear algebra community has strongly adopted such a modular approach over the past few years [10,17,1,8] and delivered subsequent production-level solvers. However, beyond this community, only few research efforts have been conducted to handle large scale codes. The main reason is that irregular problems are complex to design with a clear separation of the software layers without inducing performance loss. On the other hand, the runtime system community has strongly progressed, delivering very reliable and effective tools [6,7,14,5] up to the point that the OpenMP board is reconsidering its tasking model [1] with respect to that approach.

This paper evaluates the usability of runtime systems and of the associated modular approach in the context of complex applications, namely, the multifrontal QR factorization of sparse matrices [3], which yields extremely irregular workloads, with tasks of different granularities and characteristics as well as a variable memory consumption. For that, we consider a heavily hand-tuned state-of-the-art solver for multicore architectures, `qr_mumps` [9], we propose an alternative modular design of the solver on top of the StarPU runtime system [5] and we present a thorough performance comparison of both approaches on the architecture for which the original solver has been tuned. The penalty of delegating part of the task management system to a third party software, the runtime system, is to be regarded with respect to the impact of the numerical algorithmic choices; for that purpose, we also discuss the relative performance with respect to another state-of-the-art multifrontal QR solver for multicore architectures, the SuiteSparseQR package [11], referred to as `spqr`.

[1] `http://openmp.org/wp/presos/SC12/SC12_State_of_LC.2.pdf`

2 Multifrontal QR Factorization

The multifrontal method, introduced by Duff and Reid [12] as a method for the factorization of sparse, symmetric linear systems, can be adapted to the QR factorization of a sparse matrix thanks to the fact that the R factor of a matrix A and the Cholesky factor of the normal equation matrix A^TA share the same structure. As in the Cholesky case, the multifrontal QR factorization is based on the concept of *elimination tree* [18]. This graph, which has a number of nodes that is typically one order of magnitude or more smaller than the number of columns in the original matrix, expresses the dependencies among the computational tasks in the factorization: each node i of the tree is associated with k_i unknowns of A and represents an elimination step of the factorization. The coefficients of the corresponding k_i columns and all the other coefficients affected by their elimination are assembled together into a relatively small dense matrix, called *frontal matrix* or, simply, *front*, associated with the tree node. The multifrontal QR factorization consists in a tree traversal in a topological order (i.e., bottom-up) such that, at each node, two operations are performed. First, the frontal matrix is **assembled** by stacking the matrix rows associated with the k_i unknowns with uneliminated rows resulting from the processing of child nodes. Second, the k_i unknowns are eliminated through a **complete** QR **factorization** of the front; this produces k_i rows of the global R factor, a number of Householder reflectors that implicitly represent the global Q factor and a *contribution block* formed by the remaining rows and that will be assembled into the parent front together with the contribution blocks from all the front siblings. A detailed presentation of the multifrontal QR method, including the optimization techniques described above, can be found in Amestoy *et al.* [3].

The classical approach to the parallelization of the multifrontal QR factorization [3,11] consists in exploiting separately two distinct sources of concurrency: **tree** and **node parallelism**. The first stems from the fact that fronts in separate branches are independent and can thus be processed concurrently; the second from the fact that, if a front is big enough, multiple processes can be used to assemble and factorize it. The baseline of this work, instead, is the parallelization model proposed by Buttari [9] in the `qr_mumps` software which is based on the approach presented earlier in related work on dense matrix factorizations by Buttari *et al.* [10] and extended to the supernodal Cholesky factorization of sparse matrices by Hogg *et al.* [15]. In this approach, frontal matrices are partitioned into block-columns, which allows one to decompose the workload into fine-grained tasks. Each task corresponds to the execution of an elementary operation on a block-column or a front; five elementary operations are defined: 1) the **activation** of a front consists in computing its structure and allocating the associated memory, 2) **panel** factorization of a block-column, 3) **update** of a block-column with respect to a previous panel operation, 4) **assembly** of the piece of contribution block in a block-column in the parent front and 5) **cleanup** of a front which amounts to storing the factors aside and deallocating the memory allocated in the corresponding activation. These tasks are then arranged into a DAG where vertices represent tasks and edges the dependencies among them.

Figure 1 shows an example of how a simple elimination tree (on the left) can be transformed into a DAG (on the right); further details on this transition can be found in the paper by Buttari [9] from which this example was taken.

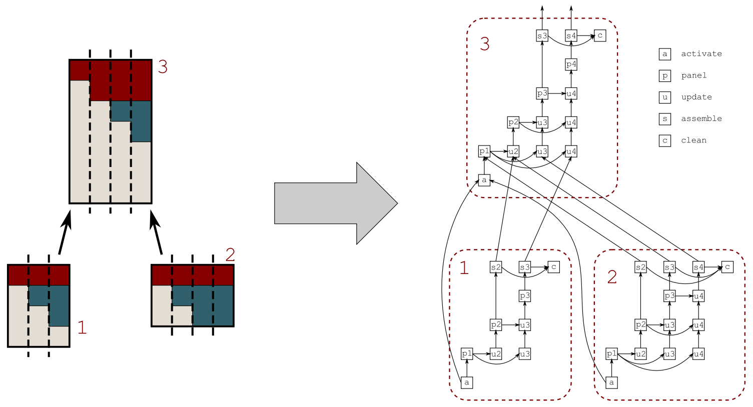


Fig. 1. An example of how a simple elimination tree with three nodes is transformed into a DAG in the `qr_mumps` code. Vertical, dashed lines show the partitioning of fronts into block-columns. Dashed-boxes group together all the tasks related to a front.

The execution of the tasks is guided by a dynamic scheduler which allows the tasks to work asynchronously. This approach is capable of achieving higher performance than the classical one thanks to the fact that tree and node types of parallelism are replaced by a single source, that is, DAG parallelism. This provides a higher amount of concurrency since dependencies are defined on a block-column basis rather than a front basis, which for instance allows one to start working on a front even if its children are not completely factorized. The execution mode, moreover, is more suited to multicore based architectures, as also shown in other related papers [10,15], because, unlike classical approaches [11,3], it does not suffer from the presence of heavy synchronizations.

3 The Task-Based StarPU Runtime System

As most modern task-based runtime systems, StarPU aims at performing the actual execution of the tasks, both ensuring that the DAG dependencies are satisfied at execution time and maintaining data consistency. The particularity of StarPU is that it was initially designed to write a program independently of the architecture and thus requires a strict separation of the different software layers: high-level algorithm, scheduling, runtime system, actual code of the tasks. We refer to Augonnet *et al.* [5] for the details and present here a simple example containing only the features relevant to this work. Assume we aim at executing the sequence $fun_1(\underline{x}, y)$; $fun_2(\ x)$; $fun_1(\underline{z}, w)$, where $fun_{i,i\in\{1,2\}}$ are functions applied on w, x, y, z data; the arguments corresponding to data which are modified submit_task(fun_1, $\underline{x}$, y, id=id_1)

submit_task(fun_2, x, id=id_2)

declare_dependency($id_3 \leftarrow id_1$)

submit_task(fun_1, $\underline{z}$, w, id=id_3)

$$\begin{tikzcd} & id_2 \\ id_1 \arrow[ur] \arrow[dr] & \\ & id_3 \end{tikzcd}$$

Fig. 2. Basic StarPU-like example (left) and associated DAG (right). Arguments corresponding to data that are modified by the function are underlined. The $id_1 \rightarrow id_2$ dependency is implicitly inferred with respect to the data hazard on x while the $id_1 \rightarrow id_3$ dependency is declared explicitly.

by a function are underlined. A task is defined as an instance of a function on a specific set of data. Because of possible data hazards [2] (here on x between fun_1 and fun_2), a so-called superscalar analysis [2] has to be performed to ensure that the parallelization does not violate dependencies. While CPUs implement such a superscalar analysis on chip at the instruction level [2], runtime systems implement it in a software layer on tasks. A task and the associated input/output data is declared with the **submit_task** instruction. This is a non blocking call that allows one to add a task to the current DAG and postpone its actual execution to the moment when its dependencies are satisfied. Although the API of a runtime system can be virtually reduced to this single instruction, it may be convenient in certain cases to explicitly define extra dependencies. For that, identification tags can be attached to the tasks at submission time and dependencies are declared between the related tags with the **declare_dependency** instruction. For instance, an extra dependency is defined between the first and the third task in Figure 2 (left). Figure 2 (right) shows the resulting DAG built (and executed) by the runtime system. Optionally, a priority value can be assigned to each task to guide the runtime system in case multiple tasks are ready for execution at a given moment. In StarPU, the scheduling system is clearly split from the core of the runtime system (data consistency engine and actual task execution). Therefore, not only all built-in scheduling policies can be applied to any high-level algorithm, but new scheduling strategies can be implemented without having to interfere with low-level technical details of the runtime system.

4 Multifrontal QR Factorization Based on StarPU

The execution of the `qr_mumps` software presented in Section 2 relies on a *ad hoc* scheduler which is extremely limited in features, relies on the knowledge of the algorithm and is, as a result, extremely lightweight. Replacing this scheduler with a complex, general purpose runtime system such as StarPU is not an easy task particularly because of several issues. First the DAG associated to the factorization of medium to large size matrices can have hundreds of thousands of tasks. Generating the whole DAG by submitting all the tasks to the runtime system may overload it and may require too much memory (see, for example, Lacoste *et al.* [16]). Second because of contribution blocks, different traversals of the DAG may result in a different memory consumption. For this reason, the activation tasks have to be carefully scheduled in order to avoid an excessive memory consumption. Third StarPU automatically infers dependencies among tasks depending on data hazards. Because, for what said above, it is not possible to allocate at once the memory needed for all the fronts in the tree, the whole DAG cannot be submitted entirely unless all the dependencies are explicitly provided to StarPU, which is largely unpractical.

The first and the third issue can be overcome by submitting tasks progressively by means of other tasks. Because activation tasks are responsible for allocating the memory of the associated frontal matrices, in our StarPU based implementation they will also be in charge of submitting the tasks for their assembly and factorization i.e., panel, update, assembly and cleanup; this is shown in Algorithm 1 *(right)*. The dependencies among these tasks can be automatically inferred by StarPU. Activation tasks, instead, are submitted all at once at the beginning of the factorization and their mutual dependencies explicitly specified to StarPU as shown in Algorithm 1 *(left)*; because they are limited in number, the runtime system will not be overloaded. As a result of this technique, the size of the DAG that the runtime system has to handle is only proportional to the number of active fronts.

Algorithm 1. Task management

Main code
(submit activation tasks):

```
1: for all $n$ in pre-computed post-order do
2:   for all children $c$ of node $n$ do
3:     declare_dependency($id_n \leftarrow id_c$)
4:   end for
5:   /* submit activation of front $f_n$ */
6:   submit_task(activation, $\underline{f_n}$, prio.=$-n$, id=$id_n$)
7: end for
```

Code of the activation task
(submit other tasks):

```
 1: allocate($f_n$)
 2: for all children $c$ of $n$ do
 3:   for all block-columns $b$ of $f_c$ do
 4:     /* submit assembly of $b$ inside $f_n$ */
 5:     submit_task(assembly, $b$, $\underline{f_n}$, prio.=3)
 6:   end for
 7:   submit_task(cleanup, $\underline{f_c}$, prio.=4)
 8: end for
 9:
10: for all block-columns $p$ in $f_n$ do
11:   /* submit panel factorization of $p$ */
12:   submit_task(panel, $\underline{p}$, prio.=2)
13:   for all block-columns $u > p$ in $f_n$ do
14:     /* submit update of $u$ wrt $p$ */
15:     submit_task(update, $p$, $\underline{u}$, prio.=1)
16:   end for
17: end for
```

The second issue, instead, can be overcome by conveniently assigning different priorities to the submitted tasks according to the idea that, as long as there is enough work to do on already activated fronts, no other front should be activated. This will keep the memory consumption under control while ensuring that there are always enough tasks for all the working threads. More precisely, each activation task is assigned a negative priority, whose value depends on a specific tree traversal order which, in our specific case, has been computed as the post-order which minimizes the memory consumption [13]. Cleanup tasks are given the highest priority because they are responsible for freeing the memory allocated by activation tasks. The other tasks, instead, are given a fixed priority which depends on the number of out-going edges in the associated DAG vertex in order to maximize the degree of concurrency; therefore, assemblies have higher priority than panels which, in turn, have higher priority than updates.

For the sake of simplicity, in Algorithm 1 we assumed that assembly operations read a single block-column b but modify an entire front f_n but in reality only a few block-columns of f_n are modified. It has to be noted that all the assembly operations at step 4 of Algorithm 1 *(right)* are independent from each other. In fact, even if multiple assemblies write on the same block-column of f_n, their modifications concern disjoint subsets of rows (not necessarily contiguous). This property is exploited by the `qr_mumps` scheduler, which was designed on purpose for this algorithm. StarPU, instead, will assume that these assemblies are dependent from each other. As a result, not only these operations cannot be performed in parallel but are forced to be executed in the same order as they have been submitted. As shown by the experimental results of Section 5, this may entail a slight performance loss.

5 Experimental Results

The native scheduler of the `qr_mumps` software was replaced with the StarPU runtime system according to the methods described in Section 4, leading to a software package that will be referred to as `qr_starpu`. This section aims at evaluating the effectiveness of the proposed techniques as well as the performance of the resulting code. For this purpose, the behavior of the `qr_starpu` code will be compared to the original `qr_mumps` one and also, briefly, to the SuiteSparseQR package (referred to as `spqr`) released by Tim Davis in 2009 [11].

Table 1. Matrices test set. The operation count is related to the matrix factorization with COLAMD column permutation.

#	Mat. name	m	n	nz	op. count (Gflops)
1	tp-6	142752	1014301	11537419	277.7
2	karted	46502	133115	1770349	279.9
3	EternityII_E	11077	262144	1572792	566.7
4	degme	185,501	659415	8127528	629.0
5	cat_ears_4_4	19020	44448	132888	786.4
6	Hirlam	1385270	452200	2713200	2401.3
7	e18	24617	38602	156466	3399.1
8	flower_7_4	27693	67593	202218	4261.1
9	Rucci1	1977885	109900	7791168	12768.1
10	sls	1748122	62729	6804304	22716.6
11	TF17	38132	48630	586218	38209.3

The experiments were conducted on a set of matrices from the the University of Florida Sparse Matrix Collection[2] presented in Table 1. The operation count is related to the factorization preceded by a COLAMD fill-reducing matrix permutation. The tests were run on the cache coherent Non Uniform Memory Access (ccNUMA) AMD Istanbul architecture equipped with 24 cores (6×4) clocked at

[2] `http://www.cise.ufl.edu/research/sparse/matrices`

2.4 GHz. The codes were compiled with the GNU v. 4.4 suite and linked to the Intel MKL sequential BLAS and LAPACK libraries. All the tests were run with real data in double precision.

Table 2 shows the factorization times (in seconds) for the matrices of the test set presented in Table 1 using `qr_starpu`, `qr_mumps` and `spqr` with different numbers of cores. Both `qr_mumps` and `qr_starpu` clearly outperform the `spqr`

Table 2. Factorization times, in seconds, on an AMD Istanbul system for `qr_starpu` (*top*), `qr_mumps` (*middle*) and `spqr` (*bottom*). The first row shows the matrix number.

		Factorization time (sec.)										
	Matrix	1	2	3	4	5	6	7	8	9	10	11
	th.											
	1	51.8	49.0	97.5	104.8	137.5	417.6	496.1	733.6	1931.0	3572.0	5417.0
`qr_starpu`	12	6.9	6.2	10.9	12.4	16.2	43.4	50.4	92.4	190.3	439.3	525.8
	24	5.7	4.4	8.0	8.5	12.4	28.1	32.9	58.0	122.7	336.3	305.9
	speedup	9.1	11.1	12.2	12.3	11.1	14.9	15.1	12.6	15.7	10.6	17.7
	1	51.5	48.8	96.9	104.6	137.1	410.8	495.2	729.7	1928.0	3571.0	5420.0
`qr_mumps`	12	5.7	5.2	10.2	10.8	14.2	39.5	46.6	69.4	177.9	392.3	479.0
	24	5.0	4.3	7.9	8.0	11.0	26.5	30.5	48.8	120.9	337.0	282.0
	speedup	10.3	11.3	12.3	13.1	12.5	15.5	16.2	14.9	15.9	10.6	19.2
	1	52.9	49.9	99.5	111.0	123.3	406.3	538.3	687.5	2081	4276	5361
`spqr`	12	17.0	14.5	26.3	33.0	32.5	85.7	90.5	131.6	468	1644	770
	24	17.0	12.3	20.7	26.2	27.8	68.6	74.1	114.2	372	1389	589
	speedup	3.1	4.0	4.8	4.2	4.4	5.9	7.3	6.0	5.6	3.1	9.1

package by a factor greater than two thanks to the powerful programming and execution paradigm based on DAG parallelism. On the other hand, `qr_starpu` is consistently but only marginally less efficient than `qr_mumps`, by a factor below 10% for eight out of eleven matrices and still only below 20% in the worst case. As a conclusion, the parallelization scheme impacts performance much more than the underlying low-level layer, validating the thesis that modular approaches based on runtime systems can compete with heavily hand-tuned codes.

Memory consumption is an extremely critical point to address when designing a sparse, direct solver. As the building blocks for designing a scheduling strategy on top of StarPU differ (and are more advanced) than what is available in `qr_mumps` (which relies on an *ad hoc* lightweight scheduler) we could not reproduce exactly the same scheduling strategy. Therefore we decided to give higher priority to reducing the memory consumption in `qr_starpu`. This cannot easily be achieved in `qr_mumps` because its native scheduler can only handle two levels of task priority; as a result, fronts are activated earlier in `qr_mumps`, almost consistently leading to a higher memory footprint as shown in Table 3. The table also shows that both `qr_starpu` and `qr_mumps` achieve on average the same memory consumption as `spqr`. On three cases out of eleven `spqr` achieves a significantly lower memory footprint; experimental results (not reported here) show that by constraining the scheduling `qr_starpu` and `qr_mumps` can achieve the same memory consumption as `spqr` while still being faster.

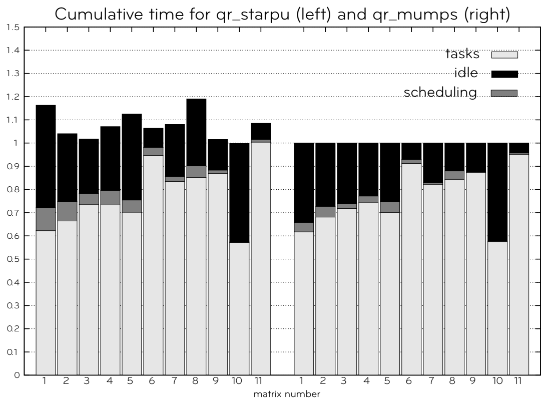


Fig. 3. Cumulative time for `qr_starpu` (*left*) and `qr_mumps` (*right*) on 24 cores. The `qr_starpu` timings are normalized with respect to the `qr_mumps` ones.

Table 3. Memory peak for the factorization of the test matrices on 24 cores

Mat.	Memory peak (MB)		
	`qr_starpu`	`qr_mumps`	`spqr`
1	1108.4	1426.5	1954.8
2	970.7	1016.5	1294.6
3	1315.1	1485.1	1344.5
4	1818.8	2040.2	2618.7
5	3111.2	3114.7	1965.6
6	4418.2	5300.9	7107.5
7	3845.1	3800.9	3535.5
8	10031.0	13608.7	6024.5
9	10070.8	8467.9	11898.7
10	54296.0	53636.8	62021.4
11	26212.7	26232.4	14424.9

In order to explain in more details the performance behavior of `qr_starpu` and `qr_mumps`, a detailed analysis of the execution times is shown in Figure 3 and Table 4. The figure shows the cumulative times spent by all threads in the three main phases of the execution of both solvers: the time spent in tasks, the time for scheduling the tasks (this includes computing the DAG in `qr_starpu`) and the idle time spent waiting for dependencies to be satisfied; these timings will be referred to as $t_c(p)$, $t_s(p)$ and $t_i(p)$, respectively, p being the number of threads (24 in Figure 3). The efficiency $e(p)$ of the parallelization can then be defined in terms of these cumulative timings as follows:

$$e(p) = \frac{t_c(1)}{t_c(p)+t_s(p)+t_i(p)} = \overbrace{\frac{t_c(1)}{t_c(p)}}^{e_l} \cdot \overbrace{\frac{t_c(p)}{t_c(p)+t_s(p)}}^{e_s} \cdot \overbrace{\frac{t_c(p)+t_s(p)}{t_c(p)+t_s(p)+t_i(p)}}^{e_p}.$$

This expression allows us to decompose the efficiency as the product of three well identified effects: e_l which measures the impact of data locality issues on the efficiency of the tasks, e_s which measures the cost of the scheduler management with respect to the actual work done and e_p which measures how well the tasks have been pipelined as a result of the scheduling decisions. Table 4 shows the corresponding values for our matrix collection.

The cumulative tasks times $t_c(1)$ (not reported here for the sake of space) are nearly identical for the two codes and stay the same when the number of threads increases as shown in Figure 3 and by the fact that the e_l values in Table 4 are comparable. The difference in the overall execution time can be explained by the higher overhead imposed by the runtime system management and by the idle time. The overhead imposed by StarPU is higher (inducing a lower efficiency e_s) because the dependencies between tasks are inferred based on the data access modes whereas in `qr_mumps` they are all defined explicitly based on the knowledge of the algorithm. However, the cost of the scheduling grows moderately with the number of threads and only accounts for a very small part of the overall execution time, especially for large matrices. The increased cumulative idle time, and the resulting lower pipeline e_p efficiency are due to two factors. First, the constraints imposed in `qr_starpu` to limit the memory consumption yield slightly lower concurrency (and, therefore, more idle time). Second, as explained in the previous section, assembly operations are serialized in `qr_starpu`; although these tasks only account for a small portion of the overall execution time, their serialization may induce delays in the pipeline that lead some threads to starvation. This second issue could be overcome by specifying to the runtime system that assembly tasks can be executed in any order and, possibly, in parallel, but this feature is not currently available in StarPU.

Finally, Table 5 shows the maximum number of tasks that the runtime system handles during the factorization versus the total number of tasks executed. The first

Table 4. Efficiency measures e_l, e_p, e_s and e for `qr_starpu` and `qr_mumps` ($e = e_l.e_p.e_s$)

						qr_starpu						
Matrix		1	2	3	4	5	6	7	8	9	10	11
	th.											
e_l	12	0.816	0.779	0.789	0.820	0.809	0.808	0.865	0.828	0.844	0.822	0.885
	24	0.711	0.680	0.669	0.730	0.694	0.666	0.768	0.689	0.733	0.752	0.774
e_p	12	0.860	0.856	0.906	0.877	0.876	0.976	0.918	0.797	0.958	0.802	0.952
	24	0.621	0.720	0.769	0.744	0.671	0.923	0.792	0.758	0.870	0.575	0.936
e_s	1	0.984	0.985	0.994	0.987	0.987	0.990	0.996	0.987	0.998	0.999	0.997
	12	0.915	0.930	0.970	0.951	0.953	0.974	0.977	0.966	0.991	0.997	0.993
	24	0.863	0.887	0.938	0.921	0.931	0.965	0.976	0.944	0.983	0.996	0.989
e	12	0.642	0.620	0.693	0.684	0.675	0.768	0.776	0.637	0.801	0.657	0.837
	24	0.381	0.434	0.482	0.500	0.433	0.593	0.594	0.493	0.627	0.431	0.716
						qr_mumps						
Matrix		1	2	3	4	5	6	7	8	9	10	11
	th.											
e_l	12	0.851	0.844	0.844	0.854	0.832	0.862	0.891	0.850	0.882	0.881	0.921
	24	0.711	0.661	0.678	0.716	0.690	0.688	0.780	0.695	0.727	0.737	0.808
e_p	12	0.915	0.915	0.898	0.936	0.922	0.977	0.937	0.985	0.963	0.812	0.992
	24	0.658	0.727	0.739	0.771	0.747	0.929	0.829	0.880	0.874	0.578	0.957
e_s	1	0.998	0.996	0.999	0.997	0.998	0.999	0.999	0.999	1.000	1.000	1.000
	12	0.949	0.973	0.989	0.985	0.963	0.977	0.995	0.982	0.997	0.998	0.997
	24	0.939	0.937	0.973	0.963	0.939	0.982	0.990	0.959	0.996	0.996	0.993
e	12	0.739	0.751	0.749	0.787	0.738	0.823	0.830	0.822	0.847	0.714	0.910
	24	0.439	0.450	0.487	0.532	0.484	0.628	0.640	0.586	0.633	0.424	0.768

Table 5. Maximum DAG size handled by StarPU during the factorization of the test matrices when using 24 threads

					DAG size						
Matrix	1	2	3	4	5	6	7	8	9	10	11
Max.	1640	1899	2023	2969	5063	4442	6965	12773	6846	11592	32978
Total	8610	10202	6058	14901	26579	49013	21192	136412	41023	33211	187101

being between 3 and 11 times smaller than the second, these data prove that the technique proposed in Section 4 is effective in reducing the runtime system overhead.

6 Conclusions and Future Work

The main objective of this work was to evaluate the usability and effectiveness of general-purpose runtime systems for parallelizing sparse factorization methods which constitute a complex and irregular workload. This was assessed implementing a new package software, `qr_starpu`, derived from `qr_mumps` by relying on the StarPU runtime system instead of the original *ad hoc* scheduler. Due to the original features of the considered algorithm, special attention had to be paid to the submission of tasks in order to contain the memory consumption, to limit the overhead imposed by the runtime system and to circumvent some limitations of StarPU (common to many other modern runtime environments). As a result, we managed to achieve an excellent memory behavior (even better than the original `qr_mumps` solver) and a very competitive performance, the overhead on elapsed time being most of the time below 10% and in any case never higher than 20%. A detailed analysis has revealed that this difference can be explained with a higher overhead imposed by the runtime system (which, however, only accounts for a very small part of the total execution time) and a more conservative scheduling of tasks to achieve a lower memory consumption.

All in all, the marginal performance loss conjugated with the excellent memory behavior show that general purpose runtime systems are very well suited for the parallelization of sparse direct methods. These powerful tools, moreover, provide several features that are likely to offer better performance and portability on architectures with higher core counts or equipped with accelerating devices (such as GPUs or MICs). These features has been evaluated in this paper and their usage is the object of ongoing research. At the same time, this document provides guidelines for the improvement of both sparse direct methods and runtime environments. Because, already on 24 cores, a considerable fraction of time is spent waiting for dependencies to be satisfied, it may be beneficial to adopt algorithms by tiles [10] for the processing of fronts in order to improve the amount of concurrency. Runtime systems, instead, can be improved by adding features that allow to cope with memory-consuming tasks or that allow to infer dependencies based on the access to memory that has not been allocated yet.

Acknowledgments. We thank Raymond Namyst and Samuel Thibault for their support with StarPU as well as Mathieu Faverge and Guillaume Sylvand for their constructive comments on a preliminary version of this document.

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. Journal of Physics: Conference Series 180(1), 012037 (2009)

2. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann (2002)
3. Amestoy, P.R., Duff, I.S., Puglisi, C.: Multifrontal QR factorization in a multiprocessor environment. Int. Journal of Num. Linear Alg. and Appl. 3(4), 275–300 (1996)
4. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory Comput. Syst. 34(2), 115–144 (2001)
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23, 187–198 (2011)
6. Badia, R.M., Herrero, J.R., Labarta, J., Pérez, J.M., Quintana-Ortí, E.S., Quintana-Ortí, G.: Parallelizing dense and banded linear algebra libraries using SMPSs. Concurrency and Computation: Practice and Experience 21(18), 2438–2456 (2009)
7. Bosilca, G., Bouteiller, A., Danalis, A., Hérault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed DAG engine for high performance computing. Parallel Computing 38(1-2), 37–51 (2012)
8. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Luszczek, P., Dongarra, J.: Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach. Scalable Computing and Communications: Theory and Practice (2013)
9. Buttari, A.: Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. To appear on the SIAM Journal on Scientific Computing (2013)
10. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Par. Comp. 35(1), 38–53 (2009)
11. Davis, T.A.: Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. ACM Trans. Math. Softw. 38(1), 8:1–8:22 (2011)
12. Duff, I.S., Reid, J.K.: The multifrontal solution of indefinite sparse symmetric linear systems. ACM Transactions on Mathematical Software 9, 302–325 (1983)
13. Guermouche, A., L'Excellent, J.-Y., Utard, G.: Impact of reordering on the memory of a multifrontal solver. Parallel Computing 29(9), 1191–1218 (2003)
14. Hermann, E., Raffin, B., Faure, F., Gautier, T., Allard, J.: Multi-GPU and multi-CPU parallelization for interactive physics simulations. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part II. LNCS, vol. 6272, pp. 235–246. Springer, Heidelberg (2010)
15. Hogg, J., Reid, J.K., Scott, J.A.: A DAG-based sparse Cholesky solver for multicore architectures. Technical Report RAL-TR-2009-004, RAL (2009)
16. Lacoste, X., Ramet, P., Faverge, M., Yamazaki, I., Dongarra, J.: Sparse direct solvers with accelerators over DAG runtimes. Research report RR-7972, INRIA (2012)
17. Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.A., Van Zee, F.G., Chan, E.: Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Trans. Math. Softw. 36(3) (2009)
18. Schreiber, R.: A new implementation of sparse Gaussian elimination. ACM Transactions on Mathematical Software 8, 256–276 (1982)
19. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)

Fast Methods for Computing Selected Elements of the Green's Function in Massively Parallel Nanoelectronic Device Simulations

Andrey Kuzmin[1], Mathieu Luisier[2], and Olaf Schenk[1]

[1] Institute of Computational Science, Universita della Svizzera italiana
CH-6900 Lugano, Switzerland
{andrey.kuzmin,olaf.schenk}@usi.ch,@usi.ch
http://www.ics.inf.usi.ch

[2] Integrated Systems Laboratory, ETH Zürich, CH-8092 Zürich, Switzerland
mluisier@iis.ee.ethz.ch
http://www.iis.ee.ethz.ch

Abstract. The central computation in atomistic, quantum transport simulation consists in solving the Schrödinger equation several thousand times with non-equilibrium Green's function (NEGF) equations. In the NEGF formalism, a numerical linear algebra problem is identified related to the computation of a sparse inverse subset of general sparse unsymmetric matrices. The computational challenge consists in computing all the diagonal entries of the Green's functions, which represent the inverse of the electron Hamiltonian matrix. Parallel upward and downward traversals of the elimination tree are used to perform these computations very efficiently and reduce the overall simulation time for realistic nanoelectronic devices. Extensive large-scale numerical experiments on the CRAY-XE6 Monte Rosa at the Swiss National Supercomputing Center and on the BG/Q at the Argonne Leadership Computing Facility are presented.

1 Introduction

Ultrascaled nanowire field-effect transistors (NWFETs) [1,2] could become the next generation logic devices when it will no longer be possible to scale the dimensions of the currently manufactured fin-shaped field effect transistor (FinFETs) and keep improving their performance. Technology computer aided design (TCAD) has established itself as a great accelerator for the development of novel transistors. However, to simulate the characteristics of NWFETs, it is necessary to go beyond classical approximations such as the drift-diffusion model and to use a quantum transport approach. Energy quantization, quantum confinement, and quantum mechanical tunneling can only be accurately captured if the Schrödinger equation is directly solved in a full-band, atomistic basis and if open boundary conditions describing the coupling of the device with its environment are included [3,4].

The non-equilibrium Green's function formalism (NEGF) is one of the most efficient techniques for performing this task. It has been widely used to study the electronic and thermal properties of nanoscale transistors and molecular switches on massively parallel architectures. The NEGF formalism is used, e.g., in the highly successful nanoelectronics modeling tools OMEN [1] [5] and TranSIESTA[2] [6]. In these applications, a subset of the entries of the inverse of complex unsymmetric matrices must be repeatedly computed, which represents a significant computational burden. This is usually achieved with a so-called recursive Green's function (RGF) algorithm [7,8].

The calculation of a subset of the entries of the inverse of a given matrix also occurs in a wide range of other applications, e.g., in electronic transport simulation [9,10], the diagonal and sometimes subdiagonal of the discrete Green's function are needed in order to compute electron density. It is therefore of utmost importance to develop and implement efficient scalable algorithms targeting the diagonal of the inverse that are faster than, e.g., inverting the entire matrix based on successive application of a sparse direct LU decomposition of A or faster than the RGF algorithm.

Consider a general sparse unsymmetric matrix $A \in C^{n \times n}$, and assume that A is not singular so that it can be factorized as $A = LU$. If the matrix A is irreducible then A^{-1} is a dense matrix [11]. In this paper, the problem of computing selected elements of the inverse A^{-1} of A is addressed. This subset of selected elements is defined by the set of nonzero entries in the factorized matrix. It was proved in [12] that both the subset and the diagonal of A^{-1} can be evaluated without computing any inverse entry from outside of the subset.

A fast direct algorithm called fast inverse using nested dissection (FIND) that was proposed in [9] is used to compute the required components of the NEGF in the simulations of nanoscale devices. The method is based on the algorithm of nested dissection. A graph of the matrix is constructed and decomposed using a tree structure. An upward and downward traversal of the tree is used to perform the computation efficiently. An alternative method is based on the Schur-complement method described in [13,14]. The fast sequential algorithm proposed for symmetric indefinite matrices was implemented in the Selinv[3] library.

1.1 Contribution

To the best of our knowledge, there is no parallel efficient software package currently available for computing an inverse subset of a general unsymmetric sparse matrix. This paper fills this gap by describing an efficient BLAS level-3 algorithm and its parallel multithreaded implementation. It is available in

[1] OMEN was awarded an honorable mention at the 2011 ACM Gordon Bell Prize for reaching a sustained performance of 1.44 PFlop/s on the Cray-XT5 Jaguar. It is available at `http://nanohub.org/resources/omenwire`

[2] `http://www.icmab.es/dmmis/leem/siesta`

[3] `https://web.math.princeton.edu/~linlin/software.html`

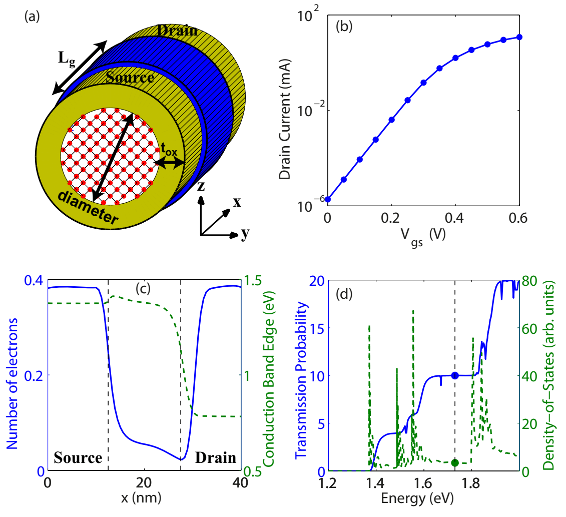


Fig. 1. (a) Schematic view of a Si ultrascaled NWFET composed of source, channel, and drain regions. Each single Si atom (red dot) is taken into account. The central semiconducting nanowire of diameter d is surrounded by an oxide layer of thickness t_{ox}. The gate contact (blue) has a length L_g. Electrons flow from source to drain along the x-axis. (b) Transfer characteristics I_d-V_{gs} (drain current vs. gate-to-source voltage) at a drain-to-source voltage V_{ds}=0.6 V of a Si NWFET with d=3 nm, L_g=15 nm, and t_{ox}=1 nm. (c) Number of electrons (solid blue line) and conduction band profile (dashed green line) along the x-axis of the same Si NWFET as in (b). (d) Electron transmission probability (solid blue line) and density of states (dashed green line) under the same conditions as in (c).

the latest version of the PARDISO[4] package. Extensive intranode performance studies were perfromed on Cray XE-6 and IBM BG/Q architectures. The method is fully integrated into the massively parallel nanoelectronic simulation code OMEN. Numerical experiments show significant advantage of method over the RGF approach.

The paper is organized as follows: in Section 2, an overview of the atomistic, quantum transport simulation approach including its RGF implementation is given. In Section 3 the idea of selected parallel inversion as an alternative to the RGF method is described. The performance of our code is finally described and analyzed in Section 4 before the paper is concluded in Section 5.

2 Overview of the Simulation Approach

2.1 Formulation of the Quantum Transport Problem

Simulating NWFET in a full-band, atomistic basis requires repeatedly solving the Schrödinger equation with open boundary conditions and dealing with large

[4] `https://www.pardiso-project.org`

matrices whose size depends on the number of atoms present in the simulation domain and on the number of atomic orbitals considered on each atom. A typical device structure is schematized in Fig. 1(a). The current flowing through the NWFET at a given gate-to-source (V_{gs}) and drain-to-source (V_{ds}) voltage represents the most relevant quantity of interest since it can be compared to experimental data. An example is given in Fig. 1(b). The electron charge density is also important since it induces an electrostatic potential through the Poisson equation, which in turn affects the Schrödinger equation. It must therefore also be calculated as illustrated in Fig. 1(c). In ballistic simulations, to obtain the current and charge density, the energy-resolved electron transmission probability $T(E)$ and density of states (DOS) $g(E)$ must be first evaluated. They are shown in Fig. 1(d).

In order to calculate $T(E)$ and $g(E)$ and then the electron and charge density, the following NEGF equation must be solved for each electron energy E:

$$\left(E\mathbf{I} - \mathbf{H} - \mathbf{\Sigma}^{RS}(E) - \mathbf{\Sigma}^{RD}(E)\right) \cdot \mathbf{G}^{R}(E) = \mathbf{I}. \tag{1}$$

In Eq. (1), the Hamiltonian matrix $\mathbf{H}$ is of size $N_A \times N_{orb}$, where N_A is the number of atoms and N_{orb} the number of orbitals in the full-band basis used to describe the device material properties. Here, a semiempirical, nearest-neighbor $sp^3d^5s^*$ tight-binding model without spin-orbit coupling is employed, which means that N_{orb}=10 [15]. In many applications, $\mathbf{H}$ is block tridiagonal, but it can also exhibit more complicated structures. The matrix $\mathbf{I}$ is the identity matrix while the self-energy matrices $\mathbf{\Sigma}^{RS}(E)$ and $\mathbf{\Sigma}^{RD}(E)$ refer to the source and drain open boundary conditions respectively, calculated as in [5]. Only the upper left corner of $\mathbf{\Sigma}^{RS}(E)$ and lower right corner of $\mathbf{\Sigma}^{RD}(E)$ are different from 0. In a device with more than two contacts, additional self-energies must be included in Eq. (1). Here, for simplicity, only a source and drain contact are accounted for. Finally, $\mathbf{G}^{R}(E)$ is the retarded Green's Function at energy E.

Once $\mathbf{G}^{R}(E)$ is calculated, either with a standard approach as in Section 2.2 or with the new algorithm proposed in Section 3, the electron density of states $g(E)$ and transmission probability $T(E)$ can be derived. First, $g(E)$ is considered. It contains as many components as contacts so that

$$g(E) = g^{S}(E) + g^{D}(E), \tag{2}$$

where $g^{S}(E)$ and $g^{D}(E)$ refer to the DOS coming from the source and drain, respectively. It can be demonstrated that [3]

$$g(E) = \frac{i}{2\pi}\text{diag}\left(\mathbf{G}^{R}(E) - \mathbf{G}^{R\dagger}(E)\right), \tag{3}$$

$$g^{S}(E) = \frac{1}{2\pi}\text{diag}\left(\mathbf{G}^{R}(E) \cdot \mathbf{\Gamma}^{S}(E) \cdot \mathbf{G}^{R\dagger}(E)\right), \tag{4}$$

$$g^{D}(E) = g(E) - g^{S}(E). \tag{5}$$

Similarly, the transmission probability $T(E)$ is given by

$$T(E) = \text{trace}\left(\mathbf{G}^{R}(E) \cdot \mathbf{\Gamma}^{S}(E) \cdot \mathbf{G}^{R\dagger}(E) \cdot \mathbf{\Gamma}^{D}(E)\right). \tag{6}$$

In both Eqs. (4) and (6), broadening functions $\mathbf{\Gamma}^{S/D}(E)$ are introduced. They are defined as

$$\mathbf{\Gamma}^{S/D}(E) = i\left(\mathbf{\Sigma}^{RS/D}(E) - \mathbf{\Sigma}^{RS/D\dagger}(E)\right). \tag{7}$$

Based on Eqs. (3)–(6) and on the fact that only the upper left corner of $\mathbf{\Sigma}^{RS}(E)$ and $\mathbf{\Gamma}^{S}(E)$ is different from 0, it appears that not the entire $\mathbf{G}^{R}(E)$ matrix is needed, but only its diagonal and its first n columns, where n is the size of the nonzero square block of $\mathbf{\Gamma}^{S}(E)$. This simplification is valid in ballistic simulations only. As soon as scattering is included, e.g., electron-phonon interactions or interface roughness, the situation becomes more complicated. However, such cases are outside the scope of this paper.

With the knowledge of $g^S(E)$, $g^D(E)$, and $T(E)$, the charge density n_{el} and current I_d can be computed as

$$n_{el} = \int dE \left(g^S(E) f(E - E_{fS}) + g^D(E) f(E - E_{fD})\right), \tag{8}$$

$$I_d = \frac{q}{\hbar}\int \frac{dE}{2\pi} T(E)\left(f(E - E_{fS}) - f(E - E_{fD})\right), \tag{9}$$

where $f(E)$ is a distribution function (here Fermi-Dirac), E_{fS} and E_{fR} the source and drain Fermi levels, q the elementary charge constant, and $\hbar$ the reduced Planck constant. The electron density n_{el} takes the form of a vector with different values on each atom. The drain current I_d is a scalar.

2.2 RGF Algorithm

When the Hamiltonian matrix $\mathbf{H}$ has a block tridiagonal structure, as in most device simulations, an efficient RGF algorithm can be used to solve Eq. (1) [7,8]. To briefly sketch the functionality of the RGF algorithm, it is assumed that $\mathbf{H}$ contains N diagonal blocks and that M_{ij} is the block with row index i and column index j in the matrix $\mathbf{M}$. The algorithm involves two recursive steps, the first one starting at the lower right corner,

$$g_i^R = \left(E - H_{ii} - H_{ii+1} g_{i+1}^R H_{i+1i}\right)^{-1} C \tag{10}$$

with

$$g_N^R = \left(E - H_{NN} - \Sigma_{NN}^{RD}\right)^{-1}, \tag{11}$$

$$g_1^R = \left(E - H_{11} - H_{12} g_2^R H_{21} - \Sigma_{11}^{RS}\right)^{-1}. \tag{12}$$

The g_i^R's are approximate solutions to Eq. (1) when H_{ii-1} and H_{i-1i} are set to 0. It then becomes clear that the exact Green's Function G_{11}^R is equal to g_1^R. Then, in a second phase, two additional recursions can be derived to calculate the exact diagonal and first column blocks of $\mathbf{G}^R(E)$ starting from the upper left corner:

$$G_{ii}^R = g_i^R + g_i^R H_{ii-1} G_{i-1i-1}^R H_{i-1i} g_i^R, \tag{13}$$

$$G_{i1}^R = g_i^R H_{ii-1} G_{i-11}^R. \tag{14}$$

No other system of equations must be solved to evaluate Eqs. (3)–(6). Note finally that the computational complexity of the RGF algorithm amounts to $O(Nn^3)$, where n is the average block size, and that it cannot be efficiently parallelized. This becomes a serious issue in large simulation domains, for which the required memory to solve Eqs. (10)–(14) is larger than the one available on each CPU.

3 A Selected Sparse Inverse Matrix Algorithm (SINV)

3.1 Sparse Inverse Supernodal Factorization

The obvious way to compute selected entries of the inverse is to invert the entire matrix and than to extract the selected entries. The standard approach for matrix inversion is to perform the LU factorization first:

$$A = LU,$$

where L and U are unit lower triangular and upper triangular matrices respectively. Using such a factorization, $A^{-1} = (x_1, x_2, ..., x_n)$ could be obtained by solving a number of linear systems $Ax_i = e_i$. Each of the systems is solved using backward and forward substitution phases, $Ly = e_j$ and $Ux_j = y$. Before the algorithm is presented, the process of computing LU factorization is reviewed. Let A be a nonsingular unsymmetric matrix. Each step of LU factorization produces the following decomposition:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} I & \\ & S \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ & I \end{bmatrix},$$

where $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ is the Schur-complement of A_{11} with respect to A. In order to simplify the derivation, it is assumed that no row or column permutation is required during the factorization. The discussion could be easily generalized for the case of pivoting algorithms used in order to improve stability. The main idea of the algorithm is that A^{-1} could be computed using the following expression:

$$A^{-1} = \begin{bmatrix} U_{11}^{-1}L_{11}^{-1} + U_{11}^{-1}U_{12}S^{-1}L_{11}^{-1}L_{21} & -U_{11}^{-1}U_{12}S^{-1} \\ -S^{-1}L_{11}^{-1}L_{21} & S^{-1} \end{bmatrix}.$$

Using the notation $\tilde{U}_{12} = -U_{11}^{-1}U_{12}$, $\tilde{L}_{12} = -L_{11}^{-1}L_{12}$ and $\tilde{D}^{-1} = U_{11}^{-1}L_{11}^{-1}$ this expression for the inverse can be simplified:

$$A^{-1} = \begin{bmatrix} \tilde{D}^{-1} + \tilde{U}_{12}S^{-1}\tilde{L}_{21} & \tilde{U}_{12}S^{-1} \\ -S^{-1}\tilde{L}_{21} & S^{-1} \end{bmatrix}.$$

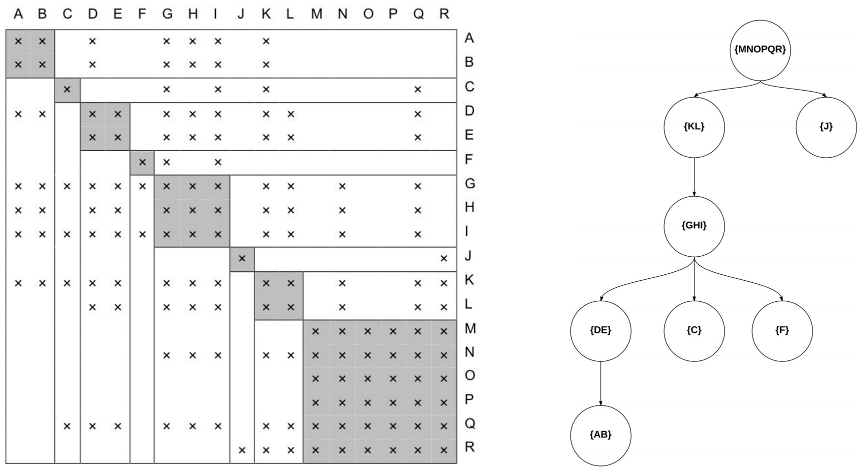


Fig. 2. Example of a supernodal partition for sparse LU factorization and the corresponding supernodal elimination tree

The idea was originally proposed in [16]. The more general approach for unsymmetric matrices is presented in this paper. This expression suggests that given the LU factorization, computation of the inverse can be reduced to computing S^{-1}. The computation of A^{-1} can be organized in a recursive manner similar to the LU factorization algorithm, but computing the sequence of the diagonal elements is organized in the opposite direction. The last diagonal element of the inverse equals a reciprocal of the last diagonal element of the U factor, $A_{nn}^{-1} = (U_{nn})^{-1}$. Starting from the last element, which is also the Schur-complement produced in the last step of the LU factorization algorithm, we proceed step by step, computing more and more blocks from the lower right corner to the upper left corner. Thus, more and more entries of A^{-1} are computed. The complexity of such an inversion algorithm is still $O(n^3)$ in the case of a dense matrix. However, computational cost can be drastically reduced if the matrix A is sparse. Rigorous consideration of this topic leads to the concept of sparse matrix elimination tree and its extension to the inverse factorization case [17,18,19]. As a consequence the complexity of the inversion process can be reduced to $O(n^2)$ for matrices from three-dimensional simulations.

The implementation is built on top of the PARDISO package that uses supernodal BLAS level-3 algorithm during the direct factorization phase. The METIS package is used to produce the fill-in reducing reordering [20]. An example of such a supernodal partitioning for LU factorization can be seen in Fig. 2. The nested dissection algorithm allows to reduce significantly the size of diagonal subblocks compared to RGF method that results in additional performance increase. Nonzero entries of LU factors are stored in dense subblocks, so that each supernode consists of many dense submatrices that could be used in efficient dense matrix-matrix multiplication subroutines.

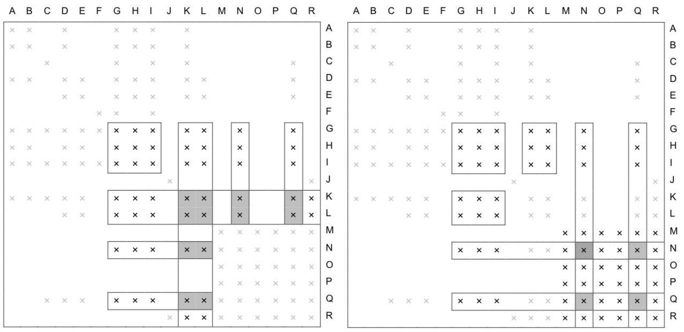

A B C D E F G H I J K L M N O P Q R
A B C D E F G H I J K L M N O P Q R

Fig. 3. Example of a dense subblock gathering during the inverse factorization process. The scheme on the left-hand side depicts the submatrix that contributes to the update of the supernode {G,H,I} by the supernode {K,L}. Contribution of the supernode {N,O,P,Q,R} to {G,H,I} is represented on the right-hand side.

In the implementation, the inverse factorization is computed in place so that the original LU factorization is overwritten by the entries of the inverse. The most computationally time consuming operations at this stage are two matrix-matrix products, namely $\tilde{U_{12}}S^{-1}$ and $-S^{-1}\tilde{L}_{21}$ where S^{-1} is a sparse matrix with supernodal storage and $\tilde{L}_{21}$, $\tilde{U}_{12}$ are, in turn, sparse block vectors stored by contiguous dense subblocks. First the computation of the product $\tilde{U_{12}}S^{-1}$ is considered. It could be split according to the supernodal partition of the Schur-complement: $\tilde{U_{12}}S^{-1} = \tilde{U_{12}}S^{-1}_{\{A,B\}} + \tilde{U_{12}}S^{-1}_{\{C\}} + \ldots + \tilde{U_{12}}S^{-1}_{\{M,N,O,P,Q,R\}}$, where $S^{-1}_{\{A,B\}}$, $S^{-1}_{\{C\}}$, $\ldots$ are supernodes of S^{-1} consisting of a lower and upper triangular part each. Each of the terms in the sum can be computed in two steps. First the required entries are gathered into a dense block using indirect addressing schemes similar to techniques described in [21] (see Fig. 3). Then, the product is computed using two dense matrix-matrix multiplications. The sum is accumulated in a temporary buffer. After $\tilde{U_{12}}S^{-1}$ has been computed, the product $\tilde{U}_{12}S^{-1}\tilde{L}_{21} = (\tilde{U}_{12}S^{-1})\tilde{L}_{21}$ could be immediately calculated using the xGEMM function. The product $-S^{-1}\tilde{L}_{21}$ is calculated in a similar manner. Thus, all the computations were performed using BLAS level-3 subroutines that guarantees optimal use of vector operations on modern architectures and effecient usage of their cache hierarchies.

3.2 Intranode Parallelization

Data dependencies in the selected inversion algorithm are represented by the inverse elimination tree. Therefore, the inversion algorithm permits parallelization strategies similar to parallel direct solvers. The level of parallelism mainly utilized in this case is the tree level parallelism.

Consider the serial inversion algorithm described in the previous section. Factorization for each supernode consists of two parts. The first part could be called internal inverse factorization (computing $\tilde{U}_{12}$, $\tilde{L}_{21}$ and $\tilde{D}^{-1}$) since it is done independently. The second part (computing $\tilde{U_{12}}S_{\{I\}}^{-1}$ and $-S_{\{I\}}^{-1}\tilde{L_{21}}$ for each supernode of S) accumulates external contributions from other supernodes and could be called the external update phase. In order to parallelize the algorithm, data dependencies must be maintained. This suggests creation of the global tasks queue. Each element of the queue is a supernode and the size of the queue is bounded by the total number of supernodes. Tasks distribution is performed dynamically: each thread fetches one element S_j from the queue and proceeds with the internal inversion phase. Computation of external contributions requires synchronization with threads working on the descendants of S_J, i.e., the thread waits until the inversion of each of the dependents is finished.

4 Numerical Experiments

4.1 Experimental Testbase

In this section the parallel performance and efficiency of OMEN equipped with the PARDISO selected inversion implementation for the NEGF equation is reported. Before moving on to the parallel scalability of the atomistic, quantum transport simulation benchmarks, this section gives a brief description of the target hardware, namely, IBM BG/Q at the Argonne Leadership Computing Facility and a Cray XE6 system installed at the Swiss National Supercomputing Center CSCS. Intranode experiments were performed on one rack of the "Mira" BG/Q which has a 1.6-GHZ 16-way quad-core PowerPC processor and 16 GB of RAM. The Cray XE6 "Monte Rosa" compute nodes were used for intranode performance experiments. The Cray XE6 consists of 2 16-core AMD Opteron 6272 2.1-GHz Interlagos processors, giving 32 cores in total per node with 32 GBytes of memory. The Interlagos CPUs implement AMD's recent "Bulldozer" microarchitecture and each Interlagos socket contains two dies, each of which contains four so-called "modules."

4.2 Intranode Performance

In this section the results on performance of the inversion algorithm based on matrices from the NEGF solver implemented in OMEN are presented. Table 1 shows the speedup of the selected inversion over the full inversion algorithm for the set of 4 Hamiltonian matrices of the size 97900 to 581900 uknowns with LU factors containing around 10^6 nonzero elements. The advantage over the full inversion grows as the problem size increases that makes it practically possible to solve the problem with more than 10^6 nonzero elements. The new method is more than 250 times faster on both architectures for the largest matrix.

Table 2 demonstrates the scalability of the inversion algorithm for the set of 2 largest matrices on one node with 2 to 32 threads compared to the RGF method that was previously used in OMEN. The observed scalability is comparable to that of the direct factorization algorithm. The shared-memory parallel efficiency of the PARDISO-SINV implementation is considerable and compelling for larger

Table 1. Average time in seconds (and Gflop/s in brackets) in PARDISO and PARDISO-SINV on 32 cores when computing all diagonal entries of the inverse of A for four selected OMEN test matrices; n represents the size of the Hamiltonian matrix

n	CRAY XE6		BG/Q	
	PARDISO	PARDISO-SINV	PARDISO	PARDISO-SINV
97900 (d=2nm)	2434.0	0.8 (51)	4174.0	3.0 (22)
212300 (d=3nm)	4198.0	7.5 (89)	19571.1	8.7 (45)
375100 (d=4nm)	20525.1	42.7 (69)	65847.6	70.7 (63)
581900 (d=5nm)	26657.1	102.1 (119)	62296.7	217.2 (68)

Table 2. Average time in seconds to calculate the density of states and the transmission probability for one energy point, as indicated by the dashed line in Fig. 1(d), for nanowire transistors with two different diameters (4 and 5 nm). The first column gives the Hamiltonian matrix size in Eq. (1), the second the number of cores, the third the solution time with the RGF algorithm [7,8], while columns 4 to 9 are dedicated to the new approach proposed in this paper. The times to reorder the Hamiltonian matrix, factorize it, compute selected elements of its inverse, solve it to obtain G^R_{N1}, and derive the DOS with Eq. (4) are reported. The last column represents the sum of columns 4 to 8.

n	Cores	RGF	Reordering	Factorization	SINV	G^R_{N1}	Eq. (4)	Total
375100 (d=4nm) IBM BG/Q	1	4179.1	55.85	601.53	1122.41	1601.13	-	3380.92
	2	-	56.01	251.55	463.25	760.15	-	1530.96
	4	-	55.63	147.91	300.77	440.86	-	945.17
	8	-	54.99	67.60	162.49	237.69	-	522.77
	16	-	54.71	37.89	84.72	150.25	-	327.57
	32	-	55.85	28.72	67.56	112.17	-	264.3
581900 (d=5nm) IBM BG/Q	1	16644.2	92.93	2042.29	3393.81	5057.64	-	10586.67
	2	-	93.21	818.52	1799.18	2655.33	-	5366.24
	4	-	93.13	460.88	996.35	1790.10	-	3340.46
	8	-	91.49	229.03	622.89	1089.41	-	2032.82
	16	-	91.61	120.61	328.37	768.19	-	1308.78
	32	-	91.86	90.82	217.21	723.23	-	1123.12
375100 (d=4nm) Cray XE6	1	1393.9	15.6	222.7	392.2	1149.5	390.1	2191.3
	2	-	16.6	163.5	311.4	671.4	327.7	1515.6
	4	-	16.4	83	171	215.9	165.4	677.3
	8	-	17.5	47.1	105.1	108.6	87.7	394.4
	16	-	17.5	25.6	64.2	59.6	45.6	242.1
	32	-	17	15.5	13.7	51.7	24.1	153.5
581900 (d=5nm) Cray XE6	1	5548.9	23.5	697.8	1250.5	1466.1	1428.4	4892.7
	2	-	25.4	533.5	1033.3	1072.1	1171.7	3865.9
	4	-	25.5	270.1	529.2	564.2	609.2	2029.2
	8	-	27.6	153.1	333.5	328.6	322.3	1199.4
	16	-	27.2	79.2	166.9	302.3	168.2	779.7
	32	-	27.7	46.7	99.3	283.7	91.8	587.9

problems (e.g., d=5 nm). The experiments show that the new implementation has the performance lower or comparable to RGF method when using 1 or 2 threads; however the RGF implementation has a low potential for scalability and the new approach is one order of magnitude faster when using 32 threads. These results show that the selected inversion algorithm can be very efficiently applied in large-scale computational nanolectronics, significantly reducing the overall simulation time for realistic devices.

5 Conclusion

The recursive Green's function algorithm that is typically used in large-scale atomistic nanoelectronic device engineering has good algorithmic efficiency in the order of $O(N_z \cdot n^3)$, where $n = n_x \cdot n_y$ is the average block size, but significant disadvantages in terms of parallelism. An alternative method based on parallel selected inversion has been presented in this paper for the NEGF that is the central computation in atomistic, quantum transport simulations. The complexity of the selected inversion method is in the order of $O(N_z^2 \cdot n^2)$. It is used to extract all diagonal entries of the inverse of a complex sparse unsymmetric matrix. The new selected inversion method overcomes the scalabilty barrier of RGF by using parallel upward and downward traversals of the elimination tree to solve the NEGF equations very efficiently. The implementation of the inversion solver showed substantial speedup over the previous approach. PARDISO-SINV solved realistically sized examples in OMEN in about 5 min compared to over 1.5 hour on the Cray XE6.

Acknowledgment. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. We acknowledge the Swiss National Supercomputing Center and for giving us access to their hardware and for their support. The work done in Switzerland is funded partially under the HP2C initiative. Mathieu Luisier acknowledges the support of SNF grant PPOOP2_133591.

References

1. Cui, Y., Zhong, Z., Wang, D., Wang, W., Lieber, C.: High performance silicon nanowire field effect transistors. Nano Letters 3, 149–152 (2003)
2. Singh, N., Agarwal, A., Bera, L., Liow, T., Yang, R., Rustagi, S., Tung, C., Kumar, R., Lo, G., Balasubramanian, N., Kwong, D.: High-performance fully depleted silicon nanowire (diameter $\leq$ 5 nm) gate-all-around CMOS devices. IEEE Electron Device Letters 27, 383–386 (2006)
3. Datta, S.: Electronic transport in mesoscopic systems. Press Syndicate University of Cambridge, Cambridge (1995)
4. Cauley, S., Jain, J., Koh, C., Balakrishnan, V.: A scalable distributed method for quantum-scale device simulation. J. Appl. Phys. 101, 123715 (2007)

5. Luisier, M., Klimeck, G., Schenk, A., Fichtner, W.: Atomistic simulation of nanowires in the $sp^3d^5s^*$ tight-binding formalism: from boundary conditions to strain Calculations. Phys. Rev. B 74, 205323 (2006)
6. Brandbyge, M., Mozos, J., Ordejon, P., Taylor, J., Stokbro, K.: Density-functional method for nonequilibrium electron transport. Phys. Rev. B 74, 165401 (2002)
7. Lake, R., Klimeck, G., Bowen, R., Jovanovic, D.: Single and multiband modeling of quantum electron transport through layered semiconductor devices. J. Appl. Phys. 81, 7845–7869 (1997)
8. Svizhenko, A., Anantram, M., Govindan, T., Biegel, R., Venugopal, R.: Two-dimensional quantum mechanical modeling of nanotransistors. J. Appl. Phys. 91, 2343–2354 (2002)
9. Li, S., Ahmed, S., Klimeck, G., Darve, E.: Computing entries of the inverse of a sparse matrix using the FIND algorithm. J. Comp. Phys. 227, 9408–9427 (2008)
10. Petersen, D.A., Song, L., Stokbro, K., Sorensen, H.H.B., Hansen, P.C., Skelboe, S., Darve, E.: A Hybrid Method for the Parallel Computation of Green's Functions. J. Comp. Phys. 228(14), 5020–5039 (2009)
11. Duff, I., Erishman, A.: Sparsity structure and Gaussian elimination. ACM SIGNUM Newsletter 23, 2–8 (2006)
12. Erishman, A., Tinney, W.: On computing certain elements of the inverse of a sparse matrix. Comm. ACM 18, 177 (1975)
13. Lin, L., Lu, L., Ying, J.: Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems. Comm. Math. Sci. 7, 755–777 (2009)
14. Lin, L., Yang, C.: Selinv - an algorithm for selected inversion of a sparse symmetric matrix. ACM Trans. on Math. Software 37 (2011)
15. Slater, J., Koster, G.: Simplified LCAO method for the periodic potential problem. Phys. Rev. 94, 1498–1524 (1954)
16. Takahashi, L., Fagan, J., Chin, M.: Formation of a sparse bus impedance matrix and its application to short circuit study. In: Proc. 8th PICA Conference, Minneapolis, Minnesota, pp. 63–69 (1973)
17. Campbell, Y., Davis, T.: Computing the Sparse Inverse Subset: an Inverse Multifrontal Approach. Technical Report TR-95-021, Computer and Information Sciences Department, University of Florida (1995)
18. Amestoy, P., Duff, I., Robert, Y., Rouet, F., Ucar, B.: On Computing Inverse Entries of a Sparse Matrix in an Out-of-Core Environment. Technical Report TR/-PA/10/59, CERFACS, Toulouse, France (2010)
19. Slavova, T.: Parallel Triangular Solution in the Out-of-Core Multifrontal Approach for Solving Large Sparse Linear Systems. Ph.D. dissertation, Institut National Polytechnique de Toulouse, Toulouse, France (2009)
20. Karypis, G., Kumar, V.: A fast and highly quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Computing 20, 359–392 (1999)
21. Schenk, O., Gärtner, K.: Solving unsymmetric sparse systems of linear equations with PARDISO. Future Gener. Comp. Systems 20, 475–487 (2004)

Topic 11: Multicore and Manycore Programming (Introduction)

Luiz Derose, Jan Treibig, David Abramson, Alastair Donaldson, William Jalby, Alba Cristina M.A. de Melo, and Tomàs Margalef

Topic Committee

Today's compute node architectures leverage impressive performance by offering more parallel resources on the chip as well as on the node level. Among parallel resources are memory interfaces (ccNUMA), cores, caches and data parallel execution units. On the other hand modern multicore designs also exhibit shared resources such as memory bandwidth on the chip level, last level cache bandwidth and capacity, and access to the network interface. An additional performance-limiting factor is the frequently high cost for synchronization. The task to make full use of parallel resources while keeping an eye on the bottlenecks imposed by the shared resources is non-trivial. Common programming models often address issues related to parallel programming in general while not covering topological issues introduced by multi- and manycore architectures. The industry is still pushing forward introducing even more powerful manycore systems like, e.g., the Nvidia Kepler and Intel MIC architectures.

While multicore chips have been present for some time now their efficient use by programmers, support of multicore aware programming models, and adapted operating systems is still in its early stages and offers many opportunities for research and innovation.

The papers of this topic have been selected based on the recommendations of four reviewers. The eight accepted papers address a wide range of topics, from multicore-aware algorithms to programming models, middleware, and operating systems.

The paper "Assessing the Performance of OpenMP Programs on the Intel Xeon Phi" by Dirk Schmidl, Tim Cramer, Sandra Wienke, Christian Terboven, Dieter An Mey and Matthias S. Müller presents a performance evaluation of the Intel Xeon Phi manycore chip. The comparisons to an Intel Sandy Bridge multicore system are based on a selection of microbenchmarks, NAS Parallel benchmarks, and real application codes.

The paper "Towards a Scalable Microkernel Personality for Multicore Processors" by Jilong Kuang, Daniel Waddington and Chen Tian introduces SilicaOS, a scalable microkernel implementation based on the Fiasco micro kernel, which is targeted to multicore machines.

The paper "A hybrid parallel Barnes-Hut algorithm for GPU and multicore architectures" by Hannes Hannak, Hendrik Hochstetter and Wolfgang Blochinger presents a modularized (hybrid) Barnes-Hut implementation which can exploit modern heterogeneous compute devices using NVIDIA GPUs.

The paper ”Lightweight Contention Management for Efficient Compare-and-Swap Operations” by Ilya Mirsky, Danny Hendler and Dave Dice proposes using lightweight contention management at the compare and swap instruction level as opposed, or, in concert with, algorithm/data-structure level contention management.

The paper ”MacroDB: Scaling Database Engines on Multicores” by Joo Soares, Nuno Preguica and João Loureno proposes MacroDB, an architecture to replicate in memory databases in multicore systems to achieve high throughput in database applications.

The paper ”Transparent Supports for Partial Rollback in Software Transactional Memories” by Alice Porfirio, Alessandro Pellegrini, Pierangelo Di Sanzo and Francesco Quaglia presents an approach for the partial rollback of a transaction in software transactional memories (STMs). The approach has been implemented based on TinySTM and its performance is evaluated with benchmarks.

The paper ”An Implementation of the Codelet Execution Model” by Joshua Suetterlein, Stphane Zuckerman and Guang R. Gao describes an implementation of the Codelet programming model using DARTS and presents two case studies to validate its advantages against traditional OpenMP implementations.

The paper ”Generic High-performance Method for Deinterleaving Scientific Data” by Eric Schendel, Steve Harenberg, Houjun Tang, Venkatram Vishwanath, Michael Papka and Nagiza Samatova presents a method for deinterleaving scientific data to improve the performance of scientific applications.

We are grateful to all authors for submitting high-quality papers to this topic and to reviewers for their effort to evaluate submitted papers. Furthermore, we would like to acknowledge the encouragement and support of the conference chairs Felix Wolf, Dieter an Mey and Bernd Mohr.

Assessing the Performance of OpenMP Programs on the Intel Xeon Phi*

Dirk Schmidl[1,3], Tim Cramer[1,3], Sandra Wienke[1,3], Christian Terboven[1,3], and Matthias S. Müller[1,2,3]

[1] Center for Computing and Communication, RWTH Aachen University, D - 52074 Aachen
[2] Chair for High Performance Computing, RWTH Aachen University, D - 52074 Aachen
[3] JARA High-Performance Computing, Schinkelstrae 2, D 52062 Aachen
{schmidl,cramer,wienke,terboven,mueller}@rz.rwth-aachen.de

Abstract. The Intel Xeon Phi has been introduced as a new type of compute accelerator that is capable of executing native x86 applications. It supports programming models that are well-established in the HPC community, namely MPI and OpenMP, thus removing the necessity to refactor codes for using accelerator-specific programming paradigms. Because of its native x86 support, the Xeon Phi may also be used stand-alone, meaning codes can be executed directly on the device without the need for interaction with a host. In this sense, the Xeon Phi resembles a big SMP on a chip if its 240 logical cores are compared to a common Xeon-based compute node offering up to 32 logical cores. In this work, we compare a Xeon-based two-socket compute node with the Xeon Phi stand-alone in scalability and performance using OpenMP codes. Considering both as individual SMP systems, they come at a very similar price and power envelope, but our results show significant differences in absolute application performance and scalability. We also show in how far common programming idioms for the Xeon multi-core architecture are applicable for the Xeon Phi many-core architecture and which challenges the changing ratio of core count to single core performance poses for the application programmer.

1 Introduction

Intel calls the new Intel Xeon Phi a coprocessor instead of using the term accelerator. Indeed it can be both, an accelerator which is used to speed up scientific applications, or a standalone SMP on a single chip. In the first case compute-intensive parts of an application can be executed on the device, as it is known from programming paradigms like CUDA or OpenCL explicitly designed for accelerators. For the Xeon Phi this can be achieved with the Intel Language Extensions for Offload (LEO). However, in this work we will focus on the second scenario and assess the behavior of Xeon Phi as an SMP machine with many cores. The coprocessor itself is able to run x86 code and supports many standard parallel programming paradigms like OpenMP or MPI which is meant to make the rewrite of a kernel or even complete applications unnecessary. The aim is to reach a much better usability and make the Xeon Phi available for a wider

* Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under Grant No. 01IH11006.

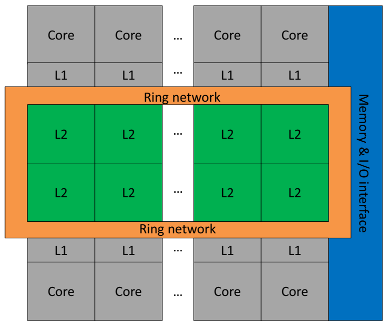
range of applications than it is possible with GPUs today. However, the fact that code optimized for multi-core architectures can run on this new many-core architecture by just recompiling does not mean that the performance for general-purpose applications is as desired. We investigate if OpenMP constructs, such as OpenMP tasks and nested parallel regions, can be used on the Xeon Phi efficiently. This would allow executing more complex programs natively in contrast to just offloading vector parallel loops as it is done for other accelerators. We present a performance evaluation with both basic kernels as well as more complex benchmarks, like a conjugate gradient solver to show that there are no general reasons which prevents an efficient use of the Xeon Phi as a SMP system on a chip. Furthermore, we evaluate the performance of the NAS parallel benchmarks and some real-world application codes, which use different methods to utilize the coprocessor.

The structure of the paper is as follows: First, we give an overview of related work done in this field in Sec. 2 and provide an overview of the Intel Xeon Phi architecture compared to Xeon architecture in Sec. 3. We start the performance evaluation in Sec. 4 benchmarking the memory subsystem, selected OpenMP constructs and CG method. Then we present the NAS parallel benchmarks and some real-world application codes, which use different methods to utilize the coprocessor, in Sec. 5 and Sec. 6, before we conclude this paper in Sec. 7.

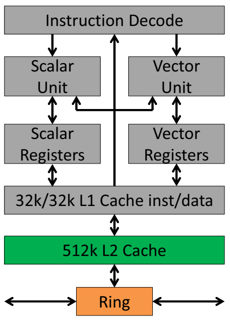
2 Related Work

Previous studies show that throughput-oriented processors like GPUs are one way to fulfill the requirement for more and more compute capabilities. This is not only valid for dense linear algebra kernels [18], but also for memory-bound kernels like sparse matrix vector multiplication [2] (depending on the matrix storage format). In order to benefit from the GPU compute capabilities in general-purpose CUDA applications it is essential to understand the underlying hardware architecture in addition to the programming model [6]. Thus, the effort for porting scientific applications to CUDA or OpenCL can be much higher compared to directive-based programming models like OpenMP [19].

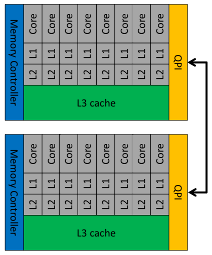
While early experiences on Intel Xeon Phi coprocessors revealed that porting scientific codes can be relatively straightforward [15], other studies also show that this does not necessarily mean that a reasonable performance can be reached without any architecture-specific optimizations like vectorization, software prefetching, SIMD intrinsics, large TLB tables, hardware-supported gather or the correct padding and alignment. [14] showed that the baseline implementation of an compute-intensive radar computation program can be slightly faster on one Xeon Phi compared to a two-socket Sandy Bridge (SNB) system, but that 2x speedup can only reached with architecture-specific optimizations and modifications of the algorithm. [20] gained similar results for multigrid methods which are widely used to accelerate the convergence of iterative solvers. In their study the baseline implementation of the code is even slower on one Xeon Phi compared to two Sandy Bridge processors, but also benefits from the coprocesser after specific optimizations. While [14,20] concentrate on one specific



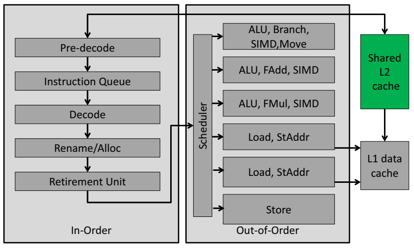
(a) High-level architecture of the Xeon Phi coprocessor.



(b) Functional units of the Xeon Phi coprocessor core.



(c) High-level architecture of the SNB system.



(d) Functional units of the Xeon processor core.

Fig. 1. CPU and core architectures of the Intel Xeon processor and the Intel Xeon Phi coprocessor

application, we focus on a wider range of smaller compute kernels as well as real-world application studies to assess the performance of OpenMP programs.

3 Architecture Comparison

In this section, we compare the architectural differences between the Intel Xeon E5 processors and the Intel Xeon Phi coprocessors.

The Intel Xeon Phi coprocessor provides a shared-memory many-core CPU that is packed on a PCI Express extension card. The specific version used here has 60 cache-coherent cores clocked at 1.053 GHz and 8 GB of coprocessor memory. Each core has 32 KB L1 instruction and data cache and a 512 KB L2 cache. A ring network connects all cores with each other and with memory and I/O devices (see. Fig. 1(a)). Every core in the SNB system has the same amount of L1 and L2 cache as the Xeon Phi cores, but there is also a shared 30 MB L3 cache on the SNB chip. The system used for our experiments consists of two 8-core chips clocked at 2.0 GHz, connected through the Intel Quick Path Interconnect (QPI) (see Fig. 1(c)). Hence, the two-socket SNB-system offers a NUMA architecture while the Xeon Phi has a flat memory model.

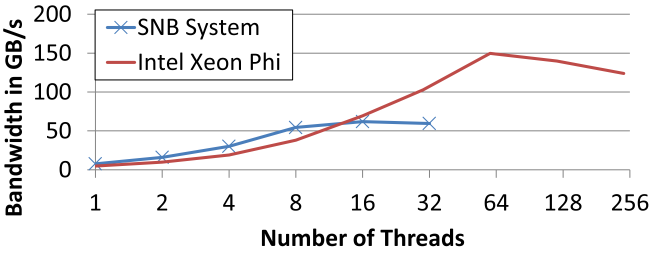


Fig. 2. Memory bandwidth on the SNB and the Intel Xeon Phi system, measured with the stream benchmark

The differences between the micro-architecture of the coprocessor and Xeon processors are more substantial (see Fig. 1(b) and 1(d)). The cores in the Xeon Phi chip are designed to deliver a high compute power per consumed watt. They are clocked comparably slowly at 1.053 GHz and execute instructions in-order. The strength of these cores is the vector unit, which can handle 512 bit vectors.

In contrast the SNB cores use a complex out-of-order engine, which is one contribution to the higher power demand of a single SNB core. However the out-of-order engine can optimize code execution on the core and depending on the executed instruction stream this may speedup execution a lot. The SNB cores are clocked at 2 GHz and support AVX vector operations with 256 bit vectors. Overall the peak performance of one core is nearly the same, the SNB core can deliver 16 GFLOPS and the Xeon Phi core 16.8 GFLOPS.

If we ignore the fact that the Xeon Phi needs a host system and look at it as a SMP system, both investigated systems consume roughly the same amount of power (250 W), space (one blade) and cost roughly the same amount of money, which makes this comparison valuable.

4 Kernel Benchmarks

This section presents some basic performance data for the Xeon Phi and SNB-system. For all benchmarks and applications investigated porting to the Xeon Phi required only to set `-mmic` as a compiler option for the Intel compiler.

4.1 Memory Benchmarks

The memory subsystem on both investigated platforms differs a lot since the SNB system uses DDR3 RAM whereas the Intel Xeon Phi is equipped with GDDR5 RAM. Here, we use simple benchmarks to highlight the differences in memory bandwidth and latency for both systems.

The stream benchmark [12] is a standard package to measure the available memory bandwidth on a system. Fig. 2 shows the results for the Triad vector operation ($\bar{a} = \bar{b} + x * \bar{c}$) for a memory footprint of roughly 2 GB. A good thread to core mapping was ensured with the affinity support of the Intel Compiler. We set `KMP_AFFINITY` to

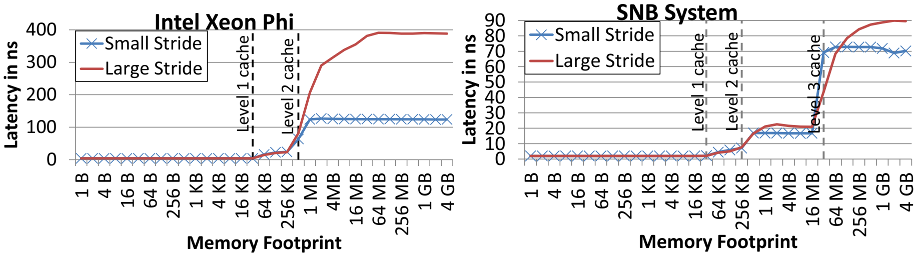


Fig. 3. Memory latency on the SNB system and the Xeon Phi system for different memory footprints. A random stride is used to walk through the memory to prevent prefetching. The small stride most probably hits on the same page whereas the large stride always hits on one of the next pages, causing also a TLB miss.

`balanced` on the Xeon Phi and to `scatter` on the SNB system, since these options delivered best results.

On both systems the bandwidth rises with the number of threads for a low number of threads. When 8 threads on the SNB or 60 threads on the Phi system are used the bandwidth reaches 62 GB/s or 150 GB/s, respectively. After this peak the bandwidth slightly drops down on both systems.

Next, we investigated the latency of the memory. We implemented a pointer chasing benchmark, similar to the latency test in lmbench [13]. We use a stride with a random offset to avoid latency hiding by hardware or software prefetching. Fig. 3 shows the latency for two different strides on both systems. One stride is small enough to hit the same memory page in most cases, whereas the large stride will always hit the next page if the memory footprint is large enough. This may cause a TLB miss if the corresponding TLB entry is not cached.

If the memory fits into the caches, the latency slightly rises with every cache level and the small or large stride does not make any difference since the TLB cache is large enough to keep all entries in the cache on both systems. With a memory footprint which only fits into the main memory the results diverge: On the SNB system, the latency is about 55 ns for the small and about 75 ns for the large stride, whereas it is 130 ns (small stride) and 400 ns (large stride) on the Xeon Phi. For the small stride the ratio of clock tick to memory latency is nearly the same on both systems since the Xeon Phi is clocked at 1 GHz and the SNB at 2 GHz. In contrast for the large stride the ratio is 400 clock ticks/cache miss and 150 clock ticks/cache miss on the Xeon Phi and SNB, respectively.

Concluding, the memory on the Xeon Phi can deliver a very high bandwidth compared to the SNB system, but the latency is worse for large strides between the data accesses.

4.2 OpenMP Constructs

The performance of the OpenMP runtime can be essential for the overall scalability of OpenMP applications. Here, we investigate the overhead of synchronization primitives in the Intel OpenMP runtime. First, we use the `syncbench` benchmark, which is part of the EPCC microbenchmarks [4] to measure the overhead of a `parallel` `for`, a `barrier` and a `reduction` operation in OpenMP. Second, we use self-implemented extensions (see [16]) of the benchmark to investigate the overhead for OpenMP tasks and for nested parallel regions, the two only ways in OpenMP to express multi-level parallelism. For nested parallel regions we use an outer parallel region with two threads and vary the number of threads in the inner region. For tasks we examine the `single-producer` (one thread creates all tasks) and the `parallelproducer` (all threads create tasks in parallel) patterns for task creation.

Table 1. Overhead in microseconds of OpenMP synchronization constructs without nesting (top), in an inner nested parallel region (middle) and for OpenMP Tasks (bottom) on the Intel Xeon Phi and on the SNB system for a different number of threads

	Intel Xeon Phi			SNB system		
EPCC syncbench						
#threads	Parallel for	barrier	reduction	Parallel for	barrier	reduction
16	13.81	5.83	21.61	3.47	2.05	5.83
32	15.85	8.21	24.80	24.36	31.78	58.90
60	17.71	9.96	29.56			
240	27.56	13.37	48.86			
Nested Parallel Regions						
#threads	Parallel for	barrier	reduction	Parallel for	barrier	reduction
2 * 8	56.67	5.47	57.83	13.89	1.79	16.86
2 * 16	117.17	7.21	130.68	27.61	2.39	32.13
2 * 30	318.93	7.74	336.03			
2 * 120	1774.59	13.14	1824.63			
OpenMP Tasks						
#threads	serial-producer	parallel-producer		serial-producer	parallel-producer	
16	81.18	1.67		63.25	0.74	
32	165.50	1.78		146.41	4.11	
60	294.55	3.54				
240	1355.90	8.39				

The systems differ in the number of cores which makes a one to one comparison of the overhead difficult. On the SNB system our experiences have shown that most applications deliver best performance for 16 (1 thread per core) or 32 threads (1 thread per hyperthread). On the Xeon Phi it takes 60 or 240 Threads, respectively, to utilize all cores or hyperthreads. We ensured the usage of one core per thread by thread binding for 16 and 60 threads on the SNB and Xeon Phi, respectively.

Table 1 shows the measurement results for the SNB system and for the Xeon Phi across different numbers of threads. For the non-nested constructs the overhead of using hyperthreads is much larger on the SNB system. If all cores start one thread, the SNB system is 3-5 times better than the Xeon Phi, whereas the time is nearly identical for the `reduction` and `parallel for`, when all hyperthreads are used.

For nested parallel regions, the overhead is much larger on the Xeon Phi system, whereas it is nearly the same (compared to the non-nested regions) on the SNB system. A reduction operation with 120 threads in the inner nested region takes 1824 microseconds, whereas the worst case on the SNB system (16 inner threads) takes only 32 microseconds. This shows that nested parallel regions introduce more overhead on the Xeon Phi system than on the SNB. For OpenMP tasks on the Xeon Phi, the overhead for the `single-producer` pattern is in the same order as for the nested parallel regions. Creating one task takes about 1355 microseconds, whereas it only takes 146 microseconds on the SNB system. However, creating tasks in parallel with the `parallel-producer` pattern works much better, here one task creation takes about 8 microseconds, which is only 2x more than on the SNB and much less than for the `serial-producer` pattern. The reason is that the tasks can be created in separate task queues with this pattern, whereas the `single-producer` pattern requires locking of the task queue when all threads steal tasks out of this queue.

We conclude that the OpenMP runtime on the Phi as SMP system on a single chip can handle thread and task creation without introducing much more overhead than on the SNB system although the number of threads is much larger on the Xeon Phi. If nested parallelism is needed to utilize the large number of threads available, the `parallel-producer` pattern with tasks seems to be an appropriate way to express this parallelism and it should be preferred over nested parallel regions if possible. Generally, if in a producer-consumer scenario only one thread is responsible for creating tasks to be executed by the other threads, increasing the core count while decreasing the single core performance (i.e. clock speed) as on the Xeon Phi may lead to the creator becoming a bottleneck.

4.3 Sparse-Matrix-Vector-Multiplication in a CG Method

To evaluate the performance of a real-world compute kernel, we use a CG solver [11]. The runtime of the algorithm is dominated by the Sparse-Matrix-Vector-Multiplication (SpMV), so we concentrate our analysis on this operation. Depending on the sparsity pattern of the matrix an adequate load balancing is needed. In the case of the CG-method the optimal load balancing can be reached by using a pre-calculated number of rows for each thread depending on the number of nonzero values per row, instead of using a static schedule of an OpenMP work sharing construct. If this is not possible for some problem class, OpenMP also offers some means for load balancing: The first is to use a dynamic schedule with an appropriated chunk size for work sharing construct, the second is to use OpenMP Tasks.

On ccNUMA machines, correct data and thread placement is essential [5]. For that reason we initialize the data in parallel in the pre-calculated version to distribute the pages over the sockets and bind the threads to the cores to avoid thread migration. For the two alternative implementations, an optimal data placement is not possible because of the dynamic scheduling, so that we use a static schedule for the data initialization. However, in [16] and [17] we have shown that at least for the tasking approach the Intel OpenMP runtime works quite well for the `parallel-producer multiple-executors` pattern. Since the two test systems differ in amount and efficient usability of hardware threads, we use different binding strategies, which are

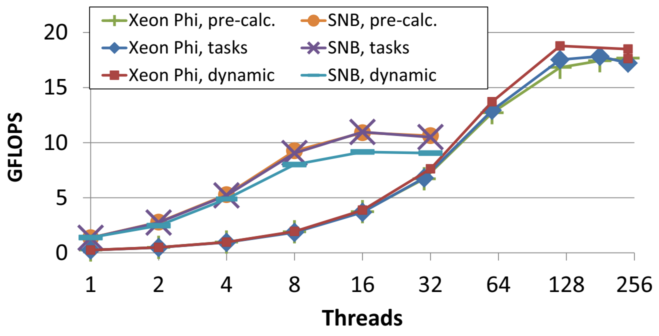


Fig. 4. SpMV performance (1000 CG iterations) on the SNB and the Intel Xeon Phi system

optimal for this kernel and the corresponding hardware. The matrix represents a computational fluid dynamics problem (Fluorem/HV15R) and is taken from the University of Florida Sparse Matrix Collection [8]. The matrix is stored in compressed row storage (CRS) format. The matrix dimension is $N = 2,017,169$ and the number of non-zero elements is $nnz = 283,073,458$, which results in a memory footprint of approximately 3.2 GB. Hence, the data set does not fit into the caches.

Fig. 4 shows the performance results for the three implementations of the SpMV (1000 CG iterations) on both systems. The version using the pre-calculated distribution of the non-zeros reaches the same performance and scalability as the tasking version on the two-socket SNB system because both have been optimized for big shared-memory systems. The measurements show that the corresponding two unmodified, cross-compiled Intel Xeon Phi implementation variants also end up at roughly the same performance. In contrast to the SNB system, the worksharing version using a dynamic schedule works even better on the Intel Xeon Phi. The reason for the different behavior of the two systems is that an optimal data distribution cannot be achieved with a dynamic schedule on ccNUMA machines, which is not an issue on the Xeon Phi. The direct comparison between the two SMP systems shows that the Xeon Phi can profit from the higher memory bandwidth for this kernel and reaches a 1.7x better performance than the two Sandy Bridges. While the peak is reached at about 11 GFLOPS on the SNB system with using only one hardware thread per core, the maximum performance on the Xeon Phi is at about 18.8 GFLOPS with 120 threads (2 hardware threads per core). In [7] we showed with the help of the Roofline Model [21] that this performance is close to the theoretical maximum taking the memory-bound character of this kernel into account.

The results show that for this kernel benchmark OpenMP tasking, as well as OpenMP worksharing with different scheduling strategies, run fine on the Intel Xeon Phi without any special tuning for the architecture.

5 NAS Parallel Benchmarks

The NAS Parallel Benchmarks [1] are a set of benchmarks designed to evaluate the parallel performance of parallel computers. We ran the reference implementation without any code change. On both systems we enabled compiler optimization and used version 13.0 of the Intel Compiler.

Table 2. Runtime (in seconds) and speedup of the NAS parallel benchmarks on the Xeon Phi and the SNB system

	SNB			Xeon Phi		
Benchmark	1 Thread	32 Threads	Speedup	1 Thread	240 Threads	Speedup
IS	23.12	1.38	16.75	192.49	2.46	78.25
EP	186.81	8.11	23.03	1518.42	13.34	113.82
MG	64.04	8.03	7.98	498.94	9.63	51.81
FT	306.11	19.19	15.95	2393.01	53.97	44.34
BT	1241.63	82.61	15.03	9433.52	132.29	71.31
SP	826.25	137.69	6.00	12264.29	164.59	74.51
LU	1109.76	62.23	17.83	9835.09	163.33	60.22

Table 2 shows the runtimes for problem size `C` of all benchmarks on both systems with one thread and with best effort performance, which was when all threads were used. The speedup on the SNB system is between 6 and 24, on the Xeon Phi system between 44 and 114. This shows that the benchmarks scale well on both systems and that the Xeon Phi system can deliver a good scalability for standard kinds of applications. But the serial performance on the Xeon Phi is much lower compared to the SNB system. With one thread the benchmarks run between 7.5 and 15 times slower on the Xeon Phi. Given that the theoretical peak performance of one phyiscal core is nearly the same for both, this is a surprising result. Although the speedup on the Xeon Phi system is quite impressive with up to 114 on 60 cores, the Xeon Phi system is in total slower for every benchmark, when all resources are used.

6 Application Case Studies

After studying the performance of kernels and benchmark codes, we investigated the performance of four real-world codes of the RWTH Aachen University, namely:

iMOOSE: The innovative Modern Object Oriented Solver Environment (iMOOSE) is a finite elements package developed by the Institute of Electrical Machines[1] at RWTH Aachen University. The native compilation of this heavy C++ code ($\sim$ $300,000$ lines of code) parallelized with OpenMP worked without any problems using the Intel Compiler. In our measurements we investigate a three-dimensional model of permanent-magnet excited synchronous machine. We only look at the solving process which uses a CG-solver and dominates the total serial run time on the host system (up to 90 %).

FIRE: The Flexible Image Retrieval Engine (FIRE) [9], developed at the Human Language Technology and Pattern Recognition Group of RWTH Aachen University, takes a set of query images and for each query image it returns a number of similar images from an image database.

NestedCP: NestedCP [10] is developed from the Virtual Reality Group of the RWTH Aachen University and is used to extract critical points in unsteady flow field datasets.

[1] `http://www.iem.rwth-aachen.de`

Table 3. Elapsed time for the applications on Intel Sandy Bridge (SNB) and Intel Xeon Phi Coprocessor

	SNB			Xeon Phi		
Application	1 Thread	best (#threads)	Speedup	1 Thread	best (#threads)	Speedup
iMOOSE	104.68	12.20 (16)	8.58	1243.54	15.59 (240)	79.74
FIRE	284.60	16.68 (32)	17.06	2672.71	38.25 (234)	98.02
NestedCP Nested	46.99	3.21 (32)	14.62	845.14	35.58 (240)	23.76
NestedCP Tasking	47.34	2.43 (32)	19.47	848.34	11.14 (240)	76.16
NINA	470.06	61.16 (16)	7.68	1381.94	27.29 (177)	50.64

Critical points are essential parts of the velocity field topologies and extracting them helps to interactively visualize the data in virtual environments. Two versions of the code were investigated, first a version parallelized with nested parallel regions and second, a version using OpenMP tasks to express the parallelism on the same levels.

NINA: The software package for the solution of Neuromagnetic INverse lArge-scale problems (NINA) was developed by Bücker, Beucker and Rupp [3] and deals with the reconstruction of focal activity in the human brain. It includes computations of matrix-vector products using a matrix of dimensions 128×512000. Here, we use an established C framework in a simplified version that mimics the original MATLAB approach.

Table 3 shows the runtime of the example applications on the SNB system and the Xeon Phi. Noticeable is that nearly all applications gain a good speedup on the Xeon Phi system of 50 to 80. The only exception is the NestedCP version parallelized with nested parallel regions, the tasking version instead delivers a speedup of 76. This confirms our assumption from Sect. 4.2 that tasking is a more appropriate way to express multi-level parallelism on the Xeon Phi system. However, although the scalability is good for all codes on both systems, the total runtime is higher on the Xeon Phi except for one code. The reason again is the serial runtime of the Xeon Phi cores. iMOOSE, FIRE and NestedCP are slower by a factor of 10 to 18 compared to one SNB core. NINA is only slower by a factor of three with one thread and because of the good scalability on the Xeon Phi, the system outperforms the SNB system by a factor of 2.2. A profile for the NINA code showed that roughly 95 % of the kernel execution time is spent in a dense matrix-vector multiplication which is performed very often. This memory access pattern and floating point operations of this operation is very similar to the stream benchmark, where two vectors are multiplied. Since all matrix elements are needed only once, the operation is memory bandwidth bound and since the accesses are consecutive in the matrix and the vector, latency is not important. The stream benchmark has shown that the Xeon Phi can reach a about 2.5 times higher memory bandwidth compared to the SNB system, this is why the NINA code performs well here. All the other codes do not have one single hotspot and they do not use dense linear algebra. Our assumption is that they profit much more from the out-of-order execution capabilities of the SNB cores and thus the SNB system outperforms the Xeon Phi for all these codes.

7 Conclusion

We investigated the performance of Intel's new Xeon Phi coprocessor, if it is used as a standalone SMP system. There are two basic differences between the Xeon Phi and the Sandy Bridge system we used for comparison. First, the Xeon Phi offers many more cores (60) and hardware threads (240) than the SNB system (16 cores / 32 hardware threads). Second, the design of the Xeon Phi cores is simpler and uses in-order execution, whereas the Sandy Bridge cores can do out-of-order execution. We find that the memory bandwidth of the Xeon Phi is about 2.5x higher than the SNB system, but the memory latency for large strides is much lower on the SNB system. Furthermore, we measured the overhead of OpenMP synchronization constructs for single and multi-level parallelism, as well as the overhead introduced by task regions. Our findings are that the overhead for these constructs are in the same order of magnitude for both systems, although a much larger number of threads is needed on the Xeon Phi to utilize all resources of the chip. Thus, the OpenMP runtime should not prevent applications from scaling to a large number of threads on the new platform. Indeed, the NAS parallel benchmarks and all user applications investigated (iMOOSE, FIRE, NestedCP and NINA) have shown a good scalability on the Xeon Phi system between 50 and 113. However, our results also show that the serial performance of one Xeon Phi core is outperformed by a SNB core by a factor of 8-12 for many applications. This leads to a better overall performance on the SNB system for most of these applications. NINA was the only application that delivered a better overall performance on the Xeon Phi where it was 2.2 times faster than on the SNB system. The code executed dense matrix vector multiplications in 95 % of the compute-intensive parts. The other codes have less predictable memory access patterns. We assume that the high memory latency of the Xeon Phi is an issue here since the in-order engine cannot hide the latency in contrast to the out-of-order engine of the SNB. According to our experience, if the Xeon Phi is used as a stand alone SMP, it does not deliver a performance comparable to a Sandy Bridge system for many applications, because of the poor single core performance. For some applications, like the NINA code, the performance is fine, but for most of our codes the SNB system is the platform of choice. Future work is to investigate which kind of applications can be tuned to perform well on the Intel Xeon Phi system and which tuning steps are necessary.

References

1. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The nas parallel benchmarks. Technical report, NASA Ames Research Center (1991)
2. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proc. of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 18:1–18:11. ACM, New York (2009)
3. Bücker, H.M., Beucker, R., Rupp, A.: Parallel Minimum p-Norm Solution of the Neuromagnetic Inverse Problem for Realistic Signals Using Exact Hessian-Vector Products. SIAM J. on Scientific Computing 30(6), 2905–2921 (2008)
4. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proc. of First European Workshop on OpenMP, pp. 99–105 (1999)

5. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Wagner, M.: Data and Thread Affinity in OpenMP Programs. In: Proc. of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?, MAW 2008, pp. 377–384. ACM (2008)
6. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. 68(10), 1370–1380 (2008)
7. Cramer, T., Schmidl, D., Klemm, M., an Mey, D.: OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In: Proc. of the Many-core Applications Research Community Symposium, pp. 38–44 (November 2012)
8. Davis, T.A.: University of Florida Sparse Matrix Collection. NA Digest 92 (1994)
9. Deselaers, T., Keysers, D., Ney, H.: Features for image retrieval: an experimental comparison. Information Retrieval 11(2), 77–107 (2008)
10. Gerndt, A., Sarholz, S., Wolter, M., an Mey, D., Bischof, C., Kuhlen, T.: Nested OpenMP for Efficient Computation of 3D Critical Points in Multi-Block CFD Datasets. In: SC 2006 Conference, Proc. of the ACM/IEEE 2006, p. 46 (November 2006)
11. Hestenes, M.R., Stiefel, E.: Methods of Conjugate Gradients for Solving Linear Systems. J. of Research of the National Bureau of Standards 49(6), 409–436 (1952)
12. McCalpin, J.: STREAM: Sustainable Memory Bandwidth in High Performance Computers
13. McVoy, L., Staelin, C.: lmbench: portable tools for performance analysis. In: Proc. of the 1996 Annual Conference on USENIX, ATEC 1996, p. 23. USENIX Association, Berkeley (1996)
14. Park, J., Tang, P.T.P., Smelyanskiy, M., Kim, D., Benson, T.: Efficient backprojection-based synthetic aperture radar computation with many-core processors. In: Proc. of the Int. Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 28:1–28:11. IEEE Computer Society Press, Los Alamitos (2012)
15. Schulz, K.W., Ulerich, R., Malaya, N., Bauman, P.T., Stogner, R., Simmons, C.: Early Experiences Porting Scientific Applications to the Many Integrated Core (MIC) Platform. Technical report, TACC-Intel Highly Parallel Computing Symposium (April 2012)
16. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing OpenMP Tasking Implementations on NUMA Architectures. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 182–195. Springer, Heidelberg (2012)
17. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Task-Parallel Programming on NUMA Architectures. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 638–649. Springer, Heidelberg (2012)
18. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Proc. of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008 (2008)
19. Wienke, S., Plotnikov, D., an Mey, D., Bischof, C., Hardjosuwito, A., Gorgels, C., Brecher, C.: Simulation of bevel gear cutting with GPGPUs - performance and productivity. Computer Science - Research and Development 26, 165–174 (2011)
20. Williams, S., Kalamkar, D.D., Singh, A., Deshpande, A.M., Van Straalen, B., Smelyanskiy, M., Almgren, A., Dubey, P., Shalf, J., Oliker, L.: Optimization of geometric multigrid for emerging multi- and manycore processors. In: Proc. of the Int. Conference on HPC, Networking, Storage and Analysis, SC 2012 (2012)
21. Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM 52(4), 65–76 (2009)

A Hybrid Parallel Barnes-Hut Algorithm for GPU and Multicore Architectures

Hannes Hannak[1,⋆], Hendrik Hochstetter[2], and Wolfgang Blochinger[1]

[1] Institute of Parallel and Distributed Systems, University of Stuttgart, Germany
first.last@ipvs.uni-stuttgart.de

[2] Computer Graphics and Multimedia Systems Group, University of Siegen, Germany
hochstetter@nt.uni-siegen.de

Abstract. Using the Barnes-Hut algorithm as an example we deal with the design of parallel algorithms that are able to exploit multicore CPUs and GPUs conjointly. Specifically, we demonstrate how to modularize a parallel application according to specific aspects of parallel execution. This allows for a flexible assignment of individual modules to the two parallel architectures based on their actual performance characteristics. Furthermore, we discuss a hybrid module for the most time consuming part of the algorithm that utilizes CPU and GPU simultaneously employing a novel load balancing heuristic. Our experimental evaluation shows that our method greatly increases overall efficiency by allowing to deploy the optimal configuration of modules for each individual computer system.

1 Introduction

In recent years, GPU-based parallel computing has attained considerable interest. However, most algorithm designs presented so far exclusively exploit the GPU for executing parallel tasks while the (potentially many) cores of the CPU are running idle.

In this paper, we discuss a modularized parallel application which enables a flexible assignment of individual parts of the algorithm for execution on the GPU, the CPU cores, or on both platforms in parallel. We chose the Barnes-Hut algorithm as an example for our studies. It computes the evolution of a large set of particles based on the forces individual particles exert on each other. In contrast to existing works (e.g. [7,8]), our approach specifically takes into account that, depending on the actual capabilities of the hosts' CPU cores and GPU, different assignments of computational parts of the Barnes-Hut algorithm to platforms may yield the most efficient solution.

Achieving such a degree of flexibility is especially important when executing parallel applications (e.g. [6,14]) in highly heterogeneous environments. A typical

⋆ Hannes Hannak is supported by a grant of the Landesgraduiertenförderung Baden-Württemberg.

example of this kind are Desktop Grids (e.g. [1,16]) which combine the computing power provided by volunteers into a global computing grid and are known to aggregate a huge variety of hardware resources [10]. Also, Cloud Computing instances exhibit a similar scenario: Even though the user is guaranteed certain minimal hardware limits, the actual configurations and capabilities can vary considerably.

2 Preliminaries

2.1 The Barnes-Hut Algorithm

N-Body methods simulate the dynamics occurring in a set of N particles based on the forces (e.g. gravity) the particles exert on each other. Simulations proceed in discrete timesteps, each resulting in new particle positions. As exact approaches consider the forces between each pair of particles, $\mathcal{O}(N^2)$ calculations are necessary per timestep. To treat larger numbers of particles, approximation methods have been proposed that considerably decrease the quadratic complexity [17].

One well known hierarchical approach to the N-body problem is the Barnes-Hut algorithm [2] which exhibits $\mathcal{O}(N \log N)$ time complexity. It employs a tree data structure to approximate forces acting on the individual particles. The leaf nodes of the *Barnes-Hut tree* hold the positions and masses of single particles whereas each inner node represents particle equivalents summarizing the positions and masses of all particles of the subtree rooted at that node. Thus, inner nodes act as *pseudo particles* which correspond to a certain area of the simulation space.

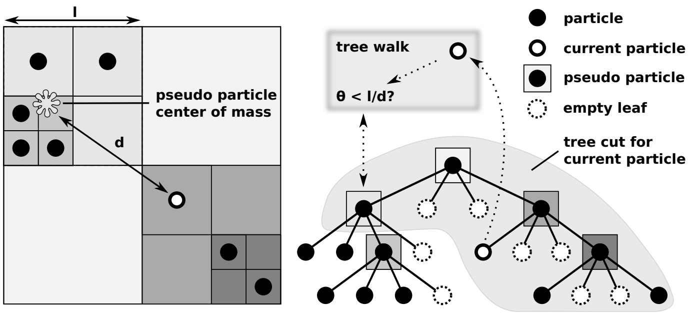


Fig. 1. Simulation space and corresponding Barnes-Hut tree (same colors denote respective pseudo particles). For clarity we show a two-dimensional space, however, the same principles hold in 3D, too. We also illustrate the parameters of the opening criterion applied for steering the tree walk for one particle and the resulting tree cut.

The computations of a timestep consist of the following consecutive steps:

S1) Tree Building: The simulation space is recursively divided into equally sized subspaces until each of them contains at most one particle (cf. Fig. 1).

These subspaces define the leaf nodes of the Barnes-Hut tree which store the position and the mass of the contained particle.
S2) Computing Pseudo Particles: For each inner node in the tree, the pseudo particle data has to be computed. The mass is calculated as the sum of the masses of the child nodes and the position as the center of mass.
S3) Force Evaluation: Forces are computed for each particle separately by traversing the tree, starting with the root node (*tree walk*). To every pseudo-particle an *opening criterion* is applied to decide whether it can be used for force evaluation or if the node must be expanded. The opening criterion is given by $\theta < l/d$, where θ is a chosen approximation factor, l the length of the subspace the node represents and d the distance of the node's position to the current particle (cf. Fig. 1). Thus, choosing a smaller value for θ decreases the simulation's approximation error but increases running time as more interactions have to be computed. If the opening criterion applies to an inner node, the algorithm executes recursively on all child nodes and adds up the individual forces. Otherwise, either the pseudo particle data is used to compute forces or, in case of a leaf node, the particle data. The set of interaction partners define a particle's *tree cut* as that part of the tree that is needed for the particle's force evaluation.
S4) Particle Update: Once the total force on each particle is known, the algorithm calculates the new positions and velocities of the particles (by applying Newton's laws). These serve as input for the next timestep.

2.2 GPGPU Computing

In contrast to CPUs, GPUs consist of a hierarchical combination of many primitive processor cores. On the nVidia platform, e.g., each *Streaming Multiprocessor (SM)* comprises a set of simple *Streaming Processors (SP)* which share a common program counter and control unit. All SPs within an SM execute instructions synchronously and thus follow the SIMD architecture model, whereas different SMs operate independently and follow the MIMD model.

We use nVidia GPUs and the CUDA SDK to speed up the Barnes-Hut algorithm. nVidia GPUs offer hardware support for the scheduling of threads. When running kernels, the scheduler selects small groups of threads, called *warps*, for execution on SMs. If single threads of a warp follow different program paths, all paths have to be followed one after another as each SM only possesses one control unit. So, to achieve high efficiency with the computational resources of GPUs, one must avoid thread divergence as if-then-else constructs have to be serialized in the SIMD model. Because of this limitation, GPUs are best suited to compute regularly structured problems.

To attenuate this constraint, modern GPUs introduce *warp vote functions* to allow all threads of a warp to evaluate conditions jointly. For a Boolean variable `var`, the warp vote function `__all(var)` returns true in each thread of the warp if `var` is true in every thread. If `var` is false in only one thread, false is returned in all threads of the warp. Thus, warp vote functions in conditional expressions ensure coherent program paths within warps and avoid thread divergence which can greatly increase efficiency.

3 A Barnes-Hut Method for Hybrid Architectures

To efficiently use the computing power provided by the CPUs and GPUs of different hosts, we designed a modular method that employs the most suitable platform for each of the steps of the Barnes-Hut algorithm.

We first describe the structure and organization of the simulation data in Section 3.1 followed by the characteristics of the individual steps that lead to our modularization in Section 3.2. As force evaluation is the most expensive step, we afterwards focus on the implementation of this module. In Section 3.3 we lay out how we use the GPU for force evaluation. To exploit the maximum available computational power, we designed an additional module that runs on both platforms in parallel (*hybrid mode*) which we discuss in Section 3.4. Furthermore, we invented a new load balancing mechanism to minimize processor idle time. We present details and advantages of this method in Section 3.5.

3.1 Data Structures and Data Organization

As described in Section 2.1, the Barnes-Hut algorithm can be divided into a sequence of distinct steps each of which depends on the outcome of preceding ones but can be solved as an independent entity. In our approach *modules* encapsulate the functionality of these steps. Depending on its implementation, each module can be executed in parallel on either one or both platforms. The definition of explicit interfaces between successive modules allows us to run the algorithm with an arbitrary assignment of modules to execution platforms. The best configuration for a specific host can be determined by initially performing a benchmark run evaluating all possible combinations.

In principle, we replicate particle and tree data in the memories of the two platforms to minimize costly data transfers between CPU and GPU memory during computations. However, if data that is needed by the subsequent module has been changed on one platform, e.g. by updating particle positions or tree building (cf. Tab. 1), it must be transferred if the execution crosses platform borders. As the resulting memory latencies reduce efficiency, we minimize idle time by overlapping data transfers with the module execution.

To facilitate modularity and to speed up data transfers and computations, we introduce two further steps/modules between tree building (S1) and computing pseudo particles (S2):

S1a) Tree Linearization: To allow for an efficient transfer between platforms and to speed up tree traversals, we transform the tree data structure into a set of linear arrays: The *tree array* reflects the recursive structure of the tree, the *next* and *more arrays* allow for a stack-free tree traversal, and the *particle indirection array* allows particles to be accessed according to their spatial position.
S1b) Particle Sorting: Spatial proximity within particles can be exploited in a number of cases during the computation to speed up execution and memory access. As particles close to each other need to interact frequently, keeping them in close memory positions increases cache efficiency. Thus, we sort particles through

the particle indirection array using three dimensional space filling Peano-Hilbert curves.

3.2 Modularization

Each of the steps described above exhibits different characteristics, especially with respect to data dependencies. In the following paragraphs, we give a detailed account of the characteristics that determined the parallelization strategy for each module. Table 1 provides an overview.

Table 1. Modules with interfaces and platforms (C = CPU, G = GPU, H = Hybrid.)

	Requires	**Provides**	**Platform**
S1	particle data	unsorted tree	C
S1a	unsorted tree	next/more, particle indirection, tree arrays	C
S1b	particle indirection array, particle data	sorted particle data	C, G
S2	tree array, sorted particle data	pseudo particles	C, G
S3	next/more, pseudo particles, sorted particle data	forces	C, G, H
S4	forces, sorted particle data	particle data	C, G

The most time consuming steps of the Barnes-Hut algorithm are tree building (S1) and force evaluation (S3) which both have a time complexity of $\mathcal{O}(N \log N)$. All four remaining steps are of linear complexity. Especially particle sorting (S1b) can be done in linear time as one can determine the actual spatial order of particles from the tree structure in one traversal.

Concerning data access patterns, tree building (S1), tree linearization (S1a) and particle sorting (S1b) are the most complex problems. Tree building is realized by inserting particles into a dynamically changing tree. The way in which the tree is arranged in memory thus depends on the order the particles are processed in. This usually leads to highly irregular access patterns. During tree linearization (S1a), the tree data structures are rearranged in the order of a depth first traversal to speed up succeeding steps. Computing pseudo particles (S2) and force evaluation (S3) benefit from tree linearization (S1a) and particle sorting (S1b) in that they have less irregular access patterns traversing the tree than the steps before. Particle sorting (S1b) allows for regular access patterns on particle data in later steps but has highly irregular access patterns, itself.

In addition to irregular access patterns, tree building (S1), linearization (S1a), and computing pseudo particles (S2) suffer from data dependencies. During tree building (S1), particles are inserted in parallel into a shared tree. Changes to the tree have to be made atomic through locking mechanisms so that no inconsistent states of the tree may arise. This limits the amount of parallelism that can be achieved. Particle sorting (S1b), force evaluation (S3) and particle update (S4), in contrast, do not involve any data dependencies and thus are trivially parallelizable.

Our partition of the Barnes-Hut algorithm into a sequence of unique steps was guided by the characteristics described above. They are summarized in Table 2.

Although force evaluation dominates the running time in sequential implementations with 97 %, all steps have to be parallelized. If only force evaluation was parallelized, following Amdahl's law a maximum speedup of 33.3 could be achieved, no matter how many processors were employed. However, implementing different parallelization approaches for each step can be easily accomplished through our modular design.

Table 2. Properties of the modules of our implementation of the Barnes-Hut algorithm

	Access patterns	Data dependencies	Time complexity	Sequential CPU running time (in %)
S1	highly irregular	yes	$\mathcal{O}(N \log N)$	≈ 2
S1a	highly irregular	yes	$\mathcal{O}(N)$	< 1
S1b	highly irregular	no	$\mathcal{O}(N)$	< 1
S2	irregular	yes	$\mathcal{O}(N)$	< 1
S3	irregular	no	$\mathcal{O}(N \log N)$	≈ 97
S4	regular	no	$\mathcal{O}(N)$	< 1

As GPUs usually use relatively slow, high-bandwidth memory, regular access patterns are crucial to achieve good performance. If irregular access patterns occur, compute time is wasted waiting for memory accesses. In contrast, on CPUs, more sophisticated memory hierarchies with larger and more cache stages are employed which can better hide latencies if irregular patterns occur. We thus expect GPUs to perform best in particle update (S4) and force evaluation (S3) and less well in steps with more irregular access patterns and data dependencies.

Due to the combination of highly irregular access patterns and data dependencies, we expect tree building (S1) and linearization (S1a) to be best suited for an execution on CPUs. The additional steps of tree linearization (S1a) and particle sorting (S1b) provide tree and particle data in an order that allows for more regular memory access patterns, so that the most time consuming step, the force evaluation (S3), can be parallelized efficiently on both CPU and GPU.

3.3 GPU Based Force Evaluation

As motivated above, GPUs are well suited to perform the force evaluation by executing one thread per particle. Because we sort particles (S1b), each warp only processes nearby particles. The force evaluation of nearby particles leads to very similar tree cuts, so we thereby greatly reduce thread divergence.

To completely avoid thread divergence, interaction lists can be employed in GPU-based force evaluation [8,4]. This approach groups nearby particles together. A tree walk computes an interaction list for each group that stores all particles and pseudo-particles that contribute to the groups' forces. Forces acting on the individual particles of a group are computed completely synchronously, using the interaction list. In contrast, like [7], we use warp vote functions to achieve a similar effect. During the tree walk, the opening criterion is evaluated using `__all`. Approximations thus are only acceptable if all threads of a warp agree. Else the threads continue the tree walk with the children of the current node. So, all threads of a warp access the same node data and don't diverge. Evaluating the opening criterion for each particle separately would cause parts of the tree to be traversed to different depths. Using `__all`, all threads of a warp follow the deepest path necessary for any one of the warp's particles, thereby increasing accuracy.

Although more interactions are computed using warp vote functions, the reduced amount of memory accesses and the absence of thread divergence greatly improve performance. Additionally, latency hiding through the scheduling hardware can work more effectively with warp vote functions than interaction lists as the memory bound tree traversal and the compute bound evaluation of interactions are not artificially separated but interleaved.

3.4 Hybrid Force Evaluation

Our modularized approach allows us to combine CPU- and GPU-based implementations of the force evaluation to yield a hybrid module that utilizes both platforms in parallel. To balance the workload among CPU and GPU, each processor evaluates forces for a subset of particles. The size of the subset depends on the processor's computational power.

We model the CPU as a set of equally powerful processors. Because of its hardware-based thread scheduling we can't control the GPU's load balancing. We thus represent the GPU as a single processor. For the same reason, it is difficult to predict GPU running times solely from the problem structure. Thus, we exploit the spatio-temporal stability of N-Body problems. We measure running times of CPU and GPU computations and adjust the GPU's computational power p_{GPU} dynamically in each timestep. That way, we are able to capture the ratio between the computing power of GPU and CPU and to distribute the load, accordingly.

If computations on the GPU (t_{GPU}) took less time than on the CPU (t_{CPU}), p_{GPU} is increased to $p_{\text{GPU}}\left(c_{\text{inc}}\frac{t_{\text{GPU}}}{t_{\text{CPU}}} + (1 - c_{\text{inc}})\right)$ so that the GPU evaluates more forces. If on the other hand the CPU took less time than the GPU, p_{GPU} is decreased to $p_{\text{GPU}}\left(c_{\text{dec}}\frac{t_{\text{GPU}}}{t_{\text{CPU}}} + (1 - c_{\text{dec}})\right)$. Else, p_{GPU} remains unchanged. The constants c_{inc}, c_{dec} and the initial value of p_{GPU} are precomputed in a benchmark run. c_{inc} and c_{dec} determine how fast p_{GPU} is increased or decreased. They are chosen in such a way that no work is stolen from the faster platform due to fluctuations in t_{GPU} or t_{CPU} that may be caused by the different ways in which CPUs and GPUs operate, the operating system, or the system's user.

3.5 A Novel Dynamic Load Balancing Scheme for Force Evaluation

To improve the hybrid force evaluation's load balancing, the varying costs to evaluate forces acting on different particles must be considered, instead of assuming a uniform cost per particle. For particles in dense areas of the simulation space more interactions have to be computed than for particles in sparse areas. Due to the spatio-temporal stability of N-body problems, particles move only

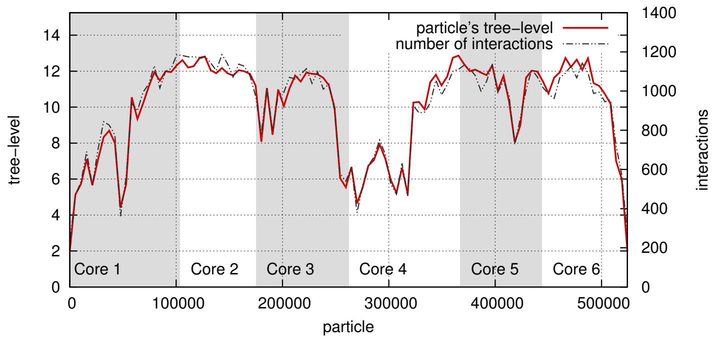


Fig. 2. Comparison between our tree-level heuristic and the number of interactions for one timestep during force evaluation using $\theta = 0.75$. Columns depict the resulting particle distribution among six CPU cores (gray and white).

slightly between timesteps. Thus, the number of interactions also changes slowly and provides a good measure for the computational cost to evaluate forces.

A common approach to load balancing is counting the number of interactions for each particle and using the data to distribute work in subsequent timesteps [17]. As GPUs use hardware-based thread scheduling, this approach isn't very well suited for our hybrid module. When counting interactions on the GPU, more resources are required and less threads can be resident in SMs, which strongly impairs the scheduling mechanism and the GPU's performance.

We thus introduce a novel heuristic approach to predict the number of interactions. It uses only data gathered during one traversal of the tree and does not introduce any overhead into the force evaluation itself. Our method is based on the tree level l of each particle x which turned out to be a good measure for the particle density in the region surrounding that particle (cf. Fig. 2). If l is large, many particles are located close to x and many interactions have to be considered to evaluate the force acting on x. The tree level thus provides a good heuristic to approximate the computational cost for a given particle.

To incorporate our heuristic into the force evaluation, we write a *cost array* containing the prefix sum of the tree level (and thus cost) of each particle. The last element of this array holds the total cost C to evaluate all forces. Each processor i, which may be a CPU core or the GPU, has a computing power p_i that determines the amount of work assigned to it. The system's total computing power is $p_{\text{total}} = \sum_i p_i$. During simulations, each processor i computes forces for a contiguous subset of particles, the lower and upper indices of which are the same as the indices of $C\frac{\sum_{k=0}^{i-1} p_k}{p_{\text{total}}}$ and $C\frac{\sum_{k=0}^{i} p_k}{p_{\text{total}}}$, respectively, in the cost array which can be determined through binary search. Fig. 2 shows a particle distribution resulting from our load balancing scheme. Note how, e.g., Core 1 is assigned a larger particle subset than Core 2, as for most particles assigned to Core 1 less interactions have to be computed.

Table 3. Configuration of the node types used for the experimental evaluation

Type	CPU (Cores / Threads / Frequency)	RAM	GPU	VRAM
I	Intel Core i7 M620 (2 / 4 / 2.67 GHz)	2 GB	nVidia GeForce NVS 3100M	256 MB
II	AMD Phenom II X6 1055T (6 / 6 / 2.8 GHz)	4 GB	nVidia GeForce GTX 660 Ti	2 GB
III	Intel Core i7-2670QM (4 / 8 / 2.2 GHz)	4 GB	nVidia GeForce GTX 570M	1.5 GB

4 Performance Evaluation

To evaluate our approach we performed test runs on a variety of different hardware (see Table 3). For each run we used two colliding galaxies of equal mass as input. Each galaxy consisted of a stellar disk surrounded by a dark matter halo following the Springel-Hernquist model. We used Starscream [5] to create the galaxies and place them on a parabolic orbit.

We denote the configuration of platforms that execute the simulation steps as strings of six characters, representing the six modules. Each module is either executed on the CPU (C), the GPU (G), or in hybrid mode (H) on CPU and GPU in parallel. Speedups are based on the sequential CPU running time on the respective test system. Figure 3 shows the speedups obtained using different combinations of modules on our test systems for varying problem sizes.

In purely CPU-based parallel configurations (CCCCCC), we achieved speedups of 1.8 (type I), about 5 (type II), and 3.7 (type III). On all three systems, the speedups of our heuristic load balancing scheme could compete with those of load balancing based on interaction counting. In the sequential configuration CCCCCC, we measured an overhead of 2 % just to count interactions. For GPU-based force evaluation this introduced an overhead of more than 30 %.

To find the most efficient module combination, we ran all possible combinations on each system. On type I, the best combination turned out to be CCCCHG which obtained speedups of 6 and above. The most efficient combination for type II, CCGGGG, achieved speedups close to 60 for the largest problem sizes. This discrepancy can be explained by the different ratio of GPU to CPU computing power on the test systems. On type II the GPU outperforms the CPU, whereas on type I both have similar peak performance. On type III, the best module combination depended on the problem size. For problem sizes of below $6 \cdot 10^6$ particles, CCCCHG yielded the highest speedups whereas for larger problems CCCGHG was fastest. We attribute this observation to the fact that GPUs only reach their maximum performance if all computational resources are fully utilized. As the GPU on type III is highly capable, this was only achieved for problems with a higher number of particles. As expected, we obtained different results among our test systems, and as such a variety of systems is common in, e.g., Desktop Grids, the results prove our idea of a flexible modular design.

The modern GPU on type II supports the warp vote function `__all`, which decreased the time of the force evaluation by over 50 %. Using the hybrid force evaluation module (CCCCHG) on type I, we measured a speedup of over 20 % compared to configuration CCCCGG. On type III, CCCGHG was over 50 % faster than CCCGGG. This shows that combining the computational power of CPU and GPU is an effective way to increase the performance of simulations.

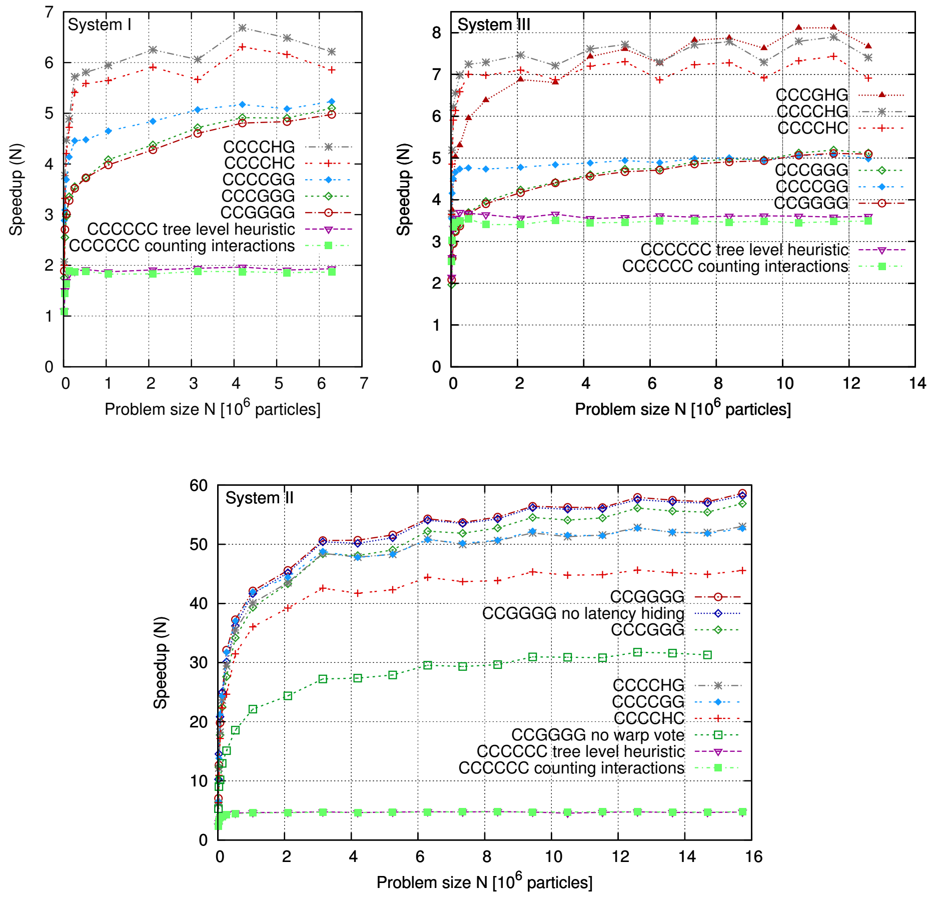


Fig. 3. Speedup of different parallel module combinations compared to purely sequential CPU implementations. Timings averaged over 10 time steps using $\theta = 0.75$.

Memory latencies constituted only about 1 % of the total running time on type II in configuration CCGGGG. Through our latency hiding approach, we reduced latencies by 50 % to below 0.5 %. On type I in configuration CCCCHG, latencies were reduced by over 66 % to less than 0.25 % of the total time.

5 Related Work

In most works, only the computationally most expensive part, the force evaluation, is computed on the GPU, whereas the CPU computes all other steps of the Barnes-Hut algorithm [15,8]. More recent works run the whole simulation on GPUs [7,4]. In contrast, our code is completely modularized to allow arbitrary combinations of CPU- and GPU-based implementations of the individual steps.

Irregular problems like cloth simulation [12] or the Barnes-Hut tree do not easily fit the parallel SIMD model of GPU programming. Construction and traversal of trees on GPUs are topics of ongoing research. Two popular approaches to parallelize construction of trees on GPUs have been proposed in literature: Tree construction can either be accomplished by inserting particles in parallel into a dynamically changing shared tree using locks to prevent race-conditions [7] or the tree can be constructed level by level, which requires particles to be sorted [4].

GPU-based force evaluation can be realized executing one thread per particle. As recursive functions are not supported on older GPU generations, tree traversals have to be done iteratively. Stack-based tree traversals were used in [8,4,7]. Tree traversals using next and more arrays were first employed in [13] and later were used on the GPU [15], as in our work. To prevent thread divergence in SMs, force evaluation can be modified to use interaction lists, as first described in [3]. They were used in GPU-based force evaluation in [8,4]. Instead, like [7], we use warp vote functions to prevent thread divergence.

Our modular design offers opportunities for latency hiding every time execution moves to another platform. As most works employ non-modular designs, little information can be found on this topic. We are aware of only one work discussing latency hiding between GPU and CPU computations. In [11] the CPU determines interaction lists and the evaluation of forces is offloaded to the GPU. However, interaction lists are computed piecewise, so that completed interaction lists can be sent to the GPU while the CPU computation continues.

6 Conclusion

In this paper we introduced a modularized parallelization of the Barnes-Hut algorithm. By defining interfaces between modules and carefully choosing data structures we facilitate efficient module implementations for CPU and GPU that allow a flexible dynamic assignment to platforms. Through the design of hybrid modules that combine the computing power of CPU and GPU, we fully utilize all available computational resources.

Our test results show that for different host systems very different combinations of GPU- and CPU-based modules yield the best overall performance, and that the best combination depends highly on the underlying hardware. Hence, by incorporating our modular design, we are able to improve an existing implementation of the Barnes-Hut algorithm for Desktop Grids [9]. The flexibility and adaptability our modular multi-platform approach offers render it an ideal model for the design of future algorithms for heterogeneous environments.

References

1. Anderson, D.P.: Boinc: A system for public-resource computing and storage. In: 5th IEEE/ACM International Workshop on Grid Computing, pp. 4–10 (2004)
2. Barnes, J.E., Hut, P.: A hierarchical O(N log N) force-calculation algorithm. Nature 324(6096), 446–449 (1986)

3. Barnes, J.E.: A modified tree code: Don't laugh; it runs. Journal of Computational Physics 87(1), 161–170 (1990)
4. Bédorf, J., Gaburov, E., Zwart, S.P.: A sparse octree gravitational n-body code that runs entirely on the GPU processor. Journal of Computational Physics 231(7), 2825–2839 (2012)
5. Billings, J.J.: Starscream – An open source galaxy modeling and simulation tool, `http://code.google.com/p/starscream/` (accessed in February 2013)
6. Blochinger, W., Dangelmayr, C., Schulz, S.: Aspect-oriented parallel discrete optimization on the cohesion desktop grid platform. In: Proc. of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), pp. 49–56 (2006)
7. Burtscher, M., Pingali, K.: An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm. In: Hwu, W.-M.W. (ed.) GPU Computing Gems Emerald Edition, ch. 6, pp. 75–92. Morgan Kaufmann Publishers Inc. (2011)
8. Gaburov, E., Bédorf, J., Zwart, S.P.: Gravitational tree-code on graphics processing units: Implementation in CUDA. Procedia CS 1(1), 1119–1127 (2010)
9. Hannak, H., Blochinger, W., Trieflinger, S.: A desktop grid enabled parallel Barnes-hut algorithm. In: Proceedings of the 31st IEEE International Performance Computing and Communications Conference (IPCCC 2012), pp. 120–129 (2012)
10. Heien, E., Kondo, D., Anderson, D.: A correlated resource model of internet end hosts. IEEE Transactions on Parallel and Distributed Systems 23(6), 977–984 (2012)
11. Jetley, P., Wesolowski, L., Gioachin, F., Kalé, L.V., Quinn, T.R.: Scaling hierarchical n-body simulations on GPU clusters. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2010)
12. Keckeisen, M., Blochinger, W.: Parallel implicit integration for cloth animations on distributed memory architectures. In: Proc. of Eurographics Symposium on Parallel Graphics and Visualization 2004, pp. 119–126 (2004)
13. Makino, J.: Vectorization of a treecode. Journal of Computational Physics 87(1), 148–160 (1990)
14. Meißner, M., Hüttner, T., Blochinger, W., Weber, A.: Parallel direct volume rendering on PC networks. In: Arabnia, H.R. (ed.) Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA 1998. CSREA Press (1998)
15. Nakasato, N.: Implementation of a parallel tree method on a GPU. Journal of Computational Science 3(3), 132–141 (2012)
16. Schulz, S., Blochinger, W., Held, M., Dangelmayr, C.: Cohesion - a microkernel based desktop grid platform for irregular task-parallel applications. Future Generation Computer Systems - The International Journal of Grid Computing: Theory, Methods and Applications 24(5), 354–370 (2008)
17. Singh, J.P., Holt, C., Totsuka, T., Gupta, A., Hennessy, J.: Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. Journal of Parallel and Distributed Computing 27(2), 118–141 (1995)

A Generic High-Performance Method for Deinterleaving Scientific Data

Eric R. Schendel[1,3,4], Steve Harenberg[1,4], Houjun Tang[1,4], Venkatram Vishwanath[3], Michael E. Papka[2,3], and Nagiza F. Samatova[1,4,⋆]

[1] North Carolina State University, Raleigh, NC 27695, USA
[2] Northern Illinois University, DeKalb, IL 60115, USA
[3] Argonne National Laboratory, Argonne, IL 60439, USA
[4] Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA
samatova@csc.ncsu.edu

Abstract. High-performance and energy-efficient data management applications are a necessity for HPC systems due to the extreme scale of data produced by high fidelity scientific simulations that these systems support. Data layout in memory hugely impacts the performance. For better performance, most simulations interleave variables in memory during their calculation phase, but deinterleave the data for subsequent storage and analysis. As a result, efficient data deinterleaving is critical; yet, common deinterleaving methods provide inefficient throughput and energy performance. To address this problem, we propose a deinterleaving method that is high performance, energy efficient, and generic to any data type. To the best of our knowledge, this is the first deinterleaving method that 1) exploits data cache prefetching, 2) reduces memory accesses, and 3) optimizes the use of complete cache line writes. When evaluated against conventional deinterleaving methods on 105 STREAM standard micro-benchmarks, our method always improved throughput and throughput/watt on multi-core systems. In the best case, our deinterleaving method improved throughput up to 26.2x and throughput/watt up to 7.8x.

1 Introduction

Emerging extreme-scale high performance computing (HPC) systems enable high fidelity scientific simulations that generate data at an increasing rate [1]. Yet, these HPC systems and data-intensive applications they support consume energy at an ever-increasing amount [2,3]. Thus, the need for performance and energy efficient data management applications is of utmost importance to maximize throughput/watt while achieving improved scalability and sustainability [4].

To improve performance during scientific data analysis, which is critical for gaining insights from the simulations, simulations often have to *deinterleave* data variables. Upon deinterleaving, the data set for each variable of the simulation is contiguous in memory and storage. This deinterleaved layout is beneficial since most data analyses span multiple time steps of a particular variable [5].

⋆ Corresponding author.

In contrast, most simulations perform calculations using instances of many variables from a current/previous time step. Hence, an *interleaved* layout in memory provides better data locality during simulation runs by keeping each group of variables together in memory for the active time steps, see Figure 1.

Deinterleaving data is frequently necessary after the completion of a simulation step before data analysis and storage. For example, simulations such as FLASH [6], S3D [7] and Nek5000 [8] have variables that are interleaved in memory while most storage and analysis, such as data compression [9,10] and variable precision analytics [11], are performed using a deinterleaved layout. Through performing numerous micro-benchmarks, we found that common deinterleaving methods have poor throughput and energy performance.

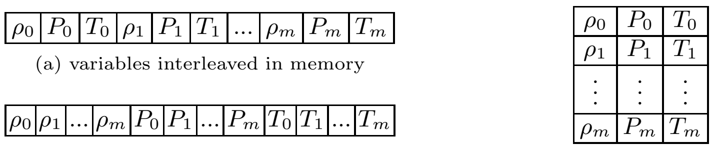
To address this problem, we propose a deinterleaving method that is high performance, energy efficient, and generic to any variable data type. To the best of our knowledge, this is the first deinterleaving method that 1) exploits data cache prefetching, 2) reduces memory accesses, and 3) optimizes the use of complete cache line writes. As a result, our method increases the throughput performance, reduces memory latency, and improves energy utilization.

Specifically, we compare the throughput performance and energy utilization of our deinterleaving method to two common deinterleaving methods. We assessed our method with 105 STREAM standard micro-benchmarks including 84 throughput and 21 energy performance test cases of varying input sizes and data types. In all cases tested, our method achieved better throughput and energy performance than the other two methods. In the best case, our method improved throughput up to 26.2x and throughput/watt up to 7.8x, when compared to the next best deinterleaving method.

2 Background

Simulations such as FLASH, S3D, and Nek5000 have variables that are interleaved in memory. These interleaved variables can be thought of as a matrix of data stored in row major format where each column corresponds to a particular variable. For multidimensional variables, each dimension has a separate column. Consider an example of FLASH simulation data with a sample of three variables ρ, P, and T corresponding to gas density, pressure, and temperature, respectively. The interleaved layout of these variables in memory can be seen in Figure 1a. Representing this data in matrix form would give an $m \times 3$ matrix where the three columns correspond to the three variables and the rows correspond to different steps of the simulation, see Figure 1c. With this interpretation, deinterleaving the data is equivalent to performing a matrix transposition, which would change the layout of the variables in memory, see Figure 1b.

There are two common techniques for deinterleaving data by performing an out-of-place matrix transposition. We refer to these techniques as *standard transposition* and *strided transposition.* These two techniques, along with our proposed method in the following section, are considered *out-of-place* due to the use of an output memory space equal to the size of the original matrix where the elements are copied. In contrast, *in-place* transposition methods use a bounded



ρ_0 P_0 T_0 ρ_1 P_1 T_1 ... ρ_m P_m T_m
ρ_0 ρ_1 ... ρ_m P_0 P_1 ... P_m T_0 T_1 ... T_m
ρ_0 P_0 T_0
ρ_1 P_1 T_1
ρ_m P_m T_m

(a) variables interleaved in memory

(b) variables deinterleaved in memory

(c) interleaved matrix format

Fig. 1. FLASH data in interleaved and deinterleaved layouts; each ρ_f, P_f, and T_f for $f = 0$ to m refers to the value of ρ, P, and T of the simulation at the f^{th} matrix row

amount of memory space and, in some cases, can slightly outperform out-of-place methods. However, in-place methods are often complex and can be performance constraining for simulations requiring variable interleaving, such as FLASH, S3D and Nek5000, to continue from where it left off in the calculation phase.

The standard and strided out-of-place transposition methods differ from each other in how they copy elements into an output memory buffer. The standard transposition method uses two loops to iterate row-wise and writes out the elements in a strided manner [12]. Alternatively, the strided transposition method uses two loops to iterate column-wise and writes out the elements contiguously.

3 Method

Our deinterleaving method performs an out-of-place transposition to transform a matrix of data stored in row major format to one stored in column major format. During the transposition process, our method combines the strength of both the standard transposition and strided transposition techniques.

In this section, we describe our deinterleaving method in detail. The method section is divided into three subsections corresponding to the three major components of our method: 1) cache prefetching on blocks of data, 2) using the registers as a vector transposition buffer, and 3) optimizing for full cache line writes. In addition, we provide a simple example for clarity.

3.1 Cache Prefetching on Blocks of Data

The benefit of cache prefetching is to hide latency time sinks associated with accessing main memory [13]. The standard transposition method, as discussed in Section 2, is able to take advantage of these benefits due to the sequential data reads inherent in its method. In contrast, the major weakness of the strided transposition method is that cache prefetching is not guaranteed and its effectiveness is dependent on the input buffer size. The cache prefetching benefits of the standard transposition method were the motivation for performing cache prefetching in our method.

Given an $m \times n$ (m rows and n columns) matrix of elements, A, stored in row major format, the first step of our deinterleaving method is to partition A into a block matrix where the blocks correspond to submatrices of A that

$$A = \begin{pmatrix} \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,n} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m_b,1} & e_{m_b,2} & \cdots & e_{m_b,n} \end{pmatrix} \\ \begin{pmatrix} e_{(m_b+1),1} & e_{(m_b+1),2} & \cdots & e_{(m_b+1),n} \\ e_{(m_b+2),1} & e_{(m_b+2),2} & \cdots & e_{(m_b+2),n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{2m_b,1} & e_{2m_b,2} & \cdots & e_{2m_b,n} \end{pmatrix} \\ \begin{matrix} \vdots & \vdots & \ddots & \vdots \end{matrix} \\ \begin{pmatrix} e_{(M-1)m_b+1,1} & e_{(M-1)m_b+1,2} & \cdots & e_{(M-1)m_b+1,n} \\ e_{(M-1)m_b+2,1} & e_{(M-1)m_b+2,2} & \cdots & e_{(M-1)m_b+2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{Mm_b,1} & e_{Mm_b,2} & \cdots & e_{Mm_b,n} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_M \end{pmatrix}$$

Fig. 2. Matrix A being partitioned into M blocks of size $m_b \times n$

will be consecutively prefetched into cache. As illustrated in Figure 2, matrix A is partitioned as an $M \times 1$ block matrix where each block is of size $m_b \times n$. Partitioning A in this manner creates M blocks each of which we label as B_k for $k = 1$ to M. The number of rows in each block, denoted m_b, is chosen so a block column can fill the entire cache line, discussed in Section 3.3.

For example, suppose the cache line is size of C bytes, which on most modern architectures is 64 or 128 bytes [14]. Suppose the elements of matrix A are each β bytes. Then, for m_b elements to fill the cache line as full as possible we want $m_b\beta = C$, and therefore make $m_b = \lfloor C/\beta \rfloor$. It is plausible that the last block will have fewer elements than the other blocks because m_b may not evenly divide the m elements. In this case, $M = \lceil m/m_b \rceil$. To process the smaller block, the matrix can be padded with values that will be disregarded [15].

The blocks, B_k for $k = 1$ to M, correspond to the submatrices of A that will be consecutively prefetched into the cache. Block of data B_{k+1} will be prefetched into cache while the block B_k is being further processed, as described in the following subsections. By prefetching blocks of elements in this manner, our method can reduce memory latency associated with loading blocks from memory.

3.2 Using the Registers as a Vector Transposition Buffer

Each block B_k can further be partitioned into submatrices using the columns as dividers, making B_k into a $1 \times n$ block matrix, referred to as a *column vector*, as seen in Figure 3a. With both partitions applied, matrix A can be viewed as a matrix of column vectors as shown in Figure 3b. Each column vector of B_k, which we denote as $V^c_{k,j}$ for $j = 1$ to n, consists of elements that are currently non-contiguous in memory due to the row major storage format of A.

The goal of our deinterleaving method is the elements of the column vectors to be contiguous in memory or, equivalently, the elements to belong to the same row in the matrix. To make the elements contiguous, each column vector gets transposed and temporarily stored in CPU registers until it is written out to a full cache line. The general notation for each transposed column vector, now referred to as a *row vector*, is denoted: $V^R_{k,j} = [e_{(k-1)m_b+1,j}, e_{(k-1)m_b+2,j}, \cdots, e_{km_b,j}]$.

$$B_k = \left(\begin{bmatrix} e_{(k-1)m_b+1,1} \\ e_{(k-1)m_b+2,1} \\ \vdots \\ e_{km_b,1} \end{bmatrix} \begin{bmatrix} e_{(k-1)m_b+1,2} \\ e_{(k-1)m_b+2,2} \\ \vdots \\ e_{km_b,2} \end{bmatrix} \begin{matrix} \cdots \\ \cdots \\ \ddots \\ \cdots \end{matrix} \begin{bmatrix} e_{(k-1)m_b+1,n} \\ e_{(k-1)m_b+2,n} \\ \vdots \\ e_{km_b,n} \end{bmatrix} \right) = \begin{pmatrix} V^C_{k,1} & V^C_{k,2} & \cdots & V^C_{k,n} \end{pmatrix}$$

(a) Partitioning of block B_k into column vectors $V^C_{k,j}$ for $j = 1$ to n

$$A = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_M \end{pmatrix} = \begin{pmatrix} V^C_{1,1} & V^C_{1,2} & \cdots & V^C_{1,n} \\ V^C_{2,1} & V^C_{2,2} & \cdots & V^C_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ V^C_{M,1} & V^C_{M,2} & \cdots & V^C_{M,n} \end{pmatrix}$$

(b) Matrix A partitioned into submatrices of column vectors

Fig. 3. Each block of matrix A partitioned into n column vectors

For clarity, consider a specific example. Suppose block B_1 is currently being partitioned into n column vectors, namely $V^c_{1,j}$ for $j = 1$ to n. The elements of a column vector $V^c_{1,j}$ consist of the elements $e_{1,j}, e_{2,j}, ..., e_{m_b,j}$ from A as seen in Figure 3a. Starting with the first column vector ($j = 1$), the elements must be loaded into a buffer of registers and in the next step written into the extra memory space that was created for the transposition matrix. Using CPU registers as a buffer to store these elements constitutes a transposition of the column vector as the elements will now be contiguous instead of strided.

The motivation for using the registers as a temporary buffer is that each column vector must be transposed into some storage location in order to achieve full cache line writes, which is the strength of the strided transposition method. The registers provide the most efficient location to store the row vectors due to their minimal CPU cycles per operation [16]. In addition, using a buffer of registers in this manner is a viable option since typically a CPU provides enough hardware registers where the buffer size is at least equal to the cache line size.

3.3 Optimizing for Full Cache Line Writes

Once the elements of a row vector are loaded into the register buffer, our method then writes out this data into the memory space that was created for the deinterleaved output. During the write process, our method utilizes the full cache line due to the row vector containing m_b elements, where m_b was chosen to fill the cache line. By utilizing full cache line writes, our method emulates the strength of the strided transposition method [17], while avoiding the inefficient write process of the standard transposition method.

During the write process, our method must leave enough room for m elements of A (an entire column) between the start of each column vector, meaning there will be a stride of m between the memory storage offset of each column vector. So, for a given row vector $V^R_{k,j}$, the elements get mapped consecutively into the new memory storage location offset starting at $(k-1)m_b + (j-1)m$.

After this process is completed and all the row vectors have been written, the process is repeated. The next block, which should already reside in cache, is partitioned into column vectors that are consecutively loaded into the register buffer and written out. The entire process is completed for each block B_k for $k = 1$ to M. Once every block has gone through this process, the output location will

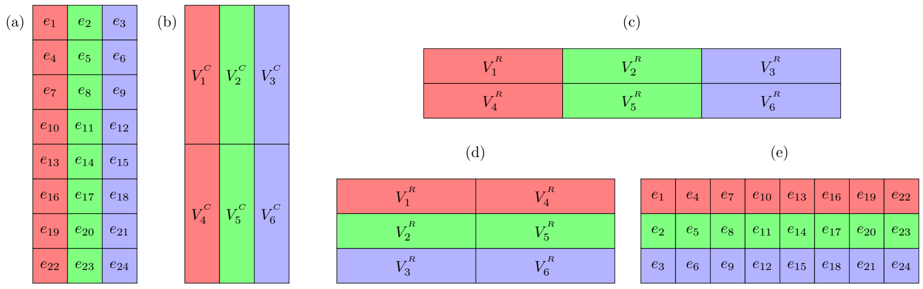

(a)
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11} e_{12} e_{13} e_{14} e_{15} e_{16} e_{17} e_{18} e_{19} e_{20} e_{21} e_{22} e_{23} e_{24}
(b)
V_1^C V_2^C V_3^C V_4^C V_5^C V_6^C
(c)
V_1^R V_2^R V_3^R V_4^R V_5^R V_6^R
(d)
V_1^R V_4^R V_2^R V_5^R V_3^R V_6^R
(e)
e_1 e_4 e_7 e_{10} e_{13} e_{16} e_{19} e_{22}
e_2 e_5 e_8 e_{11} e_{14} e_{17} e_{20} e_{23}
e_3 e_6 e_9 e_{12} e_{15} e_{18} e_{21} e_{24}

Fig. 4. The partition and transposition steps of our deinterleaving method performed on a simple 8×3 matrix of 8-byte elements optimized for cache line writes of 32 bytes

contain the transpose of matrix A. The entire deinterleaving process is illustrated by the example given in the following section.

3.4 A Simple Example of Our Deinterleaving Method

For clarity, consider a simple example of 24 data elements consisting of three different variables interleaved in memory. Figure 4a shows the matrix representation of these interleaved variables, with each column of the matrix storing data corresponding to a particular variable. For the sake of this example, suppose the elements are 8-byte doubles (common in simulation data) and the cache line size of the system is 32 bytes. The elements of the matrix are initially stored in row major format, meaning the elements are ordered as $e_1, e_2, e_3, e_4, ..., e_{24}$ in memory. The goal of our deinterleaving method is to obtain the transpose of the matrix, illustrated in Figure 4e, so that the elements of each column will be contiguous in memory and thus deinterleaved.

The initial step of our deinterleaving method is to create a new output memory space to hold the transposed matrix. Next, the matrix is partitioned into two 4×3 block matrices, B_1 and B_2 consisting of elements e_1 through e_{12} and e_{13} through e_{24}, respectively. The number of rows in each block was chosen as $m_b = 4$ so that each column within a block will entirely fill the cache line, as four 8-byte doubles is exactly the cache line size of the system.

With the matrix partitioned into two blocks, the next step is to load B_1 into the cache. The block itself is then partitioned into the three column vectors V_1^C, V_2^C, and V_3^C, as depicted in Figure 4b. After this partition, the first column vector of B_1, meaning the elements e_1, e_4, e_7, and e_{10}, is transposed into a row vector and temporarily stored in the register buffer, see Figure 4c. The full cache line is then utilized to write out the elements of the row vector into the output memory space that was created for the transposed matrix, see Figure 4d. This process is repeated on the remaining column vectors of B_1 until all of them have been written into the output memory space.

After B_1 has finished transposing and writing each of its column vectors, the same process is repeated on the second block, B_2. This block would have been prefetched into cache during the time B_1 was being processed, thus saving the time of retrieving B_2 from memory. After B_2 is processed, the matrix will be transposed and the variables deinterleaved, as illustrated in Figure 4e.

4 Performance Evaluation

In this section, we present the empirical evaluations of our deinterleaving method via a set of micro-benchmarks to evaluate throughput and energy performance. We compare the results of our deinterleaving method against those of the standard and strided transposition methods. For brevity, we will refer to our Out-of-Place Deinterleaving method as *OPD method* in the remainder of the paper.

4.1 Experimental Setup

Performance measurements were collected on the Lens Linux cluster at Oak Ridge National Laboratory and on a dedicated Intel server. The Lens cluster is primarily used for data analysis and high-end visualization. Each cluster node consists of four quad-core 2.3 GHz AMD Opteron processors and 128GB of memory. Each processor has three cache levels: L1 cache is 64KB, L2 cache is 512KB, and the shared last level cache (LLC) is 5118KB. The Intel server consists of a quad-core i7 2.93 GHz processor and 16GB of memory running CentOS-6.3. The Intel processor has three cache levels: L1 is 32KB, L2 is 256KB, and LLC is 8MB. All multi-core evaluations for both the throughput and energy experiments were done utilizing all available processors and computational cores.

For collecting performance metrics, we added micro-benchmarks of all deinterleaving methods into the STREAM [18] framework, compiled with GNU Compiler Collection (GCC) version 4.7.1. STREAM is useful for evaluating memory throughput performance of single- and multi-core I/O-intensive functions that are sensitive to system architecture characteristics [19]. We compared the throughput performance metrics collected from 105 STREAM micro-benchmarks tested across the AMD and Intel systems. The test cases spanned a diverse set of data including multiple data types, column sizes, and input buffer sizes. Specifically, the data types evaluated were bytes, single-precision floating-points, and double-precision floating-points. For each data type, the variables interleaved (columns) were $2, 4, 8$, and 16. The input buffer sizes ranged from $64, 128, \cdots, 4096$ kilobytes per core. To obtain the performance measurements seen in Figure 5 and Figure 6, each micro-benchmark was run 100 times for each deinterleaving method. The highest throughput of the 100 runs was recorded.

For our set of micro-benchmarks, we restricted our input buffer size between 64KB and 4096KB. The reason this lower bound was chosen is due to the precision of the timer used in the STREAM benchmark, which states at what point the clock measurement becomes unreliable. For input sizes less than 64KB, our deinterleaving technique ran too fast for a reliable throughput measurement. However, at sizes of 64KB and higher, the throughput could be measured accurately. The upper bound of 4096KB was chosen to represent an input size that was beyond the size of the LLC for multi-core evaluations.

4.2 Deinterleaving Throughput Performance

In all multi-core evaluations, our deinterleaving method performed better than the standard and strided transposition methods, see Figure 5 and Figure 6. In the best case, our deinterleaving method performed at a 26.2x faster throughput, when compared to the next best method. In addition, our method consistently reported gains of over 40GB/s on smaller input sizes (corresponding to lower cache levels). The performance gains of our deinterleaving method were more pronounced on smaller input buffer sizes because memory latency starts to become a significant factor on larger buffer sizes.

Another characteristic seen in our results is that neither the standard transposition nor the strided transposition was consistently better than the other. In some cases, the standard transposition would significantly outperform the strided transposition and vice versa, irrespective of the instruction set architecture being used, see Table 1. The performance inconsistency of these two techniques is another strength of our deinterleaving method, as ours consistently outperformed the other two methods.

Although not depicted in throughput performance figures, our method was also compared against the other methods when all were utilizing only a single core of the system. In this case, our method reported similar, but scaled down trends to those seen in multi-core evaluations. Even in this case, our method always had better throughput performance than the other two methods.

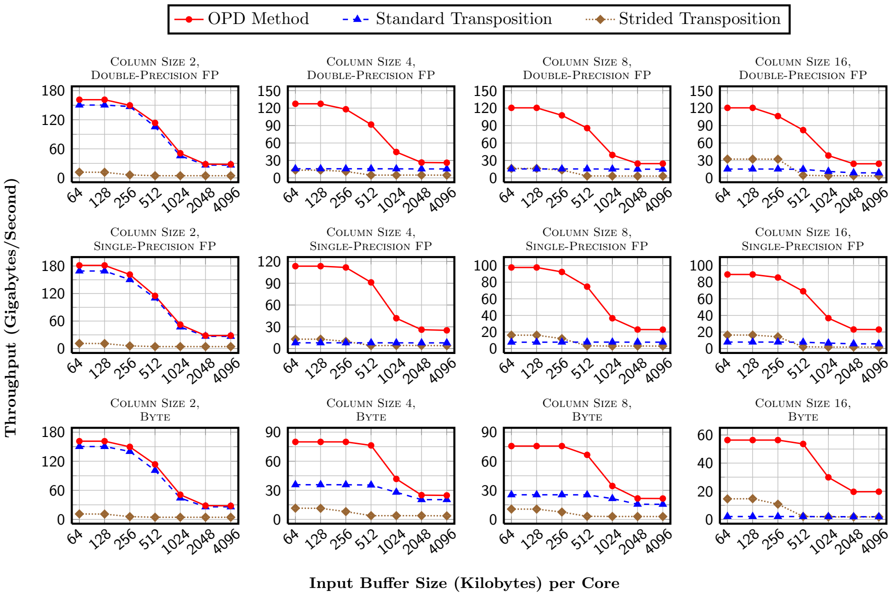


Fig. 5. Throughput performance applying STREAM micro-benchmarks when deinterleaving single-precision, double-precision floating-point (FP), and byte variables on the AMD Opteron system utilizing all 16 cores

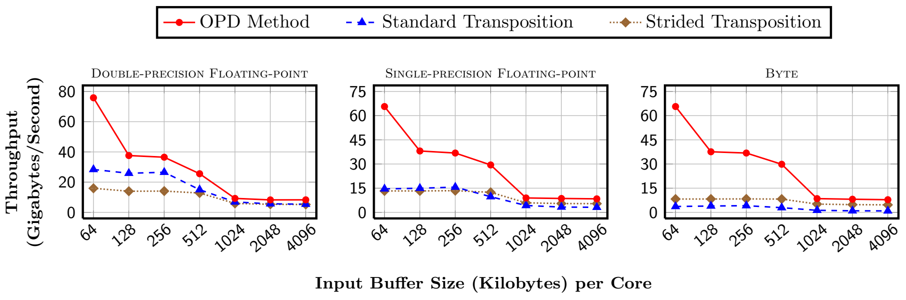


Fig. 6. Throughput performance applying STREAM micro-benchmarks when deinterleaving single-precision, double-precision floating-point, and byte variables with 16-variable interleaved data on the Intel i7 system utilizing all cores

Table 1. Instruction set architecture for deinterleaving methods

Data Type	Method	Column Size			
		2	4	8	16
Double	Standard	SSE2	x86_64	x86_64	x86_64
	Strided	x86_64	x86_64	x86_64	x86_64
Float	Standard	SSE	SSE	SSE	SSE
	Strided	SSE	x86_64	x86_64	x86_64
Byte	Standard	SSE2	SSE2	SSE2	SSE2
	Strided	x86_64	x86_64	x86_64	x86_64

4.3 Deinterleaving Energy Performance

The energy performance measurements were performed on a dedicated Intel server connected to a Watts Up Pro meter, which provides a recording of power measurements (watts) per second during the collection of throughput metrics. The power was measured for each deinterleaving method on 21 micro-benchmarks of 16-variable interleaved data of varying input sizes and data types. Energy performance normalization was done for the deinterleaving methods by calculating gigabytes per joule (throughput/watt) for each test case.

In all cases tested, our deinterleaving had better energy utilization than the other methods, with throughput/watt improvements up to 7.8x, when compared to the next best method. The results of our energy experiments can be seen in Figure 7. The improved energy performance of our method is attributed to the increased throughput (Figure 6), the effective cache utilization similar to the standard transposition method, and the optimized cache line writes like the strided transposition method.

5 Related Work

Out-of-place matrix transpositions have been studied extensively in the past. Majority of these transposition algorithms, initially proposed decades ago, focus on methodologies for optimizing use of secondary storage (tapes, disks, etc.). Although these algorithms are not well suited for modern computer systems due

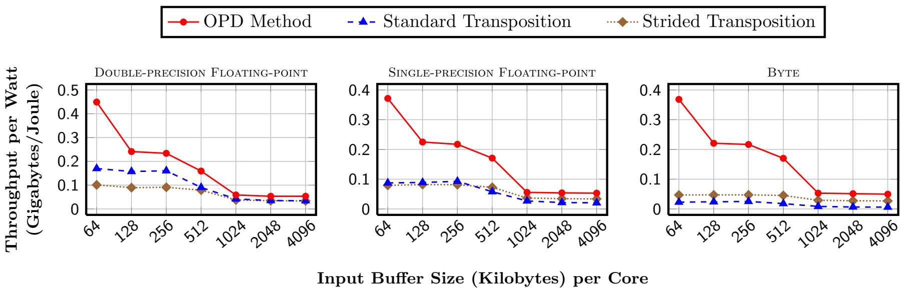


Fig. 7. Normalized energy performance measurements (throughput/watt) collected with power meter during STREAM throughput benchmarks on Intel system (Figure 6)

to processor cache inefficiency, we still use these techniques for references since secondary storage of the past is analogous to RAM in modern systems. A fast matrix transposing method was given in [20] where the algorithm was specifically designed for $2^n \times 2^n$ square matrices and it is compared with many other matrix transposition algorithms. Another algorithm called single radix algorithm was proposed in [21], and shows better performance in disk seeks and accesses. For transposing a large arbitrary matrix, PRIM was introduced in [22].

In-place matrix transpositions can be used as an alternative to out-of-place methods; however, in-place methods are often complex and can be performance inefficient for simulations requiring interleaved variables to continue with the calculation phase. Furthermore, in-place methods commonly have constraints on row and column sizes making them unusable as a generic method for deinterleaving scientific data. Six algorithms are investigated in [23] for transposing a large square matrix in-place. They use 32-bit single-precision floating-point numbers and have the length of both the row and column equal to 2^n. In their experiments, the non-linear array layout algorithm outperforms other algorithms as it uses "Morton ordering" [24]. This algorithm also uses recursion to divide the problem into smaller subproblems, as in [12], but terminates at an architecture-specific tile size. Even by using a "blocking" and "tiling" technique, a higher cache efficiency might not be achieved as claimed in [16]; instead, they proposed a buffer must be used in order to be cache efficient.

Although much attention has been paid to matrix transposition, very few of the studies focus on the utilization of cache in a specific domain requiring deinterleaving of variables. Our method applies to any data type and utilizes full cache line writes to be throughput and energy efficient when deinterleaving data. Blocking, shuffling, and compression library, Blosc, was introduced in [25], which uses a high-performance byte deinterleaving technique to reduce activity on the memory bus. Our approach differs from this technique in that we support not just byte-level but float- and double-level as well. Moreover, Blosc currently utilizes 16-byte SSE2 register writes instead of full cache line writes compared to our deinterleaving method.

6 Conclusion

We proposed a deinterleaving method that is high performance, energy efficient, and generic to any data type. Our method has increased throughput and energy performance by utilizing the system architecture in three ways: 1) improving data cache prefetching, 2) reducing memory accesses, and 3) optimizing the use of full cache line writes.

Our method results in better throughput and energy performance when compared against two common deinterleaving methods during 105 STREAM standard micro-benchmarks evaluations, which includes 84 throughput and 21 energy performance test cases. When compared to the next best case, our method improved throughput up to 26.2x and throughput/watt up to 7.8x.

Acknowledgements. We like to thank Argonne National Laboratory (ANL) and Oak Ridge National Laboratory (ORNL) for use of resources at their Leadership Computing Facilities, ALCF and OLCF respectively. Also, thanks goes to Flash Center for Computational Science at the University of Chicago for providing scientific datasets. This research and ALCF resources at ANL have been funded and supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. ORNL is managed by UT-Battelle for the LLC U.S. D.O.E. under Contract DE-AC05-00OR22725. Also, partial support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs).

References

1. Ma, K.: In situ visualization at extreme scale: Challenges and opportunities. IEEE Computer Graphics and Applications 29(6), 14–19 (2009)
2. Laurenzano, M.A., Meswani, M., Carrington, L., Snavely, A., Tikir, M.M., Poole, S.: Reducing energy usage with memory and computation-aware dynamic frequency scaling. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 79–90. Springer, Heidelberg (2011)
3. Stevens, R., White, A., Dosanjh, S., Geist, A., Gorda, B., Yelick, K., Morrison, J., Simon, H., Shalf, J., Nichols, J., Seager, M.: Scientific grand challenges: Architectures and technologies for extreme scale computing. Tech. Rep., DOE (2009)
4. Ge, R., Feng, X., Sun, X.: SERA-IO: Integrating energy consciousness into parallel I/O middleware. In: Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 204–211 (2012)
5. Latham, R., Daley, C., Liao, W., Gao, K., Ross, R., Dubey, A., Choudhary, A.: A case study for scientific I/O: Improving the FLASH astrophysics code. Computational Science & Discovery 5(1), 015001 (2012)
6. Fryxell, B., Olson, K., Ricker, P., Timmes, F., Zingale, M., Lamb, D., MacNeice, P., Rosner, R., Truran, J., Tufo, H.: FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. The Astrophysical Journal Supplement Series 131(1), 273 (2008)

7. Chen, J., Choudhary, A., De Supinski, B., DeVries, M., Hawkes, E., Klasky, S., Liao, W., Ma, K., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., Yooothers, C.S.: Terascale direct numerical simulations of turbulent combustion using S3D. Computational Science & Discovery 2(1), 015001 (2009)
8. Fischer, P.F., Lottes, J.W., Kerkemeier, S.G.: (2008), `http://nek5000.mcs.anl.gov/`
9. Schendel, E.R., Jin, Y., Shah, N., Chen, J., Chang, C., Ku, S.H., Ethier, S., Klasky, S., Latham, R., Ross, R., Samatova, N.F.: ISOBAR preconditioner for effective and high-throughput lossless data compression. In: Proceedings of the 28th International Conference on Data Engineering (ICDE), pp. 138–149. IEEE (2012)
10. Schendel, E.R., Pendse, S., Jenkins, J., Boyuka II, D.A., Gong, Z., Lakshminarasimhan, S., Liu, Q., Kolla, H., Chen, J., Klasky, S., Ross, R., Samatova, N.F.: ISOBAR hybrid compression-I/O interleaving for large-scale parallel I/O optimization. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC), pp. 61–72. ACM (2012)
11. Jenkins, J., Schendel, E.R., Lakshminarasimhan, S., Boyuka II, D.A., Rogers, T., Ethier, S., Ross, R., Klasky, S., Samatova, N.F.: Byte-precision level of detail processing for variable precision analysis. In: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis, p. 48 (2012)
12. Frigo, M., Leiserson, C., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Symposium on Foundations of Computer Science, pp. 285–297. IEEE (1999)
13. Gamoudi, O., Drach, N., Heydemann, K.: Using runtime activity to dynamically filter out inefficient data prefetches. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 338–350. Springer, Heidelberg (2011)
14. Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., Hughes, B.: Cache hierarchy and memory subsystem of the AMD Opteron processor. IEEE Micro 30(2), 16–29 (2010)
15. Dow, M.: Transposing a matrix on a vector computer. Parallel Computing 21(12), 1997–2005 (1995)
16. Gatlin, K., Carter, L.: Memory hierarchy considerations for fast transpose and bit-reversals. In: Proceedings of Fifth International Symposium High-Performance on Computer Architecture, pp. 33–42. IEEE (1999)
17. Drepper, U.: What every programmer should know about memory. Tech. Rep., Red Hat, Inc. (2007)
18. McCalpin, J.D.: STREAM: Sustainable memory bandwidth in high performance computers (2000), `http://www.cs.virginia.edu/stream/`
19. Gschwandtner, P., Fahringer, T., Prodan, R.: Performance analysis and benchmarking of the Intel SCC. In: Proceedings of International Conference on Cluster Computing, pp. 139–149. IEEE (2011)
20. Eklundh, J.: Efficient matrix transposition. In: Two-Dimensional Digital Signal Prcessing II, pp. 9–35 (1981)
21. Kaushik, S., Huang, C., Johnson, J., Johnson, R., Sadayappan, P.: Efficient transposition algorithms for large matrices. In: Proceedings of Supercomputing 1993, pp. 656–665. IEEE (1993)
22. Goldbogen, G.: PRIM: A fast matrix transpose method. IEEE Transactions on Software Engineering (2), 255–257 (1981)
23. Chatterjee, S., Sen, S.: Cache-efficient matrix transposition. In: Sixth International Symposium on High-Performance Computer Architecture, pp. 195–205. IEEE (2000)
24. Morton, G.M.: A Computer Oriented Geodetic Database and a New Technique in File Sequencing. IBM, Ltd. (1966)
25. Alted, F.: Why modern CPUs are starving and what can be done about it. Computing in Science & Engineering 12(2), 68–71 (2010)

Transparent Support for Partial Rollback in Software Transactional Memories

Alice Porfirio, Alessandro Pellegrini,
Pierangelo Di Sanzo, and Francesco Quaglia

DIAG, Sapienza, University of Rome

Abstract. The Software Transactional Memory (STM) paradigm has gained momentum thanks to its ability to provide synchronization transparency in concurrent applications. With this paradigm, accesses to data structures that are shared among multiple threads are carried out within transactions, which are properly handled by the STM layer with no intervention by the application code. In this article we propose an enhancement of typical STM architectures which allows supporting partial rollback of active transactions, as opposed to the typical case where a rollback of a transaction entails squashing all the already-performed work. Our partial rollback scheme is still transparent to the application programmer and has been implemented for x86-64 architectures and for the ELF format, thus being largely usable on POSIX-compliant systems hosted on top of off-the-shelf architectures. We integrated it within the TinySTM open-source library and we present experimental results for the STAMP STM benchmark run on top of a 32-core HP ProLiant server.

1 Introduction

Software Transactional Memory (STM) [1] stands as a programming paradigm tailored for the development of concurrent applications. By leveraging on atomic transactions, STM relieves the programmers from the burden of explicitly writing complex, error-prone thread synchronization code. In fact, programmers are only requested to wrap critical-section code within transactions. In STM, data conflicts are handled by means of conflict detection and management (CDMAN) algorithms, such as the ones presented in [2–6]. However, most of the proposed schemes rely on implementations where the rollback of a transaction entails squashing all the work carried out during its execution, despite the fact that part of the work can be still valid. To cope with this issue, in this article we present the design and implementation of a partial rollback scheme that is able to avoid rolling back an entire transaction, thus allowing portions of the carried-out transactional work to be saved. This directly provides a reduction on the number of machine instructions required for finalizing a given transaction instance. Such a reduction is achieved via minimal housekeeping overhead, in terms of machine instructions required for supporting the partial rollback scheme, since we exploit optimized approaches for the management of partial-log operations (e.g. of automatic variables) during the execution of transactional code blocks.

Further, with our partial rollback scheme, mutual consistency of shared data and thread-private data is automatically and transparently guaranteed. This removes the need for explicitly identifying and annotating the private data (e.g. the local variables) that need to be rollbackable, as instead is requested when relying on typical facilities offered by STM implementations [2, 3].

We achieve complete transparency towards the application-level code via an instrumentation tool, which adds partial log/undo capabilities without requiring any intervention by the programmer. We integrated and tested our proposal within the TinySTM open-source library [3], exploiting the Commit-Time-Locking (CTL) algorithm natively supported by TinySTM. Further, the instrumentation tool has been tailored to the Executable and Linkable Format (ELF) and to x86-64 processors, thus allowing supporting partial rollback operations for TinySTM-based applications run on top of POSIX compliant systems and widely diffused hardware architectures. Some experimental data highlighting performance advantages from our proposal, when compared to the traditional case of complete squashing for rolled back transactions, are also reported for the case of the STAMP STM benchmark [7] run on top of a 32-core HP ProLiant server equipped with 32GB of RAM memory and running Linux.

The remainder of this paper is structured as follows. In Section 2, related work is discussed. Details on our algorithmic extensions of CTL aimed at partial rollback are provided in Section 3. The actual implementation of the support for partial rollback within TinySTM is presented in Section 4. Performance data are provided in Section 5.

2 Related Work

Solutions aimed at the reduction of the waste of CPU-time associated with rolled back STM transactions can be found in [8–10]. The main approaches underlying these proposals entail (i) (dynamically) regulating the amount of concurrent threads to a well suited value (see, e.g., [9]), which is expected to avoid thrashing due to transaction aborts caused by excessive data conflicts, and/or (ii) a-priori sequentializing transactions when the abort rate exceeds specific thresholds (see, e.g., [10]). All these proposals are orthogonal to our one since none of them is tailored to the reduction of the waste of work via partial save of the effects of the execution of transactions.

Considering partial rollback in STM systems as the specific target, a few solutions have been proposed in [11–13]. Differently from what we present in this article, the proposal in [11] is limited to the management of partial rollback operations on shared data, thus not supporting rollback of thread-private data. As a consequence, mutual consistency between shared and private data within the partial rollback scheme is demanded from the programmer, while our approach enforces full transparency. The proposals in [12, 13] consist of an architectural specification of partial rollback supports, which has however not been implemented in any real environment, and has been evaluated only via simulation. Instead, we provide a real implementation within the TinySTM framework. Additionally, the work in [12, 13] bases partial rollback on a traditional approach where shared and private objects are marked as updated via dirty-bitmaps and are logged into per-object undo stacks. Instead, our proposal does not rely on any explicit management of dirty-bitmaps, and packs log information by clustering thread-private data via optimized log operations of the thread stack, which are based on ranges of memory addresses defining target regions for the memory write instructions executed along the transaction.

3 Partial Rollback

3.1 Target CDMAN: Commit-Time-Locking Plus Read Validation

We target the CTL algorithm, as used in implementations such as TL2 [2] and TinySTM [3]. The algorithm relies on a *global version clock* (*gvc*), namely a global shared counter, which is atomically incremented (e.g. via Compare-And-Swap—CAS—operations) by any thread whenever it commits a transaction that updated shared data. Also, each (size-tunable) set of shared memory objects, such as memory words for the case of word-based STM, is associated with its own meta-data consisting of (A) a lock-bit and (B) a timestamp. This association is supported by means of hash functions taking as input the shared-object memory address. When a transaction commits, the updated *gvc* value is reflected as the new timestamp of the written objects.

Upon (re-)starting a transaction, a thread stores the current value of the *gvc* into a local variable called transaction start-timestamp (*tst*). Upon a transactional read operation from a shared object, the corresponding memory address is added to the transaction read-set, while, upon a write operation, the destination-object address and the value to be stored are both added to a transaction write-set (note that the written value is not yet stored into the actual target location). In addition, when executing a transactional read operation, it is checked in advance if the shared object has already been written by the transaction (by checking the content of the transaction write-set). In the positive case, the value stored within the write-set is returned. Otherwise, the lock-bit associated with the object is sampled to check whether it is set to 1, which means that the object is currently locked by a concurrent transaction. If the lock-bit has value 1, the reading transaction gets aborted (possibly after waiting for the lock release for a while). Otherwise, the object value and its timestamp are re-sampled along with the lock-bit in order to check if (A) the timestamp is less than or equal to the *tst* of the reading transaction, and (B) the object is not currently re-locked. If both the checks succeed, it means that no concurrent transaction has modified the object in the interval between the start of the current transaction and the actual read operation, hence the value read is still *valid*. Otherwise, the transaction gets aborted and then restarted.

Upon attempting to commit a writing transaction[1], the thread tries to acquire the locks associated with all the objects belonging to the transaction write-set.

[1] For read-only transactions the commit operation is unnecessary as no shared objects have to be updated.

This is done by attempting to set the lock-bit associated with each of these objects (e.g via CAS operations). If at least one lock acquisition fails, the transaction is aborted and restarted. Otherwise, the transaction read-set gets validated. Namely, for each object belonging to the read-set, the associated current timestamp is compared with the *tst* value in order to check if it was modified after starting the transaction. Object modifications by concurrent transactions imply that the object timestamp is greater than *tst* since it reflects updates in the *gvc*, generated by successfully committing transactions. Hence, if the timestamp of at least one object has been modified, then the transaction is aborted and restarted. Otherwise the transaction can successfully commit, thus storing object-values kept within the transaction write-set in the destination memory areas and releasing all the acquired locks.

A mechanism used in combination with this scheme is called *snapshot extension*. When the thread performs a transactional read from an object that has been updated by a concurrent transaction (which would lead to an abort) this mechanism checks if all the object values returned by previously executed transactional read operations of the transaction (if any) are still valid. If yes, the snapshot seen by the transaction is still consistent, hence the transaction is not aborted. In this case, the *tst* is updated to the value of the *gvc* sampled immediately before performing the check.

3.2 The Partial Rollback Scheme

In our partial rollback scheme, we rely on snapshot extension as a basis for managing the *tst*. However, we devise an approach where snapshot extension is exploited according to a sequential validation scheme, which is used to determine the maximum amount of transactional work that can be considered as still valid on the basis of the current state of shared data. Specifically, upon the read of an invalid object-value, the previously executed read operations are revalidated in order of their occurrence within the transaction, until all of them are found to be still valid, or validation fails for one of them. The first invalid read operation along the sequence is the restoration point for our partial rollback scheme, hence all the subsequent work performed by the transaction (if any) is squashed.

Coherency between read and write sets within the partial rollback scheme has been achieved by determining causality relations (i.e. temporal ordering) among transactional read/write operations within each transaction, which are logged as part of the representation of read/write sets. Hence, all the transactional write operations that are detected as being causally dependent on invalid read operations are also squashed from the write set.

As hinted before, we have also devised a scheme for partially rolling back thread-private data, in a consistent manner with respect to squashing operations of read and write sets. This is based on identifying the memory upper/lower bound for any log/restore operation within the stack, to be used to correctly manage thread-private data within the partial rollback scheme. On the other hand, both upper and lower bounds for these operations change over time depending on how flow-control and memory updates are materialized across different routines while the transactional code block's execution is still in progress.

Overall, beyond the already depicted handling of read/write sets (reflecting the access to shared data), operated while managing incremental snapshot extensions, our partial rollback scheme deals with the management of thread-private data according to the following actions: (A1) upon invoking `TM_begin` along the thread, the stack pointer is used to determine the upper and lower bounds of the stack region (initially empty) to be logged in case updates of thread-private data occur along the transaction. This region may be enlarged (by moving its bounds) when a write operation occurs within the stack along the transaction's execution. If the write touches data above (resp. below) the upper (resp. lower) bound, such a bound is moved to the top (resp. bottom) address of the touched memory area; (A2) upon invocation of any `TM_read` operation along the thread, a recovery image for the whole memory segment in between the current upper and lower bounds for the target stack region is created, together with a recovery image for the processor context; (A3) upon an incremental-snapshot extension operation (as depicted above), the stack/processor recovery image associated with the first no more valid `TM_read` along the sequence (as determined in A2) is restored, using an incremental-restore technique similar to the one proposed in [14]; (A4) upon successful invocation of `TM_end` along the thread, which determines actual commitment, the recovery images associated with the transaction execution path (as determined in A2) are discarded.

The above scheme, in particular in point A3, provides facilities for consistently rolling back thread-private data even in cases of complete squashing of the performed transactional work (e.g. due to invalidation of the object accessed upon the first read operation along the transactional code block). This is a relevant facility along the line of simplifying the development of application code.

Two additional points devise discussion. First, generation of stack frames restoration images is subject to a set of optimizations which we will depict later on in Section 4. Second, with the devised approach, we make update operations occurring within the stack rollbackable even if they occur via pointer-based access. Specifically, whenever any routine is started-up within the thread execution flow, if any pointer is received in input which allows the access to stack memory locations (namely stack frames) associated with other functions living along the thread, then any write access is automatically handled via the recovery images depicted above.

The only case not covered, in terms of ability to rollback, is related to updates occurring within global data that are inherently outside the control of the STM layer (e.g. non-transactional global variables). However, with the STM approach, these are typically avoided since the target is synchronization-transparent management of global (inherently shared) data structures.

4 Implementation

The logic required for handling partial rollback operations within TinySTM entails: (A) Identifying the execution points where recovery images for the thread stack need to be taken; (B) Implementing the actual log/restore logic for the stack, and combining it with the management of read/write sets. Some parts of this logic have been implemented via proper modification of TinySTM internals, while an instrumentation tool [15] has been used in order to provide complete transparency to the application layer in terms of exploitation of partial roll-back capabilities. Within the whole code modification/instrumentation process, both the above targets have been achieved via instrumentation rules that allow nesting within the code a block of machine instructions to be executed right before any call from the application software to the `TM_read` function offered by the TinySTM API. This allows us to transparently take control exactly when we need to create a stack recovery image, namely before actually accessing the target transactional object in read mode. If the read operation is invalid, the additional logic included within the TinySTM layer is used in order to exploit stack restoration images for supporting partial rollback.

In order to correctly create a recovery image, which includes the current processor context right before the call to `TM_read`, the instrumentation tool has been used to transparently inline within the application ELF a functional block structured as:

```
boundaries = recompute_boundaries();
getcontext(&cpu_state);
stm_store_context_in_readset(&cpu_state);
```

With this approach, stack/processor information associated with the current state of execution of the function calling `TM_read` is sampled, with no modification of stack pointer/content and CPU image performed by the code block. On the other hand, the creation of the stack recovery image is performed by `stm_store_context_in_readset`, a function we have added to TinySTM, which performs an optimized management of stack log operations as we will explain.

Creation of Stack Recovery Images. Our approach to the creation of a recovery image for the stack of the transactional thread at a given point in the execution is based on two optimizations. The first one deals with an incremental approach for the determination of the stack portions that actually need to be logged in order to correctly achieve a recovery image for the whole stack content. The second one deals with how to perform the actual copy of the memory areas required for creation of the restoration image. The two optimizations are explicitly thought to be used in combination.

As for the first optimization, a snapshot of the stack content at a given point of execution is built by combining the latter available stack log plus a log of only those portions that have been modified up to the point of interest. This is an incremental approach, that has already been exploited in literature, but typically according to a page-based logging approach (see, e.g., [16]). Instead, we incrementally build the recovery image of the stack by logging data according to arbitrary granularity, and by organizing them in a chain (realized in the read set) as described in [14]. Also, the idea underlying our incremental scheme is to determine the stack portions to be logged in such a way that they are finally contained within a single area formed by contiguous memory locations. This allows nesting the aforementioned second optimization related to efficient support for memory copies in case of adjacency of the addresses characterizing source and destination areas for the copy operation.

To achieve incremental logging, the application level ELF is again transparently instrumented by allowing the insertion of a code block before any machine-level memory-write operation (e.g. `mov` instructions) which, by analysing the current state of the processor, determines the actual virtual address to be targeted by the update, and the size of the touched memory area. In case the update falls within the stack (namely the lower address of the area to be touched is not less than the current stack pointer value), the write operation deals with the content of the stack, and needs to be made rollbackable. In this case, the interval of virtual addresses $[I_1, I_2]$ involved in the update is determined, which in turn identifies a stack portion to be logged upon the creation of the subsequent recovery image (as the one containing all the memory locations in between the addresses I_1 and I_2). We note that for some machine-level memory instructions, write access to the stack occurs by default, such as for the case of `push` and `call`.

In fact, multiple write operations can occur before the point where the creation of the stack recovery image occurs. As an example, an additional write may involve the stack portion in the interval of addresses $[I_3, I_4]$. In such a case, instead of explicitly maintaining the list of stack portions to be logged and restored, we adopt a clustering approach where we identify the actual area to be logged as the one between a minimum address value computed as $I^- = \min(I_1, I_3)$ and a maximum address value computed as $I^+ = \max(I_2, I_4)$. In other words, we always log a contiguous segment of the stack, which contains all the modified stack locations and possibly some non-modified locations. This is done so that the actual log (namely memory copy) operation can be achieved by using a single machine instruction, such as the `movs` instruction of the x86 instruction set. This is the second optimization.

As an additional note, this approach is combined with a check on the actual top-boundary of the stack such that, when considering an incrementally built recovery image associated with stack-pointer value x, any memory location within the stack with address $y < x$, possibly belonging to the previous recovery image, is logically marked as non-relevant for the incremental construction of the current image. Further, we emphasize that our approach, based on a boundary check on the actual modified region of the stack, up to the transactional read operation, copes well with common optimizations offered by modern compiling toolchains. In particular, standard compilers might decide to use, where available, the stack base registers (e.g. `ebx` on x86 architectures) as general purpose ones. This speeds up the program's execution by enlarging the set of information which the processing unit is able to maintain within its internal state. On the other hand, this makes it impossible to determine which is the current function's stack frame. Our solution is able to cope with this scenario, since we do not need to explicitly know the base of the stack zone for any specific function.

Stack Recovery Operations. As mentioned, upon the detection of an inconsistent read, instead of relying on the classical rollback scheme, we perform a

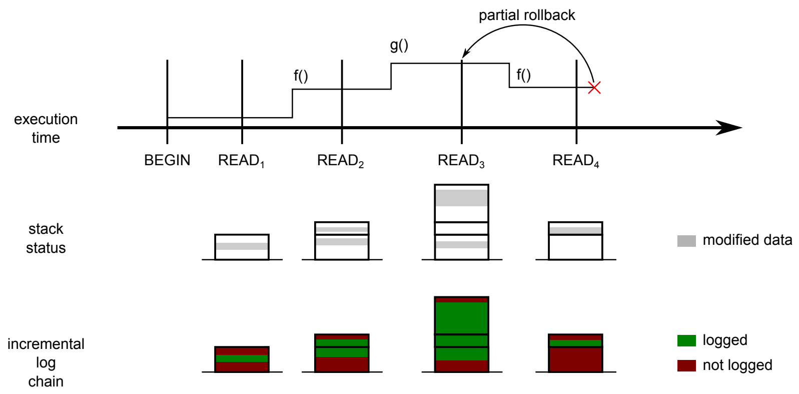
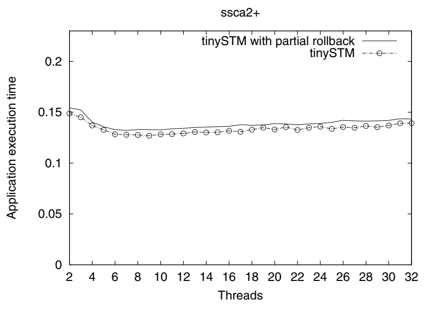


Fig. 1. Stack management within partial rollback

partial rollback operation, which entails restarting the execution of the conflicting transaction from an intermediate point such that every operation before it is still considered valid. In order to effectively restart from within a transaction, we must restore every aspect of the execution context. If on the one hand the processor state is restored via the standard System V `setcontext` library function—using a previously stored snapshot—in order to successfully cope with automatic variables we must undo any modification concerning the stack frames of the functions living along the thread execution. In Figure 1 we show how we build partial stack logs during the execution of the transaction. In the above example, upon the execution of $READ_4$, an inconsistency is discovered, and given the failure of the snapshot extension protocol we trigger our partial rollback operation. In the example, $READ_2$ is selected as being the most recent read operation entailing a still-valid value, therefore the execution is restarted from $READ_3$. In particular, in $READ_3$'s read set we can find the portion of the stack which was modified between the execution of $READ_2$ and $READ_3$. This is restored together with the aforementioned processor context, and together with other incremental portions of the stack from previous logs, as described in [14].

5 Experimental Results

We present some experimental results achieved with the STAMP STM benchmark suite [7], specifically with ssca2 and kmeans applications. The former is a transactional implementation of the Scalable Synthetic Compact Applications 2 (SSCA2) benchmark [17], where a graph kernel is used to build a directed, weighted multi-graph using adjacency arrays and auxiliary arrays. In particular, threads concurrently add nodes to the graph, and transactions are used to synchronize accesses to the adjacency arrays. Data contention in ssca2 is relatively low, making this benchmark effective for assessing the overhead produced by our partial rollback implementation with respect to the traditional rollback scheme, evaluating it mostly for the forward execution. The second one, i.e. kmeans, is



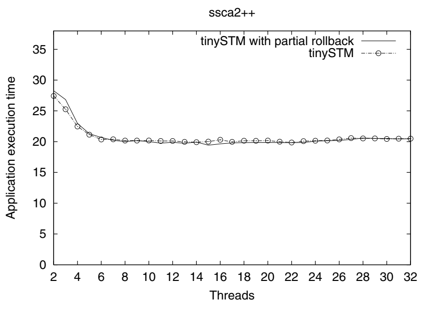


Fig. 2. Results with ssca2

a transactional implementation of a partition-based clustering method [18]. A cluster is represented by the mean value of all the objects it contains, and during the execution of this benchmark the mean points are updated by assigning each object to its nearest cluster center, based on Euclid distance. This benchmark relies on threads working on separate subsets of the data and uses transactions in order to assign portions of the workload and to store final results concerning the new centroid updates. Given the reduced amount of shared data structures being updated by transactions, in this benchmark it is more likely to incur in logical contention when a larger number of threads is used for the computation, which would allow us to better assess the benefits deriving from our partial rollback scheme. Also, we note that this is not a best case for our approach, since the amount of work saved from partial rollback is reduced, so that the overhead generated by the CPU/stack state saving/restoring is not completely amortized, which gives rise to a significative test case.

By the specification of STAMP, both the above applications can be characterized by two parameters. One is the size of the dataset, which has been changed in between '+' (indicating medium) and '++' (indicating large). The second one, particularly used for the `kmeans` benchmark, indicates the actual requirements of the transactions, in terms of, e.g., actual span of the accesses onto the dataset and, correspondingly, CPU requirements. This parameter is denoted as 'high' (indicating high demand) and 'low' (indicating reduced demand).

The execution latency that we have observed for `ssca2` while varying the number of threads (namely the number of used CPU-cores) up to 32 is shown in Figure 2, where each reported sample is the average value across 4 runs. As hinted before, this benchmark is characterized by relatively simple transactions, accessing a reduced amount of shared data, which leads the actual transactional work to be a relatively reduced percentage of the whole work carried out. This is reflected in that the actual data contention is very reduced, with the obvious outcome that partial rollback schemes cannot be expected to provide performance improvements, given the almost null amount of rolled back transactions. On the other hand, for this scenario we observe a small amount of overhead from the support for partial rollback. Specifically, for the case of the '+' configuration,

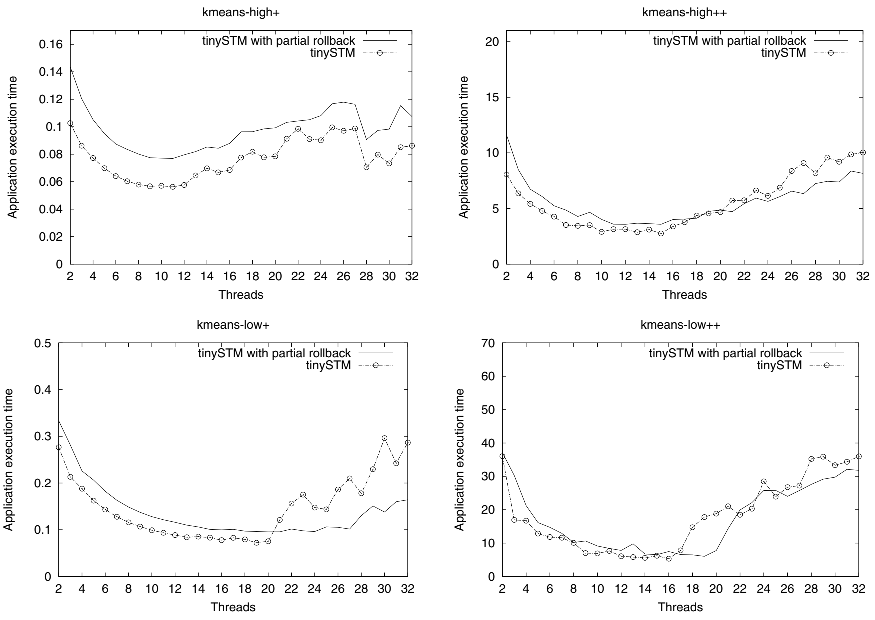


Fig. 3. Results with kmeans

we observe an overhead which is on the order of 7% for most of the considered concurrency levels (namely numbers of threads). Such an overhead is further reduced when considering the '++' configuration. The increased data set leads to reduced locality, making the execution slower than '+'. This causes the overhead from the CPU-context/stack logging mechanisms for partial rollback to be less evident.

In Figure 3 the results observed for the `kmeans` benchmark are reported, with each sample expressing again the mean value over 4 runs. In this setting, the most unfavorable configuration for our partial rollback scheme is 'high+', where the transactions have higher requirements, and access a reduced dataset. This leads to scenarios of high data conflict, likely occurring in the early phase of a transaction execution (due to reduced size of the accessed dataset). This leads the partial rollback scheme to induce a non-minimal amount of logging overhead, while not allowing a significative save of work for rolling back transactions due to the fact that rollbacks typically require transactions to resume from the beginning of their execution. This phenomenon is alleviated when considering the 'high++' configuration, where the increased size of the dataset leads to scenarios where at least a portion of the performed transactional work can be successfully saved, since the dataset largeness gives rise to dynamics where the likelihood of conflicting in the early phase of transaction execution gets reduced. This leads the partial rollback scheme to exhibit increased effectiveness, especially with a higher concurrency level. In such a case, in fact, the partial rollback scheme achieves up to 20% reduction of the benchmark execution time on 32 CPU-cores.

The 'low+' configuration of kmeans gives rise to execution dynamics that are not so far from those observable for the 'high++' configuration. Specifically, here transactions conflict while accessing a reduced data set, but they exhibit reduced resource requirements. Hence, also in this case there is no bias towards conflicting in the early phase of the execution. As a consequence, the partial rollback scheme operates effectively, especially when the level of concurrency is increased. Specifically, it provides reductions of the benchmark execution latency on the order of 40% as soon as the number of used CPU-cores is greater than 23. On the other hand, for reduced concurrency levels, the impact of transaction rollback gets reduced, which leads to the scenario where the partial rollback scheme induces logging overhead that does not get compensated by revenues while partially rolling back transactions. Such an overhead is better absorbed when running the 'low++' configuration (e.g. due to the aforementioned reduced locality phenomena within the benchmark). Hence for this scenario, we observe that partial rollback provides similar performance as the one achievable by the traditional scheme when the level of concurrency is limited, while it provides some performance advantages when this level gets increased, which leads to scenarios where the transaction rollback phenomenon is more relevant, thus rendering more useful the partial save of transactional work already carried out.

6 Summary

In this article we have presented the design and implementation of a support for application-transparent partial rollback in STM, tailored to contention managers relying on lazy (i.e. at commit-time) lock acquisition and on read validation mechanisms. It has been integrated within the open source TinySTM package, and has been tested on top of a 32-core HP ProLiant machine by running applications selected from the STAMP benchmark suite. By the data, our proposal allows for performance improvements in scenarios characterized by non-minimal data contention.

References

1. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing. ACM Press (August 1995)
2. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
3. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 237–246. ACM (2008)
4. Herlihy, M.P., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21, 289–300 (1993)

5. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory. SIGPLAN Notices 44(4), 141–150 (2009)
6. Lev, Y., Luchangco, V., Marathe, V.J., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a scalable software transactional memory. In: Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT. ACM (2009)
7. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: Proceedings of the IEEE International Symposium on Workload Characterization, ISWC (September 2008)
8. Rughetti, D., Di Sanzo, P., Ciciani, B., Quaglia, F.: Machine learning-based self-adjusting concurrency in software transactional memory systems. In: Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS, pp. 278–285. IEEE Computer Society (August 2012)
9. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Advanced concurrency control for transactional memory using transaction commit rate. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 719–728. Springer, Heidelberg (2008)
10. Yoo, R.M., Lee, H.H.S.: Adaptive transaction scheduling for transactional memory systems. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA, pp. 169–178. ACM (2008)
11. Lupei, D.: A study of conflict detection in software transactional memory. Master's thesis, University of Toronto, The Netherlands (2009)
12. Gupta, M., Shyamasundar, R.K., Agarwal, S.: Article: Clustered checkpointing and partial rollbacks for reducing conflict costs in stms. International Journal of Computer Applications 1(22), 80–85 (2010)
13. Gupta, M., Shyamasundar, R.K., Agarwal, S.: Automatic checkpointing and partial rollback in software transaction memory. US Patent 20110029490 (January 2012)
14. Pellegrini, A., Vitali, R., Quaglia, F.: Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation, PADS, pp. 45–53. IEEE Computer Society (2009)
15. Pellegrini, A.: Hijacker: Efficient static software instrumentation with applications in high performance computing (poster paper). In: Proceedings of the 2013 International Conference on High Performance Computing & Simulation, HPCS. IEEE Computer Society (July 2013)
16. Santoro, A., Quaglia, F.: Transparent state management for optimistic synchronization in the High Level Architecture. In: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation, pp. 171–180. IEEE Computer Society (June 2005)
17. Bader, D.A., Madduri, K.: Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In: Bader, D.A., Parashar, M., Sridhar, V., Prasanna, V.K. (eds.) HiPC 2005. LNCS, vol. 3769, pp. 465–476. Springer, Heidelberg (2005)
18. Bezdek, J.C.: Pattern Recognition with Fuzzy Objective Function Algorithms. Kluwer Academic Publishers, Norwell (1981)

Lightweight Contention Management for Efficient Compare-and-Swap Operations*

David Dice[1], Danny Hendler[2] and Ilya Mirsky[3]

[1] Sun Labs at Oracle
[2] Department of Computer Science & Telekom Innovation Laboratories, Ben-Gurion University of the Negev
[3] Department of Computer Science, Ben-Gurion University of the Negev

Abstract. Many concurrent data-structure implementations use the well-known *compare-and-swap* (CAS) operation, supported in hardware by most modern multiprocessor architectures, for inter-thread synchronization. A key weakness of the CAS operation is the degradation in its performance in the presence of memory contention.

In this work we study the following question: can software-based contention management improve the efficiency of hardware-provided CAS operations? Our performance evaluation establishes that lightweight contention management support can greatly improve performance under medium and high contention levels while typically incurring only small overhead when contention is low.

Keywords: Compare-and-swap, contention management, concurrent algorithms.

1 Introduction

Many key problems in shared-memory multiprocessors revolve around the coordination of access to shared resources and can be captured as *concurrent data structures* [3,13]: abstract data structures that are concurrently accessed by asynchronous threads. Efficient concurrent data structure algorithms are key to the scalability of applications on multiprocessor machines. Devising efficient and scalable concurrent algorithms for widely-used data structures such as counters (e.g., [11,14]), queues (e.g.,[1,8,18], and stacks (e.g.,[6,8]), to name a few, is the focus of intense research.

Modern multiprocessors provide hardware support of atomic read-modify-write operations in order to facilitate inter-thread coordination and synchronization. The *compare-and-swap* (CAS) operation has become the synchronization primitive of choice for implementing concurrent data structures - both lock-based and nonblocking [15] - and is supported by hardware in most contemporary multiprocessor architectures. The CAS operation takes three arguments: a memory

* Partially supported by the Israel Science Foundation (grant number 1227/10) and by the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

address[1], an old value, and a new value. If the address stores the old value, it is replaced with the new value; otherwise it is unchanged. The success or failure of the operation is then reported back to the calling thread.

A key weakness of the CAS operation, known to both researchers and practitioners of concurrent programming, is its performance in the presence of memory contention. When multiple threads concurrently attempt to apply CAS operations to the same shared variable, typically at most a single thread will succeed in changing the shared variable's value and the CAS operations of all other threads will fail. Moreover, significant degradation in performance occurs when variables manipulated by CAS become contention "hot spots": as failed CAS operations generate coherence traffic on most architectures, they congest the interconnect and memory devices and slow down successful CAS operations.

To illustrate this weakness of the CAS operation, Figure 1 shows the results of a simple test, conducted on an UltraSPARC T2 plus (Niagara II) chip, comprising 8 cores, each multiplexing 8 hardware threads, in which a varying number of Java threads run for 5 seconds, repeatedly reading the same variable and then applying CAS operations attempting to change its value.[2] The number of successful CAS operations scales from 1 to 4 threads but then quickly deteriorates, eventually falling to about 16% of the single thread performance, less than 9% of the performance of 4 threads. As we show in Section 3, similar performance degradation occurs on Intel's Xeon platform.

In this work we study the following question: can software-based contention management improve the efficiency of hardware-provided CAS operations? In other words, can a software contention management layer, encapsulating invocations of hardware CAS instructions, significantly improve the performance of CAS-based concurrent data-structures?

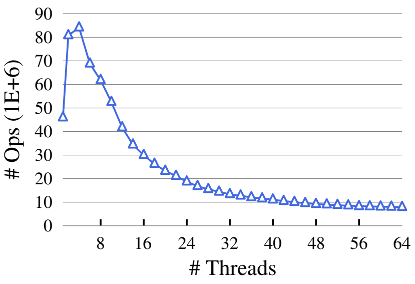


Fig. 1. SPARC: Java's CAS

To address this question, we conduct what is, to the best of our knowledge, the first study on the impact of contention management algorithms on the efficiency of the CAS operation. We implemented several Java classes that extend Java's *AtomicReference* class, and encapsulate calls to native CAS by *contention management classes*. This design allows for an almost-transparent plugging of our classes into existing data structures which make use of Java's *AtomicReference*. We then evaluated the impact of these algorithms on the Xeon, SPARC

[1] In object-oriented languages such as Java, the memory address is encapsulated by the object on which the CAS operation is invoked and is therefore not explicitly passed to the interface to the CAS operation.

[2] We provide more details on this test in Section 3.

and i7 platforms by using both a synthetic micro-benchmark and CAS-based concurrent data-structure implementations of stacks and queues.[3]

The idea of employing contention management and backoff techniques to improve performance was widely studied in the context of software transactional memory (see, e.g., [12,7]) and lock implementations (see, e.g., [2,17,4]). Backoff techniques are also used at the higher abstraction level of specific data structures implementations [9,10,18]. However, this approach adds complexity to the design of the data-structure and requires careful per-data structure tuning. Our approach, of adding contention management (and, specifically, backoff) mechanisms at the CAS instruction level, provides a simple and generic solution, in which tuning can be done *per architecture* rather than per implementation.

Our performance evaluation establishes that lightweight contention management support can significantly improve the performance of concurrent data-structure implementations as compared with direct use of Java's *AtomicReference* class. Our CAS contention management algorithms improve the throughput of the concurrent data-structure implementations we experimented with by up to a factor of 12 for medium and high contention levels, typically incurring only small overhead in low contention levels.

We also compared relatively simple data-structure implementations that use our CAS contention management classes with more complex implementations that employ data-structure specific optimizations. We have found that, in some cases, applying efficient contention management at the level of CAS operations, used by simpler and non-optimized data-structure implementations, yields better performance than that of highly optimized implementations of the same data-structure that use Java's *AtomicReference* objects directly.

Our results imply that encapsulating invocations of CAS by lightweight contention management algorithms is a simple and generic way of significantly improving the performance of concurrent objects.

The rest of this paper is organized as follows. We describe the contention management algorithms we implemented in Section 2. We report on our experimental evaluation in Section 3. We conclude the paper in Section 4 with a short discussion of our results.

2 Contention Management Algorithms

In this section, we describe the Java CAS contention management algorithms that we implemented and evaluated. These algorithms are implemented as classes that extend the *AtomicReference* class of the *java.util.concurrent.atomic* package. Each instance of these classes operates on a specific location in memory and implements the *read* and *CAS* methods.[4]

In some of our algorithms, threads need to access per-thread state associated with the object. For example, a thread may keep a record of the number of CAS

[3] We note that the lock-freedom and wait-freedom progress properties aren't affected by our contention management algorithms since in all of them a thread only waits for a bounded period of time.

[4] None of the methods of *AtomicReference* are overridden.

failures it incurred on the object in the past in order to determine how to proceed if it fails again. Such information is stored as an array of per-thread structures. To access this information, threads call a *registerThread* method on the object to obtain an index of an array entry. This thread index is referred to as *TInd* in the pseudo-code. After registering, a thread may call a *deregisterThread* method on the object to indicate that it is no longer interested in accessing this object and that its entry in this object array may be allocated to another thread.[5]

Technically, a thread's *TInd* index is stored as a thread local variable, using the services of Java's *ThreadLocal* class. The *TInd* index may be retrieved within the CAS contention management method implementation. However, in some cases it might be more efficient to retrieve this index at a higher level (for instance, when *CAS* is called in a loop until it is successful) and to pass it as an argument to the methods of the CAS contention management object.

The ConstantBackoffCAS Algorithm : Algorithm 1 presents the *ConstantBackoffCAS* class, which employs the simplest contention management algorithm that we implemented. No per-thread state is required for this algorithm. The *read* operation simply delegates to the *get* method of the *AtomicReference* object to return the current value of the reference (line **2**). The *CAS* operation invokes the *compareAndSet* method on the *AtomicReference* superclass, passing to it the *old* and *new* operands (line **4**). The *CAS* operation returns *true* in line **7** if the native CAS succeeded. If the native CAS failed, then the thread busy-waits for a platform-dependent period of time, after which the *CAS* operation returns (lines **5**–**6**).

The TimeSliceCAS Algorithm : Algorithm 2 presents the *TimeSliceCAS* class, which implements a time-division contention-management algorithm that, under high contention, assigns different time-slices to different threads. Each instance of the class has access to a field *regN* which stores the number of threads that are currently registered at the object.

The *read* operation simply delegates to the *get* method of the *AtomicReference* class (line **9**). The *CAS* operation invokes the *compareAndSet* method on the *AtomicReference* superclass (line **11**). If the CAS is successful, the method returns *true* (line **12**). If the CAS fails and the number of registered threads exceeds a platform-dependent level *CONC* (line **13**), then the algorithm attempts to limit the level of concurrency (that is, the number of threads concurrently attempting CAS on the object) at any given time to at most *CONC*. This is done as follows. The thread picks a random integer slice number in the range $\{1, \ldots, \lceil regN/CONC \rceil\}$ (line **14**). The length of each time-slice is set to 2^{SLICE} nanoseconds, where *SLICE* is a platform-dependent integer. The thread waits until its next time-slice starts and then returns *false* (lines **14**–**17**).

The ExpBackoffCAS Algorithm : Algorithm 3 presents the *ExpBackoffCAS* class, which implements an exponential backoff contention management

[5] An alternative design is to have a global registration/deregistration mechanism so that the *TInd* index may be used by a thread for accessing several CAS contention-management objects.

Algorithm 1: `ConstBackoffCAS`

```
1  public class ConstBackoffCAS<V> extends AtomicReference<V>;
2  public V read() { return get() }
3  public boolean CAS(V old, V new)
4      if ¬compareAndSet(old,new) then
5          wait(WAITING_TIME) ;
6          return false ;
7      else return true ;
```

Algorithm 2: `TimeSliceCAS`

```
8  public class TimeSliceCAS<V> extends AtomicReference<V>;
9  public V read() { return get() }
10 public boolean CAS(V old, V new)
11     if compareAndSet(old,new) then
12         return true
13     if regN > CONC then
14         int sliceNum = Random.nextInt(⌈regN/CONC⌉)
           repeat
15             currentSlice = (System.nanoTime() >> SLICE) % ⌈regN/CONC⌉;
16         until sliceNum = currentSlice;
17     return false ;
```

Algorithm 3: `ExpBackoffCAS`

```
18 public class ExpBackoffCAS<V> extends AtomicReference<V>;
19 private int[] failures = new int [MAX_THREADS] ;
20 public V read() { return get() }
21 public boolean CAS(V old, V new)
22     if compareAndSet(old,new) then
23         if failures[TInd] > 0 then
24             failures[TInd]−−;
25         return true
26     else
27         int f = failures[TInd]++ ;
28         if f > EXP_THRESHOLD then wait($2^{min(c \cdot f,m)}$);
29         return false;
```

Fig. 2. Fairness measures

	Normal stdev		Jain's Index	
	Xeon	SPARC	Xeon	SPARC
Java	0.291	0.164	0.900	0.961
CB-CAS	0.077	0.196	0.992	0.957
EXP-CAS	0.536	0.936	0.761	0.588
MCS-CAS	0.975	0.596	0.563	0.727
AB-CAS	0.001	0.822	1.000	0.638
TS-CAS	0.829	0.211	0.605	0.946

algorithm. Each instance of this class has a *failures* array, each entry of which – initialized to 0 – stores simple per-registered thread statistics about the history of successes and failures of past CAS operations to this object (line 19). The *read* operation simply delegates to the *get* method of the *AtomicReference* class (line 20).

The *CAS* operation invokes the *compareAndSet* method on the *AtomicReference* superclass (line 22). If the CAS is successful, then the *CAS* operation returns *true* (line 25).

If the CAS fails, then the thread's entry in the *failures* array is incremented and if its value f is larger than a platform-dependent threshold, the thread waits for a period of time proportional to $2^{min(c \cdot f,m)}$ where c and m are platform-dependent integer algorithm parameters (lines 27–28), eventually returning *false.*

The MCS-CAS and ArrayBasedCAS Algorithms : *MCS-CAS* and *ArrayBasedCAS* are described here briefly for lack of space. They are much more complex than the first 3 algorithms we described. Pseudo codes and full descriptions appear in a technical report [5]. *MCS-CAS* implements a variation of the Mellor-Crummey and Scott (MCS) lock algorithm [17] to serialize load-CAS operations. Since we would like to maintain the nonblocking semantics of the CAS operation, a thread t awaits its queue predecessor (if any) for at most a platform-dependent period of time. If this waiting time expires, t proceeds with the read operation without further waiting.

The *ArrayBasedCAS* algorithm uses an array-based signaling mechanism, in which a *lock owner* searches for the next entry in the array, on which a thread is waiting for permission to proceed with its load-CAS operations, in order to signal it. Also in this algorithm, waiting-times are bounded.

3 Evaluation

We conducted our performance evaluation on the SPARC, Intel's Xeon and i7 multi-core CPUs. The i7 results are very similar to those on Xeon. For lack of space, they are only described in the technical report [5]. The SPARC machine comprises an UltraSPARC T2 plus (Niagara II) chip containing 8 cores, each core multiplexing 8 hardware threads, for a total of 64 hardware threads. It runs the 64-bit Solaris 10 operating system with Java SE 1.6.0 update 23. The Xeon machine comprises a Xeon E7-4870 chip, containing 10 cores and hyper-threaded to 20 hardware threads. It runs the 64-bit Linux 3.2.1 kernel with Java SE 1.6.0 update 25.

Initially we evaluated our CAS contention management algorithms using a synthetic micro-benchmark and used the results to optimize the platform-dependent parameters used by the algorithms. We then evaluated the impact of our algorithms on implementations of widely-used data structures such as queues and stacks. No explicit threads placement was used.

3.1 The CAS Micro-benchmark

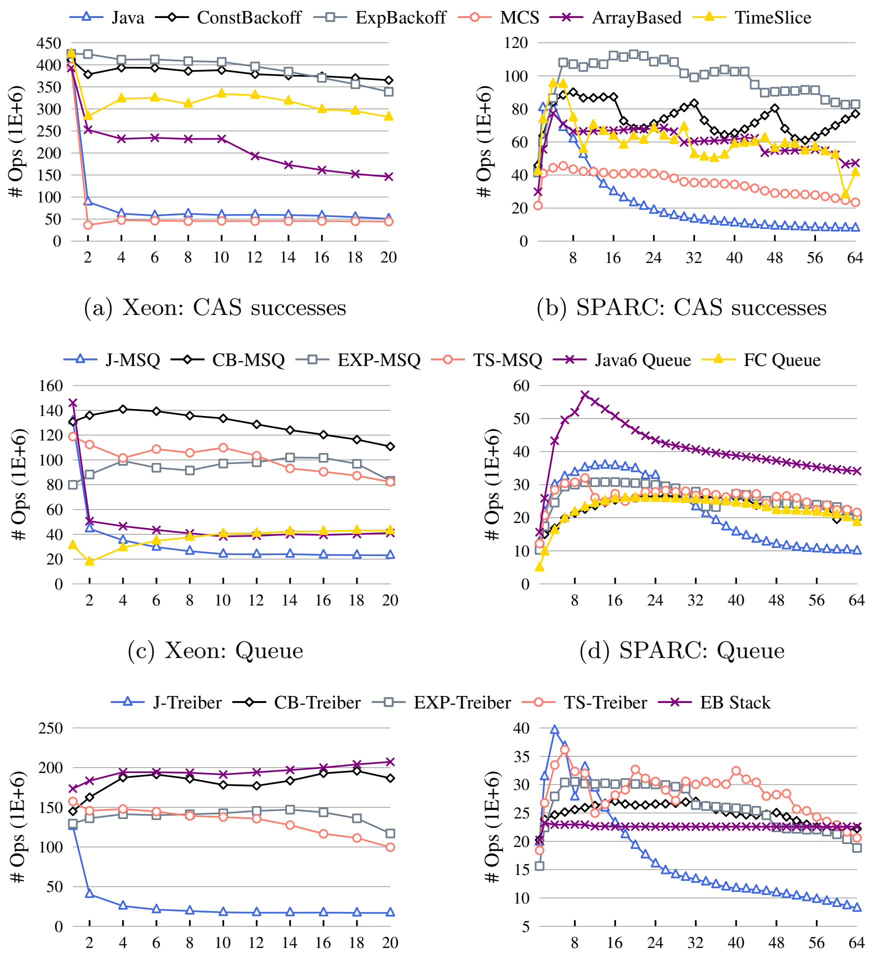
To tune and compare our CAS contention management algorithms, we used the following synthetic *CAS benchmark*. For every concurrency level k, varying from 1 to the maximum number of supported hardware threads, k threads repeatedly read the same atomic reference and attempt to CAS its value, for a period of 5 seconds. Before the test begins, each thread generates an array of 128 objects and during the test it attempts to CAS the value of the shared object to a reference to one of these objects, in a round-robin manner.

Using the CAS benchmark, we've tuned the parameters used by the algorithms described in Section 2. The values that were chosen as optimal were those that produced the highest average throughput of all concurrency levels. [6]

Figure 3a shows the throughput (the number of successful CAS operations) on the Xeon machine as a function of the concurrency level. Each data point is the average of 10 independent executions. It can be seen that the throughput of Java CAS falls steeply for concurrency levels of 2 or more. Whereas a single thread performs approximately 413M successful CAS operations in the course of the test, the number of successful CAS operations is only approximately 89M for 2 threads and 62M for 4 threads. For higher concurrency levels, the number of successes remains in the range of 50M-58M operations.

In sharp contrast, both the constant wait and exponential backoff CAS algorithms are able to maintain high throughput across the concurrency range. Exponential backoff is slightly better up until 16 threads, but then its throughput declines to below 350M and falls below constant backoff. The throughput

[6] A table with the values of tuned parameters is provided in [5].



(a) Xeon: CAS successes

(b) SPARC: CAS successes

(c) Xeon: Queue

(d) SPARC: Queue

(e) Xeon: Stack

(f) SPARC: Stack

Fig. 3. Benchmark results: throughput as a function of concurrency level

of both these algorithms exceeds that of Java CAS by a factor of more than 4 for 2 threads and their performance boost grows to a factor of between 6-7 for higher concurrency levels. The time slice algorithm is the 3'rd performer in this test, outperforming Java CAS by a factor of between 3-5.6 but providing only between 65%-87% the throughput of constant and exponential backoff. The array based and MCS-CAS algorithms significantly lag behind the simpler back-off algorithms. More insights into the results of these tests are provided in the technical report [5].

Figure 3b shows the throughput of the evaluated algorithms in the CAS benchmark on the SPARC machine. Unlike Xeon where Java CAS does not scale at all, on SPARC the performance of Java CAS scales from 1 to 4 threads but then quickly deteriorates, eventually falling to about 16% of the single thread performance, less than 9% of the performance of 4 threads. Java CAS is the worst performer for concurrency levels 12 or higher and its throughput drops to about 8M for 64 threads. The exponential backoff CAS is the clear winner on the SPARC CAS benchmark. For concurrency levels 28 or more, exponential backoff completes more than 7 times successful CAS operations compared with Java CAS and the gap peaks for 54 threads where Java CAS is outperformed by a factor of almost 12. The constant wait CAS is second best. The high overhead of MCS-CAS and array based CAS manifests itself in the single thread test, where both provide significantly less throughput than all other algorithms. For higher concurrency levels, both MCS-CAS and array based CAS perform between 30M-60M successful CAS operations, significantly more than Java CAS but much less than the constant and exponential backoff algorithms.

Analysis : As shown by Figures 3a and 3b, whereas the number of successes in the CAS benchmark on the SPARC scales up to 4 or 8 threads (depending on the contention management algorithm being used), no such scalability occurs on the Xeon. In the technical report [5] we provide a detailed explanation of the architectural reasons for this difference. For lack of space, we only provide a brief explanation here.

T2+ processors enjoy very short cache-coherent communication latencies relative to other processors. On an otherwise unloaded system, a coherence miss can be satisfied from the L2 cache in under 20 cycles. CAS instructions are implemented on SPARC at the interface between the cores and the cross-bar. For ease of implementation, CAS instructions, whether successful or not, invalidate the line from the issuer's L1. A subsequent load from that same address will miss in the L1 and revert to the L2. The cross-bar and L2 have sufficient bandwidth and latency, relative to the speed of the cores, to allow load-CAS benchmarks to scale beyond just one thread, as we see in Figure 3b.

We now describe why such scalability is not observed on the Xeon platform, as seen by Figure 3a. Modern x86 processors tend to have deeper cache hierarchies, often adding core-local MESI L2 caches connected via an on-chip coherent interconnect fabric and backed by a chip-level L3. Intra-chip inter-core communication is accomplished by L2 cache-to-cache transfers. In addition to the cost of obtaining cache-line ownership, load-CAS benchmarks may also be subject to a number of additional confounding factors on x86 which we describe in the technical report [5].

Fairness : Table 2 summarizes the fairness measures of the synthetic CAS benchmarks. We used normalized standard deviation and Jain's fairness index [16] to assess the fairness of individual threads' throughput for each concurrency level, and then took the average over all concurrency levels. The widely used Jain's index for a set of n samples is the quotient of the square of the sum and the product of the sum of squares by n. Its value ranges between $1/n$

(lowest fairness) and 1 (highest fairness). It equals k/n when k threads have the same throughput, and the other $n-k$ threads are starved. We see that CB-CAS and TS-CAS provide comparable and even superior fairness to Java CAS while the rest of the algorithms provide less fairness.

3.2 FIFO Queue

To further investigate the impact of our CAS contention management algorithms, we experimented with the FIFO queue algorithm of Michael and Scott [18] (MS-queue).[7] The queue is represented by a list of nodes and by *head* and *tail* atomic references to the first and last entries in the list, which become hot spots under high contention.

We evaluated four versions of the MS-queue: one using Java's *AtomicReference* objects (called J-MSQ), and the other three replacing them by *ConstantBackoff-CAS*, *ExpBackoffCAS* and *TimeSlice* objects (respectively called CB-MSQ, Exp-MSQ and TS-MSQ). We also compared with the flat-combining queue algorithm [8]. MCS-CAS and array based CAS were consistently outperformed and are therefore omitted from the following comparison. We compared these algorithms with the Java 6 *java.util.concurrent.ConcurrentLinkedQueue* class.[8] The *ConcurrentLinkedQueue* class implements an algorithm (henceforth simply called Java 6 queue) that is also based on Michael and Scott's algorithm. However, the Java 6 queue algorithm incorporates several significant optimizations such as performing lagged updates of the head and tail references and using *lazy sets* instead of normal writes.

We conducted the following test. For varying number of threads, each thread repeatedly performed either an *enqueue* or a *dequeue* operation on the data structure for a period of 5 seconds. The queue is pre-populated by 1000 items. A pseudo-random sequence of 128 integers is generated by each thread independently before the test starts where the *i*'th operation of thread *t* is an *enqueue* operation if integer (i mod 128) is even and is a *dequeue* operation otherwise.

Figures 3c and 3d show the results of the queue tests on the Xeon and SPARC platforms. As shown by Figure 3c, CB-MSQ is the best queue implementation on Xeon, outperforming the *AtomicReference*-based queue in all concurrency levels by a factor of up to 6 (for 16 threads). Surprisingly, CB-MSQ also outperforms the Java 6 queue by a wide margin in all concurrency levels except 1, in spite of the optimizations incorporated to the latter. More specifically, in the single thread test Java 6 queue performance exceeds that of CB-MSQ by approximately 15%. In higher concurrency levels, however, CB-MSQ outperforms Java 6 queue by a factor of up to 3.5. Java 6 queue is outperformed in all concurrency levels higher than 1 also by EXP-MSQ and TS-MSQ. The FC queue hardly scales on this test and is outperformed by almost all algorithms in most concurrency levels.

[7] We used the Java code provided in Herlihy and Shavit's book [13] without any optimizations.

[8] We used a slightly modified version in which direct usage of Java's *Unsafe* class was replaced by an *AtomicReference* mediator.

Figure 3d shows the results of the queue tests on the SPARC machine. Here, unlike on Xeon, the Java 6 queue has the best throughput in all concurrency levels, outperforming TS-MSQ - which is second best in most concurrency levels - by a factor of up to 2. It seems that the optimizations of the Java 6 algorithm are more effective on the SPARC architecture. CB-MSQ starts low but its performance scales up to 22 threads where it almost matches that of EXP-MSQ.

J-MSQ scales up to 12 threads where it performs approximately 35M queue operations, but quickly deteriorates in higher concurrency levels. For concurrency levels 50 or higher, J-MSQ is outperformed by CB-MSQ by a factor of 2 or more. Unlike on Xeon, the FC queue scales on SPARC up to 24 threads, when its performance almost equals that of the simple backoff schemes.

3.3 Stack

We also experimented with the lock-free stack algorithm of Treiber [19]. The Treiber stack is represented by a list of nodes and a reference to the top-most node is stored by an *AtomicReferece* object. We evaluated the following versions of the Treiber algorithm: one using Java's *AtomicReference* objects (called J-Treiber), and the other three replacing them by the *ConstantBackoffCAS*, *ExpBackoffCAS* and *TimSli*ceCAS (respectively called CB-Treiber, Exp-Treiber and TS-Treiber). We also compared with a Java implementation of the elimination-backoff stack (EB stack) of Hendler et al. [9].[9] The structure of the Stack test is identical to that of the Queue test.

Figure 3e shows the results of the stack test on Xeon. As with all Xeon test results, also in the stack test, the implementation using Java's *AtomicReference* suffers from a steep performance decrease as concurrency levels increase. The EB stack is the winner of the Xeon stack test and CB-Treiber is second-best lagging behind only slightly. CB-Treiber maintains and even exceeds its high single-thread throughput across the concurrency range, scaling up from less than 150M operations for a single thread to almost 200M operations for higher concurrency levels, outperforming J-Treiber by a factor of more than 10 for high concurrency levels. TS-Treiber and EXP-Treiber are significantly outperformed by the EB stack and CB-Treiber algorithms.

Figure 3f shows the results of the stack tests on SPARC. J-Treiber scales up to 6 threads where it reaches its peak performance. Then its performance deteriorates with concurrency and reaches less than 10M operations for 64 threads. From concurrency level 16 and higher, J-Treiber has the lowest throughput. TS-Treiber has the highest throughput in most medium and high concurrency levels, with EXP-Treiber mostly second best. Unlike on Xeon, EB stack is almost consistently and significantly outperformed on SPARC by all simple backoff algorithms.

[9] We used IBM's implementation available from the Amino Concurrent Building Blocks project at `http://amino-cbbs.wiki.sourceforge.net/`

4 Discussion

We conduct what is, to the best of our knowledge, the first study on the impact of contention management algorithms on the efficiency of the CAS operation. We implemented several Java classes that encapsulate calls to Java's *AtomicReference* class by CAS contention management algorithms. We then evaluated the benefits gained by these algorithms on the Xeon, SPARC and i7 platforms by using both a synthetic benchmark and CAS-based concurrent data-structure implementations of stacks and queues.

Out of the contention management approaches we have experimented with, the three simplest algorithms - constant backoff, exponential backoff and time-slice - yielded the best results, primarily because they have very small overheads. The more complicated approaches - the MCS-CAS and array based CAS algorithms - provided better results than direct calls to *AtomicReferece* in most tests, but were significantly outperformed by the simpler approaches.

Our evaluation demonstrates that encapsulating Java's *AtomicReference* by objects that implement lightweight contention management support can improve the performance of CAS-based algorithms considerably.

We also compared relatively simple data-structure implementations that use our CAS contention management classes with more complex implementations that employ data-structure specific optimizations and use *AtomicReference* objects. We have found that, in some cases, simpler and non-optimized data-structure implementations that apply efficient contention management for CAS operations yield better performance than that of highly optimized implementations of the same data-structure that use Java's *AtomicReference* directly.

Our results imply that encapsulating invocations of CAS by lightweight contention management classes is a simple and generic way of improving the performance of concurrent objects.

This work may be extended in several directions. First, we may have overlooked CAS contention management algorithms that yield better results. Second, our methodology tuned the platform-dependent parameters of contention management algorithms by using the CAS benchmark. Although the generality of this approach is appealing, tuning these parameters per data-structure may yield better results. Moreover, a dynamic tuning may provide a general, cross data-structure, cross CPU, solution.

It would also be interesting to investigate if and how similar approaches can be used for other atomic-operation related classes in both Java and other programming languages such as C/C++.

Finally, combining contention management algorithms at the atomic operation level with optimizations at the data-structure algorithmic level may yield more performance gains than applying only one of these approaches separately. We leave these research directions for future work.

Acknowledgements. We thank Yehuda Afek, Nir Shavit and Tel-Aviv's University's Multicore Computing Group for kindly allowing us to use their servers for our evaluation.

References

1. Afek, Y., Hakimi, M., Morrison, A.: Fast and scalable rendezvousing. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 16–31. Springer, Heidelberg (2011)
2. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 1(1), 6–16 (1990)
3. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn. John Wiley Interscience (March 2004)
4. Craig, T.S.: Building fifo and priority-queuing spin locks from atomic swap. Technical report (1993)
5. Dice, D., Hendler, D., Mirsky, I.: Lightweight Contention Management for Efficient Compare-and-Swap Operations. ArXiv e-prints (May 2013)
6. Fatourou, P., Kallimanis, N.D.: Revisiting the combining synchronization technique. In: PPOPP, pp. 257–266 (2012)
7. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: PODC, pp. 258–264. ACM, New York (2005)
8. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: SPAA, pp. 355–364 (2010)
9. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. J. Parallel Distrib. Comput. 70(1), 1–12 (2010)
10. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Trans. Program. Lang. Syst. 15(5), 745–770 (1993)
11. Herlihy, M., Lim, B.-H., Shavit, N.: Scalable concurrent counting. ACM Trans. Comput. Syst. 13(4), 343–364 (1995)
12. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101. ACM, New York (2003)
13. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco (2008)
14. Herlihy, M., Shavit, N., Waarts, O.: Linearizable counting networks. Distributed Computing 9(4), 193–203 (1996)
15. Herlihy, M.P.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 123–149 (1991)
16. Jain, R., Chiu, D.M., Hawe, W.: A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, DEC research, TR-301 (1984)
17. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. TOCS 9(1), 21–65 (1991)
18. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC, pp. 267–275 (1996)
19. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (April 1986)

MacroDB: Scaling Database Engines on Multicores*

João Soares, João Lourenço and Nuno Preguiça

CITI/DI-FCT-Univ. Nova de Lisboa

Abstract. Multicore processors are available for over a decade, but general purpose database management systems (DBMS) still cannot fully explore the computational resources of these platforms. This paper explores a simple and easy to deploy approach for improving DBMS performance in multicore platforms, by maintaining multiple database engines running in parallel, rather than a single instance, thus circumventing the increase in contention due to load interactions. Unlike previous works, we focus on in-memory DBMS, exploring different design solutions that combine distributed systems and concurrent programming techniques. We show that we are able to improve performance over standalone solutions, without modifying either database or application code, by up to 3 times while minimizing response times.

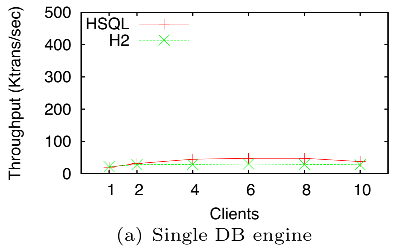
1 Introduction

Multicore processors are now available for over a decade, and still pose challenges to the design of database management systems (DBMS) [10,17,21,3,7]. Existing studies show that current DBMS engines can spend more than 30% of time in synchronization-related operations (e.g. locking and latching), even when only a single client thread is running [11]. Additionally, running two concurrent database operations in parallel can be slower than running them in sequence [26], due to workload interference. This is a limiting factor for the scalability of DBMS in current multicore platforms [19].

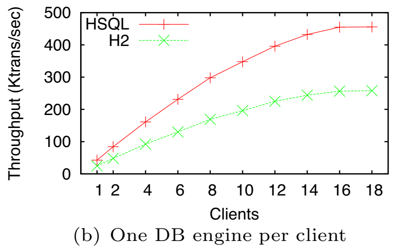
Several research solutions have been proposed to improve the use of resources offered by multicore machines. Some solutions aim at using multiple threads to execute query plans in parallel, or using new algorithms to parallelize single steps of the plan, or effectively parallelizing multiple steps [25,26,7,6,3]. Other solutions try to reuse part of the work done during the execution of multiple queries [9], or using additional threads to prefetch data that can be needed in the future [17]. Although some of these solutions start to appear in niche markets, general purpose DBMSs have been slower to adopt them, since implementing such solutions requires significant design modifications.

This paper addresses the problem of improving the scalability of DBMS on multicore machines, focusing on in-memory databases (IMDB). IMDBs provide

* This work was partially support by FCT/MCT projects PEst-OE/E-EI/UI0527/2011 and PTDC/EIA-EIA/108963/2008. João Soares was partially supported by FCT/M-CTES research grant # SFRH/ BD/ 62306/ 2009.



(a) Single DB engine



(b) One DB engine per client

Fig. 1. Scalability of read-only workload

high performance because they do not incur in disk I/O overhead. The high performance and ease of embedding them in applications have made these systems increasingly popular, being used by a large number of applications and high-performance transaction processing systems, such as Sprint [4] and H-Store [14].

Scalability issues of in-memory DBMS. Compared to disk based databases, IMDBs incur in no overhead or contention in accessing I/O. Thus, we would expect these systems to scale with the number of cores. To verify if this was true, we have run the TPC-C benchmark with a 100% read-only workload in popular HSQL and H2 IMDBs on a 16 core Sun Fire X4600 with 32 GBytes of RAM. The results of Figure 1(a) show that these engines do not scale, even when transactions do not conflict with each other. For understanding if the lack of scalability was due to lack of resources, we concurrently ran an increasing number of pairs client/DB engine in the same machine. Figure 1(b) presents the results of such experiment, showing an increasing aggregate throughput. These results make it clear that the problem lies in the design of current IMDBs.

1.1 Proposed Approach

In this paper, we explore a simple and easy to deploy mechanism for scaling IMDB, by relying on database replication and building on knowledge from distributed and replicated database systems. By treating multicore machines as an extremely low latency cluster, extended with shared memory, we deploy a middleware system, MacroDB, as a collection of coordinated IMDB replicas, for providing scalable database performance on multicores systems.

MacroDB uses a master/slave replication approach, where update transactions execute on the master replica, which holds the primary copy of the database. The slaves maintain independent secondary copies of the database, receiving read-only transactions from clients, while updates are asynchronously propagated to them upon commit on the primary. This approach minimizes contention since: *i*) Read-only transactions are fully executed on slave replicas, reducing the number of transactions each replica processes, thus distributing the load among the available replicas, and *ii*) update transactions are applied in slave replicas as sequential batches of updates, leading to no contention among them.

MacroDB provides a scalable data management solution, that does not require any changes to neither database engines or the applications. Our experiments

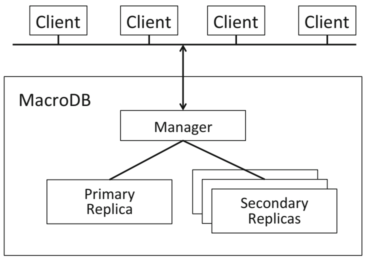


Fig. 2. MacroDB Architecture

show that MacroDB is able to provide performance benefits ranging from 40% to 180% over standalone database engines in a diverse range of benchmark workloads, such as TPC-C and TPC-W, while for write-dominated workloads, such as TPC-C, MacroDB suffers from only a 5% to 14% overhead over standalone solutions. Additionally, the memory used by the database replicas is not directly proportional to the number of replicas, as replicas share immutable Java objects, thus making MacroDB practical even with large numbers of replicas.

This paper is organized as follows, section 2 describes our system and some of the prototype considerations. Section 3 presents the evaluation results. Section 4 presents some related work, and section 5 concludes this paper.

2 MacroDB

MacroDB is a middleware infrastructure for scaling IMDBs on multicore machines. It replicates the database on several engines, all running on the same machine, while offering a *single-copy serializable view* of the database to clients [2].

It works independently of the underlying database engine, acting as a transparent layer between applications and the database. Statements received from the application are passed, without modifications, to the underlying engines. This makes MacroDB easy to deploy, since it does not require any modification to existing applications or database engines. This section details the architecture and algorithms used in the system.

2.1 Architecture

The MacroDB architecture, depicted in Figure 2, is composed by two main components: the *manager*, responsible for coordinating transaction execution in the database replicas; and the *database replicas*, i.e., the engines responsible for maintaining copies of the database. Clients remain oblivious of the replicated nature of MacroDB since it offers them a standard JDBC interface, and provides them with a *single-copy serializable view* of the database [2].

Clients do not communicate directly with the database engines, instead they communicate with the MacroDB *manager*, a JDBC compliant front-end which coordinates client queries and the underlying replicas. The *manager* receives statements from clients and forwards them, without modification, to the appropriate replica, guaranteeing their ordered execution, and replying to clients the respective results. Its main function is to: *i*) route client request to the appropriated replica, *ii*) manage operation execution to guarantee that the system provides a single consistent serializable view of the replicated database to the applications, and *iii*) to detect and recover possible replica failure. In the next section, we detail transaction execution.

2.2 Transaction Execution

MacroDB uses a master-slave replication scheme [12,23]. The master maintains the primary copy of the database, while the slaves maintain secondary replicas. Update transactions, received from clients by the *manager* are executed concurrently on the primary copy, being asynchronously propagated to the secondary replicas upon commit. This means that secondary replicas might not be completely up to date at a given moment. Read-only transactions execute concurrently on the secondaries.

Each secondary replica maintains: *i*) an associated *version*, that maintains the number of update transactions committed in the replica. This version is kept in shared memory, as an atomic *commit counter*, and can be accessed by any thread running in MacroDB; *ii*) a list for pending update batches, and *iii*) a thread responsible for executing these batches.

We will now detail the steps for executing update and read-only transactions. We assume the setReadOnly method of the JDBC interface is used for defining read-only and update transactions. The code for transaction execution is presented in Figure 3. For simplicity, we omit the code for error handling and present an explicit *begin* transaction operation - in the prototype, the code for begin transaction is executed when the first query or update operation is called after a commit or rollback.

Update Transactions. For each client connection, when an update transaction begins, a newly associated batch is created. All statements executed in this context are executed by the master replica, using the context of the caller thread, and their results returned to the client. Additionally, if the operation was an update, it is added to the batch for that transaction.

If the client decides to commit the transaction, the commit is executed in the master replica, using the caller thread. If the commit succeeds, the version number associated with the master replica is incremented and the update batch, stamped with that version number, is inserted into the lists of pending batches for the secondary replicas. For correct transaction ordering, MacroDB needs to guarantee that no new transaction starts and commits between the commit of a transaction and its ordering, thus this it the only operation that requires coordination with other threads. For guaranteeing that the commit does not block, we require the underlying database engines to use two-phase locking (instead of commit time certification strategies), which is the case in most in-memory database systems. If the client decides to rollback or if the underlying engine is

```
var global:  atomic int version[0..num replicas]
Map pendingTx[1..num replicas]
Connection connB[1..num replicas]

var per client: Connection conn[0..num replicas]
Batch txOps
int txReplica

function begin( boolean readOnly)
    active = true
    if readOnly then
        txVrs = version[0]
        txReplica = SelectReplica()
        wait until version[txReplica] >= txVrs
    else
        txReplica = 0
        txOps = new Batch

function execQuery( Statement query)
    conn[txReplica].execQuery( query)

function execUpdate( Statement update)
    conn[0].execQuery( query)
    txOps.add( update)

function commit()
    if readOnly then conn[txReplica].commit()
    else
        LOCK REPLICA 0
            result = conn[0].commit()
            if NOT result then throw CommitFailed
            newVrs = ++version[0]
        for i:= 1 to num secondary replicas
            pendingTx[i].put( newVrx, txOps)

function threadLoop( int num)
    vrs = version[ num]
    forever
        batch = pendingTx.blockingGetRemove( vrs + 1)
        connB.execBatch( batch)
        LOCK REPLICA num
            connB[num].commit()
            vrs = ++version[num]
```

Fig. 3. MacroDB code

unable to commit the transaction, a rollback is executed in the master replica and the associated batch is discarded.

The thread associated with each secondary replica waits for the next update batch to be inserted into the associated list, and executes it. Then, it atomically commits the transaction and advances the version associated with the replica. Since these updates are performed sequentially, we guarantee that no deadlock will occur on the secondaries, thus all update batches will commit successfully. By sequentially executing each commit in the master replica, and advancing the version counter, we define a correct serialization order for update transactions, without forcing an *a priori* commit order. Executing update batches in secondary replicas in the same order as in the primary guarantees that all replicas evolve to the same consistent state.

Read-only transactions. All queries from read-only transactions execute directly on secondary replicas, being executed in the context of the caller thread. For providing a single consistent view of the replicated database, MacroDB enforces that a read-only transaction will only execute on a secondary that is up-to-date, i.e., when its version is, at least, equal to the version of the primary when that transaction started. On the beginning of a read-only transaction, the manager reads the current version of the primary replica. This defines the version for the secondary replica in which the transaction will execute, waiting, if necessary, until the selected secondary is up-to-date, i.e., it waits until the version on the selected secondary is, at least, equal to the version read from the primary. This guarantees that the selected secondary is not in an old state, which could potentially lead to a violation of causality for the client. Thus, MacroDB provides clients with a *single copy serializable view* of the replicated database.

Fault Handling. When a replica fault is detected, an immediate recovery process is initiated. If the master fails, all currently executing update transactions abort, and all new update transactions are postponed until a new master is active. A new master is then selected from the set of secondary replicas, replacing the previous one after successfully executing all pending update batches. This guarantees that no update transaction is lost, and that all replicas have the same consistent state, since all update batches have been executed in all of them. At the moment of this selection, new read-only transactions are only forwarded to the remaining secondaries. At this moment the new master becomes active and the system behaves as if a secondary replica had failed. Whenever a secondary replica fails, its current transactions abort, and new read-only transactions are forwarded to the remaining replicas. A new secondary replica is then created and recovered from a non-faulty one.

2.3 Correctness

For the correctness of the system, it is necessary to guarantee that all replicas evolve to the same state after executing the same set of transactions. Also, for guaranteeing that MacroDB provides a single consistent view of the replicated database, it is necessary to guarantee that a transaction is always serialized after all update transactions that may precede it commit. This is achieved because the system enforce the following properties.

Theorem 1. *All replicas commit all update transactions in the same, serializable, order.*

Proof. At the primary, as commits execute atomically in isolation, the serializable order is defined by the order of each commit. Since secondary replicas execute update transactions in a single thread, i.e., sequentially, by the same order, all replicas commit all update transactions in the same order.

Theorem 2. *A transaction is serialized after all update transactions that precede it commit.*

Proof. For update transactions, this is guaranteed by the database engine at the primary. For read-only transactions, MacroDB enforces this property by delaying the beginning of a transaction until the secondary replica has executed all transactions committed at the moment the begin transaction was called.

2.4 Minimizing Contention for Efficient Execution

As presented earlier, the master-slave replication approach used in MacroDB executes update and read-only transactions in different replicas. Thus, read-only transactions never block update transactions and vice-versa, since these execute in distinct replicas. The execution of update transactions in secondary replicas may interfere with read-only transactions, depending on the concurrency control scheme used in the underlying database. Both H2 and HSQL support multi-version concurrency control that allows read-only transaction to not interfere with update transactions. Since only a single update transaction executes at a time, in secondary replicas, this approach guarantee serializable semantics.

Read-only transactions still need to wait until the secondary replica is up-to-date before starting. Our approach, of executing update transaction as a single batch of updates minimizes the execution time for these transactions, thus also minimizing waiting time.

We can infer, from the results presented in the introduction, that there is contention among multiple threads inside the database engine even when transactions do not conflict. We minimize this contention by reducing the number of transactions that execute in the same replica at the same time - by executing only a fraction of the read-only transactions in each secondary replica and by executing update transactions quickly in a single database operation.

3 Evaluation

In this section we evaluate MacroDB performance, comparing it with a single uncoordinated instance of the database engines (standalone versions), by measuring the throughput of each system. For this comparison we used the TPC-C benchmark, varying the number of clients and workloads. We also evaluated the performance impact of varying the number of secondary replicas of MacroDB. Additionally we also used the TPC-W benchmark.

Prototype Considerations. Our current MacroDB prototype is built in Java, and includes the necessary runtime system, as well as a custom JDBC driver. By using this simple approach, developers are able to integrate MacroDB into their applications by simply adding its library and modifying the URL used to connect to the database engine, without additional changes to the application code. The number of replicas and underlying database engines used are defined in the connecting URL. When the first client connects to the database, replicas are instantiated and the runtime system is started.

Setup. All experiments were performed on a Sun Fire X4600 M2 x86-64 server machine, with eight dual-core AMD Opteron Model 8220 processors and 32GByte of RAM, running Debian 5 (Lenny) operating system, H2 database engine version 1.3.169 and HSQL engine version 2.2.9, and OpenJDK version 1.6. All MacroDB configurations use a full database replication scheme.

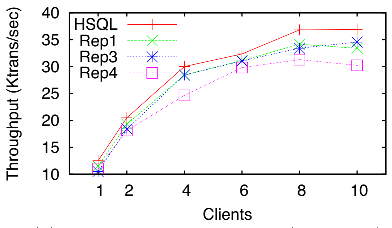
3.1 TPC-C

We ran the TPC-C benchmark using 4 different workloads, standard (8% reads and 92% writes), 50-50 (50% reads and 50% writes), 80-20 (80% reads and 20%writes) and 100-0 (100% reads), for 2 minutes, on a 4 gigabyte database. The number of clients varied between 1 and 10. The results presented are the average of 5 runs, performed on fresh database copies, disregarding the best and the worst results, and were obtained from the standalone uncoordinated versions of HSQL and H2, and MacroDB using HSQL (MacroHSQL) and H2 (MacroH2), configured with 1, 3 and 4 replicas (Rep1, Rep3 and Rep4, respectively).

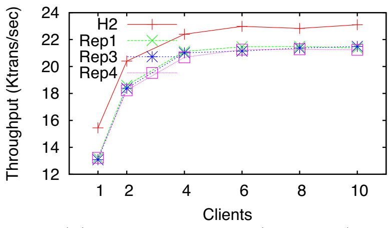
Standard Workload. Figures 4(a) and 4(b) present the results obtained running TPC-C with a standard workload. As expected, under update intensive workloads, our system is unable to benefit from the additional replicas for load balancing, since all updates must be executed on the same replica. Thus, the standalone versions of the database engines outperforms the MacroDB versions. These results also show an important aspect of MacroDB, its overhead. As put in evidence, our system is able to impose a fairly reduced overhead, compared to the standalone versions, ranging between 5% and 14%, even in update intensive workloads.

50-50 Workload. Figures 4(c) and 4(d) present the results obtained running TPC-C with a 50% read and 50% write workload. In moderate update workloads, MacroDB versions of both HSQL and H2 are able to achieve higher throughput than the standalone versions, offering up to 40% and 70% improvements over HSQL and H2 respectively. Although MacroDB is able to scale better than the standalone engines, its scalability is still limited by the nature of the workload. The secondary replicas are able to balance read-only transactions, but the moderate update nature of this workload still imposes great stress at the master replica, thus limiting scalability. This is put in evidence by the, almost negligible, performance difference when MacroDB is configured with 3 or 4 replicas, i.e., 2 or 3 secondary replicas. These results show that, even at considerable update rates, MacroDB is able to achieve a 70% improvement over standalone engines.

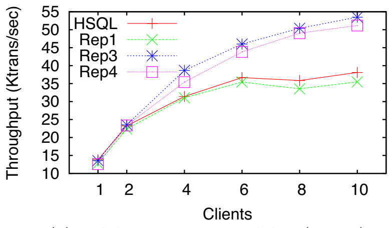
80-20 and 100-0 Workloads. The nature of read intensive workloads allows MacroDB to take full advantage of its replicated architecture. Both MacroDB versions achieve higher throughput than their standalone siblings, with an increased performance of 93% and 165%, and 166% and 176% performance increase over HSQL and H2, under an 80% (Figures 5(a) and 5(b)) and 100% (Figures 5(c) and 5(d)) read workloads, respectively.



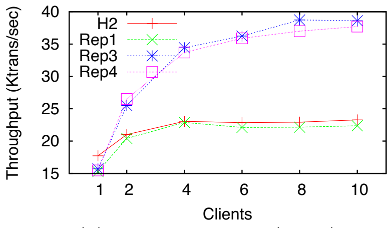
(a) HSQL vs. MacroHSQL (standard)



(b) H2 vs. MacroH2 (standard)



(c) HSQL vs. MacroHSQL (50-50)



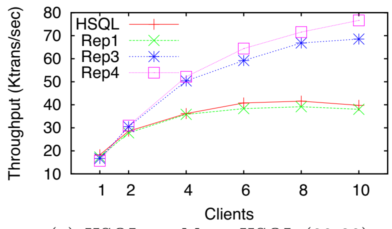
(d) H2 vs. MacroH2 (50-50)

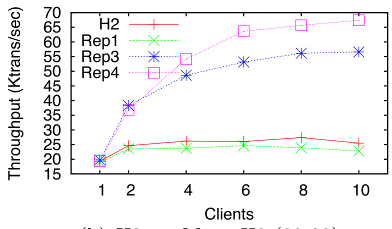
Fig. 4. TPC-C standard and 50-50 workload results

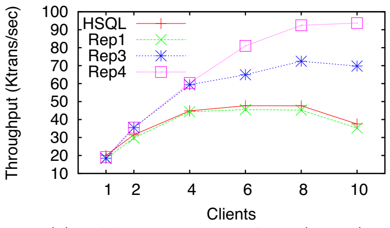
These results put into evidence the benefits of load balancing in reducing contention, since both MacroDB systems achieve higher performances with additional secondary replicas. The performance benefits of MacroDB is only limited by the number of replicas. As presented next, increasing the number of replicas allows MacroDB to further scale, achieving even higher performance figures. It also put into evidence that running 4 database replicas (1 primary and 3 secondary) is not sufficient to fully explore the processing power of our current system. Since current processors offer up to 20 threads per CPU chip [13], current engines considerably underutilize such platforms.

Also, the overhead measured by running a MacroDB with a single replica are consistent in all experiments, with a maximum value of 8%, independently of the nature of the workload, when compared to the standalone DBMS. It is also important to note the lack of scalability of the standalone versions of both database engines. These IMDBs scale fairly well up to 2 or 4 clients, but above that point performance improvements are not significant, in the majority of the experiments, thus showing that a major redesign is needed to improve DBMS performance on current multicore processors.

Additional Replicas. To further explore the computational power offered by our setup, we ran TPC-C, using the 80-20 and 100-0 workloads, on a MacroDB with 6 replicas (1 primary and 5 secondaries). The obtained results, presented in Figures 6 for MacroHSQL, show the benefits of increasing the number of replicas on a MacroDB. This increase allows MacroDB to offer performance improvements of up to 234% over standalone engines. These results also put into evidence how current chips are underutilized by current IMDB engines, since MacroDB was able to successfully improve performance, over standalone engines, even when running 6 engines on a single machine.


(a) HSQL vs. MacroHSQL (80-20)


(b) H2 vs. MacroH2 (80-20)


(c) HSQL vs. MacroHSQL (100-0)

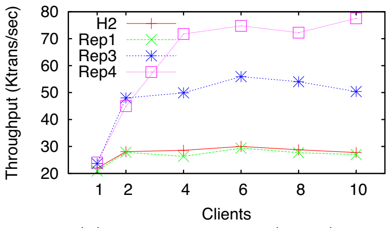
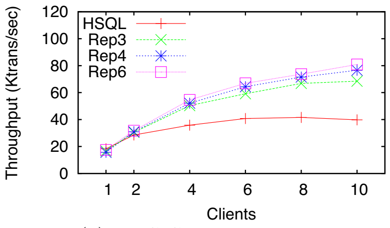

(d) H2 vs. MacroH2 (100-0)

Fig. 5. TPC-C 80-20 and 100-0 workload results


(a) TPC-C 80-20 Workload

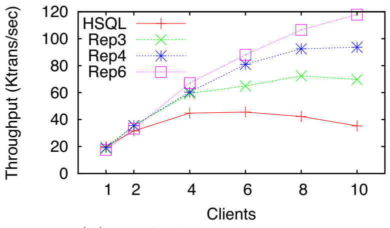

(b) TPC-C 100-0 Workload

Fig. 6. MacroHSQL with 6 replicas

Memory Usage. To measure the practicality of our proposal, we measured the memory overhead imposed by MacroDB, over the standalone database engines (Figure 7), varying the number of replicas. Contrarily to what may be expected, the memory used by MacroDB is not directly proportional to the number of replicas. This is due to the fact that replicas share immutable Java objects, such as Strings. The obtained results show that, a MacroDB configured with HSQL replicas, uses at most *2.5 times* more memory than the standalone engine, while a MacroDB configure with H2 replicas, uses at most *1.7 times* more memory than the standalone engine, when using a 4 replica configurations. This makes deploying MacroDB practical on single machine multicores, even with large numbers of replicas.

3.2 TPC-W

As an additional experiment, we compared the results obtained running TPC-W benchmark on a single, uncoordinated, H2 engine and a MacroDB using

	Replicas		
MacroDB	2	3	4
H2	1.53×	1.56×	1.76×
HSQL	1.64×	2.29×	2.56×

Fig. 7. Memory overhead

Workload	Throughput(WIPS)	
	H2	MacroDB(Rep3)
Browsing Mix	261.6	458.4
Shopping Mix	202	428.6

Fig. 8. TPC-W results

three H2 replicas (Rep3). The results obtained, presented in Figure 8, show the throughput, in web interactions per second (WIPS), obtained running TPC-W browsing and shopping mix, on the machine previously described with a database of 2 gigabytes, for 20 minutes and using 128 emulated browsers, with no thinking time. The performance improvements of MacroDB over the standalone version of H2 ranges from 75% to 112%, thus showing the benefits of our system.

4 Related Work

Several works have addressed the issues of database scalability on multicores. Most of these proposals focus on an engine redesign; on reuse of previous engine work; or on the addition of threads to automate specific procedures or to prefetch data [17,21,3,7,26,9]. These works are complementary to ours, since our focus is to allow existing engines to scale on current hardware without modification.

MacroDB, an example of a Macro-Component [15], follows the path that multicores should be seen as extremely low latency distributed systems [5,1,19,20], extended with shared memory. Thus, techniques previously developed for distributed systems are suitable for re-engineering and deploying on these platforms.

Many database replication studies have proposed solutions for improving service availability and performance [18,24,8,16]. Although complementary to our work, MacroDB builds on some of the techniques from these systems, applying them to multicore systems.

Multimed [19], an adaptation of Ganymed [18] for multicores, has previously explored database replication in single multicore machines. Although similar to our work, MacroDB presents differences that make it unique. First, unlike Multimed, we focus on in-memory databases, which presents different challenges for providing scalability, by not incurring in I/O overhead. Second, our solution aims at providing a single-copy serializable view of the database, instead of relying on weaker snapshot isolation semantics. Finally, by considering a multicore system as a distributed system *extended with shared memory*, we explore the shared memory for efficient communication between replicas and to expose data for efficient consistency management, load balancing and transaction routing.

5 Final Remarks

In this paper we presented MacroDB, a tool for scaling database systems on multicore platforms. Designed as a transparent middleware platform, it integrates replicas of existing unmodified database engines to offer increased concurrency and performance over standalone DBMS engines. MacroDB is transparent to applications, offering a single serializable view of the database, without need of rewriting the application code, while reducing contention and minimizing response times, by dividing and routing transactions according to their nature.

MacroDB is implemented using a custom JDBC driver and a self contained runtime, and can be used with any JDBC compatible database engine. Thus, performance improvements are obtained without modification to the database engine or the application. It is also easy to configure, allowing database engines and configurations to be specified by the JDBC driver URL.

Our evaluation shows that MacroDB offers 40% to 180% performance improvements over standalone in-memory DBMS, for various TPC-C workloads. Under update intensive workloads (92% update transactions), MacroDB has a reduced overhead of less than 14% when compared to standalone database engines. For TPC-W workloads, MacroDB is able to achieve improvements of up to 112%, over standalone in-memory DBMS.

The memory used by the database replicas is not directly proportional to the number of replicas, as replicas share immutable Java objects, thus making MacroDB practical even with large numbers of replicas.

References

1. Baumann, A., Barham, P., Dagand, P.E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: a new os architecture for scalable multicore systems. In: Proc. SOSP 2009 (2009)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Longman (1986)
3. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core cpus. In: Proc. SIGMOD 2011 (2011)
4. Camargos, L., Pedone, F., Wieloch, M.: Sprint: a middleware for high-performance transaction processing. In: Proc. EuroSys 2007 (2007)
5. Cecchet, E., Candea, G., Ailamaki, A.: Middleware-based database replication: the gaps between theory and practice. In: Proc. SIGMOD 2008 (2008)
6. Chekuri, C., Hasan, W., Motwani, R.: Scheduling problems in parallel query optimization. In: Proc. PODS 1995 (1995)
7. Cieslewicz, J., Ross, K.A., Satsumi, K., Ye, Y.: Automatic contention detection and amelioration for data-intensive operations. In: Proc. SIGMOD 2010 (2010)
8. Elnikety, S., Dropsho, S., Pedone, F.: Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In: Proc. EuroSys 2006 (2006)
9. Giannikis, G., Alonso, G., Kossmann, D.: Shareddb: killing one thousand queries with one stone. In: Proc. VLDB 2012 (2012)
10. Hardavellas, N., Pandis, I., Johnson, R., Mancheril, N., Ailamaki, A., Falsafi, B.: Database servers on chip multiprocessors: Limitations and opportunities. In: Proc. CIDR 2007 (2007)
11. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: Oltp through the looking glass, and what we found there. In: Proc. SIGMOD 2008 (2008)

12. Helal, A.A., Bhargava, B.K., Heddaya, A.A.: Replication Techniques in Distributed Systems. Kluwer Academic Publishers (1996)
13. Intel: Xenon processor e7 family (2012), `http://www.intel.com/`
14. Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E.P.C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D.J.: H-store: a high-performance, distributed main memory transaction processing system. In: Proc. VLDB 2008 (2008)
15. Mariano, P., Soares, J., Preguiça, N.: Replicated software components for improved performance. In: Proc. InForum 2010 (2010)
16. Mishima, T., Nakamura, H.: Pangea: an eager database replication middleware guaranteeing snapshot isolation without modification of database servers. In: Proc. VLDB 2009 (August 2009)
17. Papadopoulos, K., Stavrou, K., Trancoso, P.: Helpercoredb: Exploiting multicore technology for databases. In: Proc. PACT 2007 (2007)
18. Plattner, C., Alonso, G.: Ganymed: scalable replication for transactional web applications. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 155–174. Springer, Heidelberg (2004)
19. Salomie, T.I., Subasu, I.E., Giceva, J., Alonso, G.: Database engines on multicores, why parallelize when you can distribute? In: Proc. EuroSys 2011 (2011)
20. Song, X., Chen, H., Chen, R., Wang, Y., Zang, B.: A case for scaling applications to many-core with os clustering. In: Proc. EuroSys 2011 (2011)
21. Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., Kossmann, D.: Predictable performance for unpredictable workloads. In: Proc. VLDB 2009 (2009)
22. Vandiver, B., Balakrishnan, H., Liskov, B., Madden, S.: Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In: Proc. SOSP 2007 (2007)
23. Wiesmann, M., Schiper, A., Pedone, F., Kemme, B., Alonso, G.: Database replication techniques: A three parameter classification. In: Proc. SRDS 2000 (2000)
24. Wiesmann, M., Schiper, A.: Comparison of database replication techniques based on total order broadcast. IEEE Trans. on Knowledge and Data Engineering 17 (2005)
25. Ye, Y., Ross, K.A., Vesdapunt, N.: Scalable aggregation on multicore processors. In: Proc. DaMoN 2011 (2011)
26. Zhou, J., Cieslewicz, J., Ross, K.A., Shah, M.: Improving database performance on simultaneous multithreading processors. In: Proc. VLDB 2005 (2005)

Towards a Scalable Microkernel Personality for Multicore Processors

Jilong Kuang, Daniel G. Waddington, and Chen Tian

Computer Science Lab, Samsung Research America - Silicon Valley
{jilong.kuang,d.waddington,chen.tian}@samsung.com

Abstract. With a steady trend from singe-core to multicore processors, scalability has become a significant design issue for the Operating Systems (OS), as many critical OS functions must be re-designed in order to achieve scalable performance. While numerous efforts have been made to improve scalability of monolithic OS kernels, comparatively little work has been done for microkernels.

In this paper, we begin by studying the scalability of Fiasco.OC, a state-of-the-art microkernel implementation. We then present OmniRE, a new personality for the Fiasco.OC microkernel that is aimed at being multicore scalable. Compared to L4Re (the vanilla "off-the-shelf" Fiasco.OC personality), OmniRE aims to eliminate contention by decentralizing resource management, scheduling, and kernel access. The design also aims to minimize inter-process communication (IPC) across CPUs by localizing resource functionality such as page-fault handling. We conduct experiments to compare OmniRE against L4Re as well as Linux on a 48-core AMD server and a 6-core Intel workstation. Our results indicate that OmniRE provides better scalability than L4Re and can in fact exceed absolute performance of Linux in memory page management at higher core counts.

1 Introduction

Compared to monolithic kernels such as *Linux* and *Windows*, microkernel architectures have unique advantages in simplicity, security, robustness and customization. To date, research has predominantly focused on improving microkernel uniprocessor performance and enriching the feature set. These efforts have lead to third generation microkernels, such as *Fiasco.OC*[7], *NOVA*[15], and *OKL4* [6]. As microkernel technology has matured, it has begun to get traction in both mobile platforms (e.g., L4Android [2], OKL4 [6]) as well as embedded and safety-critical systems. More recently microkernels have been explored as a more effective platform for HPC-like applications [14] [17].

The move towards multicore processors has driven many OS communities to reexamine fundamental internal design decisions in order to improve multicore scalability. For example, in the Linux kernel, little use of coarse-grained locks remained by version 2.6; most notably code locking had been converted to data locking, advanced "lockless" data structures had been applied (e.g., Read-Copy-Update [12]) and schedulers had been re-implemented to support per-core scheduling queues.

Nevertheless, as of now, not much work has been done for microkernels. Although a number of research OSes, including Barrelfish [4], Helios [13], FOS [18], and Corey [20], have taken an approach based on the concept of *multi-kernels* to improve OS scalability on multicore processors, they have shied away from shared-memory kernels due to the need to partition resources. However, shared-memory kernels provide obvious advantage with respect to integration and resource sharing across cores. It is for this reason that we have chosen a shared-memory microkernel, Fiasco.OC [7,10], as the basis of our work.

In this paper, we propose OmniRE, a scalable runtime personality (user-land) for the Fiasco.OC microkernel. OmniRE is a direct replacement for the L4Re personality from the Fiasco.OC group [7]. OmniRE incorporates a hierarchical resource management design that eliminates central points of contention and provides sufficient flexibility to allow tailoring of the OS to underlying processor, memory and IO topologies. Compared to L4Re, OmniRE offers the following differentiation: 1) It eliminates contention on resource management by decentralization of memory management, scheduling, and access to kernel services. 2) It minimizes cross-core IPC on multicore architectures by forcing resource management, resource access and page-fault handling to be localized to the same core whenever possible.

The principal contributions and structure of the paper are as follows:

– We study and investigate the scalability potential of the Fiasco.OC microkernel and examine performance scaling for the L4Re personality (Sections 2 and 3).
– We present a design and implementation of OmniRE, a scalable multicore user-land based on the Fiasco.OC microkernel that uses a hierarchical arrangement of multi-threaded services and decentralized resource management to successfully achieve scalability (Section 4).
– We conduct experiments to empirically compare Fiasco.OC/OmniRE against Fiasco.OC/L4Re as well as Linux 3.0 running on a 48-core AMD server and a 6-core Intel workstation (Section 5).

2 Fiasco.OC and L4Re Overview

As a representative third generation microkernel, Fiasco.OC and its user-land runtime environment (L4Re), have become increasingly popular due to the availability of advanced features (e.g., capabilities, multicore support, multi-ISA support, Linux para-virtualization) and general maturity.

The basic components of an L4Re-based system are:

– *Microkernel* - provides primitives to execute programs in tasks, to enforce isolation among them, and to provide means of secure communication in order to let them cooperate. As the kernel is the most privileged, security-critical, software component in the system, it provides only a minimal set of mechanisms that are necessary to support applications.
– *L4 Runtime Environment* - L4Re comprises low-level software components that interface directly with the microkernel. The root pager *Sigma0* and the root task *Moe* are the most basic components of the runtime environment. Other services (e.g., for device enumeration) use interfaces provided by them.

– *Applications* - run on top of the system and use services provided by the runtime environment or other applications. Applications include conventional user-local programs that face the end-user, as well as virtual machine monitors, device drivers and other system services.

3 Study of Off-the-Shelf L4Re Scalability Characteristics

Although the Fiasco.OC kernel is explicitly designed to support multicore processors [9], there are scalability limitations in the current L4Re platform. To investigate the L4Re scalability characteristics, we have developed micro-benchmark applications to test memory management and thread creation (corresponding to kernel object creation) as key indicators of system scaling. The test platform is a 48-core AMD Magnycours server (see Section 6 for more details).

Figure 1 shows the memory management results, where each iteration performs a single page (4K) allocation (via std::malloc API), writes to each integer element in the page and then frees the memory (via std::free). The results show that memory management (allocation, physical-to-virtual mapping and paging) on the L4Re-based platform degrades significantly from only two cores. Figure 2 shows the thread creation results, where each iteration creates a new child thread which executes an empty function. The parent thread (per-core worker) waits for a child thread to complete before starting the next thread. Similarly, it is evident that L4Re does not scale well with respect to thread creation.

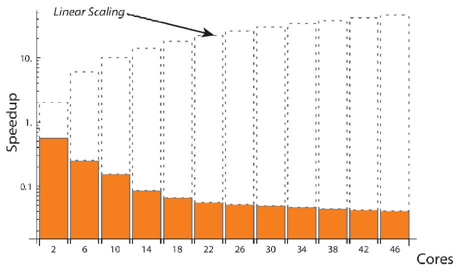


Fig. 1. L4Re Memory Allocation Scaling

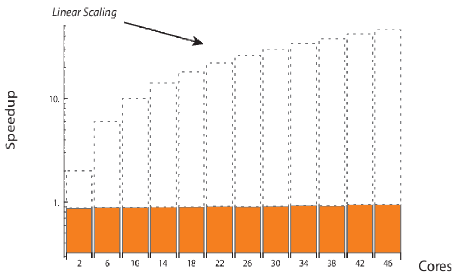


Fig. 2. L4Re Thread Creation Scaling

We believe that the current scalability limitations in L4Re-based system are predominantly a result of centralized resource management in the L4Re personality.

4 OmniRE Design

OmniRE is a new personality for the Fiasco.OC microkernel [7,10]. OmniRE directly replaces the L4Re personality [3]. Key design elements are:

– Decentralized management of memory (physical and virtual), thread/process, IO and IRQ resources.

– Minimization of cross-core IPC by localization of page-fault handling and service access.
– Explicit resource management and quota control for all resources in the system. Secure access control to resources realized through the microkernel's capability feature.

4.1 OmniRE Detailed Design

OmniRE is responsible for managing all of the resources in the system. This includes controlling allocation of kernel objects (e.g., threads, semaphores, IPC gates) as well as resources directly used by the application (e.g., memory, I/O ports). The fundamental basis of OmniRE's design is that resource management (e.g., allocation, freeing), resource access (e.g., invocation on an IPC-gate), and page-fault handling should all be localized to the same processor core whenever possible. Permissions and quotas are arranged hierarchically and managed locally. Reallocation and resource balancing is performed at a coarser granularity. The rationale for core-localization is to both minimize cross-core communications and decentralize resource management (reducing contention). Cross-core IPC is approximately ten times slower (see Section 6.1) than same core IPC. Cross-core data sharing leads to unpredictable levels of degradation due to serialization on locks and underlying side-effects such as false-sharing.

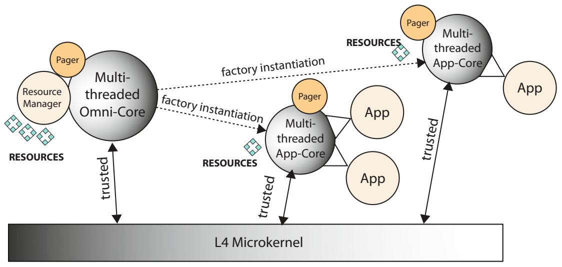


Fig. 3. OmniRE High-level Architecture

Our design includes two key elements: 1) *Omni-Core* and 2) a set of *App-Cores*. Omni-Core forms the root of the resource management tree; it manages the highest level of resource partitioning. App-Cores are localized delegates that are instantiated by Omni-Core (see Figure 3). Each App-Core is isolated in a separate process. They are assigned a coarse-grained allocation of resources from Omni-Core, which is dynamically load-balanced across App-Cores as needed. The detailed architecture is given in Figure 4.

App-Cores are the direct representative of the runtime environment for applications. They are instantiated by Omni-Core (either at boot time or dynamically) and indirectly used to load applications. The logical resource partition (i.e., set of quotas) for an application is managed by an *App-Env* (Application environment) that is instantiated inside the App-Core. Application requests to the App-Env are associated with *Service Points* that can be used to further partition the application's resources on a per-thread basis.

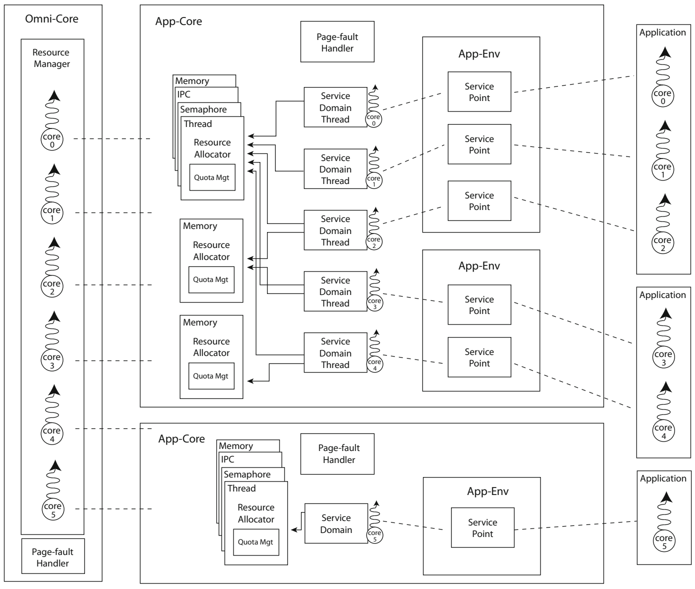


Fig. 4. Detailed OmniRE Architecture

Service Points are also useful (as a level of indirection) for transferring resources as a thread migrates between cores or applications.

Resource management in each App-Core is decentralized so that resources that have non-uniform access properties (e.g., memory, CPUs) can be separated out. To facilitate this, each App-Core maintains a number of *Service Domain Threads* that redirect resource requests to different *Resource Allocators*. Resource Allocators exist for each type of resource in the system (thread, process, memory, IPC gate, semaphore, IRQ object). They manage a strict quota of resources, defined by a secure system specification, that is assigned to the App-Core by Omni-Core during start-up. A key use of Resource Allocators is to support NUMA memory allocation across multiple memory controllers.

Essential to the design is that both Omni-Core and the App-Cores are part of the Trusted Computing Base (TCB). This means that the App-Core is trusted to, 1) prevent violation of quotas agreed with Omni-Core, and 2) not abuse its privilege to directly invoke the kernel and request kernel-level resources (e.g., threads). It is the responsibility of the App-Core to manage the application's access to resources according to defined quotas. Applications do not have direct access to either the kernel or to the Omni-Core process. Doing so would break the security model and open up potential for QoS interference between applications.

5 Case Study: Physical Memory Management

This section addresses in more detail the hierarchical resource management scheme in the context of physical memory management, which is one of the most basic functions any OS must provide. Different from monolithic kernels (e.g., Linux), where all page-level memory requests are ultimately handled in kernel mode, microkernels such as Fiasco.OC have two Physical Memory Allocators (PMA), one in the kernel and another in user-land.

Kernel functions can directly allocate physical memory through the kernel PMA. However, by default, the amount of memory managed by the kernel is less than 10% of the total. The rest of the memory is managed by a user-level PMA, which allocates memory to applications. We therefore only focus on the user-level PMA design.

Existing PMA Design. Figure 5 shows a typical sequence of operations for allocating a page in Fiasco.OC. First, a process allocates a stack variable or heap data through a virtual memory allocator such as *malloc* (step 1). When the virtual address is touched a page-fault exception is raised by the processor, which transfers execution to the kernel's page-fault handler (step 2). The handler forwards the request to a special user-level application, *Sigma0* (also known as the *pager*), which by default handles all page-faults in the system (step 3). In a multicore environment, it is possible to have multiple page-faults taking place on different cores simultaneously. In this case the kernel page-fault handler serializes them and forwards the request to *Sigma0* one by one. When *Sigma0* receives a page-fault notification IPC call, it requests a physical page from its PMA (steps 4 and 5). The result is then sent back to the kernel (step 6), which then populates the page table for the faulting process. After that, the kernel page-fault handler switches back to the faulting instruction so the application process can continue (step 7).

While the entire process is transparent to applications, it involves four context switches (steps 2, 3, 6 and 7), two of them being IPC calls (step 3 and 6). This is the cost that a microkernel design must pay for security and reliability.

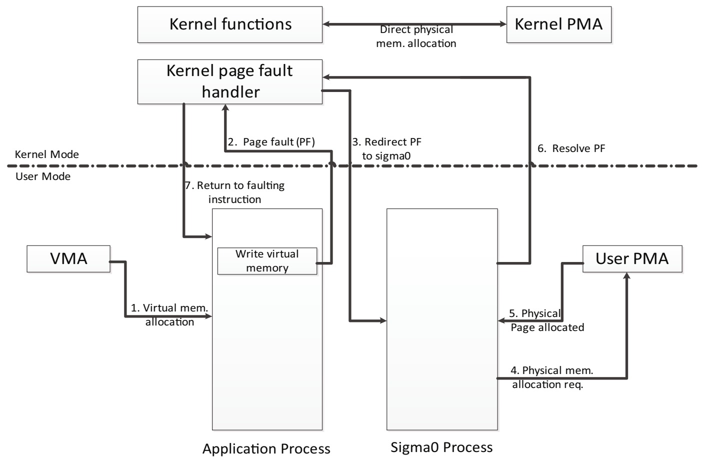

Kernel functions
Direct physical mem. allocation
Kernel PMA
Kernel page fault handler
2. Page fault (PF)
3. Redirect PF to sigma0
6. Resolve PF
Kernel Mode
User Mode
7. Return to faulting instruction
VMA
Write virtual memory
User PMA
1. Virtual mem. allocation
5. Physical Page allocated
4. Physical mem. allocation req.
Application Process
Sigma0 Process

Fig. 5. PMA Process in Fiasco.OC Kernel

Scalable PMA Design in OmniRE. The user-level PMA can be arranged globally, for each NUMA zone, for each core or even for each process. In order to minimize cross-core IPC we chose a per-core PMA and per-core pager design (see Figure 6).

The benefit of this design is that it addresses two prominent scalability inhibitors. First, it reduces pager contention by distributing page handling across cores. Second, it eliminates all cross-core IPCs of page allocations, as each core now has a pager and every process can use same-core IPC to communicate with the local pager. All page requests can be handled on the same core rather than a different core - making the behavior comparable to that of a monolithic kernel design. The result is that general performance and scalability of page allocation is largely improved. Furthermore, CPU utilization can also be maximized because no local physical memory requests interrupt applications executing on a remote cores.

The implementation of the per-core PMA scheme in OmniRE is to first partition physical memory and then construct per-core pagers as shown in Figure 7. When OmniRE is booted, it first loads *Sigma0* as the default pager and then hands off the paging for applications to Omni-Core. It is necessary to load *Sigma0* as this provides paging for the kernel and Omni-Core itself. As illustrated in Figure 7, Omni-Core first obtains all physical memory that is made available by the kernel. Management of this memory is then delegated to App-Cores which are localized with the applications.

To "link" the associated App-Core to each application, the kernel Process Control Block (PCB) pager field is modified. This effectively enables per-core paging (see Figure 6).

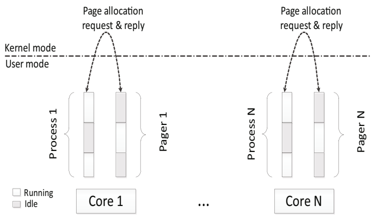


Fig. 6. Creating One Pager For Each Core Improves Scalability

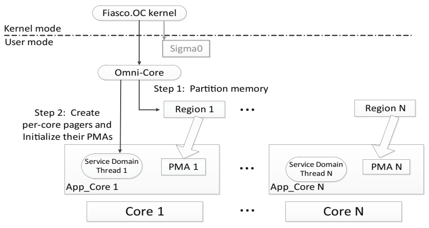


Fig. 7. Initialization of Per-core Paging and PMA

6 Experimental Results

The current OmniRE prototype is implemented on a 32-bit x86 platform. Performance and scalability results are collected from benchmark executions on both a 48-core (4x12) AMD Opteron-based server platform and a single 6-core Intel Xeon workstation. Timing measurements are taken using the on-chip time stamp counters. Measurements for performance and scalability are taken from a series of micro-benchmark applications. All benchmarks are based on replicated processes (pinned to individual cores) to remove the effects of contention on a shared page table by threads in a single process. Table 1 gives additional detail of the two test platforms.

6.1 Fiasco.OC IPC Scalability

In this section, we provides results for the scalability of IPC. We chose to include this data because of the fundamental importance of IPC performance and its broad effect

Table 1. Test Platform Specification

L4Re	Fiasco.OC Revision 36. x86 32-bit build.
Linux	Ubuntu Linux kernel 3.0.0-16 server stock build (x86_64).
Compiler	GNU GCC 4.4.6 with optimizations on (O2).
AMD Magnycours Server (Dell R815)	CPU: 4x AMD Opteron 6174 2.2MHz CPU. Each multi-chip module package (processor) combines 2 dies of 6 cores. DVFS is turned off.
	L1 cache (64KB data per core, 64KB instruction per core). L2 cache (512KB per core). L3 cache (12MB per socket).
	32GB DRAM; integrated DDR3 with support up to 42.7 GB/s memory bandwidth per CPU.
	Four x16 Hypertransport links @ up to 6.4GT/s per link.
Intel Xeon Workstation	CPU: 1x Intel W3670 3.2GHz 6-core. DVFS is turned off. HT is turned off.
	L1 Cache (64KB data per core, 64KB instruction per core). L2 cache (256KB per core). L3 12Mb shared.
	4GB DRAM on single memory controller.

on scaling on the OmniRE personality. In this benchmark processes are arranged in pairs either on a single core (same-core) or on adjacent separate cores (cross-core). For cross-core, pairs are built up in clusters on the same die. Each pair exchanged 1 million IPC messages in a ping-pong fashion. The implementation has identical semantics to the L4Re functions `l4_ipc_call` and `l4_ipc_reply_and_wait`. Total time to complete the exchange is measured and the mean taken across all cores.

Figure 8 shows the IPC scaling results. Same-core performance is approximately one order of magnitude faster than cross-core. On the AMD platform, mean (per-pair) performance actually improved by 16% over an increase of 36 cores (6-42). We believe that this increase in performance is an artifact of the hardware architecture, specifically the size and design of the Opteron's cache. A similar trend is observed for the memory management benchmark on the AMD platform, which we will describe later. On the Intel platform, the data shows negligible performance change for both cross-core and

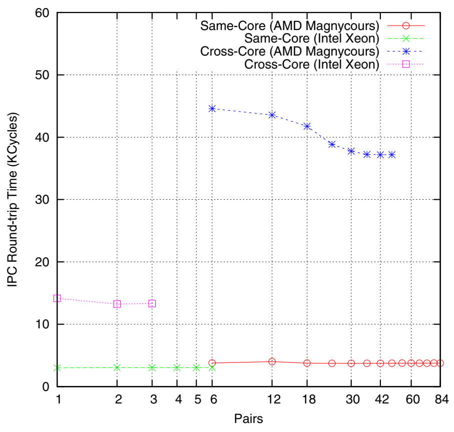


Fig. 8. Fiasco.OC IPC Performance

same-core IPC. In summary, Figure 8 validates our design in the following two aspects: 1) Minimizing cross-core IPC on multicore architecture whenever possible has significant performance gain. 2) Decentralizing resource management is preferable as both same-core and cross-core IPC are scalable.

6.2 Memory Page Management with L4Re

In this Section, we measure the basic memory page allocation, mapping and freeing in both OmniRE and L4Re. Each benchmark is executed as a single process running on a dedicated core. The benchmark performs 100 iterations of a memory allocation sequence. Each task first allocates a batch of 100 pages (1 page per allocation), then touches the first byte of each page, which invokes the physical memory allocation, and finally frees all 100 allocated pages. A *coordinator* process is used to synchronize the launch of benchmarks after loading to ensure as close as possible start-times. Similarly, L4Re benchmark uses a sequence of `alloc(..)`, `attach(..)` followed by `detach(..)` and `free(..)` during each iteration for fair comparison.

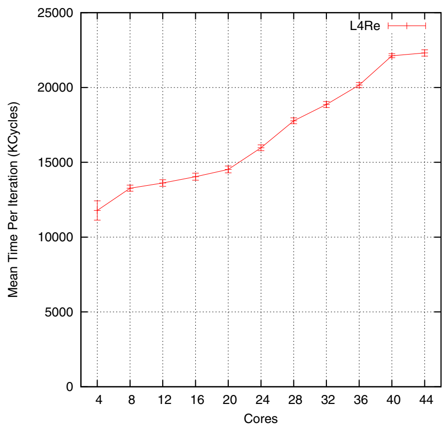


Fig. 9. Scalability of L4Re Memory Page Management (AMD Platform)

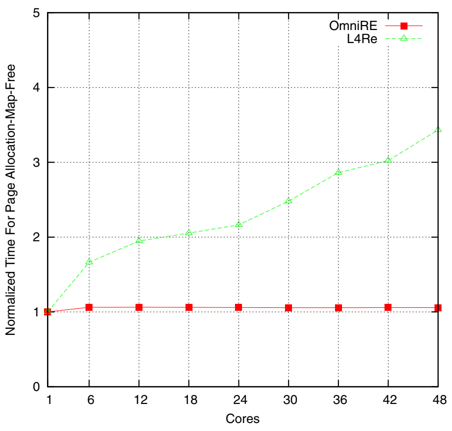


Fig. 10. Normalized Scalability Comparison with L4Re (AMD Platform)

Figure 9 shows the absolute mean time per iteration for L4Re as the number of cores increases. From this figure we can clearly see that degradation of L4Re for the equivalent benchmark using the `alloc`, `attach`, `free`, `detach` APIs is measured at 89% over 40 cores (2.2% degradation per core). We believe that the cause of the degradation in L4Re is due to contention of the page-fault handler (*Sigma0*) which is, in this implementation, single-threaded.

Figure 10 shows a normalized comparison of OmniRE and L4RE memory scaling. As the number of cores increases up to 48, the normalized time of OmniRE remains effectively flat; the degradation is only about 5% when 48 cores are used. In the case of L4Re, however, the performance degrades over 240%. Due to the multi-threaded page-fault handler in OmniRE, this resource contention is largely eliminated.

6.3 Memory Page Management Compared with Linux

In this section, we present experimental comparison data for Linux. The benchmark for OmniRE is the same as described in the previous section, except that we allocate a total number of 100K pages in batches of 600 pages (2.4MB). Each allocation is still one 4K page. The Linux implementations of this benchmark use `mmap` and `malloc` based APIs. The `mmap` API provides a means to *eagerly* map physical pages so that a page-fault is not generated. The OmniRE benchmarks also use eager mapping for fair comparison.

The results given in Figure 11 show that for single-page allocations on the AMD platform, OmniRE's page management is able to scale almost linearly whilst Linux degrades exponentially going from 9000 cycles on a single core to 1.1M on 48 cores (note the logarithmic y axis scale). Figure 12 shows the single-page allocation data for the Intel Xeon platform using 6 cores. The data shows OmniRE degradation of less than 0.8% over 6 cores and degradation of more than 35.5% for Linux. However, at this low core count absolute performance of Linux is higher than that of OmniRE, as the OmniRE's scalability advantage has yet to offset its inherent microkernel limitations (e.g., doubled IPC communications between kernel and user-land compared to a monolithic kernel). In fact, Figure 11 clearly shows that only when core count exceeds 9 does OmniRE outperform Linux.

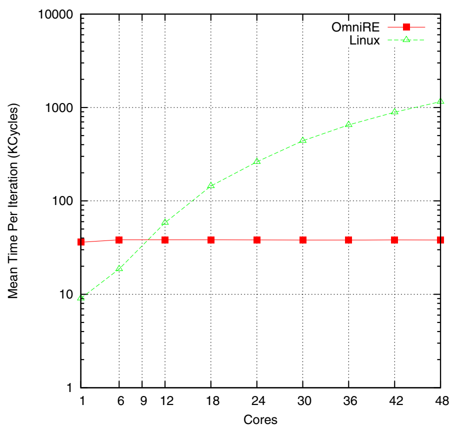


Fig. 11. Linux vs. OmniRE Comparison of Memory Page Management Scalability (AMD Platform)

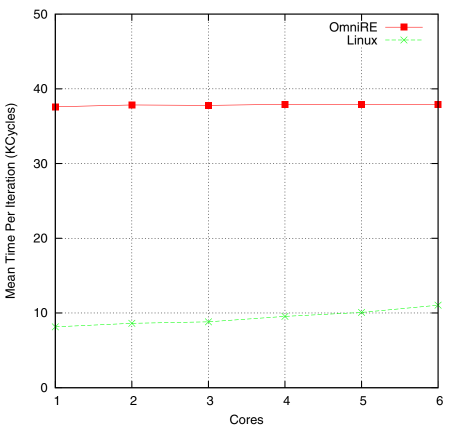


Fig. 12. Linux vs. OmniRE Comparison of Memory Page Management Scalability (Intel Platform)

The poor scalability in Linux across 48 cores demonstrates that some critical functions of today's Linux OS do not scale well for even a few cores. Both Figure 11 and Figure 12 show significant degradation of memory management as the number of cores increases. Additional in-house experiments indicate that the cause of this serialization likely relates to the locking strategy on the LRU (Least-Recently Used) page replacement list. It is worth noting that although our Linux data is congruent with data collected by other projects (FOS [19,5], Corey [20] and Barrelfish [4]), we speculate that any difference in the h/w platform (e.g., BIOS, memory) and/or the kernel build may affect the performance. The reader should not focus on this work as a criticism of Linux but simply as a comparison point.

7 Related Work

OmniRE is based on the Fiasco.OC microkernel [7] developed by the Operating Systems group of TU Dresden. As part of the Fiasco.OC work the team developed the L4Re user-level personality. However, although the Fiasco.OC kernel has been explicitly designed to support multicore processors [9] there are scalability limitations in the current L4Re mainly relating to the pager *Sigma0*, the IO server and cross-core IPC. Furthermore, L4Re does not provide a secure resource management model but allows applications to directly interact with the kernel and pager, which can potentially result in QoS crosstalk and denial-of-service issues.

The resource management philosophy of OmniRE is inspired by work done by Feske et al. in their Bastei Architecture [8]. The basic premise of their approach is to explicitly manage all resources that are required by both applications and sub-systems. Resources are securely managed through a parent-child trust relationship. The original concepts developed in this work have now been carried through into the commercially supported Genode OS Framework [1]. However, multicore scalability is not currently a primary concern for the Genode Labs group. The current design incurs many cross-core IPC invocations and scalability is limited by single-threaded contention points. OmniRE addresses these concerns at the implementation level and also introduces a different trust model from the Bastei architecture. Also worth mentioning is the resource container work done in the K42 OS [16] that also developed approaches to resource management and donation as a means to alleviate denial-of-service attacks.

The Barrelfish OS [4] developed by Microsoft Research and ETH Zurich Systems Group was started in 2008. Barrelfish is a multi-kernel design that uses the notion of User-level RPC (URPC) to facilitate high-performance IPC exchange via shared-memory region without transitions through the kernel. OmniRE uses a URPC-like approach for communications between the App-Cores and Omni-Core. As with Fiasco.OC, Barrelfish uses a capability model to perform access control to different memory regions. The current implementation is based on 64-bit x86. Data given in [4] shows that Barrelfish degradation for the `unmap` memory operation is approximately 80% at 32 cores on an 8x4 (x8 quad Opteron 8350) AMD platform.

Another prominent OS for multicore processors that is based on a multi-kernel design is the Factored Operating System (FOS) work from MIT [18] [19]. This work is driven by their work on scalable multicore MIMD processors and focuses on the use of spatial distribution to scale OS services including physical resource management, file systems, network protocols and applications. Each system service is “factored” into a collection of Internet-inspired servers that communicate via user-level message passing. The FOS solution is based on a proprietary microkernel and is currently implemented on 64-bit x86. Results collected from a 48-core AMD platform (quad Opteron 6168) reported in [19] showed that over 20 “clients”, which we assume correlates to individual processes, FOS’s page allocator performance degraded by 60%.

Finally, Tessellation OS [11] from UC Berkeley is a more recent effort to develop a multicore OS that integrates both space and time partitioning to share resources across system services and applications. The Tessellation OS design uses a hierarchical (two-level) scheduling scheme to manage global and local (partition) resource management. As with OmniRE, Tessellation also aims to provide QoS enforcement and minimization of QoS crosstalk. This OS is still in its early stages and as yet no performance and scaling results have been published.

8 Conclusions

In this paper we presented OmniRE, a new OS design based on the shared-memory Fiasco.OC microkernel that uses multi-threaded (per-core) system services and resource management delegation to eliminate points of contention and thus promote scalability. We have shown that the Fiasco.OC kernel's use of per-core data structures and internal separation, coupled with the OmniRE personality, provide a complete scalable solution. We implemented and evaluated OmniRE on both AMD and Intel platforms against L4Re and Linux 3.0. Our experimental data shows that OmniRE is able to successfully remove contention on memory management and kernel object management across 48 cores, which substantially outperforms Fiasco.OC and, at higher core counts, exceeds the scaling performance of Linux.

References

1. Genode, `http://www.genode.org`
2. L4android, `http://l4android.org/`
3. L4re, `http://os.inf.tu-dresden.de/l4re/`
4. Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems. In: Proc. of SOSP 2009 (2009)
5. Boyd-Wickizer, S., Clements, A.T., Mao, Y., Pesterev, A., Kaashoek, M.F., Morris, R., Zeldovich, N.: An analysis of linux scalability to many cores. In: Proc. of OSDI 2010 (2010)
6. G. D. Broadband: Okl4 microkernel, `http://www.ok-labs.com`
7. T. U. Dresden: Fiasco.oc microkernel, `http://os.inf.tu-dresden.de/fiasco/`
8. Feske, N., Helmuth, C.: Design of the Bastei OS Architecture. Technical report, Technische Universität Dresden (December 2006)
9. Hohmuth, M., Peter, M.: Helping in a multiprocessor environment (2001)
10. Liedtke, J.: On micro-kernel construction. SIGOPS Oper. Syst. Rev. 29, 237–250 (1995)
11. Liu, R., Klues, K., Bird, S., Hofmeyr, S., Asanović, K., Kubiatowicz, J.: Tessellation: space-time partitioning in a manycore client OS. In: Proc. of HotPar 2009 (2009)
12. Mckenney, P.E.: Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels. PhD thesis (2004)
13. Nightingale, E.B., Hawblitzel, C., Hodson, O., Hunt, G., Mcilroy, R.: Helios: Heterogeneous multiprocessing with satellite kernels. In: Proc. of SOSP 2009 (2009)
14. Schubert, L., Wesner, S., Kipp, A.: Reputing microkernels. In: Proc. of UK e-Science All Hands Meeting 2009 (2009)
15. Steinberg, U., Kauer, B.: Nova: a microhypervisor-based secure virtualization architecture. In: Proc. of EuroSys 2010 (2010)

16. Tam, A., Tam, D.K.-F., Azimi, R.: Implementing resource containers in k42 (2003)
17. Thibault, S., Deegan, T.: Improving performance by embedding hpc applications in lightweight xen domains. In: Proc. of HPCVIRT 2008 (2008)
18. Wentzlaff, D., Agarwal, A.: Factored operating systems (fos): the case for a scalable operating system for multicores. SIGOPS Oper. Syst. Rev. 43(2), 76–85 (2009)
19. Wentzlaff, D., Gruenwald III., C., Belay, A., Kasture, H., Youseff, L., Miller, J.E., Modzelewski, K., Agarwal, A.: Fleets: Scalable services in a factored operating system. Network (2011)
20. Wickizer, S.B., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., Zhang, Z.: In: Proc. of OSDI 2008 (2008)

An Implementation of the Codelet Model

Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao

University of Delaware, 140 Evans Hall, Newark, DE, USA
{jodasue,szuckerm,ggao}@capsl.udel.edu,
http://www.capsl.udel.edu

Abstract. Chip architectures are shifting from few, faster, functionally heavy cores to abundant, slower, simpler cores to address pressing physical limitations such as energy consumption and heat expenditure. As architectural trends continue to fluctuate, we propose a novel program execution model, the Codelet model, which is designed for new systems tasked with efficiently managing varying resources. The Codelet model is a fine-grained dataflow inspired model extended to address the cumbersome resources available in new architectures. In the following, we define the Codelet execution model as well as provide an implementation named DARTS. Utilizing DARTS and two predominant kernels, matrix multiplication and the Graph 500's breadth first search, we explore the validity of fine-grain execution as a promising and viable execution model for future and current architectures. We show that our runtime is on par or performs better than AMD's highly-optimized parallel library for matrix multication, outperforming it on average by 1.40× with a speedup up to 4×. Our implementation of the parallel BFS outperforms Graph 500's reference implementation (with or without dynamic scheduling) on average by 1.50× with a speed up of up to 2.38×.

Keywords: Execution model, runtime system, manycore, multicore.

1 Introduction

While the advent of many-core chips for mainstream computing is still yet to come, many-core *compute nodes* have become common in new supercomputers. A typical compute node may have 32 to 64 threads (or more), spread across several sockets. In addition, non-uniform memory access (NUMA) has become the new standard for shared-memory nodes. As thread counts increase, memory and even compute resources (such as FPUs) per core are becoming more scarce as seen in the IBM Cyclops-64 [8]. On-chip and off-chip bandwidth must also be seen as scarce resources requiring intelligent allocation among computing units. Furthermore, due to diminishing feature sizes, reducing power consumption has become a predominant obstacle forcing chip manufacturers to simplify the design of individual cores, removing power-hungry branch predictors and cache prefetchers.

The increase in available parallelism found in shared-memory nodes has lead to hard-to-exploit program execution models (PXMs). These models are still based on the sequential Von Neumann model. The semantics of traditional threads makes it extremely difficult to guarantee correct execution, and race conditions are the dreaded companion of any parallel programmer [17]. However, there are PXMs which emphasize properties such as the isolation of execution and the explicit declaration of producer-consumer relations like the dataflow program execution models [9].

This paper evaluates an implementation of the codelet model [24], a fine-grain PXM inspired by dataflow. We implemented the model using a runtime system, DARTS. We evaluate its usefulness through two case studies comparing DARTS against OpenMP on square matrix multiplication and the Graph500's breadth first search benchmark.

Section 2 provides the necessary background to understand how the codelet model works, and how DARTS implements it. Section 3 presents our two case studies, and describe in details how each problem was decomposed. Section 4 presents the related work. We conclude in Section 5.

2 Background

2.1 The Codelet Model

While most prevalent execution models in their current state are struggling to scale to future machine's peak performance, we propose the codelet execution model. The codelet PXM differs fundamentally from its Von Neumann based competitors, as it draws its roots from the dataflow model.

The Codelet Abstract Machine Model. The codelet abstract machine model (AMM) consists of many nodes connected together via an interconnection network. Each node is expected to have several chips containing hundreds of cores. Interconnects with varying latencies will connect components at multiple levels. We envision two types of cores. The first is a simple Computation Unit (CU) which is responsible performing operations. The other is a Synchronization Unit (SU) which is responsible for steering computation. Figure 1 depicts the proposed abstract machine model.

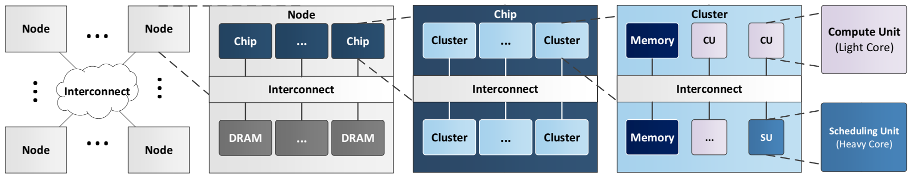


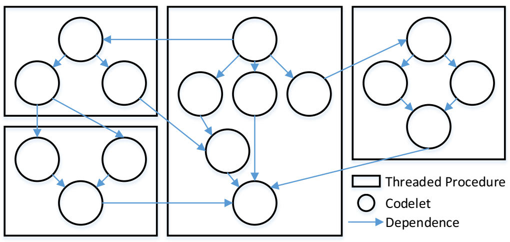
Fig. 1. The codelet abstract machine model

Codelets: Definition and Operational Semantics

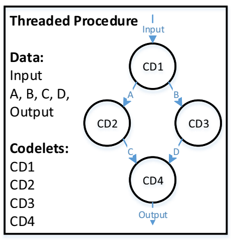
Definition. A codelet is a collection of machine instructions which are scheduled "atomically" as a non-preemptive, single unit of computation. In the codelet PXM, Codelets are the principal *scheduling quantum*. Codelets are expected (not required) to behave functionally, consuming inputs, working locally, and producing output leaving (ideally) no state behind. A codelet will only fire when all of the resources it requires are available. This entails having the necessary data local prior to execution, eliminating the need to hide latencies due to accessing remote data.

Operational Semantics. Codelets differs from a traditional task in their invocation. Similar to a dataflow actor [9], a codelet is fired once all of its dependencies (or *events*) are met. An event primarily consists of the data (i.e. arguments) required by the codelet to perform its operations; however events may include the requirement of any particular shared resource or condition such as bandwidth, power, etc. Each codelet has a requisite number of event which must be satisfied before execution. A Codelet's output is not atomic, meaning a codelet is capable of producing data, signaling other, and continuing execution differing from macro-dataflow actors.

Codelet Graphs. Codelets are linked together to form a codelet graph (CDG). In a CDG, each codelet acts as a producer and/or consumer. An initial codelet may fire, producing a result which multiple codelets can consume, giving way for more codelets to execute. Since codelets are linked together based on data dependencies, a CDG may benefit from the same properties as a dataflow graph. This includes the explicit view of parallelism and determinate execution. Figure 2(a) provides an example of various codelet graph instances. Note that codelets can signal other codelets outside of their CDG.



(a) Multiple codelet graphs linked together.



(b) A Threaded Procedure

Fig. 2. Examples of codelet graphs (CDGs) and threaded procedures (TPs). TPs are CDG containers, and allocate the space required to hold inputs, outputs, and intermediate results.

Parallel Constructs

Asynchronous Functions. Asynchronous functions are called Threaded Procedures (or TPs) in the codelet model. They are closely related to EARTH's [23] TPs. Much like its ancestor, the codelet model's TPs are containers for a codelet graph. A TP also features a *frame*, which holds the inputs passed to it, the resulting output, and values local to the contained CDG, as illustrated in Figure 2(b). A TP is invoked functionally, and exists in memory until all of the codelets in the CDG have finished executing. When a TP is instantiated, it is bound to a single cluster, equally binding the codelets. Prior to instantiation, a TP closure may be load balanced between clusters. In this way we utilize hierarchical parallelism, while providing some form of locality.

Loops. Loop parallelism is a crucial form of parallelism and a cornerstone in most useful parallel execution models. As such the codelet model provides a special loop construct enabling a CDG to be executed in successive iterations. Loops without loop-carried dependencies (for all loops) can be executed completely in parallel.

2.2 A Codelet Runtime

An execution model needs to be enforced to be useful. While using a combination of hardware and software is preferable to achieve high-performance [15], it is less time-consuming to implement everything in software that runs on off-the-shelf hardware. This section presents the Delaware Adaptive Run-Time System – DARTS.

Objectives. There already exists runtime system implementations of the codelet model currently under development, such as SWARM [16]. While they reuse the codelet object as the central unit of computation, they generally tend to stray from the original specification (see Section 4). Hence, our goal is to build a runtime system which will be true to the codelet model, but also serve as a research vehicle to evaluate and advance the model itself.

Faithfulness. DARTS is implemented to be faithful to the base codelet model. Hence, it employs codelets as the base unit of computation, but it also requires the use of threaded procedures as the containers for codelets.

Portability and Modularity. DARTS is written in C++. This language is low-level enough to ensure full control of the underlying hardware, while offering an object-oriented model which encourages modularity and component reuse. The latter point is important as we intend to use DARTS to explore and stretch the limit of the codelet PXM.

Implementation

The Codelet Abstract Machine. The codelet AMM described in Section 2.1 requires a concrete mapping to a physical machine. We reused the `hwloc` library [2] to obtain the topology of the underlying computation node. Once discovered, the runtime decides how to decompose the hardware resources (processing elements, caches, etc.) according to the user-programmer's selection of preset configurations. For example, one can elect a single socket of an SMP system to act as the AMM's cluster, and a single core on the socket to act as the synchronization unit. New mappings can easily be added to the description of the codelet AMM.

As described in the Section 2.1 each cluster contains two types of cores, one SU and several CUs. Each core runs one of two types of schedulers. Each CU runs a micro-scheduler, responsible primarily for executing codelets. An SU runs a Threaded Procedure scheduler (TP scheduler) which is responsible for load balancing TPs between clusters, instantiating codelets, and distributing codelets within a cluster. Having designed DARTS with modularity as a guiding principle, each scheduler is capable of running several different scheduling algorithms. For the scope of this work, we use a work-stealing policy similar to Cilk [1] to perform load balancing between TP schedulers. Within a cluster, micro-schedulers use a centralized queue to get work.

Codelets. The codelet specification is implemented as a `Codelet` class containing a synchronization slot (*sync slot*) and a method called `fire`. The sync slot is used to keep track of the outstanding dependencies. The codelet class must be specialized (i.e. derived) and can be instantiated once the `fire` method is expressed. `fire` is applied on a codelet by a CU's micro-scheduler when the codelet is chosen for execution.

Signaling. Each sync slot is initialized with the number of events the codelet requires to run. Codelets within a TP are known statically and can be accessed through the TP frame. The address of a codelet is required to signal codelets outside a TP, and can be provided at runtime. DARTS implements a form of argument fetching dataflow [10], as the act of signaling is dissociated from passing data. For this reason data is written first, and then a codelet is signaled.

Asynchronous Functions. DARTS uses TPs as the main way to instantiate portions of the computation graph. They act exactly as explained in Section 2.1. Much like codelets, threaded procedures are implemented as classes that must be derived by the programmer. The `ThreadedProcedure` class embeds an active codelet counter (to know when all the codelets it contains have finished executing), a pointer to a parent TP (the one which invoked it), and a member function to add a new codelet within the TP. The address of the TP frame (in practice, the pointer to the TP instance) is passed along to codelets so that they can access shared variables. Once the last codelet of an instantiated TP has finished running, the TP is deallocated along with all the codelets it contained.

Loops. Currently, DARTS implements three types of loops, a serial loop, a TP parallel for loop, and a codelet parallel for loop. Parallel for loops (*forall*) prohibit loop-carried dependencies, conceptually executing all iterations in parallel. Practically, the iterations are executed when sufficient hardware is available. The TP forall creates a TP for each iteration of the loop, permitting the iterations to run on any cluster. The codelet forall loop adds all the iterations to the invoking TP, pinning them to a single cluster.

Conceptually, a codelet loop requires two codelets, as shown in Figure 3. These "loop controllers" act as a source and sink. The source codelet is signaled normally. Upon execution, the source schedules copies of the enclosed CDG. After the loop body has finished executing, the "leaf" codelets of each iteration signal the sink codelet. Once all iterations have completed, the sink codelet deallocates the copied iterations, and signals the next codelet in the CDG. In practice, the source and sink codelets which control the loop are merged into one, to avoid useless memory allocations. Once it has performed its source action, the loop controller is reset to the number of "leaf" codelets multiplied by the number of iterations prior to scheduling the loop iterations. This approach is sufficient for supporting nested loops.

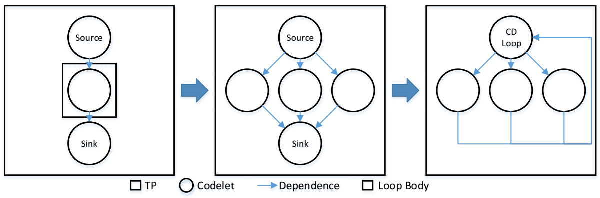


Fig. 3. Loops in the codelet model

3 Case Studies

We present two case studies, matrix multiplication and Graph 500's breadth first search. The kernels significantly differ from each other. DGEMM is compute-bound and allows for heavy data reuse, while Graph500 is memory-bound and stresses the memory subsystem with random accesses. Together, they provide an ideal base for an initial analysis of both the codelet model and DARTS.

3.1 Experimental Testbed

We evaluate our case studies on a 48-core compute node. It embeds four AMD Opteron 6234 (Interlagos) processors, clocked at 2.4 GHz. The node is equipped with 4 × 32 GB of DDR3-1333 ECC memory. Each core of the Interlagos have access to a 16 KB L1 data cache. Two cores share a 2 MB L2 unified cache. A 6 MB unified L3 cache is shared by six cores. Hence there are two L3 caches per Interlagos processor. One important architectural aspect of this processor is

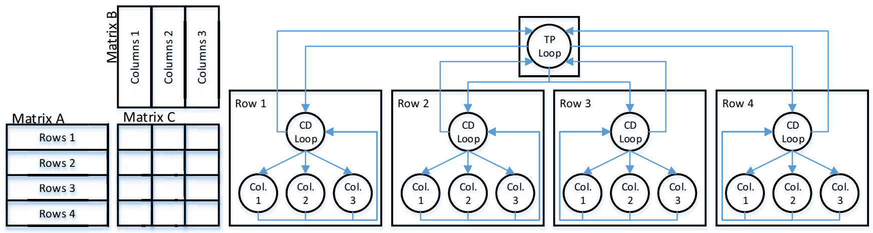


Fig. 4. Codelet representation of Matrix Multiply

that one floating-point unit is shared between two cores. It can process up to four double precision operations at once using AVX instructions.

Our testbed runs Scientific Linux 6. Both the DARTS runtime and kernels are compiled with GCC version 4.6.2 with `-O3`. In the following we compare the performance obtained with DARTS and OpenMP. All OpenMP programs were run with the threads being pinned as far away from each other as possible, to ensure they had as much cache memory to themselves as possible.

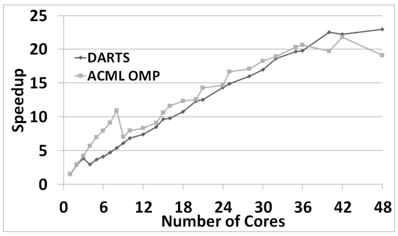
3.2 Matrix Multiplication

We use dense square matrix multiplication (DGEMM) to observe DARTS' performance on a common compute-bound kernel. Regular kernels like DGEMM typically perform well in OpenMP-like environments. In this study, we leverage AMD's Core Math Library (ACML) DGEMM kernel in two ways. First, we use ACML's sequential DGEMM kernel as an optimized building block in our DARTS implementation. Second, we compare our results against ACML's parallelized OpenMP DGEMM.

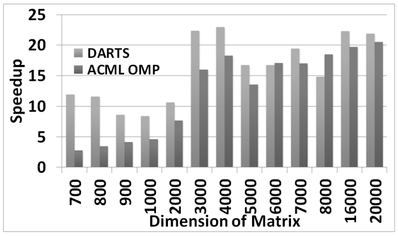
Figure 4 illustrates our decomposition of the DARTS version of DGEMM. We divide matrix A into rows and matrix B into columns producing a tile of results stored in the C matrix. We leave the "inner tiling" to the sequential ACML kernel. This partitioning is achieved using a TP forall loop to divide matrix A into even groups of rows. We further divide matrix B into columns using a codelet forall loop per spawned TP. Each parallel codelet instantiated will compute a tile in matrix C by calling ACML's DGEMM.

This partitioning translates well to the implementation of the abstract machine. A single group of rows of matrix A will be processed by a single cluster (a group of cores sharing a L3 cache). The cores within a cluster will individually compute a tile of matrix C, using exclusive groups of columns of matrix B, while sharing rows of matrix A.

Figure 5 presents our results. We present DGEMM's strong scaling for 4000×4000 matrices in Figure 5(a). When the number of cores used is small (i.e. less than 12) the OpenMP kernel clearly outperforms our DARTS implementation. As the number of cores grows, the gap becomes much more narrow ($\approx 8\%$ in favor of OpenMP). When the full node is used, DARTS achieves the highest speedup. As the number of cores increases we observe two phenomena: 1) the



(a) Strong scalability: N = 4000



(b) Weak scalability: N = 700 to 20,000

Fig. 5. Results for $N \times N$ matrix multiplication. The X axis on Figure 5(a) is the number of cores, and the size of N for Figure 5(b). In both figures, the Y axis shows the speedup w.r.t. ACML's sequential DGEMM.

FPU is more contended, as it is shared between two cores, and 2) contention on the memory banks also increases. Such contention is usually low enough w.r.t. to the number of active cores that there is no visible added latency. However with such a high number of cores simultaneously active, the delays accumulate in the OpenMP version, having a real impact on the final execution time. This phenomenon is not undocumented [12].

Figure 5(b) shows various results for runs with 48 active cores and dimension sizes ranging from 700 to 20,000. The DARTS implementation outperforms OpenMP in all but two cases. The average relative speedup between OpenMP and DARTS is 1.40×, while the maximum speedup is 4× when $N = 700$[1].

3.3 The Graph 500 Benchmark

To further evaluate DARTS, we used the Graph 500 parallel breadth-first search (BFS) algorithm[20]. BFS represents a class of irregular applications as the latencies of the memory accesses are dependent on the input data and subjected to NUMA effects. Hassaan et al. [13] present various parallel BFS algorithms in detail.

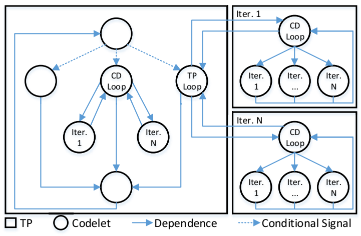
We compare a DARTS implementation of Graph 500's second kernel (BFS) to the OpenMP reference implementation. The kernel performs an in order BFS search. Each iteration traverses through a search frontier, visiting nodes and enqueuing their children for the next iteration's search frontier until all connected nodes have been visited. The kernel's output is a spanning tree.

The OpenMP kernel distributes the nodes in a search frontier using a parallel loop. After exploring a single search frontier, the OpenMP threads enter a barrier before exploring the next frontier. By default, the reference implementation uses static scheduling.

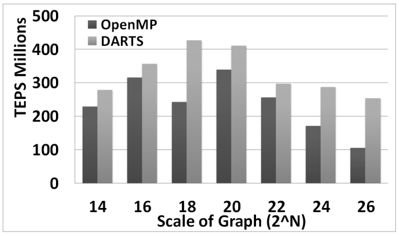
The DARTS implementation uses a similar barrier-like approach. A search frontier is distributed to one or more codelets, and a sink codelet is used upon

[1] While we present results ranging from $N = 700$ to $N = 20,000$, the overall experiments started with $N = 100$. These numbers are included in our averages.

completion. We however, take advantage of the two-level parallelism inherent in the codelet model. When the frontier is very small, we use a single codelet to process the frontier. As the frontier grows, we scale the parallelism using a codelet loop. This limits the parallelism to a single cluster, reducing the overhead of useless parallelism and increasing data locality. Once the frontier is large enough, we partition it into TPs using a TP loop and again into codelets using a codelet loop. Figure 6(a) illustrates our strategy.



(a) Codelet representation of BFS.



(b) Parallel BFS processing RMAT-generated graphs.

Fig. 6. Figure 6(a) illustrates the various strategies used to distribute the search frontier among codelets. Figure 6(b) presents our BFS results. The X axis represents the scale of the graph (2^n); the Y axis represents the harmonic mean of several runs reported in TEPS. Higher is better.

Figure 6(b) presents the number of Traversed Edges Per Second (TEPS), where the greater the number, the faster the implementation. Both implementations were provided identical graphs generated using the RMAT method [5]. Moreover, we use numactl to interleave memory. This approach was not used for DGEMM as it provided no performance gains. We did not present results for OpenMP using dynamic scheduling (creating smaller iteration chunks to increase over-subscription on the machine) as they were significantly worse. For smaller graph sizes, we see DARTS is able to narrowly outperform the reference implementation as work is easily balanced. However as the graph grows, DARTS begins to significantly outperform to the tune of 1.15-2.38×. This is due to the ease in which DARTS can exploit parallelism, scaling with the size of the search frontier. Moreover, DARTS balances the workload hierarchically, first balancing TPs and then codelets. Furthermore by decomposing the machine such that sockets map to AMM clusters, TPs will be balanced between sockets ensuring less contended accesses to DRAM. This result is a natural extension of the Codelet model, where similar approaches are possible, but require much effort.

4 Related Work

The arrival of multicore and manycore systems has rekindled the interest of efficiently running threads on shared-memory systems beyond the classical Pthread [19] and OpenMP [7] models.

Intel Threading Building Blocks [21] is a C++ library which provides several data structures and lock-free constructs to express parallelism. This includs the recent addition of augmented flow graphs (similar to a dataflow graph). The Codelet model differs as it advocates event-driven fine grained parallelism, while TBB focus on offering various types of parallelism.

Charm++ [14] also uses C++ to provide a parallel, object oriented, programing environment. Charm++'s goals are close to DARTS', with respect to resource management, energy efficiency, etc. However, while DARTS is event-driven, Charm++ is message-driven.

Cilk and its later iterations [1,18] are languages whose underlying execution model is more fine grain than classical shared-memory models. However, they do not implement dataflow features to express computations in terms of data and/or event dependencies.

Habanero Java [4], a "spin-off" of the X10 language [6], extends the initial Cilk syntax rendering it more flexible. Despite its recent additions of data driven constructs [22], its execution model does not rely on dataflow as a foundation unlike the Codelet model (and runtime implementation).

SWARM [16] is another runtime system which implements the codelet execution model. SWARM does not respect the basic semantics of individual codelets as proposed in [11], nor does it implement other advocated features, such as threaded procedures, loop constructs, etc.

The Concurrent Collections (CnC) family of languages [3] is a coordination language which is very much inspired by dynamic dataflow. CnC utilizes a separation of concerns, providing a tuning specification to achieve performance. A stand alone CnC program may not represent an event-driven codelet application, however with a proper tuning specification they could be equivalent.

5 Conclusion

In this paper we have presented an implementation of the fine-grain dataflow inspired codelet execution model. We have tested our implementation on a many-core shared-memory node, using two kernels. Our parallel implementation of DGEMM yields on average a 1.40× speedup over AMD's OpenMP-based implementation for matrix sizes ranging from 100×100 to $20,000 \times 20,000$ with a maximum speedup of 4×. We also compared ourselves to the reference implementation of the Graph500 BFS benchmark. On average, we reached a speedup of 1.50×, with a maximum of 2.38×.

Our future work includes further exploring Graph500 kernels in order to show how the codelet model eases the expression of parallelism and data dependencies between tasks. This includes exploring unordered BFS kernels which discard the barrier found at the end of each forall loop. We also want to further develop DARTS' parallel loop constructs applying software pipelining techniques, and building more general constructs to handle streams. Lastly, we would like to run our experiments on different compute node architectures, such as Intel's Ivy Bridge or other C++-supported general purpose many-core systems.

Acknowledgments. This work was partly supported by European FP7 project TERAFLUX, id. 249013. This material is based upon work supported by the Department of Energy (National Nuclear Security Administration) under the Award Number DE-SC0008717. We would also like to acknowledge the help of Tom StJohn for his expertise on the Graph500 benchmark, and Jean-Philippe Halimi for his work on DARTS.

References

1. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. 30(8), 207–216 (1995)
2. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: A generic framework for managing hardware affinities in hpc applications. In: 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 180–186 (February 2010)
3. Budimlić, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., et al.: Concurrent collections. Scientific Programming 18(3), 203–217 (2010)
4. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: The New Adventures of Old X10. In: PPPJ 2011: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (2011)
5. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: A recursive model for graph mining. Computer Science Department, 541 (2004)
6. Charles, P., Grothoff, C., Saraswat, V.A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the Twentieth Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, pp. 519–538 (October 2005)
7. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE Computational Science Engineering 5(1), 46–55 (1998)
8. Denneau, M., Warren Jr., H.S.: 64-bit cyclops principles of operation part i. Technical report, IBM Watson Research Center, Yorktown Heights, NY (2005)
9. Dennis, J.B.: First Version of a Data-Flow Procedure Language. In: Robinet, B. (ed.) Programming Symposium. LNCS, vol. 19, pp. 362–376. Springer, Heidelberg (1974)
10. Gao, G., Hum, H., Wong, Y.-B.: Parallel function invocation in a dynamic argument-fetching dataflow architecture. In: International Conference on Databases, Parallel Architectures and Their Applications, PARBASE 1990, pp. 112–116 (March 1990)
11. Gao, G.R., Suetterlein, J., Zuckerman, S.: Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond. Technical Memo – Available on request (April 2011)
12. Garcia, E., Orozco, D., Khan, R., Venetis, I., Livingston, K., Gao, G.R.: Dynamic Percolation: A case of study on the shortcomings of traditional optimization in Many-core Architectures. In: Proceedings of 2012 ACM International Conference on Computer Frontiers (CF 2012), Cagliari, Italy. ACM (May 2012)

13. Hassaan, M.A., Burtscher, M., Pingali, K.: Ordered and unordered algorithms for parallel breadth first search. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, pp. 539–540. ACM, New York (2010)
14. Kale, L.V., Krishnan, S.: Charm++: a portable concurrent object oriented system based on c++. SIGPLAN Not. 28(10), 91–108 (1993)
15. Knauerhase, R., Cledat, R., Teller, J.: For extreme parallelism, your OS is sooooo last-millennium. In: Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism, p. 3. USENIX Association (2012)
16. Lauderdale, C., Khan, R.: Towards a codelet-based runtime for exascale computing: position paper. In: Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT 2012, pp. 21–26. ACM, New York (2012)
17. Lee, E.A.: The problem with threads. Computer 39(5), 33–42 (2006)
18. Leiserson, C.E.: The cilk++ concurrency platform. In: Proceedings of the 46th Annual Design Automation Conference, DAC 2009, pp. 522–527. ACM, New York (2009)
19. Mueller, F.: Implementing posix threads under unix: Description of work in progress. In: Proceedings of the Second Software Engineering Research Forum, pp. 253–261. Citeseer (1992)
20. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.: Introducing the graph 500. Cray Users Group, CUG (2010)
21. Pheatt, C.: Intel Threading Building Blocks. J. Comput. Sci. Coll. 23(4), 298 (2008)
22. Taşırlar, S., Sarkar, V.: Data-Driven Tasks and their Implementation. In: ICPP 2011: Proceedings of the International Conference on Parallel Processing (September 2011)
23. Theobald, K.B.: EARTH: an efficient architecture for running threads. PhD thesis, McGill University, Montreal, Que., Canada, AAINQ50269 (May 1999)
24. Zuckerman, S., Suetterlein, J., Knauerhase, R., Gao, G.R.: Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper. In: Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT 2011, pp. 64–69. ACM, New York (2011)

Topic 12: Theory and Algorithms for Parallel Computation (Introduction)

Giuseppe F. Italiano, Henning Meyerhenke, Guy Blelloch, and Philippas Tsigas

Topic Committee

Parallelism permeates all levels of current computing systems, from single CPU machines, to large server farms, to geographically dispersed "volunteers" who collaborate over the Internet. The effective use of parallelism depends crucially on the availability of faithful, yet tractable, computational models for algorithm design and analysis and models of efficient strategies for solving key computational problems on prominent classes of computing platforms. Equally important are good algorithmic models of the way the different system components are interconnected. With the development of new genres of computing platforms, such as multicore parallel machines, desktop grids, clouds, and hybrid GPU/CPU-based systems, new computational models and paradigms are needed that will allow parallel programming to advance into mainstream computing. Topic 12 focuses on contributions providing new results on foundational issues regarding parallelism in computing and/or proposing improved approaches to the solution of specific algorithmic problems.

This year, papers submitted to Topic 12 covered nearly all subjects indicated in the call for papers. Among others, subjects included computation and/or communication complexity issues in various computational models, parallel algorithms and data structures for fundamental problems from various domains, and energy considerations in multiprocessor systems. Submissions indicated a significant interest of the parallel computing community towards developing new sound and solid methods for parallel problem solving in the presence of new technological challenges such as increasing core numbers per chip, deep memory hierarchies, complex distributed parallelism, and heterogeneity. Limitations and correctness of parallelism were also under investigation, for example in case of using accelerators.

Among all submissions, three high-quality papers were selected for presentation at the conference. The first paper, "Efficient Parallel and External Matching", by Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava, investigates the local-max algorithm for approximating a maximum weight matching. The algorithm is shown to run in a logarithmic number of phases, incurring work linear in the input size. Also, as their main contribution on parallel aspects of the topic, the authors provide efficient implementations in several models of computation and show that parallel local-max performs well in practice.

The second paper, "Model and complexity results for tree traversals on hybrid platforms", by Julien Herrmann, Loris Marchal, and Yves Robert, analyzes scheduling of DAGs in a streaming-like model on hybrid systems with a focus on memory usage. The employed model assumes that there are two types of execution units, where each of those has its own (limited) amount of memory. The authors provide complexity results for the question whether a given task graph can be executed given two limited memories, e.g. they show that it is impossible to approximate the optimal consumption of both memories simultaneously by any pair of constant factors.

The third paper, "Splittable Single Source-Sink Routing on CMP Grids: A Sublinear Number of Paths Suffice", by Adrian Kosowski and Przemysław Uznański, addresses the problem of power consumption for routing communication messages between cores of a single-chip multiprocessor assembled in a grid topology. It is shown that (and how) optimal power consumption (within constant factors) can be achieved by splitting requests into a sublinear number of paths.

Model and Complexity Results for Tree Traversals on Hybrid Platforms

Julien Herrmann[1], Loris Marchal[1], and Yves Robert[1,2]

[1] École Normale Supérieure de Lyon, CNRS & INRIA, France
[2] University of Tennessee Knoxville, USA

Abstract We study the complexity of traversing tree-shaped workflows whose tasks require large I/O files. We target a heterogeneous architecture with two resources of different types, each equipped with its own memory, such as a multicore node equipped with a dedicated accelerator (FPGA or GPU). Tasks in the workflow are tagged with the type of resource that is needed for their processing. Besides, a task can be processed on a given resource only if all its input files and output files can be stored in the corresponding memory. At a given execution step, the amount of data stored in each memory strongly depends upon the ordering in which the tasks are executed, and upon when communications between both memories are scheduled. The objective is to determine an efficient traversal that minimizes the maximum amount of memory of each type needed to traverse the whole tree. In this paper, we establish the complexity of this two-memory scheduling problem, provide inapproximability results, and show how to determine the optimal depth-first traversal. Altogether, these results lay the foundations for memory-aware scheduling algorithms on heterogeneous platforms.

1 Introduction

Modern computing platforms are heterogeneous: a typical node is composed of a multi-core processor equipped with a dedicated accelerator, such as a FPGA or a GPU. Our goal is to study the execution of a computational workflow, described by an out-tree, onto such a heterogeneous platform, with the objective of minimizing the amount of memory of each resource needed for its processing. The nodes of the workflow tree correspond to tasks, and the edges correspond to the dependencies among the tasks. The dependencies are in the form of input and output files: each node accepts a (potentially large) file as input, and produces a set of files, each of them to be processed by a different child node. We consider in this paper that we have two different processing units at our disposal, such as a CPU and a GPU. For sake of generality, we designate them by a color (namely *blue* and *red*). Each task in the workflow is best suited to a given resource type (say a core or a GPU), and is *colored* accordingly. To execute a task of a given color, all the input files and the output file of the task must fit within the corresponding memory. As the workflow tree is traversed, tasks of different colors are processed, and capacity constraints on both memory types must be met. In addition, when a child of a task has a different color than its parent, say for example that a blue task has a red child, a communication from the blue memory to the red memory must be scheduled before the red child can be processed (and again, the input file and all output files of this red child must fit within the red memory). All these constraints require to carefully orchestrate the scheduling of the tasks, as well as the communications between memories, in order to minimize the maximum amount of each memory that is needed throughout the tree traversal.

Memory-aware scheduling is an important problem that has been the focus of many papers (see Section 2 for related work). This work mainly builds upon the pioneering work of Liu, who has studied tree traversals that minimize the peak amount of memory used on a homogeneous system, hence with a single memory type. Liu first restricted to depth-first traversals in [5], before dealing with an optimal algorithm for arbitrary traversals in [5]. The main objective of this paper is to extend these results to colored trees with two memory types, and tasks belonging to a given type. Clearly, the traversal, i.e., the order chosen to execute the tasks, and to perform the communications, plays a key role in determining which amount of each memory is needed for a successful execution of the whole tree. The interplay between both memories dramatically complicates the scheduling: it is no surprise that the complexity of the problem, that was polynomial with a unique memory, now becomes NP-complete.

In this paper, we concentrate on memory usage, but we are fully aware that performance aspects are important too, and that even more difficult trade-offs are to be found between parallel performance and memory consumption. One could envision a fully general framework, where tasks have different execution-times for each resource type (instead of being tied to a given resource as in this paper), and where concurrent execution of several tasks on each resource type is possible (instead of the fully sequential processing of the task graph that is assumed in this paper). Altogether, this study is only a first step towards the design of memory-aware schedules on modern heterogeneous platforms with two memory types. However, despite the apparent simplicity of the model, our results show that we already face a difficult bi-criteria optimization problem when dealing with two different memory types. We firmly believe that the results presented in this paper will help to lay the foundations for memory-aware scheduling algorithms on modern heterogeneous platforms such as those equipped with multicores and GPUs. Indeed, one key contribution of the paper is the derivation of several complexity results: NP-completeness of the problem, and inapproximability within a constant (α, β) factor pair of both absolute minimum memory amounts. Here the absolute minimum memory of a given type is computed when assuming an infinite amount of memory of the other type. Another important contribution is the determination of the optimal depth-first traversal, which turns out to minimize both memories simultaneously (among all possible depth-first traversals).

The rest of the paper is organized as follows. We start with an overview of related work in Section 2. Then we detail the framework in Section 3. We deal with complexity results in Section 4, which constitutes the heart of the paper. Finally we provide some concluding remarks and hints for future work in Section 5.

2 Related Work

The work presented in this paper builds upon previous results related to memory-aware scheduling, but its applications are relevant to the field of sparse matrix factorization and of hybrid computing.

2.1 Sparse Matrix Factorization

Determining a memory-efficient tree traversal is very important in sparse numerical linear algebra. The elimination tree is a graph theoretical model that represents the storage requirements, and computational dependencies and requirements, in the Cholesky and LU factorization of sparse matrices. In a previous study, we have described how such trees are built, and how the multifrontal method organizes the computations along the tree [4]. This is the context of the founding studies of Liu [5,6] on memory minimization for postorder or general tree traversals mentioned in Section 1. Recently, still in the context of a single memory type, an extension of these results to parallel machines base been proposed in [7].

2.2 Scientific Workflows

The problem of scheduling a task graph under memory constraints also appears in the processing of scientific workflows whose tasks require large I/O files. Such workflows arise in many scientific fields, such as image processing, genomics or geophysical simulations. The problem of task graphs handling large data has been identified in [8] which proposes some simple heuristic solutions.

2.3 Pebble Game and Its Variants

On the more theoretical side, this work builds upon the many papers that have addressed the pebble game and its variants. Scheduling a graph on one processor with the minimal amount of memory amounts to revisiting the I/O pebble game with pebbles of arbitrary sizes that must be loaded into main memory before *firing* (executing) the task. The pioneering work of Sethi and Ullman [10] deals with a variant of the pebble game that translates into the simplest instance of the problem with a unique memory and where all files have weight 1. The concern in [10] was to minimize the number of registers that are needed to compute an arithmetic expression. The problem of determining whether a general DAG can be traversed with a given number of pebbles has been shown NP-hard by Sethi [9]. However, this problem has a polynomial complexity for tree-shaped graphs [10].

2.4 Hybrid Computing

Hybrid computing consists in the simultaneous use of CPUs and GPUs to optimize performance for high performance computing. Since CPUs and GPUs are powerful for specific and different tasks, its is natural to schedule a task on its "favorite" resource, that is, the resource where its execution time is minimal. This has been done successfully to increase performance in linear algebra libraries [11,3]. There also exist software tools that schedule an application composed of tasks with both CPU and GPU implementations on hybrid platforms: for instance, StarPU [1] optimizes the execution time of an application by scheduling its tasks on various resources based on predictions of execution and data transfer times.

3 Framework

As stated above, we deal with tree traversals on a two-memory system where each task belongs to a specific memory. Dependencies are in the form of input and output files: each task accepts a file as input from its parent node in the tree, and produces a set of files to be consumed by each child node.

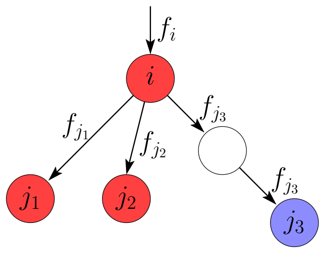


Fig. 1. Illustration of a tree node with its children

The tree work-flow $\mathcal{T}$ is composed of n nodes, or tasks, numbered from 1 to n, where $\mathit{Children}(i)$ denotes the set of the children of i. We consider here out-trees, where a parent node has to be processed before its children. As illustrated on Figure 1, each task (or node) i in the tree is characterized by the size f_i of its input file (data needed before the execution and received from its parent), which is the weight of the edge between the node and its parent, and by its *color*, which represents the specific memory where the task has to be executed. We let $\mathit{color}(i) \in \{\mathit{red}, \mathit{blue}\}$ represent the memory type of task i. If $\mathit{color}(i) = \mathit{red}$, then i is a computational node which operates in the *red* memory, which it uses to load its input file, execute its program and produce the set of output files for its children. Similarly, if $\mathit{color}(i) = \mathit{blue}$, then i is a computational node which operates in the *blue* memory. Each communication from one memory to the other is achieved through a communication node, which is uncolored. Hence, there are three types of nodes in the tree, *red* or *blue* computational nodes (or tasks), and uncolored communication nodes. Each time there is a data dependence between two tasks assigned to different memories, the output file of the source task has to be loaded from one memory into the other, using a communication node. Thus, in the model, the tree $\mathcal{T}$ does not contain direct edges between *blue* and *red* nodes; memory loads from one memory to the other occur only when processing a communication node. A *valid traversal* σ of the tree $\mathcal{T}$ is an ordered list of the nodes of $\mathcal{T}$ (including communication nodes) such that all node dependences in $\mathcal{T}$ are enforced by the schedule. Here are further details on the processing of each node type:

– Computational nodes: they represent a task executed on a specific memory. During the processing of a computational task i, the associated memory must contain the input file and its output files. Assuming that i is a *blue* task, the amounts of memory *BlueMemReq*(i) and *RedMemReq*(i) that are needed for this processing are thus:

$$\mathit{BlueMemReq}(i) = \left(\sum_{j \in \mathit{Children}(i)} f_j \right) + f_i, \qquad \mathit{RedMemReq}(i) = 0$$

After task i has been processed, the input file is discarded, while its output files are kept in memory until the processing of its children. Thus, for a traversal σ of $\mathcal{T}$, the actual amounts of memory used to process the *blue* node i are:

$$\mathit{BlueMemUsed}(\sigma, i) = \left(\sum_{j \in \mathit{Children}(i)} f_j \right) + f_i + \sum_{j \in S_{blue} \setminus \{i\}} f_j,$$
$$\mathit{RedMemUsed}(\sigma, i) = \sum_{j \in S_{red}} f_j$$

where S_{blue} (respectively S_{red}) denotes the set of files stored in the *blue* (respectively *red*) memory when the scheduler decides to execute task i. Note that S_{blue} must contain the input file of task i. After processing the *blue* node i, we have:

$$S_{blue} \leftarrow (S_{blue} \setminus \{i\}) \cup \mathit{Children}(i), \quad S_{red} \leftarrow S_{red}$$

Initially, S_{blue} contains the input file of the root and $S_{red} = \emptyset$ if the root is a *blue* node, and conversely if the root is a *red* node.

– Communication nodes represent communications between one memory and the other. Each communication node i has an input file of size f_i and an output file of the same size. It loads f_i units of memory from one memory to the other. During the processing of a communication task i from the *blue* memory to the *red* memory, both memories must contain the file of size f_i. Thus, the amount of *blue* and *red* memory needed for this processing is f_i:

$$\mathit{BlueMemReq}(i) = f_i, \quad \mathit{RedMemReq}(i) = f_i$$

After i has been processed, the input file from the *blue* memory is discarded, while the output file is kept in the *red* memory until the processing of its child. Thus, for a traversal σ of $\mathcal{T}$, the actual amounts of memory used to process the communication node i are:

$$\mathit{BlueMemUsed}(\sigma, i) = f_i + \sum_{j \in S_{blue} \setminus \{i\}} f_j, \quad \mathit{RedMemUsed}(\sigma, i) = f_i + \sum_{j \in S_{red}} f_j$$

Note that S_{blue} must contain the input file of task i. Letting j denote the unique child of communication node i, we have after the execution of i that:

$$S_{blue} \leftarrow S_{blue} \setminus \{i\}, \quad S_{red} \leftarrow S_{red} \cup \{j\}$$

It is important to stress that a communication node need not be processed right after the execution of its parent. The only constraint is that its processing must precede the execution of its unique child. This flexibility in the schedule severely complicates the search for efficient traversals.

As stated above, we face a multi-criteria optimization problem: how to minimize the amount of both memories needed for the tree traversal? The *peak memory* is the maximum usage of each memory over the whole schedule σ of the tree $\mathcal{T}$, and is defined for the *blue* and the *red* memory by:

$$M^{\sigma}_{\text{blue}}(\mathcal{T}) = \max_i \ \mathit{BlueMemUsed}(\sigma, i), \quad M^{\sigma}_{\text{red}}(\mathcal{T}) = \max_i \ \mathit{RedMemUsed}(\sigma, i)$$

Thus, we define the optimal peak for each memory needed to process a tree $\mathcal{T}$ as:

$$M^{\text{opt}}_{\text{blue}}(\mathcal{T}) = \min_\sigma \ M^{\sigma}_{\text{blue}}(\mathcal{T}), \quad M^{\text{opt}}_{\text{red}}(\mathcal{T}) = \min_\sigma \ M^{\sigma}_{\text{red}}(\mathcal{T})$$

We point out that $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ can be seen as the minimum amount of *blue* memory required to traverse the tree when there is an unbounded amount of *red* memory available: a schedule which reaches $M^{\sigma}_{\text{blue}}(\mathcal{T}) = M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ can use an arbitrary amount of *red* memory. Intuitively, one may ask what are trade-offs between the *blue* and *red* memory requirements of feasible schedules. One major objective of this paper is to provide quantitative answers to this question.

Top-down vs. bottom-up traversals. We conclude this section with two remarks on the model. First, we can handle the case where a node in the tree needs an execution file (in addition to input and output files) by adding an extra child to the node, whose input file has the size of the execution file. Second, there is a complete equivalence with top-down traversals of out-trees (the problem addressed in this paper) and bottom-up traversals of in-trees (as used in sparse matrice factorization). In a nutshell, one only needs to reverse the direction of the edges, and to execute the schedule backwards, to move form one variant to another[1]. In fact, the literature deals with both variants. The seminal paper of Liu [5] originally deals with post-order bottom-up traversals for in-trees, while we speak of depth-first top-down traversals for out-trees in this paper, but there is no actual difference.

[1] This equivalence has been formally proven in [4] for single-memory platforms, and it is straightforward to extend the proof for two-memory systems.

4 Complexity Results

This section presents several important complexity results. We start with the NP-completeness of the two-memory minimization problem in Section 4.1. Next we show in Section 4.2 that the problem reduces to traversing uncolored trees when one memory is unbounded. Then, we prove in Section 4.3 that it is impossible to approximate both minimum memories within arbitrary constant factors. Finally, we determine the optimal depth-first traversal (the equivalent of post-order traversals for in-trees). Due to lack of space, only the inapproximability proof is detailed in Section 4.3. All other proofs are available in the companion research report [2].

4.1 Hardness of the Problem

Our first result assesses the complexity of the problem, as formulated in the following definition.

Definition 1 (TwoMemoryTraversal). *Given a tree $\mathcal{T}$ with n nodes, and two fixed memory amounts M_{red} and M_{blue}, does there exist a traversal σ of the tree such that $M^{\sigma}_{\text{blue}}(\mathcal{T}) \leq M_{\text{blue}}$ and $M^{\sigma}_{\text{red}}(\mathcal{T}) \leq M_{\text{red}}$?*

Theorem 1. *The* TwoMemoryTraversal *problem is NP-complete.*

The proof relies on a reduction from the 2-partition problem: consider an instance of 2-partition with n integers a_i of sum S . The reduction uses the tree illustrated on Figure 2, with maximum memory amounts set to $M_{red} = 3S$ and $M_{blue} = 2S$. Assuming without loss of generality that R_{root} is processed before $R^{(2)}_{root}$, it is possible to prove that if the processing of the tree does not exceed the prescribed memory bounds, then a subset I of the C_i such that $\sum_{i\in I} a_i = S/2$ has to be processed before R_{big}. The detailed proof of this result is available in [2].

4.2 When One Memory is Unbounded

In this section, we focus on the computation of $M^{\text{opt}}_{\text{red}}(\mathcal{T})$ (or $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$) which represents the minimal peak memory reachable when there is no constraint on the other memory. We show that the computation of $M^{\text{opt}}_{\text{red}}(\mathcal{T})$ and $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ for a bi-colored tree $\mathcal{T}$ reduces to the computation of the minimal peak memory for an uncolored tree.

Definition 2. *Given a bi-colored tree $\mathcal{T}$, we construct the corresponding uncolored (or for convenience, single-colored) tree $\mathcal{T}_{\text{blue}}$ by turning every communication node and* red *node into a* blue *node, and by turning every* red *edge of weight f_i into a* blue *edge of weight* 0*. We construct the single-colored tree $\mathcal{T}_{\text{red}}$ in a similar way. We let M^{∞}_{blue} denote the minimal amount of memory needed to process $\mathcal{T}_{\text{blue}}$ (and similarly, M^{∞}_{red} for $\mathcal{T}_{\text{red}}$).*

The following result is straightforward.

Theorem 2. *For any bi-colored tree $\mathcal{T}$, we have $M^{\infty}_{\text{red}} = M^{\text{opt}}_{\text{red}}(\mathcal{T})$ and $M^{\infty}_{\text{blue}} = M^{\text{opt}}_{\text{blue}}(\mathcal{T})$.*

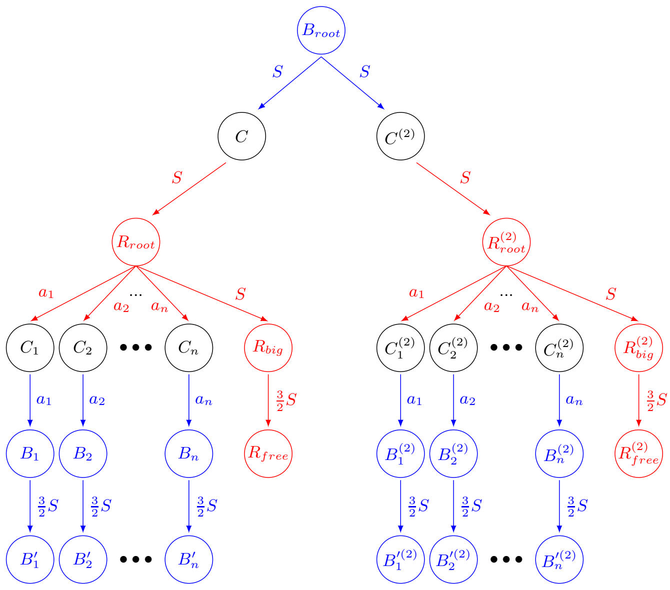


Fig. 2. Tree used in the proof of Theorem 1

4.3 Joint Minimization of Both Objectives

Since the traversal problem is NP-complete, it is natural to wonder whether there exist approximation algorithms. In this section, we prove that there does not exist schedules that approximates both minimum memories $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ and $M^{\text{opt}}_{\text{red}}(\mathcal{T})$ within arbitrary constant factors for any bi-colored tree $\mathcal{T}$. Since the (usually unfeasible) point of the Pareto diagram with coordinates $(M^{\text{opt}}_{\text{blue}}(\mathcal{T}),M^{\text{opt}}_{\text{red}}(\mathcal{T}))$ is sometimes called the *Zenith*, this result amounts to proving that there exists no *Zenith*-approximation.

Definition 3. *Given a bi-colored tree $\mathcal{T}$, we can construct the corresponding uncolored tree $\mathcal{T}_{\text{unco}}$ by turning every colored node of $\mathcal{T}$ into an uncolored node. We let $M^{\text{opt}}_{\text{unco}}(\mathcal{T}_{\text{unco}})$ be the minimal amount of memory needed to process $\mathcal{T}_{\text{unco}}$.*

The following lemma, whose simple proof can be found in [2], is helpful to prove the following theorem.

Lemma 1. *Given a bi-colored tree $\mathcal{T}$ with n nodes, consider an arbitrary traversal σ of $\mathcal{T}$ that requires an amount of* red *memory equal to $M^{\sigma}_{\text{red}}(\mathcal{T})$ and an amount of* blue *memory equal to $M^{\sigma}_{\text{blue}}(\mathcal{T})$. Then necessarily:*

$$M^{\sigma}_{\text{red}}(\mathcal{T}) + M^{\sigma}_{\text{blue}}(\mathcal{T}) \geq M^{\text{opt}}_{\text{unco}}(\mathcal{T}_{\text{unco}})$$

Theorem 3. *There exists no algorithm that is both an α-approximation for* blue *memory peak minimization and a β-approximation for* red *memory peak minimization, when scheduling bi-colored trees.*

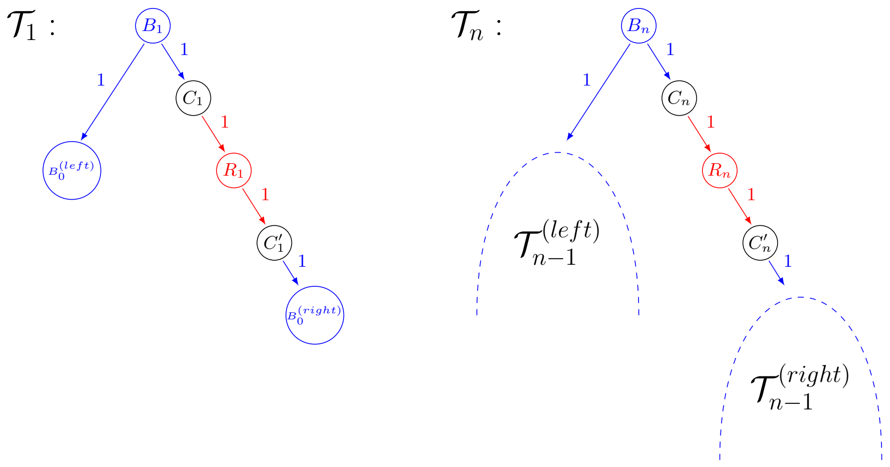


Fig. 3. Recursive definition of $\mathcal{T}_n$ in the proof of Theorem 3

Proof. To establish this result, we proceed by contradiction. We therefore assume that there is an integer α, an integer β, and an algorithm $\mathcal{A}$ that processes any bi-colored tree $\mathcal{T}$ using a *blue* peak memory that is not greater than α times the optimal *blue* peak memory $M_{\text{blue}}^{\text{opt}}(\mathcal{T})$ and using a *red* peak memory that is not greater than β times the optimal *red* peak memory $M_{\text{red}}^{\text{opt}}(\mathcal{T})$. To derive the contradiction, we use the family of tree $(\mathcal{T}_n)_{n\in\mathbb{N}}$ depicted on Figure 3. $\mathcal{T}_n$ is defined recursively using $\mathcal{T}_{n-1}$.

We prove the following statements:

– $\mathbf{\forall n \geq 2, M_{\text{blue}}^{\text{opt}}(\mathcal{T}_n) = 3}$
Consider the traversal σ_{blue} that processes $\mathcal{T}_n$ as follows:
  - If $n = 0$, σ_{blue} processes the node B_0
  - If $n > 0$, σ_{blue} processes the nodes B_n and C_n. Then $\mathcal{T}_{n-1}^{(left)}$ is processed recursively. Nodes R_n and C'_n follow. And finally $\mathcal{T}_{n-1}^{(right)}$ is processed recursively.

  At each step of this process, the traversal σ_{blue} does not use more than 3 units of *blue* memory. Since $\mathit{BlueMemReq}(B_{n-1}) = 3$, this proves that $M_{\text{blue}}^{\text{opt}}(\mathcal{T}_n) = 3$.

– $\mathbf{\forall n \geq 1, M_{\text{red}}^{\text{opt}}(\mathcal{T}_n) = 2}$
Consider the traversal σ_{red} that processes $\mathcal{T}_n$ as follows. At step k:
  - If $k = 0$, σ_{red} processes the node B_0
  - If $k > 0$, σ_{red} processes the nodes B_k. Then $\mathcal{T}_{k-1}^{(left)}$ is processed recursively. Nodes C_k, R_k and C'_k follow. And finally $\mathcal{T}_{k-1}^{(right)}$ is processed recursively.

  At each step of this process, the traversal σ_{red} does not use more than 2 units of *red* memory. Since $\mathit{RedMemReq}(R_n) = 2$, this proves that $M_{\text{red}}^{\text{opt}}(\mathcal{T}_n) = 2$.

– Let $\mathcal{T}_n^{\text{unco}}$ be the uncolored tree corresponding to $\mathcal{T}_n$ as describe in Definition 3 and $M_{\text{unco}}^{\text{opt}}(\mathcal{T}_n^{\text{unco}})$ the minimum amount of memory required to execute it. We now prove by induction that $M_{\text{unco}}^{\text{opt}}(\mathcal{T}_n^{\text{unco}}) = n + 2$ for $n \geq 2$. As show in [6], depth-first traversals (called *post-order* traversals in [6]) traversals are optimal for peak memory minimization of uncolored trees with unit costs. Besides, all depth-first traversals of $\mathcal{T}_n^{\text{unco}}$ require the same amount of memory. Thus $M_{\text{unco}}^{\text{opt}}(\mathcal{T}_n^{\text{unco}}) = M_{\text{unco}}^{\text{opt}}(\mathcal{T}_{n-1}^{\text{unco}}) + 1$ for $n \geq 2$. Since $M_{\text{unco}}^{\text{opt}}(\mathcal{T}_1^{\text{unco}}) = 2$, we have the result.

By hypothesis, algorithm $\mathcal{A}$ can process any $\mathcal{T}_n$ with $M_{\text{blue}}^{\mathcal{A}}(\mathcal{T}_n) \leq \alpha . M_{\text{blue}}^{\text{opt}}(\mathcal{T}_n) = 3\alpha$ and $M_{\text{red}}^{\mathcal{A}}(\mathcal{T}_n) \leq \beta . M_{\text{red}}^{\text{opt}}(\mathcal{T}_n) = 2\beta$. Let $n_0 = \lceil 3\alpha + 2\beta \rceil$, we have:

$$\begin{aligned} M_{\text{blue}}^{\mathcal{A}}(\mathcal{T}_{n_0}) + M_{\text{red}}^{\mathcal{A}}(\mathcal{T}_{n_0}) &\leq 3\alpha + 2\beta \\ &< \lceil 3\alpha + 2\beta \rceil + 2 \\ &= M_{\text{unco}}^{\text{opt}}(\mathcal{T}_{n_0}^{\text{unco}}) \end{aligned}$$

This contradicts Lemma 1, which means that such an algorithm $\mathcal{A}$ cannot exist.

4.4 Depth-First Traversals

Definition 4. *A depth-first traversal is a feasible traversal that processes all nodes of a tree $\mathcal{T}$ by processing the root and, then, recursively processing all sub-trees. Hence, in a post-order traversal, after processing a node i, the whole sub-tree rooted at i is completely processed before any other node that does not belong to this sub-tree. Formally, a feasible traversal σ of the tree $\mathcal{T}$ with n nodes is a depth-first traversal if and only if for each node $r \in \mathcal{T}$, with two children $i \in$ Children(r) and $j \in$ Children(r), we have:*

$$\sigma(i) < \sigma(j) \Rightarrow (\forall u \in T_i, \sigma(u) < \sigma(j))$$

where T_i is the sub-tree rooted at the node i.

In the context of single-memory trees, depth-first traversals are known to be sub-optimal [6]: worse, their memory usage can be arbitrarily high as compared to that of the optimal solution [4]. Clearly, these negative results remain true in a two-memory framework (simply assume that one memory is infinite). Still, depth-first traversals are a natural heuristic for traversing tree graphs, and they enjoy a simple implementation and memory management. As such, they are the most commonly used traversals in actual sparse solvers. Algorithm 1 computes the optimal depth-first traversal: when it encounters a blue node (respectively a red node), it applies the rule for minimizing the blue (resp. red) memory in depth-first traversals, which does not impact the amount of red (resp. blue) memory. It turns out that this traversal is optimal among all depth-first traversals for both memory usages (see proof in [2]).

Theorem 4. *Algorithm 1 returns the best depth-first traversal σ of $\mathcal{T}$ for both the* blue *and the* red *memories and the amount of memory M^{blue} and M^{red} used by σ.*

Algorithm 1: BestDepthFirstTraversal($\mathcal{T}$)

```
output: Schedule $\sigma$ with peak blue memory $M^{blue}$ and peak red memory $M^{red}$
root $\leftarrow$ the root of $\mathcal{T}$ ;
$CurrentMem \leftarrow 0$;
$(\sigma, M^{blue}, M^{red}) \leftarrow ([\text{root}], 0, 0)$;
for $i \in Children(root)$ do
    $(\sigma_i, M_i^{blue}, M_i^{red}) \leftarrow$ BestDepthFirstTraversal($T_i$);
    $CurrentMem \leftarrow CurrentMem + f_i$
if color($root$) = blue then
    for $i \in Children(root)$ in the increasing order of $M_i^{\text{blue}} - f_i$ do
        $\sigma \leftarrow [\sigma; \sigma_i]$;
        $CurrentMem \leftarrow CurrentMem - f_i$;
        $M^{blue} \leftarrow \max(M^{blue}, CurrentMem + M_i^{blue})$;
    $M^{red} \leftarrow \max_{i \in Children(root)} M_i^{red}$;
if color($root$) = red then
    for $i \in Children(root)$ in the increasing order of $M_i^{\text{red}} - f_i$ do
        $\sigma \leftarrow [\sigma; \sigma_i]$;
        $CurrentMem \leftarrow CurrentMem - f_i$;
        $M^{red} \leftarrow \max(M^{red}, CurrentMem + M_i^{red})$;
    $M^{blue} \leftarrow \max_{i \in Children(root)} M_i^{blue}$;
if the root node is an uncolored communication node then
    i $\leftarrow$ the unique child of root; $\sigma \leftarrow [\sigma; \sigma_i]$;
    if color($i$) = blue then
        $M^{blue} \leftarrow M_i^{blue}$;
        $M^{red} \leftarrow \max(f_i, M_i^{red})$;
    if color($i$) = red then
        $M^{red} \leftarrow M_i^{red}$;
        $M^{blue} \leftarrow \max(f_i, M_i^{blue})$;
return $(\sigma, M^{blue}, M^{red})$;
```

5 Conclusion

In this paper, we have studied the bi-criteria memory minimization problem that arises when traversing a task tree for a system composed of two different computing units with their own memory. After relating this problem to the well-studied one-memory problem, we have proved that the search for an optimal solution is NP-complete, and that it was impossible to approximate both memories by any pair of constant factors. In addition, we have determined the optimal depth-first traversal, which turns out to minimize both memories simultaneously.

Admittedly, the platform model used in this paper is a simplified one, but this was the key to derive complexity results in this initial study. In future work, the model should be refined in several directions, so as to more accurately account for all the characteristics of hybrid platforms (using both CPUs and GPUs);

however, this is not expected to change the NP-completeness results. A first step towards a more realistic model would be to include computation times for the tasks, and to try to minimize both the processing time of the total tree, and the amount of blue and red memories needed. A second step would consist in providing each task with two different running times rather than a color, and to give the ability for the scheduler to choose the computing unit for each task based on running time and memory. Given the complexity of the problem in the simple case, we do not expect to find approximation algorithms, but rather to design simple heuristics (as BestDepthFirst) that may be optimal under restrictive conditions, either on the traversal type or on the tree structure.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011)
2. Herrmann, J., Marchal, L., Robert, Y.: Tree traversals with task-memory affinities. Research report 8226, INRIA (2013)
3. Horton, M., Tomov, S., Dongarra, J.: A class of hybrid lapack algorithms for multicore and gpu architectures. In: 2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC), pp. 150–158 (July 2011)
4. Jacquelin, M., Marchal, L., Robert, Y., Ucar, B.: On optimal tree traversals for sparse matrix factorization. In: IPDPS 2011 (2011)
5. Liu, J.W.H.: On the storage requirement in the out-of-core multifrontal method for sparse factorization. ACM Trans. Math. Software 12(3), 249–264 (1986)
6. Liu, J.W.H.: An application of generalized tree pebbling to sparse matrix factorization. SIAM J. Algebraic Discrete Methods 8(3) (1987)
7. Marchal, L., Sinnen, O., Vivien, F.: Scheduling tree-shaped task graphs to minimize memory and makespan. Research report 8082, INRIA (2012); Accepted for publication in IPDPS 2013
8. Ramakrishnan, A., Singh, G., Zhao, H., Deelman, E., Sakellariou, R., Vahi, K., Blackburn, K., Meyers, D., Samidi, M.: Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In: CCGRID 2007. IEEE (2007)
9. Sethi, R.: Complete register allocation problems. In: STOC 1973, pp. 182–195. ACM Press (1973)
10. Sethi, R., Ullman, J.: The generation of optimal code for arithmetic expressions. J. ACM 17(4), 715–728 (1970)
11. Tomov, S., Nath, R., Du, P., Dongarra, J.: MAGMA version User's guide (2009), http://icl.eecs.utk.edu/magma/

Efficient Parallel and External Matching

Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava

Karlsruhe Institute of Technology, Karlsruhe, Germany
marcelbirn@gmx.de, {osipov,sanders,christian.schulz}@kit.edu, nodari@ira.uka.de

Abstract. We study a simple parallel algorithm for computing matchings in a graph. A variant for unweighted graphs finds a maximal matching using linear expected work and $\mathcal{O}(\log^2 n)$ expected running time in the CREW PRAM model. Similar results also apply to External Memory, MapReduce and distributed memory models. In the maximum weight case the algorithm guarantees a 1/2-approximation. Although the parallel execution time is linear for worst case weights, an experimental evaluation indicates good scalabilty on distributed memory machines and on GPUs. Furthermore, the solution quality is very good in practice.

1 Introduction

A matching M of a graph $G = (V, E)$ is a subset of edges such that no two elements of M have a common end point. Many applications require the computation of matchings with certain properties, like being maximal (no edge can be added to M without violating the matching property), having maximum cardinality, or having maximum total weight $\sum_{e \in M} w(e)$. Although these problems can be solved optimally in polynomial time, optimal algorithms are not fast enough for many applications involving large graphs where we need near linear time algorithms. For example, the most efficient algorithms for graph partitioning rely on repeatedly contracting maximal matchings, often trying to maximize some edge rating function w. Refer to [13] for details and examples. For very large graphs, even linear time is not enough – we need a parallel algorithm with near linear work or an algorithm working in the external memory model [1].

Here we consider the following simple *local max* algorithm [12]: Call an edge locally maximal, if its weight is larger than the weight of any of its incident edges; for unweighted problems, assign unit weights to the edges. When comparing edges of equal weight, use tie breaking based on random perturbations of the edge weights. The algorithm starts with an empty matching M. It repeatedly adds locally maximal edges to M and removes their incident edges until no edges are left in the graph. The result is obviously a maximal matching (every edge is either in M or it has been removed because it is incident to a matched edge). The algorithm falls into a family of weighted matching algorithms for which Preis [24] shows that they compute a 1/2-approximation of the maximum weight matching problem. Hoepman [12] derives the local max algorithm as a distributed adaptation of Preis' idea. Based on this, Manne and Bisseling [18] devise sequential and parallel implementations. They prove that the algorithm needs only a logarithmic number of iterations to compute maximal matchings by noticing that a maximal matching problem can be translated into a maximal independent set problem on the *line graph* which can be solved by Luby's algorithm [17]. However, this does not yield an algorithm with linear work since it is not proven that the edge set indeed shrinks geometrically.[1] Manne and Bisseling also give a sequential algorithm running in time $\mathcal{O}(m \log \Delta)$ where Δ is the maximum degree. On a NUMA shared memory machine with 32 processors (SGI Origin 3800) they get relative speedup < 6 for a complete graph and relative speedup ≈ 10 for a more sparse graph partitioned with Metis. Since this graph still has average degree ≈ 200 and since the speedups are not impressive, this is a somewhat inconclusive result when one is interested in partitioning large sparse graphs on a larger number of processors.

Parallel matching algorithms have been widely studied. There is even a book on the subject [16] but most theoretical results concentrate on work-inefficient algorithms. The only linear work parallel algorithms that we are aware of are randomized CRCW PRAM algorithms by Israeli and Itai [14] and Blelloch et al. [4]. We will call them IIM and BFSM, respectively. IIM runs in expected $\mathcal{O}(\log n)$ time and BFSM runs in $\mathcal{O}(\log^3 n)$ time with high probability.

Fagginger Auer and Bisseling [8] study an algorithm similar to [14] which we call red-blue matching (RBM) here. They implement RBM on shared memory machines and GPUs. They prove good shrinking behavior for random graphs, however, provide no analysis for arbitrary graphs.

Our Contributions. We give a simple approach to implementing the local max algorithm that is easy to adapt to many models of computation. We show that for computing maximal matchings, the algorithm needs only linear work on a sequential machine and in several models of parallel computation (Section 2). Moreover it has low I/O complexity on several models of memory hierarchies.

Our CRCW PRAM local max algorithm matches the optimal asymptotic bounds of IIM. However, our algorithm is simpler (resulting in better constant factors), removes higher fraction of edges in each iteration (IIM's proof shows less than 5% per iteration, while we show at least 50%) and our analysis is a lot simpler. We also provide the first CREW PRAM algorithm which performs linear work and runs in expected $\mathcal{O}(\log^2 n)$ time.[2]

In Section 3 we explain how to implement local max on practical massively parallel machines such as MPI clusters and GPUs. Our experiments indicate that the algorithm yields surprisingly good quality for the weighted matching problem and runs very efficiently on sequential machines, clusters with reasonably partitioned input graphs, and on GPUs. Compared to RBM, the local max implementations remove more edges in each iteration and provide better quality

[1] Manne and Bisseling show such a shrinking property under an assumption that unfortunately does not hold for all graphs.

[2] While a generic simulation of IIM on the CREW PRAM model will result in a $\mathcal{O}(\log^2 n)$ time algorithm, the simulation incurs $\mathcal{O}(n \log n)$ work due to sorting.

results for the weighted case. Some of the results presented here are from the diploma thesis of Marcel Birn [2].

2 Parallel Local Max

Our central observation is:

Lemma 1. *Each iteration of the local max algorithm for the unit weight case removes at least half of the edges in expectation.*

Proof. Consider the graph remaining in the currently considered iteration where $d(v)$ denotes the degree of a node and m the remaining number of edges. Consider the end point at node v of an edge $\{u, v\}$ as *marked* if and only if some edge incident to v becomes matched. Note that an edge is removed if and only if at least one of its end points becomes marked. Now consider a particular edge $e = \{u, v\}$. Since any of the $d(u) + d(v) - 1$ edges incident to u and v is equally likely to be locally maximal, e becomes matched with probability $1/(d(u) + d(v) - 1)$.[3] If e is matched, this event is responsible for setting $d(u) + d(v)$ marks, i.e., the expected number of marks caused by an edge is $(d(u)+d(v))/(d(u)+d(v)-1) \geq 1$. By linearity of expectation, the total expected number of marks is at least m. Since no edge can have more than two marks, at least $m/2$ edges have at least one mark and are thus deleted.[4] ∎

Métivier et al. [22] uses a similar proof technique to define “preemptive removal” of nodes for distributed maximal independent set problem.

Assume now that each iteration can be implemented to run with work linear in the number of surviving edges (independent of the number of nodes). Working naively with the expectations, this gives us a logarithmic number of rounds and a geometric sum leading to linear total work for computing a maximal matching. This can be made rigorous by distinguishing *good* rounds with at least $m/4$ matched edges and bad rounds with less matched edges. By Markov’s inequality, we have a good round with constant probability. This is already sufficient to show expected linear work and a logarithmic number of expected rounds. We skip the details since this is a standard proof technique and since the resulting constant factors are unrealistically conservative. An analogous calculation for median selection can be found in [20, Theorem 5.8]. One could attempt to show a shrinking factor close to $1/2$ rigorously by showing that large deviations (in the wrong direction) from the expectation are unlikely (e.g., using Martingale tail bounds). However this would still be a factor two away from the more heuristic argument in Footnote 4 and thus we stick to the simple argument.

[3] For this to be true, the random noise added for tie breaking needs to be renewed in every iteration. However, in our experiments this had no noticeable effect.

[4] This is a conservative estimate. Indeed, if we make the (over)simplified assumption that m marks are assigned randomly and independently to $2m$ end points, then only one fourth of the edges survives in expectation. Interestingly, this is the amount of reduction we observe in practice – even for the weighted case.

There are many ways to implement an iteration which of course depend on the considered model of computation.

Sequential Model. For each node v maintain a candidate edge $C[v]$, originally initialized to a dummy edge with zero weight. In an iteration go through all remaining edges $e = \{u, v\}$ three times. In the first pass, if $w(e) > w(C[u])$ set $C[u]:= e$ (add random perturbation to $w(e)$ in case of a tie). If $w(e) > w(C[v])$ set $C[v]:= e$. In the second pass, if $C[u] = C[v] = e$ put e into the matching M. In the third pass, if u or v is matched, remove e from the graph. Otherwise, reset the candidate edge of u and v to the dummy edge. Note that except for the initialization of C which happens only once before the first iteration, this algorithm has no component depending on the number of nodes and thus leads to linear running time in total if Lemma 1 is applied.

CRCW PRAM Model. In the most powerful variant of the *Combining CRCW PRAM* that allows concurrent writes with a maximum reduction for resolving write conflicts, the sequential algorithm can be parallelized directly running in constant time per iteration using m processors.

MapReduce Model. The CRCW PRAM result together with the simulation result of Goodrich et al. [11] immediately implies that each iteration of local max can be implemented in $\mathcal{O}(\log_M n)$ rounds and $\mathcal{O}(m \log_M n)$ communication complexity in the MapReduce model, where M is the size of memory of each compute node. Since typical compute nodes in MapReduce have at least $\Omega(m^\epsilon)$ memory [15], for some constant $\epsilon > 0$, each iteration of local max can be performed in MapReduce in constant rounds and linear communication complexity.

External Memory Models. Using the PRAM emulation techniques for algorithms with geometrically decreasing input size from [5, Theorem 3.2] the above algorithm can be implemented in the external memory [1] and cache-oblivious [9] models in $\mathcal{O}(sort(m))$ I/O complexity, which seems to be optimal.

2.1 $\mathcal{O}(\log^2 n)$ Work-Optimal CREW Solution

In this section, we present a $\mathcal{O}(\log^2 n)$ CREW PRAM algorithm, which incurs only $\mathcal{O}(m)$ work.

Consider the following representation of the graph $G = (V, E)$. Let V be a totally ordered set, i.e., given two vertices $u, v \in V$ we can uniquely determine whether $u < v$ or not. Let `E` be an array of undirected edges with each entry `E`$[k]$ storing all the information of a single edge $\{u, v\} \in E$, i.e., vertex endpoints u and v, its weight or any other auxiliary data. Let `A` be an array of tuples (v, e_k), where $v \in V$ and e_k is the *pointer* to `E`$[k]$ representing the edge $\{u, v\}$. Let `A` be sorted by the first entry, i.e. all tuples (v, e_k) pointing to the edges incident on the same vertex v are in contiguous space in `A`.

Note that any edge `E`$[k] = \{u, v\}$ contains two corresponding entries in `A` pointing to it: (u, e_k) and (v, e_k). During our algorithm, a processor responsible for (u, e_k) might need to find and update entry (v, e_k) (and vice versa). The following lemma describes how to compute for each entry (u, e_k) the index of the corresponding entry (v, e_k) in `A`.

Lemma 2. *For every edge* $\mathtt{E}[k] = \{u, v\}$ *entries* $\mathtt{A}[i] = (u, e_k)$ *and* $\mathtt{A}[j] = (v, e_k)$ *of* `A` *can compute each other's index in* `A` *in* $\mathcal{O}(1)$ *time and* $\mathcal{O}(|\mathtt{A}|)$ *work in the CREW PRAM model.*

Proof. For every $\mathtt{E}[k] = \{u, v\}$ we show how $\mathtt{A}[j] = (v, e_k)$ can compute the index of the corresponding entry $\mathtt{A}[i] = (u, e_k)$ in `A` for $u < v$. The indices for the other half of the entries are computed symmetrically.

The algorithm proceeds in two phases. In the first phase, each entry $\mathtt{A}[i] = (u, e_k)$, stores the value i in $\mathtt{E}[k] = \{u, v\}$ iff $u < v$. In the second phase, each entry $\mathtt{A}[j] = (v, e_k)$ reads the stored value i from $\mathtt{E}[k] = \{u, v\}$ iff $v > u$.

If we assign a separate processor to each entry of `A`, each processor performs only $\mathcal{O}(1)$ steps. Moreover, there are no concurrent writes because, at each step only one of the two vertices of the edge e_k writes to $\mathtt{E}[k]$. Note, we need a concurrent read to $\mathtt{E}[k] = \{u, v\}$ to determine the relative order of u and v. ■

Lemma 3. *Using our graph representation, each node* v *in the graph can apply an associative operator* $\oplus$ *to all edges incident on* v *in* $\mathcal{O}(\log |\mathtt{A}|)$ *time and* $\mathcal{O}(|\mathtt{A}|)$ *work on the CREW PRAM model.*

Proof. First, we read for each entry $(v, e_k) \in \mathtt{A}$ the value from $\mathtt{E}[k]$ on which to apply the operator. Next, we run segmented prefix sums with $\oplus$ operator on these values, where segments are the portions of `A` representing the neighbors of a single node and are easily identified from the definition of `A`. Finally, each entry of $(v, e_k) \in \mathtt{A}$ applies its result of segmented prefix sums to the edge $\mathtt{E}[k]$, while using the technique of Lemma 2 to avoid write conflicts. Each step of the algorithm can be implemented in $\mathcal{O}(\log |\mathtt{A}|)$ time using $\mathcal{O}(|\mathtt{A}|)$ work. ■

Now we are ready to describe the solution to the matching problem. We perform the following in each phase of the local max algorithm.

1. Each edge $\mathtt{E}[k]$ picks a random weight w_k.
2. Using Lemma 3, each vertex v identifies k' such that $\mathtt{E}[k']$ is the heaviest edge incident on v by applying the associative operator MAX to the edge weights picked in the previous step.
3. Using Lemma 2, each entry $(v, e_{k'})$ checks if $\mathtt{E}[k'] = \{u, v\}$ is also the heaviest incident edge on u. If so, the smaller of u and v adds $e_{k'}$ to the matching and sets the deletion flag $f = 1$ on $\mathtt{E}[k']$.
4. Using Lemma 3, each entry $(v, e_{k'})$ spreads the deletion flag over all edges incident on v by applying MAX associative operator on the deletion flags of incident edges on v. Thus, if at least one edge incident on v was added to the matching, all edges incident on v will be marked for deletion.
5. Now we must prepare the graph representation for the next phase by removing all entries of `E` and `A` marked for deletion, compacting `E` and `A` and updating the pointers of `A` to point to the compacted entries of `E`. To perform the compaction, we compute for each entry $\mathtt{E}[k]$, how many entries $\mathtt{E}[i]$ and

$\texttt{A}[i], i \leq k$ must be deleted. This can be accomplished using parallel prefix sums on the deletion flags of each entry in E and A. Let the result of prefix sums for edge $\texttt{E}[k]$ be d_k and for entry $\texttt{A}[i]$ be r_i. Then $k - d_k$ is the new address of the entry $\texttt{E}[k]$ and $i - r_i$ is the new address of $\texttt{A}[i]$ once all edges marked for deletion are removed.

6. Each entry $\texttt{E}[k]$ that is not marked for deletion copies itself to $\texttt{E}[k - d_k]$. The corresponding entry $(v, e_k) \in \texttt{A}$ updates itself to point to the new entry $\texttt{E}[k - d_k]$, i.e., (v, e_k) becomes (v, e_{k-d_k}), and copies itself to $\texttt{A}[i - r_i]$.

The algorithm defines a single phase of the local max algorithm. Each step of the phase takes at most $\mathcal{O}(|\texttt{A}|) = \mathcal{O}(m)$ work and $\mathcal{O}(\log |\texttt{A}|) = \mathcal{O}(\log m) = \mathcal{O}(\log n)$ time in the CREW PRAM model. Over $\mathcal{O}(\log m)$ phases, each with geometrically decreasing number of edges, the local max algorithm takes overall $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(m)$ work in the CREW PRAM model.

3 Implementations and Experiments

We now report experiments focusing on computing approximate maximum weight matchings. We consider the following families of inputs, where the first two classes allow comparison with the experiments from [19].

Delaunay Instances are created by randomly choosing $n = 2^x$ points in the unit square and computing their Delaunay triangulation. Edge weights are Euclidean distances.

Random graphs with $n := 2^x$ nodes, αn edges for $\alpha = \{4, 16, 64\}$, and random edge weight chosen uniformly from $[0, 1]$.

Random geometric graphs with 2^x nodes (rgg$x$). Each vertex is a random point in the unit square and edges connect vertices whose Euclidean distance is below $0.55 \ln n/n$. This threshold was chosen in order to ensure that the graph is almost connected.

Florida Sparse Matrix. Following [8] we use 126 symmetric non-0/1 matrices from [6] using absolute values of their entries as edge weights, see [3] for the full list. The number of edges of the resulting graphs $m \in (0.5 \ldots 16) \times 10^6$. See [3] for a detailed list.

Graph Contraction. We use the graphs considered by KaFFPa for partitioning graphs from the 10'th DIMACS Implementation Challenge [25].

We compare implementations of local max, the red-blue algorithm from [8] (RBM) (their implementation), heavy edge matching (HEM) [10], greedy, and the global path algorithm (GPA) [19]. HEM iterates through the nodes (optionally in random order) and matches the heaviest incident edge that is nonadjacent to a previously matched edge. The greedy algorithm sorts the edges by decreasing weights, scans them and inserts edges connecting unmatched nodes into the matching. GPA refines greedy. It greedily inserts edges into a graph G_2 with maximum degree two and no odd cycles. Using dynamic programming on the resulting paths and even cycles, a maximum weight matching of G_2 is computed.

Algorithms involving sorting use standard STL Visual Studio 2010 sort routine.

Sequential and shared-memory parallel experiments were performed on an Intel i7 920 2.67 GHz quad-core machine with 6 GB of memory. We used a commodity NVidia Fermi GTX 480 featuring 15 multiprocessors, each containing 32 scalar processors, for a total of 480 CUDA cores on chip. The GPU RAM is 1.5 GB. We compiled all implementations using CUDA 4.2 and Microsoft Visual Studio 2010 on 64-bit Windows 7 Enterprise with maximum optimization level.

3.1 Sequential Speed and Quality

We compare solution quality of the algorithms relative to GPA. Via the experiments in [19] this also allows some comparison with optimal solutions which are only a few percent better there. Figure 1 shows the quality for Delaunay graphs (where GPA is about 5 % from optimal [19]). We see that local max achieves almost the same quality as greedy which is only about 2 % worse than GPA. HEM, possibly the fastest nontrivial sequential algorithm is about 13 % away while RBM is 14 % worse than GPA, i.e., HEM and RBM almost double the gap to optimality of local max. Looking at the running times, we see that HEM is the fastest (with a surprisingly large cost for actually randomizing node orders) followed by local max, greedy, GPA, and RBM. From this it looks like HEM, local max, and GPA are the winners in the sense that none of them is dominated by another algorithm with respect to both quality and running time. Greedy has similar quality as local max but takes somewhat longer and is not so easy to parallelize. RBM as a sequential algorithm is dominated by all other algorithms. Perhaps the most surprising thing is that RBM is fairly slow. This has to be taken into account when evaluating reported speedups. We suspect that a more efficient implementation is possible but do not expect that this changes the overall conclusion. In [3] we report similar results for the rgg instances and random graphs.

Looking at the wide range of instances in the Florida Sparse Matrix collection leads to similar but more complicated conclusions. Figure 2 shows the solution

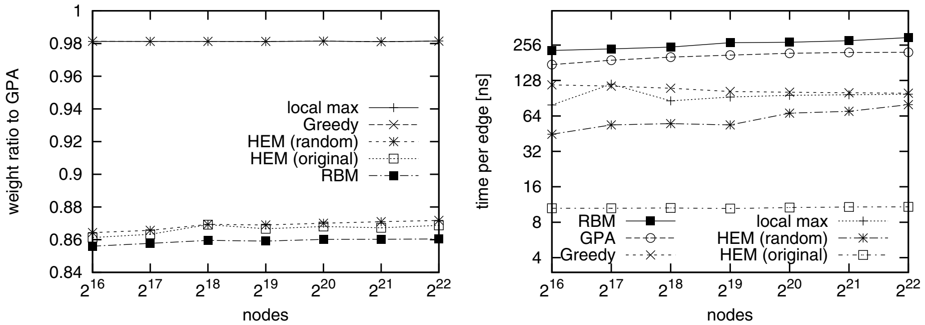


Fig. 1. Ratio of the weights computed by GPA and other algorithms for Delaunay instances and running times

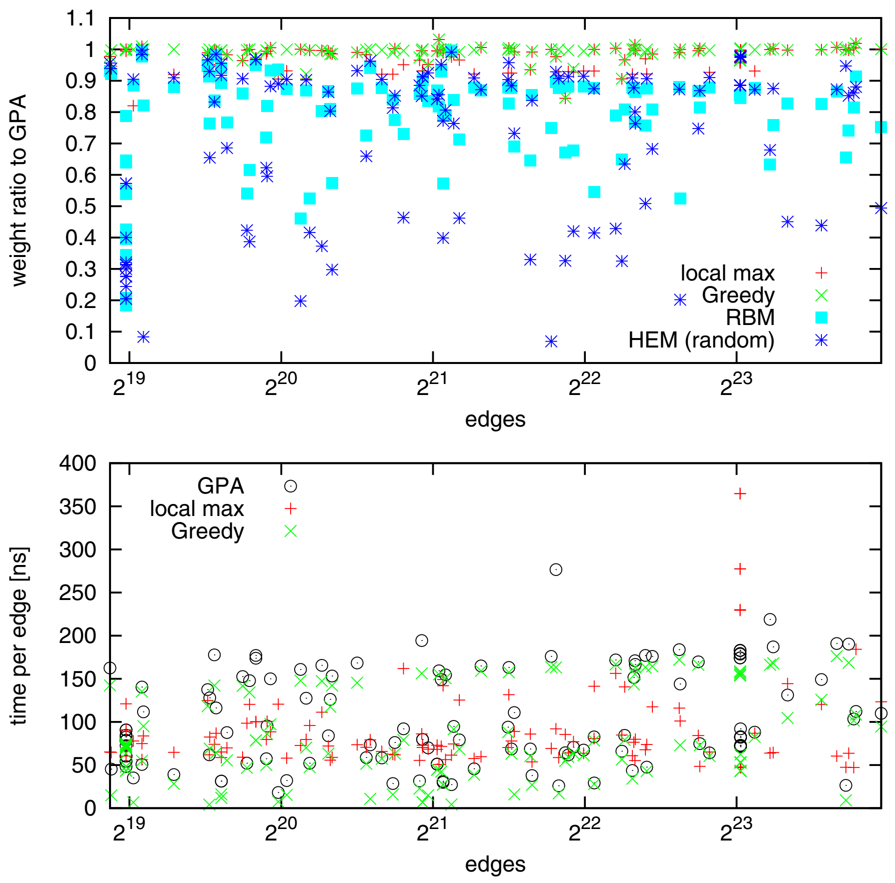


Fig. 2. Ratio of the weights computed by GPA and other sequential algorithms for sparse matrix instances and running time

qualities for greedy, local max, RBM and HEM relative to GPA. RBM and even more so HEM shows erratic behavior with respect to solution quality. Greedy and local max are again very close to GPA and even closer to each other although there is a sizable minority of instances where greedy is somewhat better than local max. Looking at the corresponding running times one gets a surprisingly diverse picture. HEM which is again fastest and RBM which is again dominated by local max are not shown. There are instances where local max is considerably faster then greedy and vice versa. A possible explanation is that greedy becomes quite fast when there is only a small number of different edge weights since then sorting is quite an easy problem.

Experiments on the graph contraction instances in [2] show local max about 1 % away from GPA. For these instances the average fraction of remaining edges after an iteration is well below 25 %. Notable exceptions are the graphs *add20* and *memplus* which both represent VLSI circuits. Nevertheless, none of the instances considered required more than 10 iterations.

3.2 Distributed Memory Implementation

Our distributed memory parallelization (using MPI) on p processing elements (PEs or MPI processes) assigns nodes to PEs and stores all edges incident to a node locally. This can be done in a load balanced way if no node has degree exceeding m/p. The second pass of the basic algorithm from Section 2 has to exchange information on candidate edges that cross a PE boundary. In the worst case, this can involve all edges handled by a PE, i.e., we can expect better performance if we manage to keep most edges locally. In our experiments, one PE owns nodes whose numbers are a consecutive range of the input numbers. Thus, depending on how much locality the input numbering contains we have a highly local or a highly non-local situation. We have not considered more sophisticated ways of node assignment so far since our motivating application is graph partitioning/clustering where almost by definition we initially do *not* know which nodes form clusters – this is the intended *output*. Since Lemma 1 also applies to the subgraph relevant for a particular PE, we can expect that the graph shrinks fairly uniformly over the entire network.

We performed experiments on two different clusters at the KIT computing center both using compute-nodes with two quad-core processors each. Refer to [2] for details. We ran experiments with up 128 compute-nodes corresponding to 1024 cores with one MPI process per core.

Figure 3 illustrates how our distributed local max implementation scales for the random geometric graphs *rgg23* and *rgg24* (using random edge weights) which have fairly good locality. We plot the decrease in running time for successive doubling of p, i.e., a value of two stands for perfect relative speedup for this step and a value below one means that parallelization no longer helps.

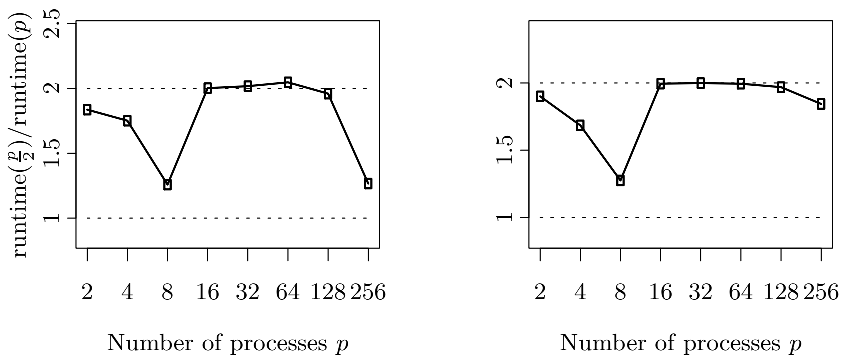


Fig. 3. Scaling results of the parallel local max algorithm on random geometric graphs with random edge weights. Left: rgg23 (≈63 million edges). Right: rgg24 (≈ 132 million edges).

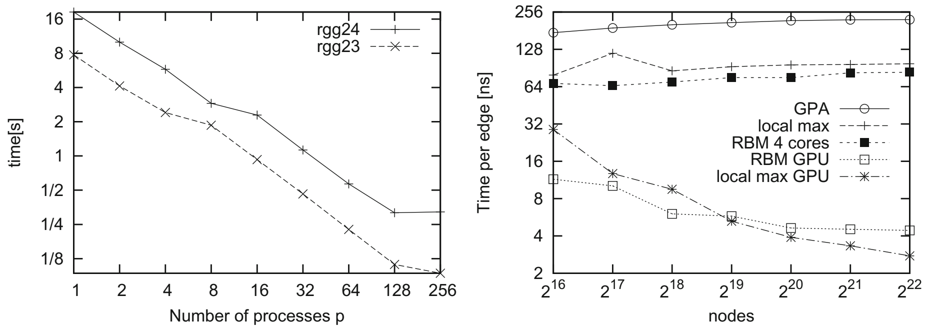


Fig. 4. Running time for distributed memory implementation on rgg23 and rgg24 (left). Time per edge of sequential and GPU algorithms for Delaunay instances (right).

We see values slightly below two for the steps $1 \to 2$ and $2 \to 4$ which is typical behavior of multicore algorithms when cores compete for resources like memory bandwidth. For $p = 8$ we start to use two compute-nodes (with 4 active cores each) and consequently we see the largest dip in efficiency. Beyond that, we have almost perfect scaling until the problem instance becomes too small. We have similar behavior for other graphs with good locality. For graphs with poor locality, efficiency is not very good. However the ratios stay above one for a very long time, i.e., it pays to use parallelism when it is available anyway. This is the situation we have when partitioning large graphs for use on massively parallel machines. Considering that the matching step in graph partitioning is often the least work intensive one in multi-level graph partitioning algorithms we conclude that local max might be a way to remove a sequential bottleneck from massively parallel graph partitioning. See Figure 4 (left) for the absolute timing and refer to [2] for additional data.

3.3 GPU Implementation

Our GPU algorithm is a fairly direct implementation of the CRCW algorithm. We reduce the algorithm to the basic primitives such as segmented prefix sum, prefix sum and random gather/scatter from/to GPU memory. As a basis for our implementation we use back40computing library by Merrill [21].

Figure 4 (right) compares the running time of our implementation with GPA, sequential local max, the RBM algorithm parallelized for 4 cores, and its GPU parallelization from [8]. While the CPU implementation has troubles recovering from its sequential inefficiency and is only slightly faster than even sequential local max, the GPU implementation is impressively fast in particular for small graphs. For large graphs, the GPU implementation of local max is faster. Since local max has better solution quality, we consider this a good result. Our GPU code is up to 35 times faster than sequential local max. We may also be able to learn from the implementation techniques of RBM GPU for small inputs in future work.

For random geometric graphs and random graphs, we get similar behavior (see [3] for details). The results for rgg are slightly worse for GPU local max – speedup is up to 24 over sequential local max and a speed advantage over GPU RBM only for the very largest inputs. As for random graphs, the denser the graph the larger is our speedup over the sequential and GPU RBM implementations. Thus, for $\alpha = 64$ our implementation is faster than GPU RBM already for $n = 2^{15}$. For $n = 2^{18}$ it is 65% faster than GPU RBM and 30 times faster than the sequential local max.

4 Conclusions and Future Work

The local max algorithm is a good choice for parallel or external computation of maximal and approximate maximum weight matchings. On the theoretical side it is provably efficient for computing maximal matchings and guarantees a 1/2-approximation. On the practical side it yields better quality at faster speed than several competitors including the greedy algorithm and RBM. Somewhat surprisingly it is even attractive as a sequential algorithm, outperforming HEM with respect to solution quality and other algorithms with respect to speed.

We have learned about the linear work algorithm by Blelloch et al. [4] from an anonymous reviewer during the review process. While our algorithm guarantees better expected asymptotic runtime, the practical results in [4] seem to be quite promising. However, lack of optimized shared memory implementation of our algorithm for multicores, use of different compilers and operating systems, and different set of test cases makes a thorough and fair comparison of the two algorithms unfeasible in the short period of time and is left for future work.

Many interesting questions remain. Can we omit re-randomization of edge weights when computing maximal matchings? The result of Blelloch et al. [4] partially answers this question by performing randomization only once at the expense of the performance guaratee. Is there a linear work parallel algorithm with polylogarithmic execution time that computes 1/2-approximations (or any other constant factor approximation). Can we even do 2/3-approximations with linear work in parallel [7,23]?

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM 31(9), 1116–1127 (1988)
2. Birn, M.: Engineering fast parallel matching algorithms. Diploma Thesis, Karlsruhe Institute of Technology (2012)
3. Birn, M., Osipov, V., Sanders, P., Schulz, C., Sitchinava, N.: Efficient parallel and external matching. CoRR, abs/1302.4587 (2013)
4. Blelloch, G.E., Fineman, J.T., Shun, J.: Greedy sequential maximal independent set and matching are parallel on average. In: SPAA, pp. 308–317 (2012)
5. Chiang, Y.-J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: SODA, pp. 139–149 (1995)

6. Davis, T.: The University of Florida Sparse Matrix Collection (2008), http://www.cise.ufl.edu/research/sparse/matrices
7. Drake, D.E., Hougardy, S.: Improved linear time approximation algorithms for weighted matchings. In: Arora, S., Jansen, K., Rolim, J.D.P., Sahai, A. (eds.) APPROX 2003+RANDOM 2003. LNCS, vol. 2764, pp. 14–23. Springer, Heidelberg (2003)
8. Fagginger Auer, B.O., Bisseling, R.H.: A GPU algorithm for greedy graph matching. In: Keller, R., Kramer, D., Weiss, J.-P. (eds.) Facing Multicore-Challenge II 2011. LNCS, vol. 7174, pp. 108–119. Springer, Heidelberg (2012)
9. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: FOCS, pp. 285–298 (1999)
10. Kumar, V., Karypis, G.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20(1), 359–392 (1998)
11. Goodrich, M.T., Sitchinava, N., Zhang, Q.: Sorting, searching and simulation in the mapreduce framework. In: Asano, T., Nakano, S.-I., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 374–383. Springer, Heidelberg (2011)
12. Hoepman, J.-H.: Simple distributed weighted matchings. CoRR, cs.DC/0410047 (2004)
13. Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a Scalable High Quality Graph Partitioner, pp. 1–12 (2010)
14. Israeli, A., Itai, A.: A fast and simple randomized parallel algorithm for maximal matching. Information Processing Letters 22(2), 77–80 (1986)
15. Karloff, H.J., Suri, S., Vassilvitskii, S.: A model of computation for mapreduce. In: SODA, pp. 938–948 (2010)
16. Karpinski, M., Rytter, W.: Fast parallel algorithms for graph matching problems, vol. 98. Clarendon Press (1998)
17. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM Journal on Computing 15(4), 1036–1053 (1986)
18. Manne, F., Bisseling, R.H.: A parallel approximation algorithm for the weighted maximum matching problem. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 708–717. Springer, Heidelberg (2008)
19. Maue, J., Sanders, P.: Engineering algorithms for approximate weighted matching. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 242–255. Springer, Heidelberg (2007)
20. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures — The Basic Toolbox. Springer (2008)
21. Merrill, D.: Back40computing: Fast and efficient software primitives for GPU computing, http://code.google.com/p/back40computing/
22. Yves, M., Robson, J.M., Nasser, S.-D., Zemmari, A.: An optimal bit complexity randomized distributed MIS algorithm (Extended abstract). In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 323–337. Springer, Heidelberg (2010)
23. Pettie, S., Sanders, P.: A simpler linear time $2/3 - \epsilon$ approximation for maximum weight matching. Technical Report MPI-I-2004-1-002, MPII (2004)
24. Preis, R.: Linear time $\frac{1}{2}$-approximation algorithm for maximum weighted matching in general graphs. In: Meinel, C., Tison, S. (eds.) STACS 1999. LNCS, vol. 1563, pp. 259–269. Springer, Heidelberg (1999)
25. Sanders, P., Schulz, C.: High Quality Graph Partitioning. In: Proceedings of the 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering, pp. 1–17. AMS (2013)

Splittable Single Source-Sink Routing on CMP Grids: A Sublinear Number of Paths Suffice

Adrian Kosowski and Przemysław Uznański

INRIA Bordeaux – Sud-Ouest, 33405 Talence CEDEX, France
{adrian.kosowski,przemyslaw.uznanski}@inria.fr

Abstract. In single chip multiprocessors (CMP) with grid topologies, a significant part of power consumption is attributed to communications between the cores of the grid. We investigate the problem of routing communications between CMP cores using shortest paths, in a model in which the power cost associated with activating a communication link at a transmission speed of f bytes/second is proportional to f^{α}, for some constant exponent $\alpha > 2$.

Our main result is a trade-off showing how the power required for communication in CMP grids depends on the ability to split communication requests between a given pair of node, routing each such request along multiple paths. For a pair of cores in a $m \times n$ grid, the number of available communication paths between them grows exponentially with n, m. By contrast, we show that optimal power consumption (up to constant factors) can be achieved by splitting each communication request into k paths, starting from a threshold value of $k = \Theta(n^{1/(\alpha-1)})$. This threshold is much smaller than n for typical values of $\alpha \approx 3$, and may be considered practically feasible for use in routing schemes on the grid. More generally, we provide efficient algorithms for routing multiple k-splittable communication requests between two cores in the grid, providing solutions within a constant approximation of the optimum cost. We support our results with algorithm simulations, showing that for practical instances, our approach using k-splittable requests leads to a power cost close to that of the optimal solution with arbitrarily splittable requests, starting from the stated threshold value of k.

1 Introduction

The increase in the level of integration of single chip multiprocessors (CMPs) creates demand for high-speed communication on-chip, which in turn increases the power consumption on CMP. This trend is predicted to continue in the future [7]. Numerous studies concern the optimization of power cost in integrated chip designs, taking into account that both processors and communication buses may operate at variable frequency, determining the speed of computations or transmissions (cf. [8,11,13,15]). The increase of power cost with the third power of workload in such designs is a well-established relation (cf. e.g. [3,6,15]).

A significant part of power in CMPs is consumed by maintaining communications within the chip, and that makes efficient allocation of communication routes a very important issue [14]. On CMP grids, links with dynamic frequency and/or voltage scaling are used ([12,16]), and the dissipated power P on a link is related to the frequency f and voltage V on it by the following relation supported by both theory and experiments $P \sim f \cdot V^2$ (cf. e.g. [2]). However, for most designs, an increase in operating frequency also results in an increase in voltage, roughly according to the relation $V \sim f$ (cf. e.g. [16]), which results in the relation between power cost and transmission speed given as $P \sim f^3$ ([4,5]). Such a model of power consumption was recently studied in the context of splittable Manhattan-path routing by Benoit *et al.* [4]. They introduced several routing schemes in an effort to minimize the total power cost, but observed that this may require the splitting of each communication request, and routing its fragments along a potentially very large number of communication paths. Splitting a request, taking care of the route for each part, and merging it at the target imposes additional time and power overhead.

In this work, our goal in this work is to show how to *limit path splitting as much as possible*, without excessively increasing communication power cost. Specifically, we consider the problem of optimizing the power consumption cost of communication between two given cores, which may sometimes require the routing of multiple requests. (Our scenario can also be seen as a rough approximation of the general case of multicore communication, under the simplifying assumption that the total communication rate due to communication between all pairs of cores other than the distinguished pair may be treated as the same for each link, and so excluded from optimization.) Our power consumption model assumes that if an edge is transmitting at rate f, the power cost of maintaining the frequency over an edge is proportional to f^{α} for a given constant $\alpha > 2$, identical for every edge. We make the practically-motivated assumption [12,16] that only the dynamic part (associated with transmission) is dominant for high communication rate, and static effects need not be considered in optimization.

Outline and Results. Our study concerns routing between a single source-sink pair of nodes using Manhattan paths on a grid CMP. Communication between these nodes is assumed to be static, i.e. constant over time, and the cost of a transmission along an edge is assumed to be proportional to a fixed power of the transmission rate. The considered model, power cost function, and rules of routing are formally presented in Section 2. We briefly outline the theory of Manhattan-path routing with arbitrarily splittable requests (Max-MP). We provide an optimal convex programming formulation of the problem, leading to a routing scheme denoted as OPT, and recall the properties of the $\mathcal{C}$ routing scheme introduced in [4]. We also provide a convenient formulation of Manhattan routings in terms of transmission through nodes.

Our main results are given in Section 3. They concern the variant of the Manhattan routing problem in which each request can be satisfied by at most k communication paths, where k is a parameter of the model (k-MP). We study the value of the ratio of the cost of the optimal solution in this case, denoted OPT_k, to the cost of the routing scheme OPT with arbitrarily splittable paths. We establish that in general, $\text{cost}(\text{OPT}_k)/\text{cost}(\text{OPT}) = O(1 + \frac{n}{k^{\alpha-1}})$, whereas for the special case of $d \geq 1$ identical requests of the same size, this ratio is given precisely as $\Theta(1 + \frac{n}{(kd)^{\alpha-1}})$. This means that for $k = o(n^{1/(\alpha-1)})$, the requirement that requests can be split into at most k paths impacts the cost of the routing scheme asymptotically, i.e., increases the cost by an unbounded factor for sufficiently large n. On the other hand, for k larger than the threshold value of $\Theta(n^{1/(\alpha-1)})$, the obtained k-splittable routings are within a constant factor of the optimal solution to Max-MP.

The proposed bounds are obtained through the analysis of three efficiently implementable algorithmic schemes for solving k-MP: $\mathcal{F}_k$ routing and $\mathcal{D}_k$ routing (for uniform requests), and $\mathcal{A}_k$ (for non-uniform requests). The latter two are shown to have a constant approximation ratio with respect to the cost of OPT_k for all k, while the former converges to the cost of OPT as k goes to infinity. The design of such approximate techniques results from the observation that OPT_k is NP-hard.

Finally, in Section 4, we perform a validation, using simulations, of the determined threshold value of $k = \Theta(n^{1/(\alpha-1)})$, showing the effect of smaller and larger values of k on the cost of the routing. We also experimentally compare the performance on $\mathcal{F}_k$ routing and $\mathcal{D}_k$ routing, studying their convergence to asymptotic behavior for increasing values of k and different values of the power cost exponent $\alpha \approx 3$.

2 Framework

Platform and Power Consumption Model. We model our platform as a grid graph on a set of $m \times n$ uniform nodes $V_{i,j}$, with $1 \leq i \leq m$ and $1 \leq j \leq n$. Without loss of generality, we assume that $m \geq n$. We will also assume for the purpose of analysis that the sides of the grid are of the same order of magnitude, i.e., $m = O(n)$. Nodes are connected by bidirectional edges. The horizontal edge $E_{i,j}$ connects $V_{i,j}$ and $V_{i,j+1}$ (for $1 \leq i \leq m$, $1 \leq j \leq n-1$), and the vertical edge $E'_{i,j}$ connects $V_{i,j}$ and $V_{i+1,j}$ ($1 \leq i \leq m-1$, $1 \leq j \leq n$), see Fig. 1 for an illustration.

The power consumed on each edge is closely related to the amount of data sent through this edge in a unit of time. To simplify the analysis of the model, we discard constant factors, and (following [4]) set the cost of transmission at rate x as $C(x) = x^{\alpha}$, where $\alpha > 2$ is an absolute constant of the model (it is reasonable to assume $\alpha \approx 3$).

Communication and Routing Rules. The study of routing with Manhattan-type paths (of shortest length) is motivated by practical concerns, in particular, the need to minimize communication latency, and to confine communications between nearby processors to a local area of the grid. For the purpose of the study of single source-sink communications, it is assumed that the source and target are placed in the opposite corners of the grid; for communications between a different pair of nodes, considerations can be restricted to the respective rectangular sub-grid.

A *routing R* of a single communication request of size s is a weighted set of paths, $\{(w_1, p_1), \ldots, (w_k, p_k)\}$, where each path p_i starts at the same source vertex $V_{1,1}$, and ends at the same target vertex $V_{m,n}$ in the opposite corner of the grid. The real-valued weights w_i satisfy $w_i \geq 0$ and $\sum_i w_i = s$. This definition of a routing extends naturally to a set of $d \geq 1$ requests, which may be *uniform* (with identical request size $s = K/d$), or *non-uniform* (with possibly distinct request sizes $s_1, \ldots, s_d$). Given a routing R, we define $R(e)$ as the size of the transmission going through an edge e, i.e.: $R(e) = \sum_{i:\, e \in p_i} w_i$. (We will use this notation accordingly for routings denoted by letters different from R.)

The *routing policy* is expressed by the bound k on the splitability of each request:

– In *k-Path Manhattan Routing (k-*MP*)*, communication for each request can be split into any number of $k' \leq k$ (partially overlapping) source-sink paths, where k is a parameter of the model.

– In *Max-Paths Manhattan Routing (*Max-MP*)*, the number of paths allowed for each request is unbounded ($k = +\infty$).

Problem Definition. For a given routing policy with parameter k and power coefficient α, we define our optimization problem as follows: *Given a $m \times n$ grid and a set of requests of sizes $(s_1, \dots, s_d)$, with $\sum_{i=1}^{d} s_i = K$, find a routing R of this set of requests minimizing the total power cost of transmission through all the edges of the grid, expressed by the cost function:*

$$\mathrm{cost}(R) = \sum_{i=1}^{m} \sum_{j=1}^{n-1} R(E_{i,j})^{\alpha} + \sum_{i=1}^{m-1} \sum_{j=1}^{n} R\left(E'_{i,j}\right)^{\alpha}.$$

Solution to Max-MP **Routing.** For Max-MP, the routing policy does not impose a bound on k. We will denote the optimal solution to Max-MP by OPT and use it as a reference for k-splittable routing algorithms. The adopted definition of routing cost leads directly to a convex-programming formulation of Max-MP routing, and thus applications of convex programming algorithms lead to polynomial-time schemes with arbitrarily good approximation of OPT (cf. e.g. [1,9] for a discussion of convex programming in the context of finding min-cost flows).

We remark on the following lower bound on the size of OPT. Consider any Max-MP routing which transmits requests of total size K. The edges adjacent to node $V_{1,1}$, i.e., $\{E_{1,1}, E'_{1,1}\}$, have to transmit requests of size K in total. It follows that:

$$\mathrm{cost}(\mathrm{OPT}) \geq \left(\tfrac{K}{2}\right)^{\alpha} = \Theta\left(K^{\alpha}\right). \tag{1}$$

Remarkably, as shown in [4], this lower bound is tight regardless of the size of the grid, since it can be achieved using a specific routing scheme. We will provide a definition of a scheme called $\mathcal{C}$ which has equivalent properties, but is described from a different perspective, based on load balancing on so-called *vertex diagonals*. We will then use this scheme as a starting point for schemes solving k-MP.

Our Approach: Load Balancing on Vertex Diagonals. In all of the routing schemes which we propose in this paper, we will attempt to perform "load balancing" of paths with respect to transmission through vertices rather than edges. Hence, in a similar fashion to the notation $R(e)$ for an edge e, we define $R(v)$ as the total transmission size going through a vertex v in routing R.

We introduce the notion of the l-th *vertex diagonal*, denoted as DV_l $(1 \leq l \leq n + m - 1)$ by splitting the set of vertices according to their distance from the source, as follows (see Fig. 1 for an illustration): $V_{i,j} \in \mathrm{DV}_l$, iff $i + j = l + 1$. Likewise, by the l-th *edge diagonal*, denoted DE_l $(1 \leq l \leq n + m - 2)$, we mean the set of edges connecting vertices from DV_l and DV_{l+1}, namely: $E_{i,j}, E'_{i,j} \in \mathrm{DE}_l$, iff $i + j = l + 1$.

We start by observing that the values of $R(v)$ uniquely determine the values of $R(e)$. This property will allow us to design routing schemes simply by setting $R(v)$ for all nodes.

Routing Scheme $\mathcal{C}$ for Max-MP. We define the routing scheme $\mathcal{C}$ for Max-MP by putting a limit on the transmission going through vertices. Since each diagonal of vertices has a total transmission of exactly K, we set an equal value of transmission for all vertices in the layer:

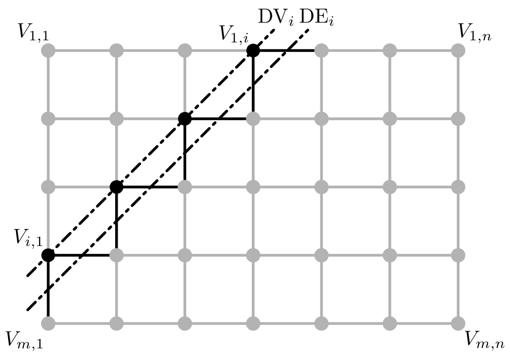


Fig. 1. Vertex diagonal DV_i and edge diagonal DE_i

$$\forall_{v \in \mathrm{DV}_j} \mathcal{C}(v) = \frac{K}{|\mathrm{DV}_j|}. \tag{2}$$

To verify that this routing is well-defined, we compute transfers over each edge based on the transfers on vertices (a complete implementation and analysis is provided in the Appendix[1]). Examples of transfers obtained using this algorithm are shown in Fig. 2.

The scheme $\mathcal{C}$ corresponds to the differently formulated algorithm studied in [4], where it was shown that it admits a constant approximation ratio for Max-MP.

Theorem 1 ([4]). $\mathrm{cost}(\mathcal{C}) = \Theta(K^\alpha) = \Theta(\mathrm{cost}(\textsc{Opt}))$.

Although such a solution has optimal, up to a constant factor, power cost, it can result in a single request being split into a large number of paths. Indeed, for a given graph $G = (V,E)$ and any flow f on G, f can be represented as the union of at most $|E|$ weighted paths. It follows that both OPT routing (computed through convex optimization) and $\mathcal{C}$ routing require $O(nm)$ splits per request. In the next section, we will show that it is possible to preserve a constant approximation ratio of the optimal cost, while using a much smaller number of splits, sublinear in the dimensions of the grid.

3 Schemes for k-Splittable Routing

In this section, we present three schemes for solving the k-Path Manhattan Routing problem (k-MP). The first two, denoted $\mathcal{F}_k$ and $\mathcal{D}_k$, are designed for uniform sets of requests. As the bound k on the number of allowed paths per request tends to infinity, these approaches will be shown to converge to the performance of schemes OPT and $\mathcal{C}$ for Max-MP, respectively. The third scheme, denoted $\mathcal{A}_k$, is an extension of $\mathcal{D}_k$ which also works for non-uniform sets of requests.

3.1 1-Splittable Routing with Uniform Requests

We start by considering the 1-MP routing policy, meaning that no splitting of requests is allowed. We can treat this problem as a discrete version of a continuous Max-MP prob-

[1] A full version of the paper is also available at: `http://hal.inria.fr/hal-00737611`

lem. First, we will consider uniform requests (of equal sizes); without loss of generality, we can assume that the input consists of d requests of size 1, each.

This considered problem can be solved by the flow-based $\mathcal{F}_1$ routing approach presented in Algorithm 1. The obtained solution is optimal, i.e., for uniform instances, we have $\text{cost}(\text{OPT}_1) = \text{cost}(\mathcal{F}_1)$. Moreover, using a classical min-cost flow algorithm, a $\mathcal{F}_1$ routing can be found in polynomial time with respect to parameters n, m, and d.

Algorithm 1. $\mathcal{F}_1$ routing scheme $\{$optimal solution to uniform 1-MP$\}$

```
Input: A set of $d$ unsplittable requests of size $s = 1$ in a $m \times n$ grid.
Solution:
  1. Construct a multigraph $G'$ such that $V(G') = V(G)$.
  2. For every directed edge $e \in E(G)$, add $d$ weighted directed edges to $G'$, having the same
     endpoints as $e$, and weights given as: $1^\alpha, 2^\alpha - 1^\alpha, \ldots, d^\alpha - (d-1)^\alpha$.
  3. Return the min-cost flow of size $d$ in $G'$, using the two opposite corners of the grid as the
     source and sink.
```

We will now provide asymptotic bounds on the size of the (optimal) solution to the uniform 1-MP problem. We obtain the lower bound by combining the lower bound for problem Max-MP (formula (1) with $K = d$), with an additional factor resulting from the discrete nature of 1-MP.

Lemma 1. *For every $R \in$ uniform 1-MP: $\text{cost}(R) = \Omega(d^\alpha) + \Omega(nd)$.*

(Proofs omitted due to space constraints are provided in the Appendix.)

To provide a complementary upper bound on the size of 1-MP routings, we do not analyze the optimal scheme $\mathcal{F}_1$, but instead propose an approximation scheme called $\mathcal{D}_1$ routing, which turns out to be easier to analyze.

We design the $\mathcal{D}_1$ routing through a discretization of the construction of $\mathcal{C}$ routing proposed in the previous section for Max-MP. Similarly to equation (2), we will place limits on the size of the transfer going through vertices. Consider the vertex diagonal DV_p with $1 \le p \le n+m-1$, and let $i = |\text{DV}_p|$. Suppose that the vertices of DV_p are ordered by decreasing first coordinate, as $\text{DV}_p = \{v_1, \ldots, v_i\}$. Then, for $1 \le j \le i$, we successively set $\mathcal{D}_1(v_j)$ so that at each step, the following condition holds: $\mathcal{D}_1(v_1) + \ldots + \mathcal{D}_1(v_j) = \lfloor d \cdot \frac{j}{i} \rfloor$. This is achieved by setting:

$$\mathcal{D}_1(v_j) = \left\lfloor d \cdot \tfrac{j}{i} \right\rfloor - \left\lfloor d \cdot \tfrac{j-1}{i} \right\rfloor . \tag{3}$$

To verify the correctness of this construction, we deduce transfer values over vertical and horizontal edges from values over vertices; a formal implementation of $\mathcal{D}_1$ routing is provided in Algorithm 2. An exemplary comparison of the vertex and edge transfers for $\mathcal{C}$ routing and $\mathcal{D}_1$ routing is shown in Fig. 2.

We start the analysis of the cost of $\mathcal{D}_1$ routing with the following lemma.

Lemma 2. *Let DV be an arbitrary vertex diagonal, and let $|\text{DV}| = i$. Then:*

$$\sum_{v \in DV} \mathcal{D}_1(v)^\alpha = \begin{cases} i\left(\left(\frac{d}{i}\right)^\alpha + O\left(\left(\frac{d}{i}\right)^{\alpha-2}\right)\right), & \text{for } i < d \\ d, & \text{for } i \ge d. \end{cases}$$

Algorithm 2. $\mathcal{D}_1$ routing scheme {for uniform 1-MP}

Input: A set of d unsplittable requests of size $s = 1$ in a $m \times n$ grid.
Solution: For each diagonal DE$_j$ of the grid, $1 \le j < n+m$, set the flow on its successive horizontal edges e_i and vertical edges e'_i, $1 \le i \le j$, as follows:

– If $1 \le j < n$, set:

$$\mathcal{D}_1(e_i) = \left\lfloor d\tfrac{i}{j} \right\rfloor - \left\lfloor d\tfrac{i}{j+1} \right\rfloor, \qquad \mathcal{D}_1(e'_i) = \left\lfloor d\tfrac{i}{j+1} \right\rfloor - \left\lfloor d\tfrac{i-1}{j} \right\rfloor.$$

– If $n \le j < m$, set:

$$\mathcal{D}_1(e_i) = \left\lfloor d\tfrac{i}{n} \right\rfloor - \left\lfloor d\tfrac{i-1}{n} \right\rfloor, \qquad \mathcal{D}_1(e'_i) = 0.$$

– If $m \le j < n+m$, set:

$$\mathcal{D}_1(e_i) = \left\lfloor d\tfrac{i}{n+m-j} \right\rfloor - \left\lfloor d\tfrac{i-1}{n+m-j-1} \right\rfloor, \qquad \mathcal{D}_1(e'_i) = \left\lfloor d\tfrac{i}{n+m-j-1} \right\rfloor - \left\lfloor d\tfrac{i}{n+m-j} \right\rfloor.$$

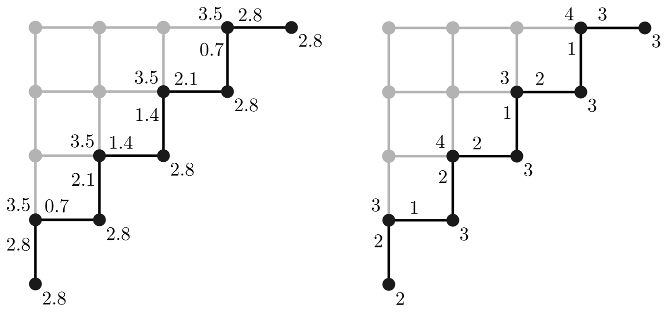


Fig. 2. Comparison of transfer values over one diagonal for a $\mathcal{C}$ routing with $K = 14$ (on the left) and a $\mathcal{D}_1$ routing with $d = 14$ (on the right)

Using the above lemma, we compute the cost of a $\mathcal{D}_1$ routing as cost$(\mathcal{D}_1) = \Theta(d^\alpha) + \Theta(nd)$. By Lemma 1, this cost is asymptotically the best possible for 1-MP.

Theorem 2. *For a uniform set of d requests (with total size $K = d$):*

$$\text{cost}(\mathcal{D}_1) = \Theta(d^\alpha) + \Theta(nd) = \text{cost}(\mathcal{F}_1).$$

3.2 k-Splittable Routing with Uniform Requests

We now proceed to extend our results from the previous section to the case of k-MP uniform routing. We will consider sets of d requests of total size K, i.e., of size K/d each. A natural generalization of $\mathcal{D}_1$ routing, called $\mathcal{D}_k$ routing, is presented in Algorithm 3.

Algorithm 3. $\mathcal{D}_k$ routing scheme {for uniform k-MP}

Input: A set of d k-splittable requests, of size K/d each, in a $m \times n$ grid.
Solution: Split each of the requests into k smaller ones, each of size $\frac{K}{kd}$. Return the $\mathcal{D}_1$ routing of this new set of requests.

Since in a $\mathcal{D}_k$ routing, we split the transmission of each request equally along its k paths, the cost of such a routing is the same as that of a $\mathcal{D}_1$ routing on the extended set of kd requests of size $\frac{K}{kd}$ each. Hence, the following result follows directly from Theorem 2 by a scaling argument: $\text{cost}(\mathcal{D}_k) = \Theta(K^\alpha) + \Theta(K^\alpha \frac{n}{(kd)^{\alpha-1}})$. Next, we show that although $\mathcal{D}_k$ only splits requests into paths of equal weight, one cannot achieve a better asymptotic result by using unequal splits, i.e., for any $R \in$ uniform k-MP : $\text{cost}(R) = \Omega(K^\alpha) + \Omega(K^\alpha \frac{n}{(kd)^{\alpha-1}})$. Combining these results, we obtain the following theorem, stating the optimality of $\mathcal{D}_k$ in the class of k-splittable routings.

Theorem 3. *For a uniform set of d requests with total size K:* $\text{cost}(\mathcal{D}_k) = \Theta(K^\alpha) + \Theta\left(K^\alpha \frac{n}{(kd)^{\alpha-1}}\right) = \text{cost}(\text{OPT}_k)$, *where* OPT_k *denotes the optimal cost solution to the considered set of requests for k-*MP.

Combining the bound on $\text{cost}(\text{OPT}_k)$ in the above Theorem with the bound on $\text{cost}(\text{OPT})$ in Theorem 1 for Max-MP routing, we obtain our main result: the threshold value of k for which imposing a limit of k into which each request can be split does not affect the asymptotics of power cost.

Theorem 4. *For uniform requests, imposing a routing policy with a split limit of* $k = \Theta\left(\frac{1}{d} \cdot n^{\frac{1}{\alpha-1}}\right)$ *does not affect the power cost, i.e.:* $\text{cost}(\text{OPT}_k) = \Theta(\text{cost}(\text{OPT}))$.

We end this subsection with a remark on the asymptotic behavior of the considered routing schemes for uniform instances, when $k \to +\infty$. Taking into account Theorems 1 and 3, we obtain:

Proposition 1. *For a grid of fixed dimension:* $\lim_{k\to+\infty} \text{cost}(\mathcal{D}_k) = \text{cost}(\mathcal{C})$.

Since, in general $\text{cost}(\mathcal{C}) > \text{cost}(\text{OPT})$, it is natural to ask for a different routing schemes for k-MP with improved limit behavior. A natural candidate is $\mathcal{F}_k$ routing, obtained by a natural generalization of $\mathcal{F}_1$ routing, as given by Algorithm 4.

Algorithm 4. $\mathcal{F}_k$ routing scheme {for uniform k-MP}

Input: A set of d k-splittable requests, of size K/d each, in a $m \times n$ grid.
Solution: Split each of the requests into k smaller ones, each of size $\frac{K}{kd}$. Return the $\mathcal{F}_1$ routing of the new set of requests.

This algorithm turns out to by asymptotically optimal as $k \to +\infty$.

Proposition 2. *For a grid of fixed dimension:* $\lim_{k\to+\infty} \text{cost}(\mathcal{F}_k) = \text{cost}(\text{OPT})$.

3.3 k-Splittable Routing with Non-uniform Requests

We close our considerations with a discussion of the general (non-uniform) case, where no assumptions are made about the sizes of the routed requests. We first observe that the considered problem is computationally hard.

Theorem 5. *The following decision version of non-uniform* 1-MP *routing is NP-complete: "Given* $(n,m,K=(K_1,\ldots,K_i),C,\alpha)$*, decide if it is possible to perform 1-MP routing with cost* $\leq C$*."*

Despite the hardness of the studied problem, one can try to look for approximate solutions. Note that applying $\mathcal{D}_k$ routing naively to a set of non-uniform requests could lead to excessively large additional cost. However, by applying a careful modification of $\mathcal{D}_k$ routing, called the $\mathcal{A}_k$ routing scheme (Algorithm 5), we obtain a good tool for routing non-uniform requests on the grid.

Theorem 6. *For non-uniform requests, $\mathcal{A}_k$ finds a solution to k-*MP *whose cost is within a constant factor of the optimum k-splittable routing:* $\mathrm{cost}(\mathcal{A}_k) = \Theta(\mathrm{cost}(\textsc{Opt}_k))$.

Algorithm 5. $\mathcal{A}_k$ routing scheme $\{$for non-uniform k-MP$\}$

```
Input: A set of $d$ $k$-splittable requests, of given sizes $S = (s_1, s_2, \ldots, s_d)$ (with $1 = s_1 \leq s_2 \leq \ldots \leq s_d$), in a $m \times n$ grid.
Solution:
1. Partition the set of request sizes into the union of disjoint subsets, $S = S_0 \cup S_1 \cup \ldots$, such that $\forall_{s \in S_i}\ 2^i \leq s < 2^{i+1}$.
2. For all non-empty sets $S_i$:
   – Find a $\mathcal{D}_k$ routing for the uniform instance consisting of $|S_i|$ requests of size $2^{i+1}$ each.
   – For all $1 \leq j \leq |S_i|$, route the $j$-th input request belonging to $S_i$ using the paths assigned to the $j$-th request in the corresponding $\mathcal{D}_k$ routing.
```

We end this section with a similar threshold theorem as Theorem 4 for the uniform case, obtaining bounds on value of k for which a split limit of k no longer affects the asymptotics of the cost of the routing. However, in this case the threshold depends on the structure of the set of requests, hence we only provide lower and upper bounds.

Theorem 7. *For non-uniform requests, imposing a routing policy with a split limit of k:*

1. *does not affect the asymptotic power cost (i.e.* $\mathrm{cost}(\textsc{Opt}_k) = \Theta(\mathrm{cost}(\textsc{Opt}))$*) when* $k = \Omega\left(n^{\frac{1}{\alpha-1}}\right)$,
2. *always increases the asymptotic power cost (i.e.* $\mathrm{cost}(\textsc{Opt}_k) = \omega(\mathrm{cost}(\textsc{Opt}))$*) when* $k = o\left(\frac{1}{d} \cdot n^{\frac{1}{\alpha-1}}\right)$.

4 Simulations Results

In this section we provide the results of experimental evaluation, through simulations, of the algorithms presented in the previous section. We analyze the effect of n, k and α on the efficiency of solutions found for k-MP routing of instances with uniform (identical-size) requests. Throughout the section, we choose the number of requests as $d = 1$ (for uniform instances, other values result only in a scaling factor for k in k-MP, and do not affect Max-MP).

We focus on the approximation ratio, looking at the cost of the routing obtained using the two schemes designed for uniform k-MP ($\mathcal{D}_k$, $\mathcal{F}_k$), relative to the cost of the optimal solution Opt to Max-MP, which is treated as the reference solution. In some graphs, we also provide the cost of the sub-optimal Max-MP routing $\mathcal{C}$ as an additional reference. Keep in mind, that both $\mathcal{C}$ and Opt use as much as $\Theta(nm)$ routing paths.

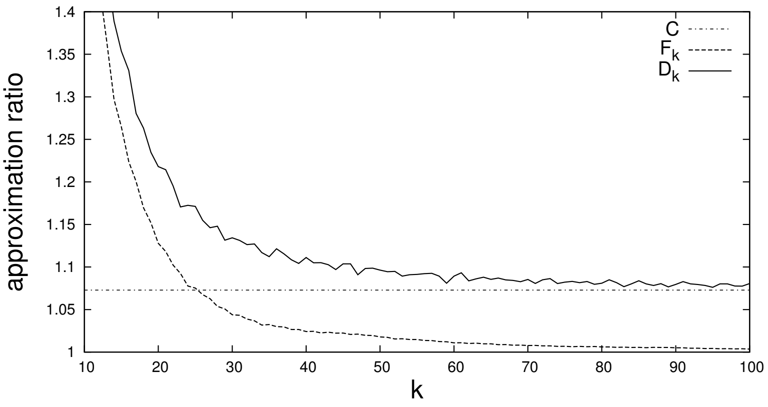


Fig. 3. Effect of k on the cost of $\mathcal{F}_k$ and $\mathcal{D}_k$ routing for a 30×30 grid

We recall that the value of cost of the optimal solution to Max-MP, $\text{cost}(\text{OPT}_k)$, is bounded from below by $\text{cost}(\text{OPT})$, and from above by both $\text{cost}(\mathcal{D}_k)$ and $\text{cost}(\mathcal{F}_k)$.

The implementation and tests were performed within a software package, written by the authors for this purpose in GNU C++. The min-cost flow subroutines were implemented using the standard cycle-canceling method [10] with some optimizations for faster performance. The presented results of the tests are deterministic and fully reproducible, independent of the test environment and the details of the implementation of the flow algorithms.

Impact of k on the Routing Cost. We begin by studying the approximation ratio of algorithms $\mathcal{F}_k$ and $\mathcal{D}_k$ for increasing values of k, the allowed number of splits of each requests. In the first plot (Fig. 3), we fix the dimensions of the grid $n,m = 30$, model power cost exponent $\alpha = 2.5$, plotting the values of $\text{cost}(\mathcal{F}_k)/\text{cost}(\text{OPT})$ and $\text{cost}(\mathcal{D}_k)/\text{cost}(\text{OPT})$ for k in the range $k \in [10, 100]$. For reference, we also provide the approximation ratio of $\mathcal{C}$ routing for the studied instance.

We observe that, as predicted by theory (Propositions 1 and 2), $\lim_{k\to+\infty} \text{cost}(\mathcal{F}_k) = \text{cost}(\text{OPT})$ and $\lim_{k\to+\infty} \text{cost}(\mathcal{D}_k) = \text{cost}(\mathcal{C})$, and the respective costs converge to their limits quickly, reaching a point 10% over the respective limit already for $k < n$. In general, the convergence need not be monotone for either of the approximation algorithms, since partitioning a request into $k+1$ equally-weighted paths may give worse results than partitioning it into k equally-weighted paths.

In our second plot (Fig. 4), we present more compelling evidence of the relation $k = \Theta\left(n^{1/(\alpha-1)}\right)$ for the threshold split value resulting in asymptotically-optimal cost, derived theoretically as Theorem 4. Once again, we choose model parameter $\alpha = 2.5$. In the experiment, we consider square grids of increasing size in the range $n = m \in [10, 120]$, testing three different relations between n and k ($k = \lfloor 2n^{1/2} \rfloor$, $k = \lfloor \frac{3}{2} n^{2/3} \rfloor$, $k = n$). For each of these relations, we plot the approximation ratios $\text{cost}(\mathcal{F}_k)/\text{cost}(\text{OPT})$ and $\text{cost}(\mathcal{D}_k)/\text{cost}(\text{OPT})$. Based on the plot, we can presume that:

– For the relation $k \sim n^{1/2}$, we have in the limit:

$$\text{cost}(\mathcal{F}_k)/\text{cost}(\text{OPT}) \to +\infty, \qquad \text{cost}(\mathcal{D}_k)/\text{cost}(\text{OPT}) \to +\infty.$$

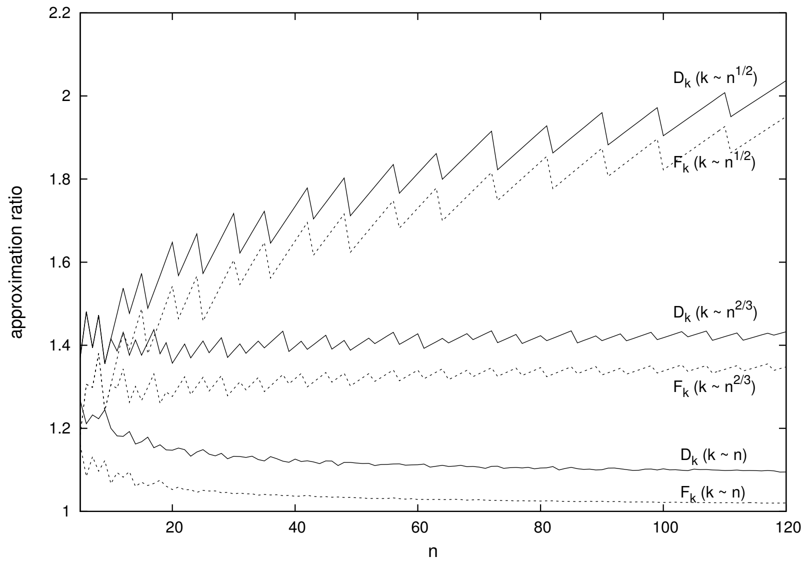


Fig. 4. Approximation ratio for $\mathcal{F}_k$ and $\mathcal{D}_k$ routing with split parameter $k \sim n^{\beta}$, for β greater, equal and smaller than $1/(\alpha-1)$. $[\alpha = 2.5]$

– For the relation $k \sim n^{2/3}$, we have in the limit:

$$\mathrm{cost}(\mathcal{F}_k)/\mathrm{cost}(\textsc{Opt}) \to const, \qquad \mathrm{cost}(\mathcal{D}_k)/\mathrm{cost}(\textsc{Opt}) \to const.$$

– For the relation $k \sim n$, we have in the limit:

$$\mathrm{cost}(\mathcal{F}_k)/\mathrm{cost}(\textsc{Opt}) \to 1, \qquad \mathrm{cost}(\mathcal{D}_k)/\mathrm{cost}(\textsc{Opt}) \to const.$$

We remark that the relation $k \sim n^{2/3}$ precisely corresponds to the threshold exponent $1/(\alpha-1) = 2/3$ for the considered value of α. Thus, the limit behavior of all the algorithms is consistent with the theory derived in the previous section. We note that the cost achieved by both $\mathcal{F}_k$ routing and $\mathcal{D}_k$ routing is highly satisfactory, and that the performance of $\mathcal{F}_k$ routing proves to be superior to $\mathcal{D}_k$ in all of the performed tests.

Effect of Power Exponent α. In auxiliary experiments, we studied the effect of the power exponent α (which is a constant of the model) on the required threshold value of split parameter k. We tested the rate of convergence of the approximation ratio $\mathrm{cost}(\mathcal{F}_k)/\mathrm{cost}(\textsc{Opt})$ to 1 in a grid of dimensions $n = m = 30$ for three different values of the power exponent, $\alpha \in \{2.5, 3, 3.5\}$. It was observed that the convergence is faster for larger values of α. This is consistent with the theoretical threshold, $k = \Theta\left(n^{1/(\alpha-1)}\right)$, whose growth rate decreases with the increase of α.

5 Conclusions

The contribution of our study is twofold. On the one hand, we advance the theory of splitting of requests in Manhattan routing on the grid, and point out that in practice, only a relatively small number of splits per request will be beneficial from a power-cost perspective. On the other hand, we propose efficient approximation schemes for such a k-path routing problem. Simulations provide evidence that corroborates the theoretical results, showing that the designed algorithms lead to routings with a cost which is, in practice, even superior to that resulting from our theoretical bounds.

In future work, it would be beneficial to improve the constant bounds on the approximation ratios of our algorithms, establishing more tightly their dependence on the power exponent α. Another promising direction of study would extend our results to routing requests between multiple sources and targets on the grid. Such a study would have a purely experimental nature, since the thresholds which appear in multi-core communication scenarios are difficult to capture theoretically, depending on the observed traffic patterns.

References

1. Ahuja, R.K., Hochbaum, D.S., Orlin, J.B.: Solving the convex cost integer dual network flow problem. In: IPCO 1999. LNCS, vol. 1610, pp. 31–44. Springer, Heidelberg (1999)
2. Andrei, A., Schmitz, M.T., Eles, P., Peng, Z., Al-Hashimi, B.M.: Simultaneous communication and processor voltage scaling for dynamic and leakage energy reduction in time-constrained systems. In: Proc. Int. Conf. on Computer-Aided Design, pp. 361–367. IEEE (2004)
3. Aydin, H., Yang, Q.: Energy-aware partitioning for multiprocessor real-time systems. In: Proc. Int. Parallel and Distributed Processing Symp., p. 113. IEEE (2003)
4. Benoit, A., Melhem, R.G., Renaud-Goud, P., Robert, Y.: Power-aware manhattan routing on chip multiprocessors. In: Proc. Int. Parallel and Distributed Processing Symp., pp. 189–200. IEEE (2012)
5. Benoit, A., Renaud-Goud, P., Robert, Y.: On the performance of greedy algorithms for power consumption minimization. In: Proc. Int. Conf. on Parallel Processing, pp. 454–463. IEEE (2011)
6. Chen, J.-J., Kuo, T.-W.: Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics. In: Proc. Int. Conf. on Parallel Processing, pp. 13–20. IEEE (2005)
7. Blake, R.D.G., Mudge, T.: A survey of multicore processors. Signal Processing Magazine 26(6), 26–37 (2009)
8. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: Proc. Int. Symp. on Low Power Electronics and Design, pp. 197–202 (1998)
9. Karzanov, A.V., McCormick, S.T.: Polynomial methods for separable convex optimization in unimodular linear spaces with applications. SIAM J. on Computing 26(4), 1245–1275 (1997)
10. Klein, M.: A primal method for minimal cost flows. Management Sci. 14, 205–220 (1967)
11. Langen, P., Juurlink, B.: Leakage-aware multiprocessor scheduling. J. of Signal Processing Systems 57, 73–88 (2009)

12. Lee, S.E., Bagherzadeh, N.: A variable frequency link for a power-aware network-on-chip (NoC). Integration 42(4), 479–485 (2009)
13. Mishra, R., Rastogi, N., Zhu, D., Mossé, D., Melhem, R.: Energy aware scheduling for distributed real-time systems. In: Proc. Int. Parallel and Distributed Processing Symp., p. 21 (2003)
14. Owens, J.D., Dally, W.J., Ho, R., Jayasimha, D.J., Keckler, S.W., Peh, L.-S.: Research challenges for on-chip interconnection networks. IEEE Micro 27, 96–108 (2007)
15. Pruhs, K., van Stee, R., Uthaisombut, P.: Speed scaling of tasks with precedence constraints. In: Erlebach, T., Persinao, G. (eds.) WAOA 2005. LNCS, vol. 3879, pp. 307–319. Springer, Heidelberg (2006)
16. Shang, L., Peh, L.-S., Jha, N.K.: Dynamic voltage scaling with links for power optimization of interconnection networks. In: Proc. Int. Symp. on High-Performance Computer Architecture, pp. 91–102. IEEE (2003)

Topic 13: High-Performance Networks and Communication

(Introduction)

Olav Lysne, Torsten Hoefler, Pedro López, and Davide Bertozzi

Topic Committee

Networks have always been a central component in the realization of parallel processing. Having multiple processors working concurrently to solve a problem inherently means that there has to be a reliable and well-performing network over which the processors can communicate. Ever since the first parallel machines we have therefore seen research on topics like topologies, switches, links, routing, device drivers, traffic control, fault tolerance, and congestion control specialized for parallel machines.

In over the last decade or so, parallel processing has moved out of the dedicated supercomputers and into main stream computing. This means that the study of high performance networks and communication as vehicles for parallelism has become a much broader field. It now ranges from communication solutions between components on the same chip, via traditional interconnection networks in clusters and supercomputers, to datacenter networks in cloud facilities, and finally to nation and continent spanning internet-based protocols.

The submission to this years track on "high performance networks and communication" mirror this development by covering a broad set of topics. We also think that the papers selected for presentation represents a good view of the breadth that this area has grown into. We have one paper on switch design that studies buffer usage, and another on a technology to combine multiple InfiniBand networks together through routers. There is a contribution on optical switching and another on protocol tuning for data-transfers over longer distances.

There are two groups of people that deserves special thanks for their effort to this track. The first are the authors of the submitted papers. There was unfortunately not enough room for all of them in the program, but we have read through a lot of interesting work that we expect to see published elsewhere in the near future. The second group are the reviewers. Countless hours of work from field experts have gone into identifying ways in which the submitted papers could be improved, and selecting the papers that finally went into the program. We are grateful for the efforts from all of them.

Making the Network Scalable: Inter-subnet Routing in InfiniBand

Bartosz Bogdański[1], Bjørn Dag Johnsen[1], Sven-Arne Reinemo[2], and José Flich[3]

[1] Oracle Corporation, Oslo, Norway
{bartosz.bogdanski,bjorn-dag.johnsen}@oracle.com
[2] Simula Research Laboratory, Lysaker, Norway
svenar@simula.no
[3] Universidad Politécnica de Valencia, Valencia, Spain
jflich@disca.upv.es

Abstract. As InfiniBand clusters grow in size and complexity, the need arises to segment the network into manageable sections. Up until now, InfiniBand routers have not been used extensively and little research has been done to accommodate them. However, the limits imposed on local addressing space, inability to logically segment fabrics, long reconfiguration times for large fabrics in case of faults, and, finally, performance issues when interconnecting large clusters, have rekindled the industry's interest into IB-IB routers. In this paper, we examine the routing problems that exist in the current implementation of OpenSM and we introduce two new routing algorithms for inter-subnet IB routing. We evaluate the performance of our routing algorithms against the current solution and we show an improvement of up to 100 times that of OpenSM.

1 Introduction

Until recently, the need for routers in InfiniBand (IB) networks was not evident and all the essential routing and forwarding functions were performed by layer-2 switches. However, with the increased complexity of the clusters, the need for routers becomes more obvious, and leads to more discussion about native IB routing [1,2,3]. Obsidian Research was the first company to see the need for routing between multiple subnets, and provided the first hardware to do that in 2006 [4].

There are several reasons for using routing between IB subnets with the two main being address space scalability and fabric management containment. Address space scalability is an issue for large installations whose size is limited by the number of available *local identifiers* (LIDs). Hosts and switches within a subnet are addressed using LIDs and a single subnet is limited to 49151 unicast LIDs. If more end-ports are required, then the only option is to combine multiple subnets by using one or more IB routers. Because LID addresses have local visibility, they can be reused in the subnets connected by routers, which theoretically yields an unlimited addressing space. It is worth observing that there are multiple suggestions to expand the address space of IB without introducing routers. One of the more mature proposals aims at extending the LID addressing space to 32 bits [5], however, it would not be backward compatible with older hardware, which limits its usability.

Fabric management containment has three major benefits: 1) fault isolation, 2) increased security, and 3) intra-subnet routing flexibility. First, by dividing a large subnet into several smaller ones, faults or topology changes are contained to a single subnet and the subnet reconfiguration will not pass through a router to other subnets. This shortens the reconfiguration time and limits the impact of a fault. Second, from a security point of view, segmenting a large fabric into subnets using routers means that the scope of most attacks is limited to the attacked subnet [6]. Third, from a routing point of view, fabric management containment leads to more flexible routing schemes. This is particularly advantageous in case of a hybrid fabric that consists of two or more regular topologies. For example, a network may consist of a fat-tree part interconnected with a mesh or a torus part (or any other regular topology). The problem with managing this in a single subnet is that it is not straightforward to route each part of the subnet separately because intra-subnet routing algorithms have a subnet scope. Moreover, there are no general purpose agnostic routing algorithms for IB that will provide optimal performance for a hybrid topology. However, if a hybrid topology is divided into smaller regular subnets then each subnet can be routed using a different routing algorithm that is optimized for a particular subnet. For example, a fat-tree routing algorithm could route the fat-tree part and the dimension-order routing could route the mesh part of the topology. This is because each subnet can run its own subnet manager (SM) that configures only the ports on the local subnet and routers are non-transparent to the subnet manager.

In this paper, we present two inter-subnet routing algorithms for IB. The first one, *inter-subnet source routing* (ISSR), is an agnostic algorithm for interconnecting any type of topology. The second one is fat-tree specific and only interconnects two or more fat-trees. With these algorithms we solve two problems: how to optimally choose a local router port for a remote destination and how to best route from the router to the destination. We compare the algorithms against the solution that is implemented in OpenSM. Inter-subnet routing in OpenSM is at the time of writing very limited, the configuration is tedious and the performance is only usable for achieving connectivity - not for high performance communication between multiple sources and destinations [2]. It is, however, the only available inter-subnet routing method for IB.

The rest of this paper is organized as follows: we discuss related work in Sect. 2, and we introduce the IB Architecture in Sect. 3. We follow with a description of our proposed layer-3 routing for IB in Sect. 4. Next, we continue with a presentation and discussion of our results in Sect. 5. Finally, we conclude in Sect. 6.

2 Related Work

Obsidian Strategics was the first company to demonstrate a device marketed as an IB-IB router (the Longbow XR) in 2006 [4]. That system highlighted the need for subnet isolation through native IB-IB routing. The Longbow XR featured a content-addressable memory for fast address resolution and supported up to 64k routes. The drawbacks of the router included a single 4x SDR link, and its primary application was disaster recovery - it was aimed at interconnecting IB subnets spanning large distances as a range extender. Furthermore, the Longbow XR appears to the subnet manager as a transparent switch, so the interconnected subnets are merged together into one large subnet. When releasing the router, Obsidian argued that while the IB specification 1.0.a defines the router hardware well, the details of subnet management interaction (like routing) are not fully addressed. This argument is still valid for the current release of the specification [7]. In 2007, Prescott and Taylor verified how range extension in IB works for campus area and wide area networks [8]. They demonstrated that it is possible to achieve high performance when using routers to build IB wide area networks. However, they did not mention the deadlock issues that can occur when merging subnets, and they only focused on remote traffic even though local traffic can be negatively affected by suboptimal routing in such a hybrid fabric. In 2008, Southwell presented how native IB-IB routers could be used in System Area Networks [1]. He argued that IB could evolve from being an HPC-oriented technology into a strong candidate for future distributed data center applications or campus area grids. While the need for native IB-IB routing was well-demonstrated, Southwell did not address the routing, addressing and deadlock issues. In 2011, Richling et al. [9] addressed the operational and management issues when interconnecting two clusters over a distance of 28 kilometers. They described the setup of hardware and networking components, and the encountered integration problems. However, they focus on IB-IB routing in the context of range extension and not on inter-subnet routing between local subnets.

When reviewing the literature, we noticed that the studies of native IB-IB routing is focused on disaster recovery and interconnection of wide area IB networks. Our work explores the foundations of native IB-IB routing in the context of performance and features in inter-subnet routing between local subnets. Furthermore, we assume full compliance with the IB specification and we deal with issues previously not mentioned including the deadlock problem and path distribution.

3 The InfiniBand Architecture

InfiniBand is a serial point-to-point full-duplex interconnection network technology, and was first standardized in October 2000 [7]. The current trend is that IB is replacing proprietary or low-performance solutions in the high performance computing domain [10], where high bandwidth and low latency are the key requirements. The de facto system software for IB is OFED developed by dedicated professionals and maintained by the OpenFabrics Alliance [5].

Every IB subnet requires at least one subnet manager (SM), which is responsible for initializing and bringing up the network, including the configuration of all the IB ports residing on switches, routers, and host channel adapters (HCAs) in the subnet. At the time of initialization the SM starts in the *discovering state* where it does a sweep of the network in order to discover all switches and hosts. During this phase it will also discover any other SMs present and negotiate who should be the master SM. When this phase is completed the SM enters the *master state*. In this state, it proceeds with LID assignment, switch configuration, routing table calculations and deployment, and port configuration. At this point the subnet is up and ready for use. After the subnet has been configured, the SM is responsible for monitoring the network for changes.

A major part of the SM's responsibility is routing table calculations. Routing of the network aims at obtaining full connectivity, deadlock freedom, and proper load balancing between all source and destination pairs in the local subnet. Routing tables must be calculated at network initialization time and this process must be repeated whenever the topology changes in order to update the routing tables and ensure optimal performance. Despite being specific about intra-subnet routing, the IB specification does not say much about inter-subnet routing and leaves the details of the implementation to the vendors.

IB is a lossless networking technology, and under certain conditions it may be prone to *deadlocks* [11,12]. Deadlocks occur because network resources such as buffers or channels are shared and because packet drops are usually not allowed in lossless networks. The IB specification explicitly forbids IB-IB routers to cause a deadlock in the fabric irrespective of the congestion policy associated with the inter-subnet routing function. Designing a generalized deadlock-free inter-subnet routing algorithm where the local subnets are arbitrary topologies is challenging. In this paper we limit our scope to fat-tree topologies and by making sure our routing functions use only the standard up/down routing mechanism, we eliminate the deadlock problem.

3.1 Native InfiniBand Routers

The InfiniBand Architecture (IBA) supports a two-layer topological division. At the lower layer, IB networks are referred to as subnets, where a subnet consists of a set of hosts interconnected using switches and point-to-point links. At the higher level, an IB fabric constitutes one or more subnets, which are interconnected using routers. Hosts and switches within a subnet are addressed using LIDs and a single subnet is limited to 49151 LIDs. LIDs are local addresses valid only within a subnet, but each IB device also has a 64-bit *global unique identifier* (GUID) burned into its non-volatile memory. A GUID is used to form a GID - an IB layer-3 address. A GID is created by concatenating a 64-bit subnet ID with the 64-bit GUID to form an IPv6-like 128-bit address. In this paper, we when using the term GUID we mean a port GUIDs, i.e. the GUIDs assigned to every port in the IB fabric.

IB-IB routers operate at the layer-3 of IB addressing hierarchy and their function is to interconnect layer-2 subnets as shown in Fig. 1(a). A thorough description of the inter-subnet routing scheme is currently out of scope of the IBA specification and much freedom is given to the router vendors when implementing inter-subnet routing. The inter-subnet routing process defined in the IBA specification is similar to the routing in TCP/IP networks. First, if an end-node want to send a packet to another subnet, the address resolution makes the local router visible to that end-node. The end-node puts the local router's LID address in the *local routing header* (LRH) and the final destination address (GID) in the *global routing header* fields. When the packet reaches a router, the packet fields are replaced (the source LID is replaced with the LID of the router's egress port, the destination LID is replaced with the LID of the next-hop port, and CRCs are recomputed) and the packet is forwarded to the next hop. The pseudo code for the rest of the packet relay model is described in [7] on page 1082. In this paper, we will only consider topologies similar to that presented in Fig. 1(a), i.e. cases where one or more subnets are directly connected using routers. Furthermore, each subnet must be a fat-tree topology and it must be directly attached to the other subnets without any transit subnets in between.

4 Layer-3 Routing in InfiniBand

Up until now, IB-IB routers were considered to be superfluous. Even the concept of *routing*, which in IP networks strictly refers to layer-3 routers, in IB was informally applied to forwarding done by layer-2 switches that process packets based only on their LID addresses. With the increasing size and complexity of subnets the need for routers has become more evident. There are two major problems with inter-subnet routing: which router should be chosen for a particular destination (first routing phase) and which path should be chosen by the router to reach the destination (second routing phase). Solving these problems in an optimal manner is not possible if adhering to the current IB specification: the routers are non-transparent subnet boundaries (local SM cannot see beyond), so full topology visibility condition is not met. However, in this paper, by using regularity features provided by the fat-tree topology, we propose a solution for these problems. Nevertheless, for more irregular networks where the final destination is located behind another subnet (at least two router hops required) there may be a need for a super subnet manager that coordinates between the local subnet managers and establishes the path through the transit subnet. We consider such scenarios to be future work. In this section we present two new routing algorithms: Inter-Subnet Source Routing (ISSR) and Inter-Subnet Fat-Tree Routing (ISFR). ISFR is an algorithm designed to work best on fat-trees while ISSR is a more generic algorithm that works well on other topologies also. However, in this paper we only focus on fat-trees and fat-tree subnets as the deadlock problem becomes more complex when dealing with irregular networks. Nevertheless, we plan to address deadlock free inter-subnet IB routing in a more general manner in subsequent publications.

4.1 Inter-subnet Source Routing

We designed ISSR to be a general purpose routing algorithm for routing hybrid subnets. It needs to be implemented both in the SM and the router firmware. It is a deterministic oblivious routing algorithm that always uses the same path for the same pair of nodes. In general, it offers routing performance comparable to ISFR algorithm provided a few conditions explained in Sect. 5 are met.

The routing itself consists of two phases. First, for the local phase (choosing an ingress router port for a particular destination) this algorithm uses a mapping file. Whereas the *find_router*() function which chooses the local router looks almost exactly the same (it just matches the whole GID) for ISSR algorithm as for the OpenSM routing algorithm, the main difference lies in the setup of the mapping file. In our case, we provide full granularity meaning that instead of only a subnet prefix as for the OpenSM inter-subnet routing, the file now contains a fully qualified port GID. This means that we can map every destination end-port to a different router port while OpenSM routing can only match a whole subnet to a single router port. In the case of ISSR, an equal number of destinations is mapped to a number of ports in a round robin manner. In our example, *dst_gid1* and *dst_gid3* are routed through port 1 and port 2 on router A, and *dst_gid2* and *dst_gid4* are routed through the same ports on router B. Backup and default routes can also be specified.

Code 1. A high-level example of a mapping file for ISSR and ISFR algorithms

```
1: dst_gid1     router_A_port_1_guid
2: dst_gid2     router_B_port_1_guid
3: dst_gid3     router_A_port_2_guid
4: dst_gid4     router_B_port_2_guid
5: #default route
6:     *        router_A_port_1_guid
7:     *        router_B_port_1_guid
```

Second difference is the implementation of the additional code in the router firmware. A router receiving a packet destined to another subnet will source route that packet. The routing decision is based both on the source LID (of the original source or the egress port of the previous-hop router in a transit subnet scenario) and the destination LID (final destination LID or the LID of the next-hop ingress router port). The router knows both these values because it sees the subnets attached to it. To obtain the destination LID, a function mapping the destination GID to a destination LID or returning the next-hop LID based on the subnet prefix located in the GID is required. In our case, this function is named *get_next_LID* (line 2 of the pseudo code in Algorithm 1).

The algorithm calculates a random number based on the source and destination LIDs. This is done in a deterministic manner so that a given src-dst pair always generates the same number, which prevents out of order delivery when routing between subnets and, unlike round-robin, makes sure that each src-dst always uses the same path through the network. This number is used to select a single egress port from a set of possible ports. There is a set of possible ports because a router may be attached to more than two subnets and therefore a two-step port verification is necessary: first choose the ports attached to the subnet

Algorithm 1. choose_egress_port() function in ISSR

```
Require: Receive an inter-subnet packet
Ensure: Forward the packet in a deterministic oblivious manner
1: if $received\_intersubnet\_packet()$ then
2:     $dstLID = get\_next\_LID(dGID)$
3:     $srand(srcLID + dstLID)$
4:     $port\_set = choose\_possible\_out\_ports()$
5:     $e\_port = port\_set[(rand()\%port\_set.size)]$
6: end if
```

(or in the direction of the subnet) in which the destination is located, and then, by using a simple hash based on a modulo function, choose the egress port.

4.2 Inter-subnet Fat-Tree Routing

As mentioned previously, a problem that needs to be solved is the communication between SMs that are in different subnets and are connected through non-transparent routers. Our solution is based on the fact that the IBA specification does not give the exact implementation for inter-subnet routing, so our proposal provides an interface in the routers through which the SMs will communicate. In other words, we implement handshaking between two SMs located in neighboring subnets. The algorithm uses the previously defined file format containing the GID-to-router port mappings in Code 1. The ISFR algorithm is presented in Algorithm 2. Like the ISSR algorithm, it is also implemented in the router device. ISFR works only on fat-trees and with fat-tree routing running locally in every subnet. It will fall back to ISSR if those conditions are not met.

Algorithm 2. query_down_for_egress_port() function in ISFR

```
Require: Local fat-tree routing is finished
Require: Received the mapping file
Ensure: Fat-tree like routing tables throughout the fabric
1: if $received\_mapping\_files$ then
2:     for all $port\ in\ down\_ports$ do
3:         $down\_switch = get\_node(port)$
4:         $lid = get\_LID\_by\_GID(GID)$
5:         if $down\_switch.routing\_table[lid] == primary\_path$ then
6:             $e\_port = port$
7:         end if
8:     end for
9: end if
```

Every single router in a subnet receives the port mappings from its local SM and is thereby able to learn which of its ports are used for which GIDs. Next, for each attached subnet, the router queries the switches in the destination subnets to learn which of the switches has the primary path to that subnet's HCAs. If we assume a proper fat-tree (full bisection bandwidth) with routers on the top of the tree, then after such a query is performed, each router will have one path per port in the downward direction for each destination located in a particular subnet. In other words, if we substituted the top routers with switches, the routing tables for the pure fat-tree and the fat-tree with routers on top would be the same.

5 Simulations

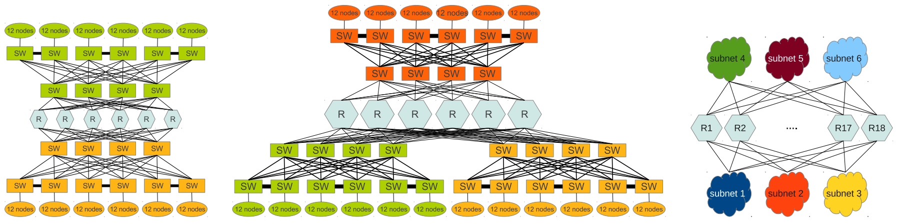
To perform large-scale evaluations and verify the scalability of our proposal, we use an InfiniBand model for the OMNEST/OMNeT++ simulator [13]. The IB model consists of a set of simple and compound modules to simulate an IB network with support for the IB flow control scheme, arbitration over multiple virtual lanes, congestion control, and routing using linear routing tables. In each of the simulations, we used a link speed of 20 Gbit/s (4x DDR) and Maximum Transfer Unit (MTU) equal to 2048 bytes. Furthermore, we use uniform, non-uniform and HPCC traffic patterns. We used synthetic traffic patterns to show baseline performance as these patterns have a predictable and easily understandable behavior, and are general rather than specific to a given application. We do not provide the baseline results for the same topologies implemented as a single subnet because ISFR routing provides exactly the same performance.

The simulations were performed on three different topologies shown in Fig. 1. Each of the topologies can be classified as a 3-stage (i.e. having three routing/switching stages and one node stage) fat-tree with routers placed on top of the tree (instead of normally placing root switches there). Even though there is a dedicated fat-tree routing algorithm delivering high performance on almost any fat-tree, we still decided to subnet a fat-tree fabric. The reason for that is that we consider the fat-tree topology to be a very good proof-of-concept topology for inter-subnet routing testing.

The fat-tree topology is scalable and by changing the number of ports we are able to vary the size of the topologies and show how our algorithms scale with regards to the number of nodes and subnets that are interconnected. All our subnets are 2-stage fat-trees that are branches in a larger 3-stage fat-tree so we can use routers and our routing algorithms to demonstrate how to seamlessly interconnect smaller fat-tree installations without using oversubscription. We chose a 3-stage 648-port fat-tree as the base fabric because it is a common configuration used by switch vendors in their own 648-port systems [14,15,16]. Additionally, such switches are often connected together to form larger installations like the JuRoPA supercomputer [17].

5.1 Routing Algorithm Comparison

We perform three sets of simulations: with uniform traffic, with non-uniform traffic and we run the HPCC benchmark. For non-uniform traffic we vary packet

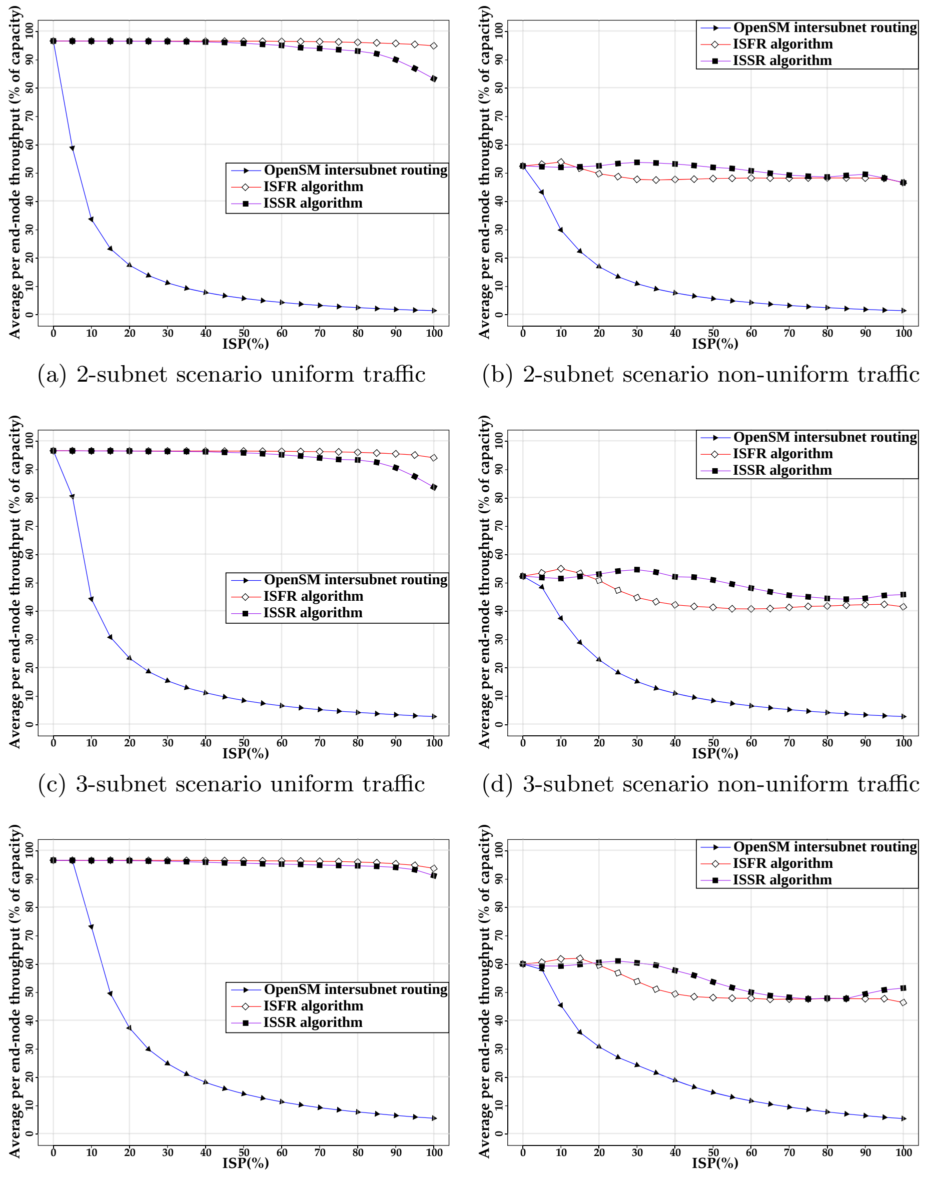


(a) 2-subnet fabric (b) 3-subnet fabric (c) 6-subnet fabric

Fig. 1. Topologies used for the experiments

length from 84 bytes to 2 kB, keep the message length constant at 2 kB. We also introduce some randomly preselected hot spots (different for every random seed, not varying in time): one hot spot per subnet, with a probability of $ISP * 0.05$ for remote traffic and the same hot spot with a probability of $(1 - ISP) * 0.05$ for local traffic. The ISP (Inter-Subnet Percentage) value is the probability that a message will be sent to the local or the remote subnet. It varies from 0% (where all messages remain local) to 100% (where all messages are sent to remote subnets). The non-hot spot destined part of the traffic selects their destination randomly from all other available nodes. This means that there could be some other random hot spots that vary in time, and some nodes could also contribute unknowingly to the preselected hot spots. We express the measured throughput as the percentage of the available bandwidth for all the scenarios. The parameters for that traffic pattern were chosen to best illustrate the impact of congestion on the routing performance, which is a good baseline for algorithm comparison.

Uniform Traffic. The results for this scenario are shown in Fig. 2. When it comes to uniform traffic, we can establish that the performance of the OpenSM inter-subnet routing deteriorates in the presence of even a very small amount of inter-subnet traffic. At ISP equal to 20%, throughput is reduced to 17.5% for the 2-subnet scenario in Fig. 2(a), 23.46% for the 3-subnet scenario in Fig. 2(c), and 37.5% for the 6-subnet scenario in Fig. 2(e). The increase in performance for a larger number of subnets is explained by the fact that traffic is spread across more routers, i.e. each subnet in the topology uses a different ingress port locally. ISFR algorithm provides almost constant high performance under uniform traffic conditions whereas the performance of ISSR algorithm deteriorates slightly for very high ISP values as shown in Fig. 2(a) and Fig. 2(c). This is caused by the fact that egress ports from the routers may not be unique as they are chosen randomly. However, the deterioration is smaller when the number of subnets increases (12% decrease for the 3-subnet scenario compared to 5% decrease for the 6-subnet scenario at ISP=100%) as shown in Fig. 2(e). This occurs because the more subnets we have in the fabric and the higher is the ISP value, the more inter-subnet traffic pairs are created, so the hash function has a higher probability to utilize more links from the defined subnet-port-set as there are more random numbers generated.



(a) 2-subnet scenario uniform traffic

(b) 2-subnet scenario non-uniform traffic

(c) 3-subnet scenario uniform traffic

(d) 3-subnet scenario non-uniform traffic

(e) 6-subnet scenario uniform traffic

(f) 6-subnet scenario non-uniform traffic

Fig. 2. Throughput as a function of ISP with uniform and non-uniform traffic

Non-uniform Traffic. Whereas under uniform traffic our algorithms gave almost optimal performance, the situation worsens if some non-uniformity is added as seen in Fig. 2.

What is first noticeable is the fact that ISFR algorithm is clearly outperformed by the ISSR algorithm for the middle range of the ISP values (20% to 70%), as best seen in Fig. 2(d). Second, we observe that ISSR algorithm deteriorates for ISP values greater than 40%, which is best seen in Fig. 2(f). ISFR, on the other hand, becomes stable at ISP values close to 40%. This behavior is explained by the addition of the hot spots to our traffic pattern, the occurrence of head-of-line (HOL) blocking, and the migration of the root and the branches of the congestion trees. For lower ISP values ($\leq$50%) local traffic is dominant and in every subnet 5% of such traffic is destined to the local hot spot. In such a case the branches of the congestion tree will mostly influence the local traffic, but they will also grow through the single dedicated downward link (a thick branch) to influence the victim nodes in other subnets if the ISFR algorithm is used. For ISSR, the same will happen, but there is no dedicated downward link and the branches growing through multiple downward links will be much thinner, therefore, influencing the local traffic in other subnets to a lesser extent. This happens because ISSR spreads the traffic destined towards the hot spot across multiple downward links. This is the reason why ISFR algorithm is outperformed by ISSR in such a hot spot scenario for almost all ISP values.

For ISFR algorithm, for higher ISP values ($\geq$50%), the root of the congestion tree will move from the last link towards the destination to the first downward dedicated link towards the destination (i.e. a router port), and the congestion tree will influence mostly the inter-subnet traffic as there will be little or no local traffic, which is why ISFR algorithm reaches stability at around 50% ISP. For ISSR, for higher ISP values, the root of the congestion tree will not move and the branches will grow much thicker (as there is more incoming remote traffic). This will not only slightly influence the local traffic (that is low for high ISP values) in the contributor's subnets, but also it will cause HOL blocking for the downward traffic that uses the same links that the hot spot traffic uses to reach other destinations. It happens because ISSR does not use dedicated paths for downward destinations. Such a deterioration can be best observed in Fig. 2(d) or Fig. 2(f).

Another vital observation is the fact that by increasing the number of subnets, we increase the performance of all the routing algorithms. This is best visible when comparing Fig. 2(d) and Fig. 2(f). The explanation for that is the segmentation of the hot spot contributors. In other words, the more hot spots there are, the weaker is the influence of the head-of-line blocking (the congestion tree branches are thinner).

We also see that OpenSM routing still yields undesirable performance for every scenario. However, an important observation here is that the congestion does not originate from the hot spots, but from the utilization of a single ingress link to transmit the traffic to the other subnet.

HPC Challenge Benchmark. We implemented a ping-pong traffic pattern that was used to run the HPC Challenge Benchmark [18] tests in the simulator. We used a message size of 1954KB and kept the load constant at 100%. The tests were performed on 500 ring patterns: one natural-ordered ring (NOR) and 499 random-ordered rings (ROR) from which the minimum, maximum and average results were taken. In this test each node sends a message to its left neighbor in the ring and receives a message from its right neighbor. Next, it sends a message

Table 1. The HPC Challenge Benchmark results (in MB/s)

Measurement	OpenSM	ISSR	ISFR
NOR BW	1572.49	1572.49	1572.49
ROR BW min/max/avg			
2 subnets	528/878/703	847/1166/1001	1064/1314/1187
3 subnets	345/611/482	753/993/867	946/1165/1069
6 subnets	202/343/270	709/875/775	841/1018/933

back to its right neighbor and receives a return message from its left neighbor. We treated the whole fabric as a continuous ring and we disregarded the subnet boundaries.

Table 1 presents the HPCC Benchmark results. For any fat-tree the NOR bandwidth results give the maximum throughput as there is no contention in the upward or the downward direction. However, when we compare the results for the ROR, we observe differences between the routing algorithms. For the 2-subnet scenario, we observe an increase in throughput of 536 MB/s (102%) when comparing the minimum throughput for the OpenSM and the ISFR algorithms. Furthermore, we observe that the average throughput for the ISFR algorithm is higher than the maximum throughput for the ISSR algorithm in all cases. For the average throughput we observe an increase of 484 MB/s (69%) compared to OpenSM routing. For the 3-subnet scenario the trend is the same as for the previous scenario, but we observe that the throughput is lower than for the 2-subnet scenario. This happens because the larger the topology, the higher the probability that the destination is chosen from a set of non-directly connected nodes. For a 144-node fabric, each source can address 143 end-nodes and 23 out of those end-nodes (15.7%) are reachable through a non-blocking path (11 at the local switch, 12 at the neighbor switch). For a 216-node fabric, the same number of nodes is reachable through a non-blocking path, but the overall number of nodes is larger, which gives only 10.6% of nodes reachable through a non-blocking path. This means that a ROR pattern in a larger fabric has a lower probability for reaching a randomly chosen node in a non-blocking manner. Furthermore, in larger topologies more nodes are non-local, which means that the routing algorithm uses the longest hop path to reach them (traversing all stages in a fat-tree), which further decreases the performance. For the 6-subnet scenario, we observe a similar situation as for the 3-subnet scenario: that there is an overall decrease in performance. The explanation is the same as for the 3-subnet scenario: more nodes are used to construct a ROR, but the number of nodes accessible in a non-blocking manner stays the same, so the generated ROR pairs have an even lower probability to use a non-blocking path.

The general observation is that for the HPCC benchmark, the ISFR algorithm delivers the best performance. It is because this traffic pattern does not create any destination hot spots and the congestion occurs only on the upward links towards the routers, while the dedicated downward paths are congestion-free. Despite using the same upward path as the ISFR algorithm, ISSR algorithm may not provide a dedicated downward path, which is why there is a performance difference between these two algorithms.

6 Conclusions and Future Work

Native IB-IB routers will make the network scalable, and designing efficient routing algorithms is the first step towards that goal. In this paper, we laid the groundwork for layer-3 routing in IB and we presented two new routing algorithms for inter-subnet routing: the inter-subnet source routing and the inter-subnet fat-tree routing. We showed that they dramatically improve the network performance compared to the current OpenSM inter-subnet routing.

In future, we plan to generalize our solution to be able to support many different regular fabrics in a deadlock-free manner. Another candidate for research will be evaluating the hardware design alternatives. Looking further ahead, we will also propose a deadlock-free all-to-all switch-to-switch routing algorithm.

References

1. Obsidian Strategics: Native InfiniBand Routing (2008), http://www.nsc.liu.se/nsc08/pres/southwell.pdf
2. Southwell, D.: Next Generation Subnet Manager - BGFC. In: Proceedings of HPC Advisory Council Switzerland Conference 2012 (2012)
3. InfiniBand Trade Association: Introduction to InfiniBand for End Users (2010)
4. Obsidian Strategics: Native InfiniBand Routing (2006), http://www.obsidianresearch.com/archives/2006/Mellanox_Obsidian_SC06_handout_0.2.pdf
5. The OpenFabrics Alliance: Issues for Exascale, Scalability, and Resilience (2010)
6. Yousif, M.: Security Enhancement in InfiniBand Architecture. In: 19th IEEE International Parallel and Distributed Processing Symposium, pp. 105a. IEEE (April 2005)
7. InfiniBand Trade Association: Infiniband Architecture Specification, 1.2.1 edn. (November 2007)
8. Prescott, C., Taylor, C.: Comparative Performance Analysis of Obsidian Longbow InfiniBand Range-Extension Technology (2007)
9. Richling, S., Kredel, H., Hau, S., Kruse, H.G.: A long-distance InfiniBand interconnection between two clusters in production use. In: State of the Practice Reports on - SC 2011. ACM Press, New York (2011)
10. Top 500 Supercomputer Sites (November 2012), http://top500.org/
11. Dally, W.J., Towles, B.: Principles and practices of interconnection networks. Morgan Kaufmann (2004)
12. Duato, J., Yalamanchili, S., Ni, L.: Interconnection Networks an Engineering Approach. Morgan Kaufmann (2003)
13. Gran, E.G., Reinemo, S.A.: Infiniband congestion control, modelling and validation. In: 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools 2011, OMNeT++ 2011 Workshop) (2011)
14. Oracle Corporation: Sun Datacenter InfiniBand Switch 648, http://www.oracle.com/us/products/servers-storage/networking/infiniband/046267.pdf

15. Mellanox Technologies: Voltaire Grid Director 4700, `http://www.voltaire.com/assets/files/Datasheets3/Grid-Director-4700-DS-WEB-020711.pdf`
16. Mellanox Technologies: IS5600 - 648-port InfiniBand Chassis Switch, `http://www.mellanox.com/related-docs/prod_ib_switch_systems/IS5600.pdf`
17. Forschungszentrum Jülich: JuRoPA - Jülich Research on Petaflop Architectures
18. Luszczek, P., Dongarra, J.J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., McCalpin, J., Bailey, D., Takahashi, D.: Introduction to the HPC Challenge Benchmark Suite (April 2005)

BBQ: A Straightforward Queuing Scheme to Reduce HoL-Blocking in High-Performance Hybrid Networks

Pedro Yebenes Segura[1], Jesus Escudero-Sahuquillo[1], Crispin Gomez Requena[1], Pedro Javier Garcia[1], Francisco J. Quiles[1], and Jose Duato[2]

[1] Dept. of Computing Systems, University of Castilla-La Mancha, Spain
{pedro.yebenes,jesus.escudero,crispin.gomez, pedrojavier.garcia,francisco.quiles}@uclm.es
[2] Dept. of Computer Engineering, Universitat Politècnica de València, Spain
jduato@disca.upv.es

Abstract. Head-of-Line (HoL) blocking is a well-known phenomenon that may dramatically degrade the performance of the modern high-performance interconnection networks. Many techniques have been proposed to solve this problem, most of them based on separating traffic flows into different queues at switch ports. However, the efficiency of these proposals may vary depending on the network topology or routing algorithm, as many of them are not aware of any specific network configuration. By contrast, other schemes are tailored to specific topologies like fat-trees, achieving a greater efficiency than "topology-agnostic" schemes. In this paper we propose a straightforward queuing scheme intended to be used in an efficient, recently-proposed hybrid topology. Our proposal significantly boosts network performance with respect to other queuing schemes while requiring similar or fewer resources. Moreover, the implementation of this scheme in InfiniBand-based networks is elementary thanks to the mapping of Service-Levels to Virtual-Lanes supported by this specification.

Keywords: Interconnection Networks, Hybrid Topologies, Dimension-Order Routing, Head-of-line Blocking, InfiniBand.

1 Motivation

High-performance, switch-based interconnection networks are key elements for the different systems currently used for parallel computing: massive parallel processors, clusters of PCs, and Networks-on-Chip. In such environments, the interconnection network plays a prominent role in the performance achieved by the whole system, especially as the number of endnodes increases. Indeed, the network may become the system bottleneck if it does not meet the communication needs of the applications. Therefore, achieving good network performance is an essential issue for interconnect researchers, designers, and manufacturers.

One of the most dangerous menaces to the performance of current high-performance interconnection networks is the Head-of-Line (HoL) blocking effect. This well-known effect may limit switch throughput to about 58% of its peak value [1] in switches that implement buffer-based queues at input/output ports to store packets waiting to be forwarded. It is worth pointing out that commercial switches of current high-speed interconnect technologies (InfiniBand [2], Myrinet [3], etc.) support some type of buffer-based queuing scheme, thus they may suffer HoL-blocking. Basically, HoL-blocking occurs when a packet at the head of a queue blocks the rest of packets in that queue, even if they request access to available ports. As a consequence, average packet latency eventually increases and network throughput decreases. This effect is strongly related to high traffic loads and congestion situations [4], although it may happen also in other traffic scenarios. Moreover, especially in congestion situations, HoL-blocking tends to "propagate", as HoL-blocked queues fill up and subsequently block (by flow control) other queues in other switches. This last phenomenon is known as high-order HoL-blocking [5], in contrast with the "original" HoL-blocking, which is called low-order HoL-blocking.

Taking into account the negative impact of HoL-blocking, many techniques have been proposed that may help to solve this problem. In general, any solution focused on limiting traffic load in the network or on avoiding or removing congestion situations may lower HoL-blocking probability, but there exist many techniques specially designed to reduce HoL-blocking (see Section 2.2). Most of the them are based on dividing the buffer space at the switch ports into different queues, then mapping packet to queues so that certain packet flows do not share queues with other flows. Although some of these queuing schemes prevent HoL-blocking completely [6,7,8], they require expensive and/or additional resources that are not supported by current commercial switches. Thus, other, feasible queuing schemes are far more popular even though they reduce HoL-blocking just partially [9,10,11]. However, we have found that queuing schemes can be more efficient in reducing HoL-blocking if they are designed for specific network topologies and routing algorithms, in this way the use of queues can be optimized. In that sense, in [12,13] queuing schemes were proposed that exploit the properties of fat-trees with DESTRO routing algorithm [14] to reduce HoL-blocking more efficiently than more "generic" schemes that require a higher (or similar) number of queues per port.

Following this approach, in this paper we present a queuing scheme that exploits the properties of an efficient, recently-proposed hybrid topology [15] to straightforwardly reduce HoL-blocking. Indeed, although this hybrid topology has been reported as more efficient than others (either direct or indirect topologies, see Section 2.1), its performance decreases when high traffic loads or congestion situations produce a massive appearance of HoL-blocking in the network. By contrast, if our proposed queuing scheme is used in this network topology, HoL-blocking is significantly reduced, as we show in Section 4. Specifically, our proposal is based on a simple but clever mapping of packet destinations to queues, so that every queue at every switch port in the network is used to separate the packet flows that may cross that port. We call this scheme Band-Based Queuing (BBQ). Moreover, as we explain in Section 3.1, this queuing scheme could be directly applied to InfiniBand-based networks, as the aforementioned mapping of packet destinations to queues can be implemented by mapping packets destinations to Service Levels (SLs), then SLs to Virtual Lanes (VLs).

The rest of the paper is organized as follows: Section 2 offers an overview of both proposed hybrid topologies and existing techniques that prevent or reduce HoL-blocking. The proposed queuing scheme is described in Section 3, along with some ideas for its implementation in real hardware. We also evaluate our proposal in Section 4, where simulation results are shown to compare its efficiency with that of other schemes. Finally, in Section 5 some conclusions are drawn.

2 Related Work

2.1 Hybrid Networks

Traditionally, network topologies are classified as either direct or indirect. Examples of such topologies appear in the TOP500 list, where tori and fat-trees are widely used. However, these two types of topologies have several drawbacks.

On one hand, direct topologies usually adopt an orthogonal structure, where nodes are organized in a n-dimensional space and connected in each dimension in a ring or array fashion. 2D or 3D direct topologies are relatively easy to build as each topology dimension is mapped to a physical dimension. Direct topologies with more than three dimensions imply not only increasing its wiring complexity but also the length of its links when they are mapped to our 3D physical space, thereby increasing the communication latency and negatively impacting performance. In addition, low dimension topologies tend to have a large number of nodes per dimension which also leads to an increase in communication latency.

On the other hand, the most common indirect topologies are multistage interconnection networks (MINs) where switches are organized as a set of stages. Indirect topologies usually provide better performance for a large number of nodes than direct topologies, at the cost of using a high number of switches and links, and increasing the wiring complexity, which grows with the size of the network. However, in direct topologies the complexity grows with the number of dimensions.

Recently, hybrid and hierarchical topologies have been proposed to get the benefits from both direct and indirect topologies. That is, hybrid topologies aim to provide high-performance as indirect topologies, but at a similar cost to direct topologies. With this aim in mind, the authors of [16] propose the Flattened-Butterfly, which is a derived topology from the Fat-tree in which all the switches from a column of the fat-tree are collapsed into a single switch, becoming a generalized hypercube. Extending this topology to exploit high-radix switches, in [17] the Dragonfly topology is proposed. In this topology, switches from the same row of the cube are totally connected forming a group. Groups are connected among them by a parameterizable number of links. The problem with hierarchical topologies is that they require the same number of switches than a direct topology with the same number of nodes, but switches are markedly more complex. So global complexity and cost of the network is noticeably higher than in direct networks, despite being lower than in indirect ones.

In order to overcome this drawback, k-ary n-direct s-indirect topologies were proposed [15]. In this topology nodes are organized in n dimensions, like in a direct topology, but the nodes of a given dimension are not connected as in meshes or tori. Instead, these nodes are directly connected by means of an indirect network. This indirect network could be even a simple crossbar or switch, as in Fig. 1. If the number of nodes per dimension exceeds the number of ports of the switches used in the indirect network, a MIN is required. Therefore, the proposed family of topologies is defined by three parameters. Two of them are inherited from direct networks: the number of dimensions n and the number of nodes per dimension k. In addition, there is an extra parameter, s, the number of stages of the indirect subnet. The number of processing nodes that this topology can interconnect is given by $N = k^n$. In this paper, we focus on k-ary

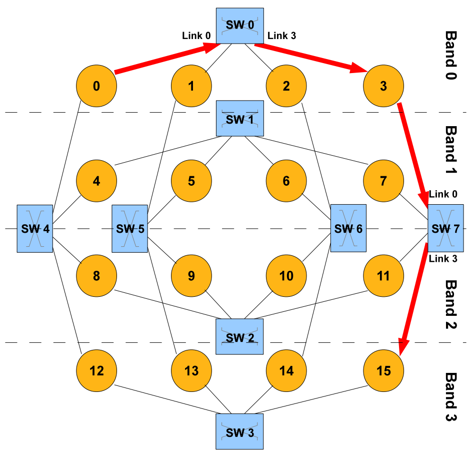


Fig. 1. *2*-ary *4*-direct *1*-indirect topology. The route with Hybrid-DOR from node 0 to node 15 is highlighted.

n-direct *1*-indirect topologies (hereafter hybrid networks), since using a MIN to connect the nodes of a dimension skyrockets the number of switches of the network.

In [15], the authors also propose a deterministic routing algorithm derived from Dimension-Order Routing (DOR) for these topologies called Hybrid-DOR. In Hybrid-DOR, network dimensions are crossed in an established order to guarantee deadlock freedom as in DOR, specifically the *X* dimension is crossed first, then the *Y* dimension (i.e. a *X*-*Y* routing algorithm). However, packets do not perform several hops at each dimension. Instead, in Hybrid-DOR, nodes directly forward packets through the unique link that connects them to the dimension through which they have to send a packet. Then, the packet is received in a crossbar where it is just forwarded through the link indicated by the destination component of the corresponding dimension, requiring just another hop to reach the next endnode. For instance, Fig. 1 shows (highlighted) the route from node 0 (0,0) to node 15 (3,3). As it can be seen, the packet leaves node 0 through its only link in the *X* dimension, reaching a switch (sw0) where it is forwarded through link 3, then reaching node 3 (0,3). Then the process is repeated in the next dimension. Notice that no matter the distance that a packet has to travel in a dimension, it can be traversed in just 2 hops. In [15], it is shown that this hybrid topology with Hybrid-DOR obtains better performance figures of merit than other network configurations, such us tori, fat-trees or flattened butterflies (provided that size is similar in all the cases).

2.2 Solutions for the HoL-Blocking Problem

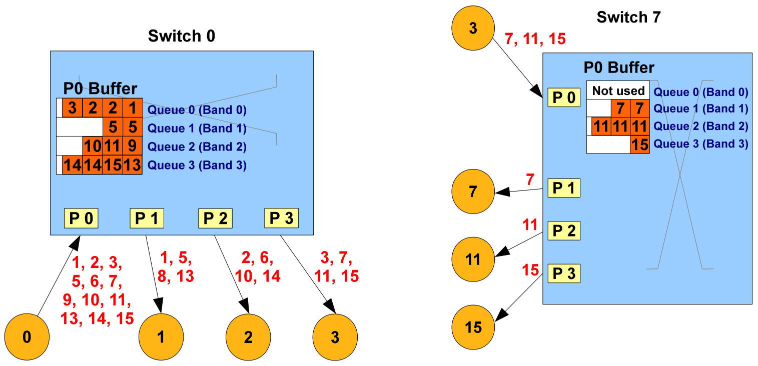
As mentioned above, there exist several approaches to solve the problem of HoL-blocking in high-performance interconnects, but in this section we focus only on those solutions that address this problem explicitly.

In that sense, probably the most efficient of these solutions are based on the dynamic allocation of queues (or Virtual Channels [10]) to isolate the packet flows that contribute to congestion (usually referred to as "hot flows"), so that the HoL-blocking these flows could cause to others ("cold flows") is prevented. Thus, they assume that the HoL-blocking between any two cold flows is not significant. These techniques require some mechanism to detect congestion, in order to identify (then separate) hot flows. Solutions like [18,19] identify hot flows based on their final destination, while other solutions [7,8] are able to identify hot flows based on their path towards their final destination (i.e. the former solutions assume that congestion originates only at endpoints while the latter consider that congestion may originate also at internal points of the network). Although, as mentioned above, these techniques are quite efficient, they may have problems when the number of simultaneous hot-spots in the network is greater than the number of queues available at the ports to isolate hot flows. Most important, they require some type of control memories at switch ports to keep track of hot-spots, and also other additional resources and/or control messages, that in general are not supported by current commercial interconnects.

Other solutions deal with HoL blocking also by mapping packet flows to different queues, but keeping constant both the number of active queues at each port and the mapping policy, that does not depend on traffic conditions. Note this means that, in contrast with the aforementioned strategies based on the dynamic allocation of queues, these solutions do not "react" against congestion situations but "statically" separate packet flows into the available queues so that the impact of HoL-blocking is reduced regardless of its origin. This approach results in different "static" queuing schemes based on simple mapping criteria, that in general do not require additional resources apart from the set of queues at each port. However, it does not mean that these schemes have always affordable or feasible implementations. For instance, Virtual Output Queues at network level (VOQnet) [6] uses at each switch port as many queues as destinations in the network, so that any packet is mapped to the queue corresponding to its destination, only sharing that queue with other packets addressed to the same destination. This scheme totally prevents low- and high-order HoL-blocking, but it is unfeasible due to the high number of queues per port, as each queue needs a minimum memory space.

By contrast, other static queuing schemes are far more feasible as they just require a reduced number of queues per port. Among them, Virtual Output Queues at switch level (VOQsw) [11] uses at each port as many queues as output ports in the switch, so that each incoming packet is mapped to the queue assigned to its next output port. Hence, VOQsw totally prevents low-order HoL-blocking, but not the high-order. Other similar (although not identical) queuing schemes that reduce Hol-blocking only partially are Dynamically Allocated Multi-Queues (DAMQs) [9], Destination-Based Buffer Management (DBBM) [20] and Dynamic Switch Buffer Management (DSBM) [21].

In general, the aforementioned static queuing schemes are not aware of the routing algorithm and network topology. As a consequence, the reduced set of queues per port is not always efficiently leveraged to reduce Hol-blocking, thus the performance of these techniques may drop in certain topologies when HoL-blocking appears. By contrast, other queuing schemes like Output-Based Queue Assignment (OBQA) [12,13] and vFTree [22] take into account network topology (specifically, fat-trees) and routing algorithms (respectively, the ones proposed in [14] and [23]) and exploit their characteristics to reduce HoL-blocking as effectively as (or better than) schemes such as VOQsw, while requiring half, or even the quarter, the number of queues per port



(a) X-axis Switch #0. (b) Y-axis Switch #7.

Fig. 2. Packet destination distribution for BBQ on a *2*-ary *4*-direct *1*-indirect network

required by other schemes. Obviously, neither OBQA nor vFTree are designed for hybrid topologies, but the basic idea of a queuing scheme that "fits" this specific topology to efficiently reduce HoL-blocking as OBQA and vFTree do in fat-trees lead us to propose BBQ, which is described in the next section.

3 BBQ Description

In this section, we describe our straightforward proposal for HoL-blocking reduction in *k*-ary *n*-direct *s*-indirect networks that use the Hybrid-DOR routing algorithm. Basically, our proposal consists in a "static" queuing scheme based on a simple but clever policy to map packets to queues at each switch port, which significantly reduces HoL-blocking. Specifically, the proposed mapping policy selects the queue for every packet according to this formula:

$$SelectedQueue = \frac{Destination \times \#Queues}{\#EndNodes} \tag{1}$$

Where *Destination* is the destination of the packet, *#Queues* is the number of queues that BBQ configures at each switch port (i.e. the number of queues each port buffer is divided into), and *#EndNodes* is the number of endnodes in the network. This queue-selection policy "virtually" divides the hybrid network in a number of horizontal zones (bands), each one consisting of one or more consecutive rows of destinations (endnodes) that result in the same "SelectedQueue" when they are introduced as "Destination" in the formula. This is the reason to call the proposal Band-Based Queuing (BBQ). Note that BBQ divides the network into a number of bands equal to the number of queues configured at each port. For instance, in Figure 1, four queues per port are assumed, thus the network is divided into four bands, each one consisting in a row of endnodes as there are four rows in the network.

As a consequence of this mapping policy, at any port of the network, a specific packet is mapped to the same queue as any other packet whose destination is in the same band, i.e. packets addressed to the same band are stored in the same queue, and this happens in the ports of the switches of the *X* dimension and in the ports of the switches of the *Y* dimension. Moreover, note that, as Hybrid-DOR is assumed, packets addressed to any band may be present in any port of the switches of the X dimension and also in any port of the switches of the Y dimension, as all the latter switches are directly connected to endnodes belonging to any band in the network. Thus, all the queues defined at any port of the switches may receive packets, so that they are efficiently used to separate packet flows, i.e. to reduce HoL-blocking probability. Besides, note that packets addressed to a specific band never share queues with packets addressed to any other band, thus if a hot-spot appears in one of the bands, only packets addressed to that band may suffer HoL-blocking caused by the hot flows contributing to that hot-spot.

Fig. 2 shows an example of the internal behavior of BBQ both in switch #0 of Fig. 1 (see Fig. 2a) and in switch #7 of Fig. 1 (see Fig. 2b). Notice that each link is depicted as an arrow labelled with possible destinations of packets that may cross through that link in the sense indicated by the arrow, and packets are depicted as squares inside their corresponding queues and are labelled with their respective destinations. For instance, packets with destinations 1, 2, and 3 (all in Band 0) are stored in queue 0 at port P0 of switch 0. Note that the only queue that is not used in switch #7 is queue 0, but just because only 3 destinations are received at port P0. However, note that queue 0 would be used if band 0 consisted of more than one row, in order to store packets that require to change rows inside the same band.

In contrast with BBQ, other feasible queuing schemes such as DBBM [20] would not exploit all the available queues in both dimensions if used in the hybrid topology. In particular, DBBM selects the queue to store a packet according to the formula $Destination\, MOD\ \#Queues$, where $\#Queues$ is again the number of queues per port (i.e. DBBM maps packets to queues by consulting the least-significant $log_2(\#Queues)$ bits of the destination ID of each packet). Note this formula divides the network into vertical zones consisting of non-consecutive columns of endnodes, so that packets addressed to the same vertical zone are mapped to the same queue. As a consequence, if DBBM is applied to hybrid networks with Hybrid-DOR, switches in the Y dimension would not use all the queues at their ports to reduce HoL-blocking, thereby network performance dropping (as we show in Section 4). For instance, if DBBM were used (configured with four queues per port) in the same scenario as BBQ in Fig. 2, at port P0 of switch #7 all the queues but one would be always wasted, as all the packets that could be received at this port would be always mapped to the same queue.

On the other hand, if either VOQsw [11] or OBQA [12] were used, packets would be mapped to a queue depending on the output port requested at each switch. This reduces the impact of low-order HoL-blocking, but note that packet flows share queues with different flows depending on the current switch. Thus, hot flows contributing to a hot-spot may cause (high-order) HoL-blocking to many flows, the performance of these schemes dropping in hot-spot scenarios. By contrast, as explained above, when BBQ is used only packets addressed to the band where the hot-spot appears are affected.

In conclusion, the BBQ queuing scheme exploits the characteristics of the hybrid topology and Hybrid-DOR to efficiently reduce HoL-blocking. Note that this queuing scheme is based on a simple operation, that can be performed even prior to packet injection into the network. This simplicity can be exploited to easily apply BBQ to real network technologies such as Infiniband, as it is explained in next Section.

Table 1. Evaluated Network Configurations

#	Network Topology	Processing Nodes	#Routers (Endnodes)	#Switches (indirect network)
1	8-ary 2-direct 1-indirect	64	64	16
2	16-ary 2-direct 1-indirect	256	256	32
3	32-ary 2-direct 1-indirect	1024	1024	64

3.1 BBQ Implementation for InfiniBand

The simplicity of the BBQ queuing scheme permits a straightforward implementation in InfiniBand, as the band each packet is addressed to can be computed before its injection, and then the result applied to assign the packet a Service-Level (SL) and a Virtual Lane (VL), as supported by InfiniBand. Specifically, when each InfiniBand packet is generated, it is assigned a specific SL that is mapped at each switch to a given VL, each VL being in practice an independent queue at the buffer of the ports of InfiniBand switches (i.e. InfiniBand flow-control is performed at VL-level). Thus, packets with the same SL are always mapped to the same VL, so to the same queue. The specific SL-to-VL mapping is implemented by tables that are filled by the InfiniBand Subnet Manager. Taking all that into account, BBQ can be easily implemented in InfiniBand by assigning each packet an SL equal to the selected queue given by the Formula 1 for the specific packet destination, and filling the SL-to-VL tables so that VL=SL. In this way, packets will be assigned to VLs (i.e. to queues) following the BBQ scheme. Indeed, as a future work, we plan to compute the BBQ formula into the Subnet Manager in a real InfiniBand-based network, in order to assign SLs to packets before their injection.

4 Evaluation

In this section, we evaluate the BBQ technique based on simulation experiments, in comparison to other queuing schemes. Next, we describe the simulation model used in our experiments and discuss the simulation results.

4.1 Simulation Model

The simulation tool [24] used in the experiments has been built from the OMNeT++ platform [25]. Basically, our simulator is able to model several interconnection network configurations, by means of simple and compound modules. These OMNeT++ modules define the structure and behavior of the network, supporting the routing algorithms, queuing schemes, arbitration over multiple queues per switch port and flow control. This simulation tool has been validated against real systems.

Table 1 shows the different hybrid topologies (i.e. k-ary n-direct s-indirect) that we have modeled in the simulator with different sizes, in order to test the scalability of the BBQ proposal. Note that we indicate separately the number of switches in the indirect network and the number of endnodes.

For all the network configurations, we assume serial full-duplex pipelined links with 2.5 GB/s (20 Gbps) bandwidth, 6 nanoseconds of link propagation delay (i.e. a length of 1.2 meters and a delay of 5 ns/m, these values being based on the InfiniBand specification [2]), both for switch-to-switch and node-to-switch links. Similarly, in all the cases the switching technique is Virtual Cut-Through, the flow control policy is credit-based and packet size (i.e. MTU) is 2048B. We also assume the use of the Hybrid-DOR routing algorithm described in section 2.

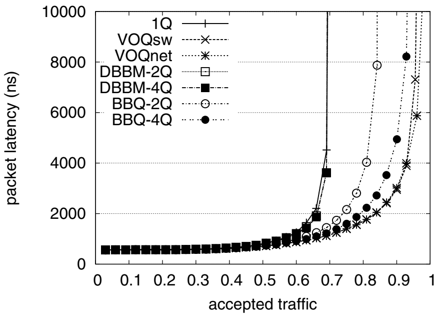
Regarding the switch model, we use switches with 8 ports for network configuration #1, 16 ports network configuration #2 and 32 ports network configuration #3[1]. Besides, the modeled switch architecture follows an Input-Queued (IQ)-scheme, i.e. RAMs are present only at switch input ports. RAM organization depends on the queuing scheme. We have modeled the following queuing schemes for HoL-blocking prevention:

- **Single Queue** (1Q). This is the simplest case, with only one queue at each input RAM, whose size is set to 128 KB. Hence, there is no HoL-blocking reduction policy at all. Particularly, this scheme allows to evaluate the performance achieved by the Hybrid-DOR algorithm without any technique alleviating HoL-blocking.
- **Virtual Output Queues at switch level** (VOQsw). 128 KB buffers per input port are used, statically and equally divided into as many queues as switch output ports, in order to store each incoming packet in the queue corresponding to its requested output port. As the number of queues per port is equal to the switch radix, note that 8 queues are used in network configuration #1, 16 queues are used in network configuration #2 and 32 queues are used in network configuration #3.
- **Virtual Output Queues at network level** (VOQnet). The RAM at each input port is divided into as many queues as destinations in the network, so that two packets may share a queue only if they have the same destination. VOQnet requires larger RAMs as each queue needs a minimum size[2]. Specifically, RAM size for VOQnet is 256 KB, 1024 KB and 4096 KB respectively for network configurations #1, #2 and #3. Although this scheme is almost unfeasible for medium-size or large networks, it shows the theoretical maximum effectiveness in HoL-blocking prevention.
- **Destination-Based Buffer-Management** (DBBM). We assume a buffer of 128 KB per input port, statically and equally divided among 2 or 4 queues. The mapping of packet of queues is performed according to the modulo-mapping function $SelectedQueue = Destination\ MOD\ \#Queues$.
- **Band-Based Queuing** (BBQ). We also assume a buffer of 128 KB per input port, statically and equally divided among 2 or 4 queues. Each packet is mapped to a queue according to the formula 1 in section 3.

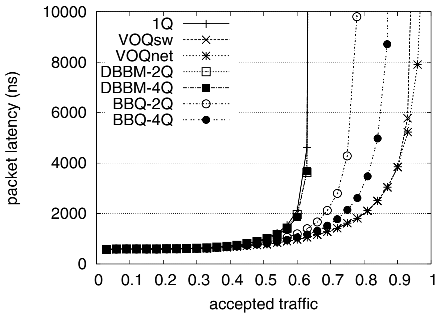
Endnodes are connected to switches by means of Host Channel Adapters (HCAs), modeled with as many admittance queues as endnodes in the network, and each generated packet is stored in the admittance queue assigned to its destination, so that HoL-blocking is prevented at traffic generation level. Besides, packets are transferred from admittance queues to injection queues, which are organized following the same scheme as that established at the switch input ports (e.g. for BBQ, each HCA has 2 or 4 injection queues). Injection queues are flow-controlled from the switch input ports.

[1] Note that these numbers of ports are common in current commercial switches, but we could also have used switches with a greater radix.

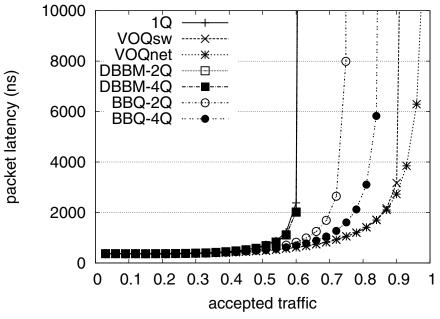
[2] Considering flow-control restrictions, packet size, link bandwidth and link delay, the minimum queue size is 4 KB (i.e. 2 packets).



(a) Network Configuration #1



(b) Network Configuration #2



(c) Network Configuration #3

Fig. 3. Packet Latency versus Generated traffic. Traffic case #1 (Uniform)

Regarding traffic modeling, we use two synthetic traffic patterns. The first one follows a completely uniform (random) distribution, traffic generation rate incrementally increasing from 0% up to 100% of link bandwidth. The second one is a hot-spot pattern: a percentage of sources (25%) generate traffic addressed to one single (hot-spot) destination, the rest of the nodes generating only uniform traffic. The generation rate in the second pattern also increases incrementally from 0% up to 100%.

Finally, among all the metrics that the simulator offers, we base our evaluation on the metrics commonly used for measuring network performance, such as Normalized Efficiency (i.e. the average amount of traffic delivered by the network per time unit, being normalized against the maximum overall amount of traffic that can generate the endnodes), and the packet latency (in nanoseconds), defined as the average time from packet generation to packet delivery.

4.2 Uniform Traffic Results

Fig. 3 shows Packet latency results for network configurations #1, #2 and #3 (see Table 1), when the uniform traffic pattern is used. As can be seen, in Figure 3a, the network saturation point is about 70% when HoL-blocking is not reduced in any way (i.e. 1Q scheme). DBBM achieves the same results as 1Q, regardless of its number of queues per port, due to the drawbacks explained in Section 3. As expected, VOQnet achieves the highest performance, but requiring 64 queues per port, while VOQsw, which uses 8 queues per port, has nearly the same results. Notice that BBQ needs only 4 queues per port to achieve similar results to VOQsw. Furthermore, note that BBQ is simpler to implement than VOQsw, as the former can calculate the queues to store a packet prior to its injection, while the latter needs to recalculate the queue at each switch along the packet route. Note also that BBQ significantly outperforms (around 25%) the performance achieved by 1Q or DBBM.

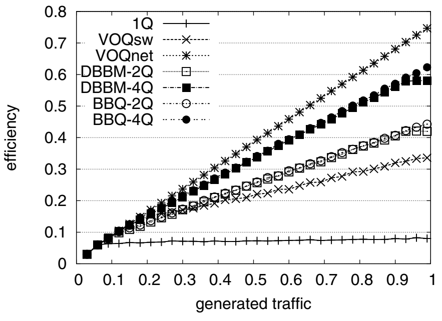
Similar conclusions can be extracted from Figs. 3b and 3c. For a network with 256 endnodes (Fig. 3b), BBQ configured with 4 queues per port reaches the saturation point just about 5% lower than VOQsw. However, VOQsw needs 16 queues per port because there are more endnodes per dimension. As in the previous scenario, DBBM and 1Q achieve the worst results. Note that, again, the reason for the poor behavior of DBBM is that it wastes many queues in the switches of the indirect network. Similarly, Fig. 3c shows the result for an hybrid network with 1024 endnodes. Note that this network is four times bigger than the previous one. VOQnet, which now needs 1024 queues per port and uses memories of 4096 KB, achieves the best performance, but VOQsw configured with just 32 queues per port achieves a similar performance. However, BBQ with only 4 queues achieves a performance only 8% lower than VOQsw, while the former using an eighth of the number of queues per port used by the latter. In conclusion, under uniform traffic scenarios, BBQ achieves nearly the same performance as VOQsw, while drastically reducing the required number of queues per port.

4.3 Hot-Spot Traffic Results

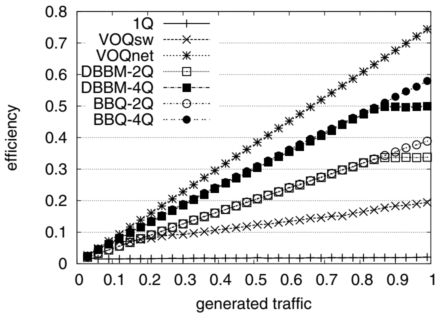
Fig. 4 shows the Network Efficiency for network configurations #1, #2 and #3 (see Table 1) when the hot-spot traffic pattern is used. It is worth reminding that in this case a 25% of endnodes generate traffic addressed to a single destination, whereas the rest of the traffic is randomly distributed. Thus, a maximum of 75% of network efficiency can be achieved when complete HoL-blocking prevention is provided.

In Fig. 4a (hybrid network with 64 endnodes), 1Q barely achieves a 9% of network efficiency when 100% of traffic load is generated, while the ideal network efficiency achieved by VOQnet is about $\approx$ 75%. VOQsw also suffers a strong degradation, as its maximum network efficiency is about 33%. This is because VOQsw suffer specially the high-order HoL-blocking that arises in hot-spot scenarios, as explained in Section 3. By contrast, notice that DBBM and BBQ using only 2 queues per port achieve better performance than VOQsw (that requires 8 queues per port), while if they are configured with 4 queues per port, their network efficiency improves significantly. Besides, note that BBQ outperforms DBBM in about 10% when high traffic loads are injected in the network. The reason for this good behavior of BBQ is that it efficiently separates the hot flows while taking advantage of all the queues available in every switch port. Thus, the HoL-blocking effect is reduced as it only appears among packets addressed to the same network band, as explained in Section 3. By contrast, DBBM wastes queues in the switches in the *Y* dimension.

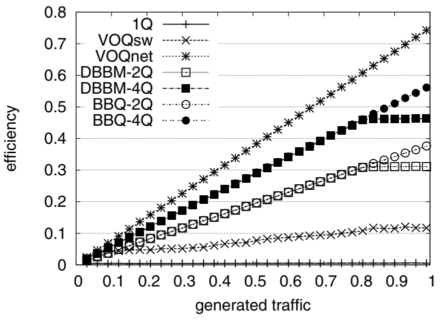
Similar conclusions can be drawn from Figs. 4b and 4c. Although VOQnet achieves almost 20% of improvement over BBQ, note that the latter only uses 4 queues per port



(a) Network Configuration #1.



(b) Network Configuration #2.



(c) Network Configuration #3.

Fig. 4. Network Efficiency versus Accepted Traffic. Traffic case #2 (Hot-Spot)

to achieve its best results, while the former uses 256 or 1024 queues per port. Note also that BBQ outperforms DBBM in more than 15% of efficiency (even though they use the same number of queues per port), and it achieves an 80% of improvement over VOQsw.

Summing up, analysis of hot-spot results leads to similar conclusions than those obtained from the uniform traffic analysis: BBQ reaches the best performance (except VOQnet) with a smaller or equal number of queues per port (only 4) than other schemes.

5 Conclusions and Future Work

Head-of-Line (HoL) blocking is a phenomenon that may dramatically degrade the performance of current high-speed interconnection networks if not properly managed. Although many techniques have been proposed to deal with this problem, only some of them are actually feasible in current switches, mostly based on queuing schemes. Many of these feasible queuing schemes are not aware of network topology or routing algorithm, thus they do not leverage the available queues in certain scenarios. In this paper we propose Band-Based Queuing (BBQ), a straightforward queuing scheme that efficiently reduce HoL-blocking exploiting the properties of networks with a specific hybrid topology that use Hybrid-DOR as routing algorithm. From the results of this paper, we can conclude that BBQ significantly reduces HoL-blocking either in uniform or hot-spot traffic scenarios while requiring a small number of queues per port (even in networks with thousands of endnodes), outperforming or matching other queuing schemes that require more queues per port. Moreover, as BBQ is based on a simple but smart policy to map packets to queues, it could be easily implemented in real InfiniBand networks, as we have explained in the paper and as we plan to do in the near future.

Acknowledgment. This work has been jointly supported by the Spanish MINECO and European Commission under the project TIN2012-38341-C04-04 and by the JCCM under project and POII10-0289-3724.

References

1. Karol, M.J., Hluchyj, M.G., Morgan, S.P.: Input versus output queuing on a space-division packet switch. IEEE Transactions on Communications COM-35, 1347–1356 (1987)
2. InfiniBand Trade Association: InfiniBand architecture specification volume 1. Release 1.2.1
3. Myrinet, 2000 Series Networking (February 2013), `https://www.myricom.com/`
4. García, P.J., Flich, J., Duato, J., Johnson, I., Quiles, F.J., Naven, F.: Dynamic Evolution of Congestion Trees: Analysis and Impact on Switch Architecture. In: Conte, T., Navarro, N., Hwu, W.-M.W., Valero, M., Ungerer, T. (eds.) HiPEAC 2005. LNCS, vol. 3793, pp. 266–285. Springer, Heidelberg (2005)
5. Jurczyk, M., Schwederski, T.: Phenomenon of Higher Order Head-of-Line Blocking in Multistage Interconnection Networks under Nonuniform Traffic Patterns. IEICE Transactions on Information and Systems E79-D(8), 1124–1129 (1996)
6. Dally, W., Carvey, P., Dennison, L.: Architecture of the Avici terabit switch/router. In: 6th Hot Interconnects, pp. 41–50 (1998)
7. García, P.J., Flich, J., Duato, J., Johnson, I., Quiles, F.J., Naven, F.: Efficient, Scalable Congestion Management for Interconnection Networks. IEEE Micro 26(5), 52–66 (2006)
8. Escudero-Sahuquillo, J., Garcia, P.J., Quiles, F.J., Flich, J., Duato, J.: An Effective and Feasible Congestion Management Technique for High-Performance MINs with Tag-Based Distributed Routing. IEEE Transactions on Parallel and Distributed Systems PP(99) (2012)
9. Tamir, Y., Frazier, G.: Dynamically-Allocated Multi-Queue Buffers for VLSI Communication Switches. IEEE Transactions on Computers 41(6), 725–737 (1992)
10. Dally, W.: Virtual-Channel Flow Control. IEEE Trans. on Parallel and Distributed Systems 3(2), 194–205 (1992)
11. Anderson, T., Owicki, S., Saxe, J., Thacker, C.: High-Speed Switch Scheduling for Local-Area Networks. ACM Transactions on Computer Systems 11(4), 319–352 (1993)
12. Escudero-Sahuquillo, J., García, P.J., Quiles, F.J., Flich, J., Duato, J.: OBQA: Smart and cost-efficient queue scheme for Head-of-Line blocking elimination in fat-trees. J. Parallel Distrib. Comput. 71(11), 1460–1472 (2011)
13. Escudero-Sahuquillo, J., García, P.J., Quiles, F.J., Flich, J., Duato, J.: Cost-effective queue schemes for reducing head-of-line blocking in fat-trees. Concurrency and Computation: Practice and Experience 23(17), 2235–2248 (2011)
14. Gomez, C., Gilabert, F., Gomez, M., Lopez, P., Duato, J.: Deterministic versus Adaptive Routing in Fat-Trees. In: Workshop CAC in Conjunction with the IPDPS, p. 235 (March 2007)
15. Penaranda, R., Gomez, C., Gomez, M., Lopez, P., Duato, J.: A New Family of Hybrid Topologies for Large-Scale Interconnection Networks. In: 2012 11th IEEE International Symposium on Network Computing and Applications (NCA), pp. 220–227 (August 2012)

16. Kim, J., Dally, W.J.: Flattened butterfly: A cost-efficient topology for high-radix networks. In: Proc. of the Intl. Symp. on Computer Architecture (2007)
17. Kim, J., Dally, W.J., Scott, S., Abts, D.: Technology-Driven, Highly-Scalable Dragonfly Topology. In: ISCA 2008: Proceedings of the 35th Annual International Symposium on Computer Architecture, pp. 77–88 (2008)
18. Katevenis, M., Serpanos, D., Spyridakis, E.: Credit-Flow-Controlled ATM for MP Interconnection: The ATLAS I Single-Chip ATM Switch. In: Proc. 4th HPCA, pp. 47–56 (1998)
19. Guay, W.L., Reinemo, S.A., Lysne, O., Skeie, T.: dFtree: A fat-tree routing algorithm using dynamic allocation of virtual lanes to alleviate congestion in infiniband networks. In: Proceedings of the First International Workshop on Network-aware Data Management, NDM 2011, pp. 1–10. ACM, New York (2011)
20. Nachiondo, T., Flich, J., Duato, J.: Buffer Management Strategies to Reduce HoL-Blocking. IEEE Transactions on Parallel and Distributed Systems 21(6), 739–753 (2010)
21. Olesinski, W., Eberle, H., Gura, N.: Scalable alternatives to virtual output queueing. In: Proc. IEEE ICC, pp. 1–6 (2009)
22. Guay, W.L., Bogdanski, B., Reinemo, S.A., Lysne, O., Skeie, T.: vFtree - A Fat-Tree Routing Algorithm Using Virtual Lanes to Alleviate Congestion. In: Proc. of IPDPS, pp. 197–208 (2011)
23. Zahavi, E., Johnson, G., Kerbyson, D.J., Lang, M.: Optimized InfiniBand^{TM} fat-tree routing for shift all-to-all communication patterns. Journal of CCPE 22(2), 217–231 (2010)
24. Yebenes, P., Escudero-Sahuquillo, J., Garcia, P., Quiles, F.: Towards Modeling Interconnection Networks of Exascale Systems with OMNet++. In: 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 203–207 (2013)
25. Varga, A.: OMNeT++ User Manual, `http://omnetpp.org/doc/omnetpp/manual/usman.html`

Accelerating Communication-Intensive Parallel Workloads Using Commodity Optical Switches and a Software-Configurable Control Stack

Diego Lugones[1], Konstantinos Christodoulopoulos[2], Kostas Katrinis[3], Marco Ruffini[2], Donal O'Mahony[2], and Martin Collier[1]

[1] The Rince Institute, Dublin City University
`diego@rince.ie`
[2] School of Computer Science and Statistics, Trinity College Dublin, Ireland
`christok@tcd.ie`
[3] IBM Research, Ireland
`katrinisk@ie.ibm.com`

Abstract. In response to the need for faster and fatter networks for large-scale HPC cluster systems, hybrid optical/electrical networks have been proposed as an affordable and high-capacity solution. Still, there is no prior work evaluating the performance of HPC workloads over such types of networks. To fill this gap, this work presents a hybrid network architecture comprising commodity-only equipment, shows its price competitiveness against fat-tree alternatives and presents a prototype implementation. We evaluated several HPC workloads over our prototype, showing that our hybrid optical/electrical network manages to significantly accelerate tested workloads, without incurring any extra cost compared to an all-electronic fat-tree network.

Keywords: high-performance computing, high-speed networks, interconnects, distributed systems.

1 Introduction

The increasing compute density in modern High-Performance Computing (HPC) system as a result of higher-core integration and the use of specialized accelerators is among others pushing the need for high-speed networks that are faster and with higher capacity across all levels of their hierarchies. The latter requirement, especially when seen at large-scale, leads to massive capital and management costs, magnifying the contribution of the network to total system cost. In response to this, prior research has proposed using commercial off-the-shelf optical switches for aggregating traffic between racks, partly or entirely replacing the multiple hierarchies of electronic networks and leveraging on the interesting features exhibited by such devices, such as lower cost/port and immense rate/port capability. However, very little is still known about the impact that hybrid optical/electrical networks have to HPC applications' performance and at what cost.

To address this challenge, we lay out in this paper the architecture of such a system at scale using commodity-only single-vendor equipment, calculate its total price using current list prices and compare it against the total investment for a conventional fat-tree at various capacity levels, showing that our hybrid solution is more affordable, up to 31%. We then present a fully-functional research prototype of our system architecture, featuring among others: a) a network controller that is capable of accepting workload communication pattern input and re-configuring the network in a manner that optimizes application execution and b) an end-system shim-layer to allow compute servers to route over our network without modifications to running applications or the operating system. The controller implements optimization heuristics presented in our previous work [1][2] and exposes a high-level programming interface that can be use to try out further topology optimization algorithms.

We deployed our software stack in a 40-servers/4-rack testbed in our lab and used our experimental setup to compare the performance of communication-intensive HPC kernels and pseudo-applications running over our architecture against the performance obtained over equal-cost fat-tree setups. Our results manifest that - at equal cost to fat-trees - our hybrid network system implementation manages to accelerate the workloads tested, yielding roughly up to 35% faster application execution.

This paper is structured as follows. Section 2 puts past related research in the context of our work. Section 3 presents our system architecture and its price competitiveness against fat-trees. We outline in Section 4 the main components of our system prototype that we used throughout our experimentation to obtain the results reported in Section 5. Section 6 summarizes the findings and contributions of this work and outlines future work in this field.

2 Related Work

Various hybrid interconnects have been proposed for high-performance clusters [3] and datacenter architectures [4] [5] [6]. The basic differences among these proposals can be found in the network level at which optical switching occurs, in the number and rate of the optical ports per connection point, and in the use of single vs. multi-hop connections over the optical network. In Helios [4], the optical network interconnects *pods* (i.e. sets of racks comprising 1024 servers), while in c-Through [5] and OSA [6] the basic block is the rack. Both [4] and [5] report only single-hop transmissions over the optical network, while [6] considers multi-hop connectivity, however, without including this feature as part of its topology re-configuration heuristic. It must be noted that including multi-hop connections in the optimization makes the topology computation quickly unaffordable for large systems and the implementation of the control software more challenging.

Beyond the architectural features and algorithmic approaches that we innovate on, this paper deviates from related work in the perspective we take on the problem. That is, in addition to assessing price competitiveness of hybrid optical/electrical interconnects, addressing the required system adaptations and showing viability, our end goal is to deliver on the untouched hypothesis as to whether such systems lead to better performance for parallel/distributed applications and to what extent. Our workload-centric approach is reflected in our work and specifically in this paper by developing a fully-functional prototype used to evaluate the cost-constrained performance of target parallel workloads.

3 System Architecture and Competitiveness

Incrementally to the architectures mentioned in section 2, we are interested in exploring the scalability limits posed by the hybrid architecture under the constraint of using commercially available equipment (cf. section 4), as well as comparing the cost competitiveness of the approach against currently employed network solutions.

3.1 Data- and Control-Plane Architecture

We depict a full-scale embodiment of our system architecture in Figure 1, comprising 320 server racks and a dual network option: a) a high-speed single-level circuit-switched network driven by high-speed (10Gbps in this embodiment) Ethernet Top of Rack (TOR) switches (depicted as *TOR-X* in Figure 1) and b) a lower-rate, packet-switched Ethernet network driven by lower-rate (1Gbps in this embodiment) Ethernet TOR switches (depicted as *TOR-B-X* in Figure 1). The server integration factor (32 servers/rack) stems from the currently "standard" 64-port density of high-end 10Gbps Ethernet switches used as TOR switches, allowing construction of full bisection bandwidth trees for racks of such integration. The high-speed network is implemented with commodity Micro Electro Mechanical Systems (MEMS) optical switches that exhibit interesting features (cost/port, rate-free, protocol-agnostic, low power consumption) to be used as cluster/datacenter interconnects. The ability to arbitrarily cross-connect any pair of ports of any MEMS switch enables direct low-latency connectivity between racks, as opposed to the cumbersome switching and high-latency that multi-level electronic interconnects suffer from (e.g. fat-trees). Still, MEMS optical switches suffer inherently from high - relative to the transmission time of a typical packet or message size at 10Gbps - switching latency (in the order of tenths of milliseconds) and therefore can only be perceived as circuit switching elements, carrying high-volume, long-lived flows between pairs of racks in our system. Lower-rate communication (e.g. short messages, barriers, application signaling), occurs in our system via the lower-end electronic network built out of inexpensive Ethernet switches that are arranged in a highly over-subscribed tree topology (bisection bandwidth is 12.5% of the full in the embodiment shown in Figure 1).

In the control-plane, the low-rate electronic part of our network architecture can be realized with well-researched solutions (e.g. [7]) for implementing large-scale networks over redundant topologies of Ethernet switches without applying

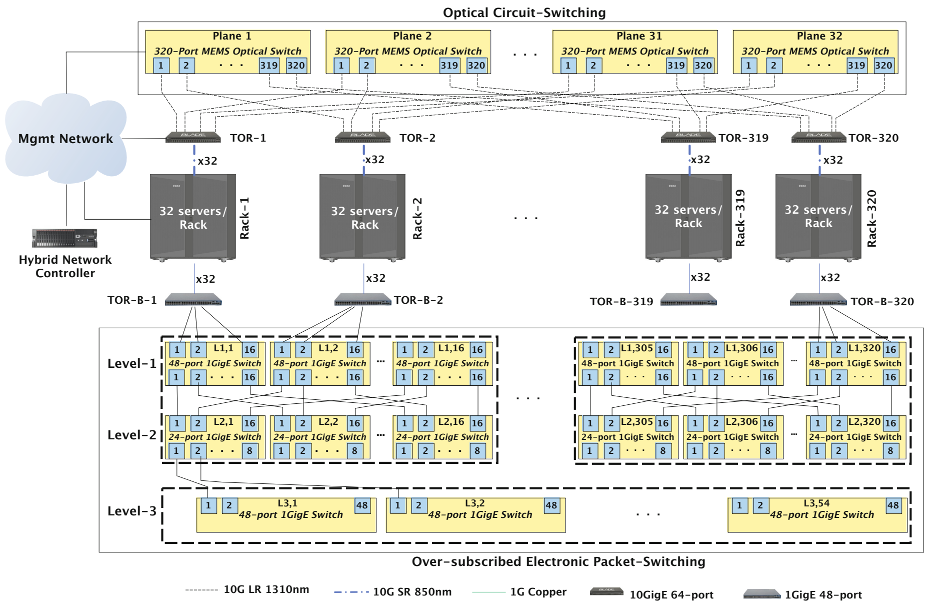


Fig. 1. Cluster architecture comprising a hybrid optical/electrical network spanning 10240 servers at an integration factor of 32 servers/rack

modifications to hardware or protocol standards. We believe that such solutions have matured, therefore we focus our research and system prototyping efforts on the control-plane of the optical part of our network architecture. As shown in Figure 1, all devices comprising the network (MEMS optical switches and TOR switches) and server racks connect via a low-rate management network to a dedicated server hosting specialized software that implements a *network controller*. The primary role of the network controller is to periodically, or upon application request, configure the optical part of the network in a manner that benefits the execution of parallel/distributed applications. At a high-level, the controller delivers this in a three step process: a) ingest input specifying application task mapping to server racks, b) calculate an optical network topology that maximizes throughput, constrained on available network resources (spare optical ports) and c) implement the computed optical network topology by applying corresponding cross-connections to the array of MEMS optical switches and by applying required state changes to TOR switches facing the optical MEMS switches. We elaborate further in the workings of our network controller and the specificities of the control-plane of the optical network part in section 4.

Our previous work [1] proposed efficient algorithms for solving steps a) and b) of the above process and showed the performance improvement they bring at various scales via simulation with application traces as input. The present work closes the loop of this part by addressing step c) of the aforementioned cycle in commodity systems, as well as by showing system-level feasibility and performance improvement brought to real workloads in a real system prototype.

Recognizing that in advance knowledge of the application communication pattern cannot be assumed across all parallel applications, we limit the scope of this work to a class of applications that we term "static". The term static refers here to the fact that these workloads exhibit per application logic (e.g. mesh simulations) a logical communication pattern that is invariant over application executions and that can be profiled through a test execution. Extending the scope of our research to embrace applications exhibiting dynamic communication patterns, as well as evaluate the performance impact of their co-existence with static parallel applications is part of our ongoing work.

3.2 Competitiveness Analysis

Since the changes we are proposing are to a great extent disruptive and not just incremental to an existing architecture, we assess in the following the estimated total list price of our network architecture and compare it against a conventional electronic packet network of equal nominal bisection bandwidth performance, namely a fat-tree implemented with top-end Ethernet switches [7]. The fact that our architecture is employing solely commodity off-the-shelf equipment is helpfull in assessing the price competitiveness of our approach. We recognize that list prices can be highly volatile, subject to market demand and the maturity of technology and therefore the following analysis can only serve as a snapshot of today that may not endure over time. Still, we contend that this is the only objective approach for drawing cost-related conclusions, when it is nearly impossible to scientifically reason about any mid- or long-term price trends (e.g. we had a hard time validating the cost trends reported in [4], even two years after their appearance).

We start with pricing our hybrid optical/electrical network solution for a cluster size of 10K server, equivalent to the system depicted in Figure 1. A hybrid network of this size can be readily built today using commercially available 320-port MEMS optical switches (e.g. Calient S320), while the rest of the equipment is commonly used in building high-end clusters and datacenters. We list the equipment description, corresponding prices per item and symbols used to refer to each equipment type in Table 1. All list prices used were drawn from publicly available sources [8], with the exception of the list price of the optical MEMS switch port, for which we used an averaged representative list price after discussions we had with respective vendors. Implementing the three levels of the low-rate electronic part of the hybrid network requires $\#S1_{1G} = 694$ 48-port 1G switches, $\#S2_{1G} = 320$ 24-port 1G switches and $\#C_C = 7680$ copper cables. We then calculate the total price of the optical part of the network, as a function of the nominal bisection bandwidth of the optical network part. For this, we use an integer parameter β that denotes the divisor that needs to be applied to the full bisection bandwidth to derive the nominal bisection bandwidth of the optical part of the network. For instance, β=1 corresponds to the full-bisection bandwidth setup shown in Figure 1, while β=4 corresponds to applying 1:4 over-subscription to the network that optically connects racks (or equivalently, that each server can source/sink at a maximum off-rack traffic rate of 2.5Gbps

Table 1. List of equipment and corresponding list price/item used in the analysis

Symbol	Equipment Name	Equipment Description	Price/Item [$]
S_{10G}	IBM RackSwitch G8264R	10Gbps Ethernet TOR switch	30,000
T_{LR}	IBM SFP+ SR Transceiver	10Gbps 850nm Transceiver	665
T_{SR}	IBM SFP+ LR Transceiver	10Gbps 1310nm Transceiver	1600
OPT	MEMS Switch (96-320 ports)	Price per optical port	340
$S1_{1G}$	Juniper 48 Port 1Gb EX2200	1Gbps Ethernet Switch	3595
$S2_{1G}$	Juniper 24 Port 1Gb EX2200	1Gbps Ethernet Switch	1995
C_{MM}	LC-LC 50μm Fiber Cable	Multi-mode fiber cable	28
C_{SM}	9μm Fiber Cable	Single-mode fiber cable	25
C_C	Cat5 Copper Cable	Copper cable for Gb Ethernet	10

at full network load). We note that over-subscription leads to fewer fiber links between the TOR switches and the optical array and thus to a reduction in the number of optical MEMS switches ("optical planes") required. Following from the above, implementing the optical part of the network for 10K servers requires $\#OPT = \frac{32}{\beta}$ 320-port MEMS optical switches, $\#T_{LR} = \frac{320\cdot32}{\beta}$ LR-transceivers, $\#C_{SM} = \frac{320\cdot32}{\beta}$ single-mode fiber cables, $\#S_{10G} = 320$ 10G Ethernet TOR switches, $\#T_{SR} = 320\cdot32$ SR-transceivers and $\#C_{MM} = 320\cdot32$ multi-mode fiber cables. Next, we breakdown the equipment quantity required to realize a fully electronic fat-tree (again parametrically to its bisection bandwidth) built out of 64-port 10G low-latency Ethernet switches (e.g. IBM RackSwitch G8264R) as in [7]. Due to space limitations, we defer here a concise presentation of the fat-tree structure and dimensioning and refer the reader to [9]. Parametrically to β carrying the semantic defined above, implementing an all-electronic fat-tree in this manner requires three levels and particularly: $\#S_{10G} = 320 + \frac{320}{\sqrt{\beta}} + \frac{160}{\beta}$ 10G 64-port Ethernet switches, $\#T_{SR} = 10240 + \frac{20480}{\sqrt{\beta}} + \frac{20480}{\beta}$ SR-transceivers and $\#T_{SR} = 5120 + \frac{10240}{\sqrt{\beta}} + \frac{10240}{\beta}$ multi-mode fiber cables. We note that the $\sqrt{\beta}$ factor comes from the fact that over-subscription is applied in uniform multiplicative steps as we move from the first to the third level of the fat-tree.

Using the price values listed in Table 1 and the item quantities calculated for each network separately above, we calculated the total list price of a hybrid (resp. a fat-tree) network interconnecting a 10K server cluster and plot the results at various capacity levels in Figure 2. We observe that the hybrid network is by 31% cheaper at maximum capacity and remained cheaper compared to the all-electronic fat-tree throughout for all capacity levels up to the minimum capacity that is conceivable for the hybrid network (corresponding to β=32 or 10Gbps exiting the rack using one optical MEMS switch).

We note here that the cluster size picked for our analysis is the maximum, for we used the maximum size of MEMS switches that are commercially available today. Scaling such setups beyond this limit would require either building denser switches (1024-port MEMS switches have been shown in-vitro [3]) or experimenting with multi-stage alignments of existing commercial optical switches (constrained on optical performance requirements). The latter path forms part of

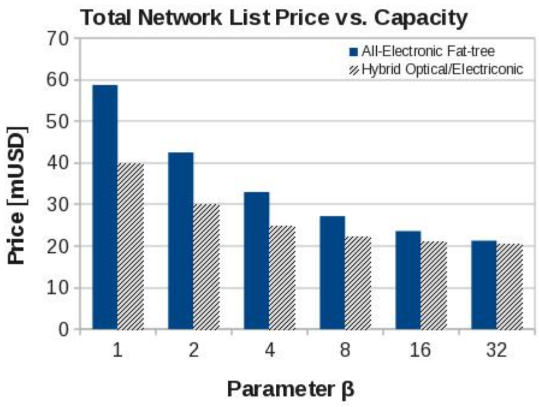


Fig. 2. List price of a hybrid network (resp. a full-electronic fat-tree) interconnecting a 10K server cluster at various capacity levels. The parameter β denotes the nominal bisection bandwidth of the network as the fraction of full bisection bandwidth.

our future agenda for creating wider scale-out designs. Last, we note that there is potential in enhancing our cost sizing with a sensitivity analysis on prices of specific equipment. This forms a task of our future work on the problem.

4 Network Control and Host Adaptations

4.1 Network Controller

The role of the network controller is to ingest input expressing the communication requirements of a mapped workload, compute a "good" configuration of the (re-configurable) optical network for the given input and take all necessary control-plane actions to enforce the computed configuration on all involved devices. We depict a toy but illustrative example showing the steps taken by the controller upon receiving a request to match the optical infrastructure to an input workload in Figure 3. The input comes in the form of a traffic matrix, whereby each matrix element corresponds to the (normalized) volume of communication between two processing elements (cores). Following a clustering step to derive the rack-level traffic matrix and given the physical connectivity (wiring between TORs and optical switches) that the controller discovers during its initialization phase, the controller computes in the next step a connectivity graph between the racks involved, aiming at minimizing average traffic load throughout workload execution and thus speeding up workload completion. We haved presented the theoretical and optimization underpinnings of these steps and evaluated the performance of our topology configuration heuristic via simulation in [1]. We have implemented these in our network controller prototype for the purpose of showing viability and to obtain the performance evaluation results reported in the next section.

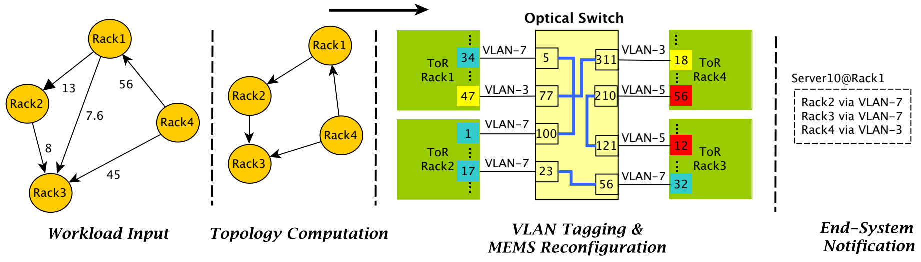


Fig. 3. Steps undertaken by the hybrid network controller to match the optical part of the hybrid network to the input workload

In the next phase, the controller enforces the computed workload-specific topology to the physical network. For this, it sends the right set of commands (via the specialized TCP-based API) to all optical switches involved to cross-connect the pairs of ports corresponding to the computed connectivity between TOR switches. In parallel, the controller tags the TOR switch ports with VLAN-IDs, whereby each circuit is assigned a distinct (in the broadcast domain it touches) VLAN-ID. The reason we choose to operate our forwarding substrate using VLANs is to allow parallel links, rings and generally setup of paths that would otherwise be impossible, had we used Ethernet's spanning tree routing. A similar approach has been employed in [10] using static VLAN allocation, which is though shown not to scale. Instead, we measured that our controller is capable of installing VLANs for up to 32 TOR switch ports in less than 1.5 seconds and therefore we employ dynamic VLAN allocation for increased scalability. Generally, the multi-threaded implementation of our controller achieves state installation across all network devices in less than 2 seconds, which is a negligible overhead compared to the runtime of scaled-out workloads. Last, it is important to note that unlike alternative solutions, our system is able to utilize ”multi-hop” communication, i.e. have a flow traverse the optical array multiple times until it reaches its destination TOR. This increases the search space for good topologies, while we have also been able to obtain better sharing and thus utilization of the optical resources.

4.2 End-System Support

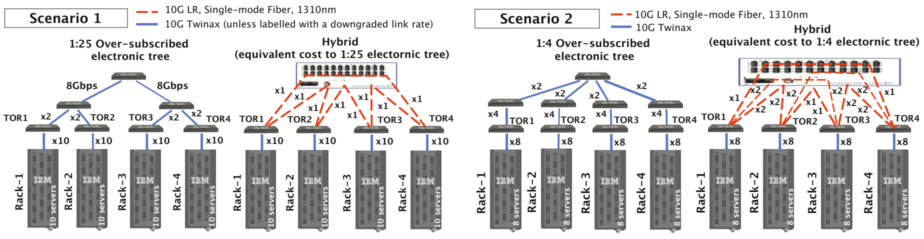
We leverage on the vast set of configuration tools and networking software available at commodity servers to send/receive packets to/from the two networks existing in the hybrid system, in fact without applying any modification to the underlying operating system or the application(s). For this, we injected a custom *translation shim layer* into the network stack of each server. The translation service uses as input the connectivity information communicated by the network controller (see last step in Figure 3) and creates the required virtual network interfaces accordingly. Given that each server has two network interfaces (leading to the optical network or the low-rate electronic network), the shim layer needs to decide which interface to pick to forward the packets of a specific workload. We accomplish this in a manner transparent to applications by having our shim layer rewrite the IP source/destination header values of each packet using the NAT feature of iptables. Specifically to the optical network case, the shim layer rewrites the IP address headers in a way that the packets are routed via the right VLAN and thus the circuit that leads to the destination rack of the packets. We defer here due to space limitations a more thorough presentation of the internal workings of our shim layer, which we plan to report in future public communication.

5 System Validation and Evaluation Results

We conducted various trials to validate our system prototype and targeted experiments to compare the performance of our solution to that of standard electronic tree-based solutions. Our testbed comprised 40 servers (12 cores each) mounted in 4 racks, eight 10G Ethernet ToR switches (IBM RackSwitch G8264) and one MEMS optical switch with 96 bi-directional ports (Crossfiber Liteswitch 96). Each server connects via a 10G SR-transceiver to each rack's ToR switch and each of the 4 ToR switches that have servers attached connects via 10 LR single-mode transceivers to the MEMS switch. We also used four additional 10G Ethernet switches to create a slice of an all-electronic fat-tree network. Our network controller ran on a dedicated server that connects via an 1G management network to the management ports of the TOR switches and the optical switch, as well as to all servers.

The rationale behind the creation of our experimental scenarios is as follows: constrained on the scale of our prototype (10 fiber links from each TOR to the optical switch), our goal was to compare instantiations of our hybrid network prototype against **equal-cost** instantiations of an all-electronic fat-tree; and in fact do so using real parallel workloads. To this end, we used our cost models to obtain two scenarios, each comprising two equal-cost network instantiations: a) scenario-1 compares a hybrid network with 6 TOR-to-optical fiber links against a 1:25 over-subscribed fat-tree and b) scenario-2 compares a hybrid network with 20 TOR-to-optical fiber links against a 1:4 over-subscribed fat-tree.

Our use-case involves a 10K multi-tenant cluster (or datacenter) with 32 servers per rack and a user requesting to execute a parallel job. The user is effectively allocated the requested number of servers in different racks. To address the general case, where this allocation may lead to racks without physical proximity - due to resource fragmentation or for better resilience against shared risk failures - we force inter-rack communication in the three-level fat-tree case to traverse the root of the tree. For scenarios 1 and 2 we assume the use of 10 and 8 servers in each of the 4 racks, respectively, and a uniform bandwidth allocation to servers in each rack. As such, in both the hybrid and the fat-tree networks the 10 servers in each rack in scenario-1 are allocated 1/3 of the available inter-rack bandwidth (8 Gbps and 2x10 Gbps for the tree and the hybrid respectively), while the 8 servers in scenario 2 are allocated 1/4 of the available


Scenario 1
10G LR, Single-mode Fiber, 1310nm
10G Twinax (unless labelled with a downgraded link rate)
1:25 Over-subscribed electronic tree
8Gbps
8Gbps
Hybrid (equivalent cost to 1:25 electornic tree)
TOR1
TOR2
TOR3
TOR4
x1
x2
x10
Rack-1
Rack-2
Rack-3
Rack-4
10 servers
IBM
Scenario 2
10G LR, Single-mode Fiber, 1310nm
10G Twinax
1:4 Over-subscribed electronic tree
Hybrid (equivalent cost to 1:4 electornic tree)
x4
x8
8 servers

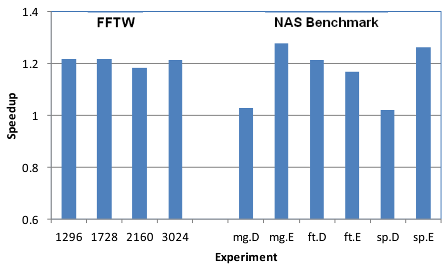
(a) Scenario 1 (b) Scenario 2

Fig. 4. Network configurations implementing the two experiment scenarios for the two network types under test in our testbed

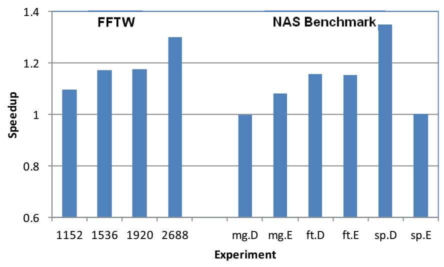
inter-rack bandwidth (2x10 Gbps and 5x10 Gbps for the tree and the hybrid respectively). Figures 4a and 4b illustrate the instantiations of the fat-tree and hybrid networks according to the two scenarios outlined above.

We executed the following MPI parallel applications over all four network configurations: FFTW [11] which is a discrete Fast-Fourier Transform kernel, the FT (discrete 3D fast Fourier Transform) kernel, the MG (Multi-Grid on a sequence of meshes) kernel, and the SP (Scalar Penta-diagonal solver) pseudo-application; the last three are part of the NAS Parallel Benchmarks (NPB) suite [12]. For scenario-1 (40 servers in total) we used 4 input sizes for the FFTW ranging from 1296x1296x1296 to 3024x3024x3024, while for scenario-2 (32 servers in total) we used again 4 input sizes ranging from 1152x1152x1152 to 2688x2688x2688 . For FT, MG, SP NAS benchmarks we executed class D and E problem sizes. In the case of the hybrid network, the parallel execution involved using our network controller stack and utilizing our VLAN and translation shim-layer solution. Note that in the hybrid network configurations of both scenarios the constructed topologies include loops, which would be broken by disabling one or more link, if standard Ethernet switching was used. Instead, our VLAN-based routing enabled the loops, thus yielding higher throughput over the same network configuration.

Figure 5a shows the *speedup* results obtained in the set of experiments for scenario-1 that is, for the case of 1:25 tree and the equivalent cost hybrid network and 40 servers in total. In this context, speedup is defined as the ratio of the completion time of a single application execution in the tree network over the completion time in the hybrid network. Results show a measurable acceleration across all tested workloads, reaching up to 30%. Figure 5b depicts the speedup results for scenario-2, that is, for the case of 1:4 tree and the equivalent cost hybrid network and 32 servers in total. The improvements in this scenario are similar to those observed in the 1:25 case. For small size problems (small fft problem sizes and NAS class D experiments) accelaration is low, for the capacity provided to the network in both cases is enough to satisfy the communication needs of the executed workloads. For large problems though, acceleration was higher, up to 35%. These workloads are bandwidth demanding and there is a



(a) Scenario 1



(b) Scenario 2

Fig. 5. Evaluation results of speedup achieved by the hybrid optical/electrical network over the all-electronic fat-tree to the various workloads tested in scenario-1(left) and scenario-2(right)

clear advantage of capacity in favor of the hybrid network, which accelerates the tested workloads.

6 Conclusions

Despite all the research effort put on system specification, prototyping and evaluation of system features for hybrid optical/electrical interconnects in support of large-scale server co-locations (HPC clusters and datacenters), none of these efforts has to the best of our knowledge reached production deployment to date. Although we recognize that the shift from an entirely packet-switched to a hybrid circuit-/packet-switched world is per se not easy, we contend that the above is to a great extent due to the lack of evidence with regard to the benefit that such a shift can bring to applications. To address this gap, this work presented essentially a cost/benefit analysis for such systems. In particular, we delivered a concise price analysis of a hybrid interconnect comprising commodity parts and showed that it is cheaper compared to its most prominent competitor, namely a full electronic fat-tree. To deliver on the benefit part, we prototyped a network controller that computes efficient workload-input specific topologies and is capable of orchestrating the control-planes of the various network devices and the network stack of end-systems involved to create an optical substrate transparent to applications using it. We deployed our system prototype in a 4-rack testbed and showed through real experimentation that in most cases tested parallel workloads are accelerated at an equal network investment with a state-of-the-art solution.

Based on these promising findings, we are conducting work on expanding the range of applications evaluated, as well as scaling-out our testbed to enable larger-scale experiments. Our future agenda contains also dealing with applications with dynamically changing communication patterns and evaluating them in a multi-application scenario.

References

1. Christodoulopoulos, K., Ruffini, M., O'Mahony, D., Katrinis, K.: Topology configuration in hybrid EPS/OCS interconnects. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 701–715. Springer, Heidelberg (2012)
2. Lugones, D., Katrinis, K., Collier, M.: A reconfigurable optical/electrical interconnect architecture for large-scale clusters and datacenters. In: Conf. Computing Frontiers, pp. 13–22 (2012)
3. Barker, K.J., Benner, A., Hoare, R., Hoisie, A., Jones, A.K., Kerbyson, D.K., Li, D., Melhem, R., Rajamony, R., Schenfeld, E., Shao, S., Stunkel, C., Walker, P.: On the feasibility of optical circuit switching for high performance computing systems. In: Proc. ACM/IEEE SC 2005 Conf. Supercomp. (2005)
4. Farrington, N., Porter, G., Radhakrishnan, S., Bazzaz, H.H., Subramanya, V., Fainman, Y., Papen, G., Vahdat, A.: Helios: A hybrid electrical/optical switch architecture for modular data centers. SIGCOMM Comput. Commun. Rev. 40, 339–350 (2010)
5. Wang, G., Andersen, D.G., Kaminsky, M., Papagiannaki, K., Ng, T.E., Kozuch, M., Ryan, M.: c-through: Part-time optics in data centers. SIGCOMM Comput. Commun. Rev. 40, 327–338 (2010)
6. Chen, K., Singlay, A., Singhz, A., Ramachandranz, K., Xuz, L., Zhangz, Y., Wen, X., Chen, Y.: Osa: An optical switching architecture for data center networks with unprecedented flexibility. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI 2012, p. 18. USENIX Association, Berkeley (2012)
7. Greenberg, A., et al.: Vl2: A scalable and flexible data center network. In: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication, SIGCOMM 2009, pp. 51–62. ACM, New York (2009)
8. IBM: ibm.com. (2013), `http://www.ibm.com/`
9. Dally, W., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers Inc., San Francisco (2003)
10. Zhang, X.J., Wagle, R., Giles, J.: Vlan-based routing infrastructure for an all-optical circuit switched LAN. In: Proceedings of the 28th IEEE Conference on Global Telecommunications, GLOBECOM 2009, pp. 365–370. IEEE Press, Piscataway (2009)
11. Frigo, M., Johnson, S.: The design and implementation of fftw3. Proceedings of the IEEE 93(2), 216–231 (2005)
12. NASA: Nas parallel benchmarks (2013), `http://www.nas.nasa.gov/publications/npb.html/`

Dynamic Protocol Tuning Algorithms for High Performance Data Transfers

Engin Arslan, Brandon Ross, and Tevfik Kosar

Department of Computer Science & Engineering
University at Buffalo (SUNY), Buffalo NY 14260, USA
{enginars,bwross,tkosar}@buffalo.edu

Abstract. Obtaining optimal data transfer performance is of utmost importance to today's data-intensive distributed applications and wide-area data replication services. Doing so necessitates effectively utilizing available network bandwidth and resources, yet in practice transfers seldom reach the levels of utilization they potentially could. Tuning protocol parameters such as pipelining, parallelism, and concurrency can significantly increase utilization and performance, however determining the best settings for these parameters is a difficult problem, as network conditions can vary greatly between sites and over time. In this paper, we present four application-level algorithms for heuristically tuning protocol parameters for data transfers in wide-area networks. Our algorithms dynamically tune the number of parallel data streams per file, the level of control channel pipelining, and the number of concurrent file transfers to fill network pipes. The presented algorithms are implemented as a standalone service as well as being used in interaction with external data scheduling tools such as Stork. The experimental results are very promising, and our algorithms outperform existing solutions in this area.

Keywords: Application-level protocol tuning, throughput optimization, wide-area networks, data-intensive applications, data replication.

1 Introduction

Despite the increasing availability of high-speed wide-area networks and the use of modern data transfer protocols designed for high performance, file transfers in practice often only attain fractions of theoretical maximum throughputs, leaving networks underutilized and users unsatisfied. This fact is often due to a number of confounding factors, such as underutilization of end-system CPU cores, low disk I/O speeds, server implementations not taking advantage of parallel I/O opportunities, traffic at inter-system routing nodes, and unsuitable system-level tuning of networking protocols.

The effects of some of these factors can be mitigated to varying degrees through the use of techniques such as command pipelining, transfer-level parallelism, and concurrent transfers using multiple control channels. The degree to which these techniques are utilized, however, has the potential to negatively impact the performance of the transfer and the network as a whole. Too little use of one technique, and the network might be underutilized; too much, and the network might be overburdened to the detriment of the transfer and other users. Furthermore, the optimal level of usage for each technique varies depending on network and end-system conditions, meaning no combination of parameters is optimal for every scenario.

Dynamic optimization techniques provide a method for determining which combination of parameters is "just right" for a given transfer. This paper proposes optimization techniques that try to maximize transfer throughput by choosing optimal parallelism, concurrency, and pipelining levels through file set analysis and clustering. Our algorithms also re-provision idle control channels dynamically to improve the performance of "slower" file clusters, ensuring that resources are effectively utilized.

In this paper, we present four application-level algorithms for heuristically tuning protocol parameters for data transfers in wide-area networks. Our algorithms can tune the number of parallel data streams per file (for large file optimization), the level of control and data channel pipelining (for small file optimization), and the number of concurrent file transfers to fill network pipes (a technique useful for all types of files) in an efficient manner. The developed algorithms are implemented as a standalone service as well as being used in interaction with external data scheduling tools such as Stork [10,12]. The experimental results are very promising, and our algorithms outperform other existing solutions in this area.

2 Related Work

Liu et al. [14] developed a tool which optimizes multi-file transfers by opening multiple GridFTP control channels. The tool increases the number of concurrent flows up to the point where transfer performance degrades. Their work only focuses on concurrent file transfers, and other transfer parameters are not considered.

Globus Online [1] offers fire-and-forget GridFTP file transfers as a service. The developers mention that they set the pipelining, parallelism, and concurrency parameters to fixed values for three different file sizes (i.e. less than 50MB, larger than 250MB, and in between). However, the tuning Globus Online performs is non-adaptive; it does not change depending on network conditions and transfer performance.

Other approaches aim to improve throughput by opening flows over multiple paths between end-systems [17,8], however there are cases where individual data flows fail to achieve optimal throughput because of end-system bottlenecks. Several others propose solutions that improve utilization of a single path by means of parallel streams [2,6,16,23], pipelining [5,4,3], and concurrent transfers [13,11,14]. Although using parallelism, pipelining, and concurrency may improve throughput in certain cases, an optimization algorithm should also consider system configuration, since end-systems may present factors (e.g., low disk I/O speeds or over-tasked CPUs) which can introduce bottlenecks.

In our previous work [21], we proposed network-aware transfer optimization by automatically detecting bottlenecks and improving throughput by utilizing network and end-system parallelism.

We developed three highly-accurate models [24,22,9] which would require as few as three sampling points to provide accurate predictions for the optimal parallel stream number. These models have proved to be more accurate than existing similar models [7,16] which lack in predicting the parallel stream number that gives the peak throughput. We have developed algorithms to determine the best sampling size and the best sampling points for data transfers by using bandwidth, Round-Trip Time (RTT), or Bandwidth-Delay Product (BDP) [20].

3 Dynamic Protocol Tuning

Different transfer parameters such as pipelining, parallelism, and concurrency play a significant role in affecting achievable transfer throughput. However, setting the optimal levels for these parameters is a challenging problem, and poorly-tuned parameters can either cause underutilization of the network or overburden the network and degrade the performance due to increased packet loss, end-system overhead, and other factors.

Among these parameters, **pipelining** specifically targets the problem of transferring a large numbers of small files. In most control channel-based transfer protocols, an entire transfer must complete and be acknowledged before the next transfer command is sent by the client. This may cause a delay of more than one RTT between individual transfers. With pipelining, multiple transfer commands can be queued up at the server, greatly reducing the delay between transfer completion and the receipt of the next command. **Parallelism** sends different portions of the same file over parallel data streams (typically TCP connections), and can achieve high throughput by aggregating multiple streams and getting an unfair share of the available bandwidth. **Concurrency** refers to sending multiple files simultaneously using parallel control channels, and is especially useful for taking advantage of I/O concurrency in parallel disk systems.

The models developed in our previous work [21,24,22,9] lay the foundations of the dynamic protocol tuning algorithms presented in this paper, where we utilize all three parameters in combination to heuristically determine near-optimal network throughput.

In this paper, we present four dynamic protocol tuning algorithms:

1. The "Single-Chunk (SC)" approach, which divides the set of files into chunks based on file size, and then transfers each chunk with its optimal parameters;
2. the "Multi-Chunk (MC)" approach which likewise creates chunks based on the file size, but, rather than scheduling each chunk separately, it co-schedules and runs small-file chunks and large-file chunks together in order to balance and minimize the effect of poor performance of small file transfers;
3. the "Pro-Active Multi-Chunk (ProMC)" approach, which, instead of allocating channels equally among chunks, considers chunk size and type, and improves the performance if small chunks dominate the dataset; and

4. the “Max Fair MC (FairMC)” approach, which aims to make use of simultaneous chunk transfers but also tries to be fair in terms of network resource usage by limiting the maximum number of simultaneous chunk transfers.

3.1 Single-Chunk (SC) Algorithm

Files with different sizes need different transfer parameters to obtain optimal throughput. For example, pipelining and data channel reuse would mostly improve the performance of small file transfers, whereas per-file parallelism would be beneficial if the files are large. Optimal concurrency levels for different file sizes would be different as well. Instead of using the same parameter combinations for all files in a mixed dataset, we partition the dataset into chunks based on file size and Bandwidth Delay Product (BDP), and use different parameter combinations for each chunk.

As shown in Algorithm 1, we initially partition files into different chunks, then we check if each chunk has a sufficient number of files using the `mergePartitions` subroutine. We merge a chunk with another if it is deemed to be too small to be treated separately. After partitioning files, we calculate the optimal parameter combination for each chunk in `findOptimalParameters`. When calculating the density of a chunk, we take the average file size of the chunk and find its density in a similar way we do in `mergePartitions`.

Pipelining and concurrency are the most effective parameters at overcoming poor network utilization for small file transfers, so it is especially important to choose the best pipelining and concurrency values for such transfers. We set the pipelining values by considering the BDP and average file size of each chunk (lines 23, 27, 31 and 35); set the parallelism values by considering the BDP, average file size, and the TCP buffer size (lines 24, 28, 32, and 36); and set the concurrency values by considering the BDP, average file size, number of files in each chunk, and the maximum concurrency level (lines 25, 29, 33, and 37) in Algorithm 1.

As the average file size of a chunk increases, we decrease the pipelining value since it does not further improve performance, and can even cause performance degradation by poorly utilizing concurrent control channels. The method of selecting parallelism prevents using unnecessarily large parallelism levels for small files and insufficiently small parallelism levels for large files. Concurrency is set to larger values for small files, whereas for large files it is limited to smaller values, as higher concurrency values might cause unfair usage of end-system and network resources.

We tested our algorithms with concurrency levels up to 10. Although higher concurrency levels could possibly further increase throughput, the testing was performed on a shared testbed where it was against policy to open more than 10 file transfer connections at a time. Presumably many other shared network environments implement similar policies. Furthermore, throughput gains were found to experience diminishing returns as the concurrency level was increased. These factors led us to limit the maximum concurrency level our algorithms could reach to some safe fixed value – specifically 10.

Algorithm 1. — Partitioning Dataset and Setting Parameter Values

```
1: function PARTITIONFILES(allFiles,BDP)
2:     Chunk Small, Middle, Large, Huge
3:     while $allFiles.count() > 0$ do
4:         $File f = allFiles.pop()$
5:         if $f.size < \frac{BDP}{10}$ then
6:             Small.add(f)
7:         else if $f.size < \frac{BDP}{2}$ then
8:             Middle.add(f)
9:         else if $f.size < BDP * 20$ then
10:            Large.add(f)
11:        else
12:            Huge.add(f)
13:        end if
14:    end while
15:    allChunks.add(Small,Middle,Large,Huge)
16:    mergePartitions(allChunks)
17: return allChunks
18: end function
19: function FINDOPTIMALPARAMETERS(chunk,BDP,bufferSize,concurrency)
20:    Density d = findDensityofPartition(chunk)
21:    $avgFileSize = findAverage(chunk)$
22:    if $d == SMALL$ then
23:        $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil - 1$
24:        $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 1$
25:        $concurrency = Min(\frac{BDP}{avgFileSize}, chunk.count(), concurrency)$
26:    else if $d == MIDDLE$ then
27:        $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil$
28:        $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 1$
29:        $concurrency = Min(\frac{BDP}{avgFileSize}, chunk.count(), concurrency)$
30:    else if $d == LARGE$ then
31:        $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil + 1$        ▷ This chunk should have pipelining
32:        $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 2$
33:        $concurrency = Min(2, chunk.count(), concurrency)$
34:    else if $d == HUGE$ then
35:        $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil - 1$        ▷ Pipelining will be zero in most cases
36:        $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 2$
37:        $concurrency = Min(2, chunk.count(), concurrency)$
38:    end ifreturn pipelining,parallelism,concurrency
39: end function
```

After deciding the best parameter combination for each chunk, the Single-Chunk (SC) algorithm transfers each chunk one-by-one.

3.2 Multi-Chunk (MC) Algorithm

In the Multi-Chunk (MC) method, the focus is mainly on minimizing the effect of small file chunks on the overall throughput. Based on the results obtained from the SC approach, we deduced that even after choosing the best parameter combination for each chunk, the throughput obtained during the transfer of the small file chunks (called Small and Middle in Algorithm 1) is significantly worse compared to large chunks (Large and Huge in Algorithm 1) due to the high overhead of reading too many files from disk and underutilization of the

Algorithm 2. — Pro-Active Multi-Chunk (ProMC) Algorithm

```
function TRANSFER(source,destination,BW,RTT,concurrency)
    $BDP = BW * RTT$
    $allFiles = fetchFilesFromServer()$
    $chunks = partitionFiles(allFiles, BDP)$
    for $i = 0; i < chunks.length; i++$ do
        if $chunks[i] == SMALL$ then
            $weights[i] = 6 * chunks[i].size$
        else if $chunks[i] == MIDDLE$ then
            $weights[i] = 3 * chunks[i].size$
        else if $chunks[i] == LARGE$ then
            $weights[i] = 2 * chunks[i].size$
        else if $chunks[i] == HUGE$ then
            $weights[i] = 1 * chunks[i].size$
            $totalWeight = totalWeight + weights[i]$
        end if
    end for
    for $i = 0; i < chunks.length; i++$ do
        $weights[i] = weights[i]/totalWeight$      ▷ Calculate proportional weight of each chunk
        $channelAllocation[i] = \lfloor concurrency * weights[i] \rfloor$
    end for
    $transferChunks(chunks)$      ▷ Run chunks concurrently
end function
```

network pipe. Depending on the weight of small files relative to the total dataset size, overall throughput can be much less than the throughput of large file chunk transfers. Thus, we developed the MC method which aims to minimize the effect of poor transfer throughput of a dataset dominated by small files.

The MC method distributes data channels among chunks using round-robin in the order of Huge–Small–Large–Middle. The ordering of chunks provides better chunk distribution if the number of channels is less than the number of chunks. After channel distribution is completed, MC schedules chunks concurrently using the calculated concurrency level for each chunk.

The estimated completion time for each chunk is calculated every five seconds by dividing the remaining data size by the throughput of the chunk (i.e. the sum of the throughput for all channels for a given chunk). When the transfer of all files in a chunk is completed, the channels of the chunk are scheduled for other chunks based on their estimated completion time.

3.3 Pro-Active Multi-Chunk (ProMC) Algorithm

The way the MC approach distributes channels among chunks might be non-optimal if the weights of chunks are different. For example, if we have a dataset dominated by small files, then round-robin scheduling of channels may lead to sub-optimal channel allocation. This can cause sub-optimal transfer throughput since large chunks can be transferred more quickly than smaller chunks. The Pro-Active Multi-Chunk (ProMC) approach concentrates on more effectively distributing chunks among channels to improve the effectiveness of concurrency between the small and large chunks.

Channel allocation in the ProMC approach is demonstrated in Algorithm 2. ProMC also considers the type of a chunk when calculating its weight since the

Algorithm 3. — Max-Fair Multi-Chunk (FairMC) Algorithm

```
function TRANSFER(BW,RTT,BufferSize)
    BDP = BW*RTT
    allFiles = fetchFilesFromServer()
    chunks = partitionFiles(allFiles,BDP)
    if chunks contains Huge&Large chunk c then
        $c.channels++$                       ▷ Allocate a channel for huge&large chunk
        $concurrency--$
        if chunks contains Small&Middle chunks then
            allocateChannels(Small,Middle,concurrency);   ▷ If there exist Small&Middle chunks then allocate rest of channels to them
        else
            allocateChannel(Large,Huge,1);
        end if
    else
        allocateChannel(Small,Middle,concurrency) ▷ If there is no large chunk, then distribute given channels among Small&Middle chunks
    end if
    $transferChunks(chunks)$                 ▷ Run chunks concurrently
end function
```

transfer time of a chunk heavily depends its file distribution. Another way of achieving fairness among chunks in ProMC is dynamic channel allocation. It calculates the transfer completion time of each chunk periodically (by default, it is set to check every five seconds, similar to MC). If a chunk's completion time is calculated to be significantly less than another chunk's completion time for three consecutive periods, then a channel is taken over by the slow chunk from the faster chunk. Since channel transfer from one chunk to another is a costly operation, the threshold must be chosen carefully when comparing completion time differences. Also, rather than deciding on channel allocation after each period, ProMC waits three periods to make sure the estimated completion time difference is not a temporary condition.

3.4 Max-Fair Multi-Chunk (FairMC) Algorithm

The idea behind the Max-Fair Multi-Chunk (FairMC) approach is to make use of concurrent chunk transfers and to keep network and end-system utilization at a fair level. FairMC first calculates how many channels are needed for each chunk. Then, if small and large chunks exist, it opens only one channel for large chunks and uses the rest of the available channels for small chunks as shown in lines 5-9 of Algorithm 3. Otherwise, the channels are shared between small or large chunks as shown in lines 11 and 14. The goal here is to achieve high performance throughput without violating network fairness policies.

4 Performance Evaluation

We tested our experiments on XSEDE [18] and LONI [15] production-level high-bandwidth networks. Although both of the networks have 10G network bandwidth between sites, XSEDE provides higher throughput in end-to-end (disk-to-disk) transfers despite the high RTT between its sites. This is mainly due to the highly tuned and parallelized disk sub-systems at the XSEDE sites.

Table 1. Network specifications of test environments

Specs	**XSEDE**		**LONI**
	(Lonestar-Gordon)	**(Blacklight-Trestles)**	**(Queenbee-Painter)**
Bandwidth	10 Gbps	10Gbps	10 Gbps
RTT	60 ms	71 ms	10 ms
TCP Buffer Size	32 MB	32 MB	16 MB
BDP	75 MB	90 MB	9 MB

On XSEDE, we tested our dynamic protocol tuning algorithms between two different site pairs – Lonestar-Gordon and Blacklight-Trestles – with specifications given in Table 1. We also tested our algorithms using two different datasets where file sizes range between 3MB and 20GB. The datasets differ in the proportion of small files to the total dataset size. In the first one (referred to as "mixed"), small files are almost 35-40% of the total dataset, whereas they make up 55-65% of the second dataset (referred to as "small"). The purpose of using two different datasets is to demonstrate how our algorithms perform when the dataset is dominated by small or large files.

To analyze the effects of different parameters on the transfer of different file sizes, we initially conducted experiments for each of the parameters separately, as shown in Figure 1. We transferred each dataset, only changing one parameter (i.e., pipelining, parallelism, or concurrency) at a time to observe the individual effect of each parameter. Then we introduced other parameters one-by-one. These results show that concurrency is the most influential parameter for all file sizes on both networks, with parallelism being the second most. For this reason, we use concurrency as the pivot parameter in the comparison of different algorithms in this section. In all of our algorithms, it is assumed that RTT, bandwidth, and TCP buffer capacities are known beforehand. However, one can easily obtain RTT and TCP buffer capacity with negligible overhead. Available bandwidth can also be measured via bandwidth estimator tools (e.g. Iperf) with the cost of a couple of seconds.

We compared the performance of our four dynamic protocol tuning algorithms with Globus Online [1], PCP [19], and optimized globus-url-copy (GUC). Globus Online is a well-known data transfer service which uses a heuristic approach for transfer optimizations. The heuristic they use is similar to our basic Single-Chunk (SC) method in terms of dividing the dataset into chunks and running each chunk sequentially using different parameter sets. However, SC and Globus Online differ in the way they divide the chunks and in choosing the parameter set for each chunk. PCP employs a similar divide-and-transfer approach like SC and Globus Online using its own specific heuristic. For GUC, we set the pipelining to 30, parallelism to 4, and changed concurrency to different values (specifically, 2, 6, and 10) for different runs. We chose the pipelining and parallelism parameters in a way that they give close-to-best results based on our prior observations.

Results for Globus Online transfers are shown for concurrency level two as it always uses two channels for every chunk it creates. Although the PCP algorithm does not use a statically defined concurrency level, we observed that it

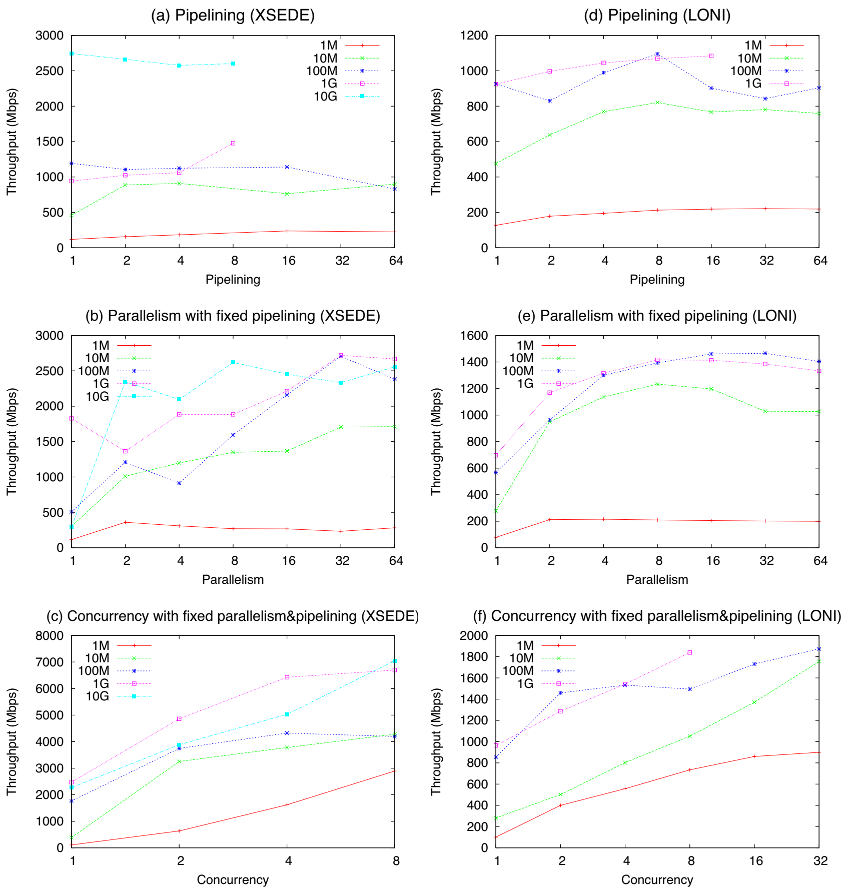


Fig. 1. Effect of combining parameters on throughput

generally choses a concurrency level between one and three; we set its performance histogram on concurrency level two. When concurrency level is set to two for all algorithms, almost all of our dynamic tuning algorithms perform better than PCP, Globus Online, and GUC. As we increase the concurrency level (as our dynamic algorithms do so), we can see significant performance improvement for all algorithms. However, SC is unable to improve the performance after concurrency level six while MC makes use of concurrency more effectively and its performance continues to increase in proportion to the concurrency level.

FairMC also improves in performance as concurrency increases. However, it does not perform as well as MC and ProMC, since it limits concurrency levels for large files and aims for fairness in lieu of maximum performance. ProMC and MC achieve similar performance in this case, since ProMC plays a significant role when small files dominate the data set. ProMC and MC perform better than GUC for all concurrency levels, which is mostly due to the efficient channel management of these two algorithms.

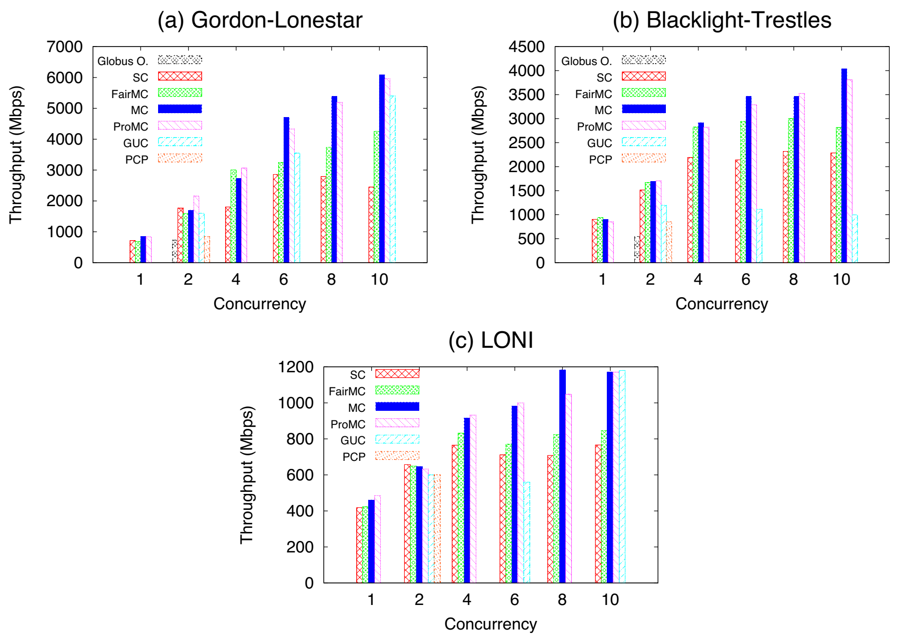
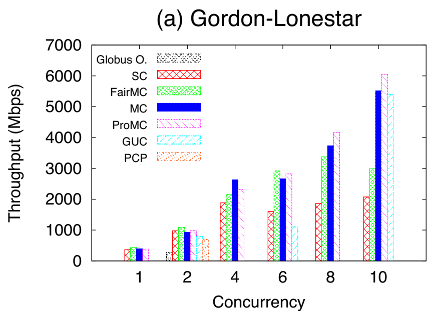


Fig. 2. Disk-to-disk transfer performance comparison with the mixed dataset

Figures 2(a) and 2(b) show the performance of optimization methods when used between different site pairs on XSEDE for the same dataset. Since disk read/write performance on Blacklight-Trestles is less than that on Gordon-Lonestar, throughput values obtained between these hosts are relatively smaller. GUC performance is very low compared to MC, since, when the pipelining is set in the GUC transfer, it statically sets the pipelining level for each channel to the number of transfer tasks. This means that it is possible for files to be assigned to the channels unequally. For example, one channel can be assigned to transfer the set of files contributing to the dominant portion of total dataset size. This will cause inefficient usage of channels, since, even if some channels finish their transfer tasks earlier, they will not be able to help others by sharing the remaining tasks.

We observed that LONI end-system disk performance is much lower than on XSEDE sites. This affected the results we obtained from LONI considerably as shown in Figure 2(c). Although the increased concurrency contributed positively to the throughput on LONI, the improvement is not as noticeable as it is on XSEDE.

In Figure 3, we observe that ProMC performed better than MC for almost all concurrency levels, as it allocates the channels to chunks more efficiently. It also monitors each chunk's performance and acts to re-allocate a channel from one chunk to another to minimize the negative effect of small file transfers on overall transfer performance.



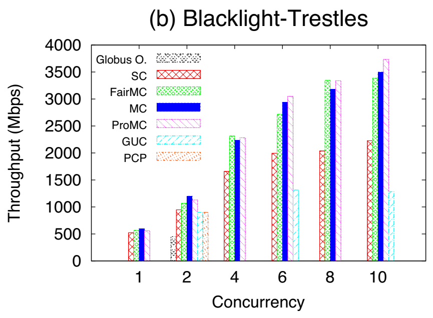


Fig. 3. Disk-to-disk performance comparison with the small file dominant dataset

5 Conclusions

We have presented four application-level algorithms for heuristically tuning protocol parameters for data transfers in wide-area networks. The parameters dynamically tuned by our algorithms (parallelism, pipelining, and concurrency levels) have shown to be very important factors in determining the ultimate throughput and network utilization obtained by many data transfer applications. Though determining the best combination for these parameter values is not a trivial task, we have shown that our algorithms can choose parameter combinations which yield demonstrably higher throughputs than those used in unoptimized transfers or chosen by less sophisticated heuristics.

Our algorithms were designed to be client-side techniques and operate entirely in user space, and thus special configurations at the server side or at the kernel level are not necessary to take advantage of them. The algorithms can be implemented as standalone transfer clients or as part of an optimization library or service. We plan to include these (and future algorithms based thereupon) in the Stork data scheduler [10,12] as well as our new Cloud-hosted transfer optimization suite, StorkCloud.

Acknowledgments. This project is in part sponsored by NSF under award numbers CNS-1131889 (CAREER), OCI-0926701 (STCI-Stork), and CCF-1115805 (CiC-Stork).

References

1. Allen, B., Bresnahan, J., Childers, L., Foster, I., Kandaswamy, G., Kettimuthu, R., Kordas, J., Link, M., Martin, S., Pickett, K., Tuecke, S.: Software as a service for data scientists. Communications of the ACM 55(2), 81–88 (2012)
2. Altman, E., Barman, D.: Parallel TCP sockets: Simple model, throughput and validation. In: Proceedings of IEEE INFOCOM (2006)
3. Bresnahan, J., Link, M., Kettimuthu, R., Fraser, D., Foster, I.: Gridftp pipelining. In: Proceedings of TeraGrid (2007)

4. Farkas, K., Huang, P., Krishnamurthy, B., Zhang, Y., Padhye, J.: Impact of TCP variants on HTTP performance. In: Proceedings of High Speed Networking, vol. 2 (2002)
5. Freed, N.: SMTP service extension for command pipelining, `http://tools.ietf.org/html/rfc2920`
6. Hacker, T.J., Noble, B.D., Athey, B.D.: Adaptive data block scheduling for parallel TCP streams. In: Proceedings of HPDC (2005)
7. Hacker, T.J., Noble, B.D., Atley, B.D.: The end-to-end performance effects of parallel TCP sockets on a lossy wide area network. In: Proc. of IPDPS (2002)
8. Khanna, G., Catalyurek, U., Kurc, T., Kettimuthu, R., Sadayappan, P., Foster, I., Saltz, J.: Using overlays for efficient data transfer over shared wide-area networks. In: Proceedings of SC, Piscataway, NJ, USA (2008)
9. Kim, J., Yildirim, E., Kosar, T.: A highly-accurate and low-overhead prediction model for transfer throughput optimization. In: Proceedings of ACM SC 2012 DISCS Workshop (2012)
10. Kosar, T.: A new paradigm in data intensive computing: Stork and the data-aware schedulers. In: Proceedings of IEEE HPDC 2006 CLADE Workshop (2006)
11. Kosar, T., Balman, M.: A new paradigm: Data-aware scheduling in grid computing. Future Generation Computing Systems 25(4), 406–413 (2009)
12. Kosar, T., Balman, M., Yildirim, E., Kulasekaran, S., Ross, B.: Stork data scheduler: Mitigating the data bottleneck in e-science. The Phil. Transactions of the Royal Society A 369(3254-3267) (2011)
13. Kosar, T., Livny, M.: Stork: Making data placement a first class citizen in the grid. In: Proceedings of ICDCS 2004, pp. 342–349 (March 2004)
14. Liu, W., Tieman, B., Kettimuthu, R., Foster, I.: A data transfer framework for large-scale science experiments. In: Proceedings of DIDC Workshop (2010)
15. LONI: Louisiana optical network initiative (LONI), `http://www.loni.org/`
16. Lu, D., Qiao, Y., Dinda, P.A., Bustamante, F.E.: Modeling and taming parallel TCP on the wide area network. In: Proceedings of IPDPS (2005)
17. Raiciu, C., Pluntke, C., Barre, S., Greenhalgh, A., Wischik, D., Handley, M.: Data center networking with multipath TCP. In: Proceedings of Hotnets-IX (2010)
18. XSEDE: Extreme Science and Engineering Discovery Environment, `http://www.xsede.org/`
19. Yildirim, E., Kim, J., Kosar, T.: How gridftp pipelining, parallelism and concurrency work: A guide for optimizing large dataset transfers. In: Proceedings of Network-Aware Data Management Workshop (NDM 2012) (November 2012)
20. Yildirim, E., Kim, J., Kosar, T.: Optimizing the sample size for a cloud-hosted data scheduling service. In: Proc. of IEEE/ACM CCGrid CCSA Workshop (2012)
21. Yildirim, E., Kosar, T.: Network-aware end-to-end data throughput optimization. In: Proceedings of Network-Aware Data Management Workshop (NDM 2011) (2011)
22. Yildirim, E., Yin, D., Kosar, T.: Prediction of optimal parallelism level in wide area data transfers. IEEE TPDS 22(12) (2011)
23. Yildirim, E., Yin, D., Kosar, T.: Balancing TCP buffer vs parallel streams in application level throughput optimization. In: Proceedings of DADC Workshop (2009)
24. Yin, D., Yildirim, E., Kosar, T.: A data throughput prediction and optimization service for widely distributed many-task computing. IEEE TPDS 22(6) (2011)

Topic 14+16: High-Performance and Scientific Applications and Extreme-Scale Computing

(Introduction)

Turlough P. Downes, Sabine Roller, Ari P. Seitsonen, Sophie Valcke, David Keyes, Marie-Christine Sawley, Thomas Schulthess, and John Shalf

Topic 14 and 16 Committees

As our understanding of the world around us increases it becomes more challenging to make use of what we already know, and to increase our understanding still further. Computational modeling and simulation have become critical tools in addressing this challenge. The requirements of high-resolution, accurate modeling have outstripped the ability of desktop computers and even small clusters to provide the necessary compute power. Many applications in the scientific and engineering domains now need very large amounts of compute time, while other applications, particularly in the life sciences, frequently have large data I/O requirements. There is thus a growing need for a range of high performance applications which can utilize parallel compute systems effectively, which have efficient data handling strategies and which have the capacity to utilise current and future systems. The High Performance and Scientific Applications topic aims to highlight recent progress in the use of advanced computing and algorithms to address the varied, complex and increasing challenges of modern research throughout both the "hard" and "soft" sciences. This necessitates being able to use large numbers of compute nodes, many of which are equipped with accelerators, and to deal with difficult I/O requirements.

Following seven orders of magnitude improvement in performance on applications over the past 24 years the road to extreme performance is encountering different challenges and becoming much steeper. Traditionally, our scientific computing code base is focused on optimizing floating point operations and improving the execution rate of those that remain, but this will not suffice in the future. On one hand, diverging exponentials in hardware subsystem performance require more attention to parallel programming. High concurrency and power-efficient design of the individual cores bring opposite pressures: greater data locality and greater freedom to redistribute data and computation. For reasons of energy efficiency and system acquisition cost, we must now focus on squeezing out synchronizations, total memory footprint, and limiting memory transfers. On the other hand, scientific applications are become more complex, for example, in order to address multiscale/multiphysics, reinforcing the requirements for numerical accuracy, synchronization and resiliency. Rethinking programming models, algorithmic implementations, and even mathematical models preferred as starting points will therefore be major milestones on the way to extreme scale. Topic 16 "Extreme-Scale Computing"[1] therefore covers papers charting the path of extreme simulation and data analytics in science and engineering onto emerging architectures, at all levels of the modeling chain.

The papers accepted for these topics address issues from scaling to large numbers of compute nodes where the algorithm in use makes this an inherently challenging prospect through to community detection in networked environments. Two papers are in what might be generally recognised as traditional HPC areas: ocean modeling and systems biology. A third paper tackles directly the challenge of getting data efficiently to the relevant compute device within a node in the context of bioinformatics. The subject matter of the remaining paper in this track reflects the broadening relevance of high performance computing and applications and it addresses the problem of identifying communities in online environments. We believe the collection of papers assembled for this topic are both excellent and inspiring and we invite you to take part in the growth of high performance applications through reading and being provoked by the research of our colleagues.

Hu *et al.* demonstrate a novel strategy for dramatically improving the scalability of the barotropic mode in the Parallel Ocean Program in "A Scalable Barotropic Mode Solver for the Parallel Ocean Program". Their approach achieves a significant reduction in execution time of the POP. In "Heterogeneous Combinatorial Candidate Generation" Khalid *et al.* present an approach for enumerating elementary flux modes using compute nodes enabled with GPU accelerators. The key point here is that while significant advances have been made in addressing this challenge for shared memory and SMP systems, not much work has yet addressed accelerating this process in the context of heterogeneous compute nodes. Förster and Naumann deal with the issue of using algorithmic differentiation by source transformation in combination with OpenMP compiler directives in "Solving a Least-Squares Problem with Algorithmic Dientiation and OpenMP". The exemplar problem, as indicated in the title, is that of least-squares fitting, which is a widely applicable algorithm. The speed-ups achieved are significant and the memory footprint of the code is reduced. Cheong *et al.* in "Hierarchical Parallel Algorithm for Modularity-Based Community Detection using GPUs" make use of GPUs and multiple levels of parallelism in finding community detection to great effect. In addition to offering a speed-up of a factor of 5, the proposed algorithm finds higher quality communities. Finally, in "Streaming Data from HDD to GPUs for Sustained Peak Performance" Beyer & Bentinesi deal with the difficult problem of shipping large amounts of data from disk to the desired compute element, in this case one or more GPUs. The application is in genomics and, as such, the main bottleneck in this problem is data transport. The method presented shows a significant speed-up in comparison to an optimised CPU implementation, and also scales well with the addition of extra GPUs to the compute node.

[1] Topic 16 "Extreme-Scale Computing" was introduced for Euro-Par 2013 and did not receive a significant number of submissions. Therefore, the Program Committee decided to merge it with Topic 14.

A Scalable Barotropic Mode Solver for the Parallel Ocean Program

Yong Hu[1,2], Xiaomeng Huang[1], Xiaoge Wang[2], Haohuan Fu[1], Shizhen Xu[1,2], Huabin Ruan[1,2], Wei Xue[1,2], and Guangwen Yang[1,2]

[1] Ministry of Education Key Laboratory for Earth System Modeling, Center for Earth System Science, Tsinghua University, Beijing, 100084, China
[2] Tsinghua National Laboratory for Information Science and Technology (TNList)
{huyong11,xsz12,rhb09}@mails.tsinghua.edu.cn,
{hxm,wangxg,haohuan,xuewei,ygw}@tsinghua.edu.cn

Abstract. This paper represents a novel strategy to improve the scalability of the barotropic mode in the Parallel Ocean Program (POP), by theoretically analyzing the barotropic communications bottleneck. POP discretizes the elliptic equations of the barotropic mode into a linear system $Ax = b$ and solves it using the Preconditioned Conjugate Gradient (PCG) method. PCG scales poorly on distributed systems because of the time-consuming global reductions needed by the inner products in each iteration. A performance model is developed to quantify the scaling bottleneck of PCG. Based on this model, the classical Stiefel iteration (CSI), which was originally supposed to be less efficient than PCG, is identified as being promising for massive parallelism. In contrast to PCG, the recurrence parameters of CSI are determined by the spectrum of the coefficient matrix A instead of the inner product of the residuals in previous iterations. The Lanczos method is used to resolve the difficulty of estimating the eigenvalues of the large-scale matrix A. It constructs a small-scale tridiagonal matrix that has eigenvalues close to A. By replacing PCG with CSI, global reductions and their inherent poor scalability are eliminated in the barotropic mode. The implementation of CSI in POP with a 0.1 degree resolution can accerlate one barotropic step by five times, from 1.23s to 0.26s, on 15,000 cores.

Keywords: Massive Parallelism, Preconditioned Conjugate Gradient, Classical Stiefel Iteration, Parallel Ocean Program, Barotropic Mode.

1 Introduction

Much research in high performance computing now focuses on how to adapt scientific applications for massive parallelism. Without scalable applications, large supercomputers cannot provide the application acceleration that leads to scientific progress in many important problems, such as ocean modeling.

Numerical ocean models that run on supercomputers will increase our ability to simulate and comprehend oceanic processes, monitor and predict the state of the oceans. The computational requirements of ocean simulation will become enormous as the resolution of ocean models increases, so optimization for massive parallelism is essential in ocean models.

In this paper, we focus on the performance optimization of POP, which is an important multi-agency ocean model that was developed at Los Alamos National Laboratory and is used for global ocean modeling. POP has been widely used for eddy-resolving ocean simulations[1] and coupled ocean-ice and atmosphere-ocean simulations[2] and was officially adopted as a component of the famous Community Earth System Model (CESM). POP utilizes three-dimensional primitive equations with hydrostatic and Boussinesq approximations. To avoid the severe time step restrictions imposed by fast waves, it divides the time integration into two parts: the baroclinic mode, which describes the original dynamic process in three dimensions, and the barotropic mode, which solves the vertically-integrated momentum and continuity equations[3].

Much attention is currently focused on the performance of POP, especially the poor scaling of the barotropic mode. Jones et al. [4] tested the portability of POP 1.4.3 on both vector architectures and commodity clusters and found that the baroclinic mode is dominated by computation, while the barotropic mode is dominated by communications overhead, including halo updates and global reduction operations. Stone et al. [5] found that the time consumption ratio of the barotropic mode increases from 10% on hundreds of processors to more than 50% on more than 10,000 processors. Worley et al. [6] and Dennis et al. [7] tested POP 2.0.1 on nearly 30,000 cores as a component of CESM. They found that POP is the most expensive component in most production simulations and confirmed that the performance of POP at large process counts is dominated by the communications overhead in the barotropic mode.

The main reason for the poor scalability of the barotropic mode is the implicit solver used for the linear system. The barotropic mode of POP is approximated as $Ax = b$, and a typical PCG solver is used to solve this linear system. The PCG solver involves two inner products in each iteration. When harnessing hundreds of thousands processors, the global communications and synchronization operations needed by the inner product becomes the main bottleneck. There are currently many solutions for reducing the negative effect of the PCG solver. Some solutions attempt to reduce the global communications overhead[8] and to overlap communication with computation[9]. Other solutions use land elimination and load-balance strategies[10, 11] to decrease the number of processes and the associated global reduction overhead.

These solutions are somewhat efficient. However, they are not intended to eliminate the root of this problem, which is the global reduction overhead. In this paper, we first construct a performance model of the PCG solver to quantitatively analyze the scalability of the barotropic mode. The model identifies the gradual increase of global communications as the source of the scalability bottleneck. Accordingly, we design a novel and scalable solver based on the classical Stiefel iteration (CSI) to break this bottleneck. The recurrence parameters of CSI are determined by the spectrum of the coefficient matrix A instead of the communication-intensive inner product of the residuals of previous iterations.

This feature makes CSI more scalable than PCG on massively parallel architectures due to the elimination of global reduction. The Lanczos method is used to estimate the eigenvalues of A. The Lanczos method constructs a low-order tridiagonal matrix T that has eigenvalues close to A and thus resolves the difficulty of directly obtaining the eigenvalues of A. The extra cost of estimating the eigenvalues introduced by CSI is as low as one barotropic step. Experiments show that PCG scales well on fewer than 1,000 cores but that the execution time of PCG increases when more than 5,000 cores are used. In contrast, CSI scales well until 10,000 cores are used, and it reduces the execution time of one barotropic mode from 1.23 seconds to 0.26 seconds on 15,000 cores.

The remainder of this paper is organized as follows. Section 2 reviews the mathematical model of the barotropic mode of POP and constructs a performance model to evaluate the scalability of the iterative methods. Section 3 introduces the design of CSI in POP. Section 4 presents experiments that compare the scalabilities of PCG and CSI on various numbers of cores. Finally, related work is described in section 5 and and the conclusion is presented in section 6.

2 Barotropic Mode Review and Bottleneck Analysis

The main procedure of the barotropic mode of POP is to solve an elliptic system of the sea surface height (SSH) [3]. To damp the computational modes associated with gravity waves and Rossby waves raised by pure leapfrog discretization, POP adopts an implicit scheme in the barotropic mode and simplifies the elliptic equations as a linear system $Ax = b$.

The implicit elliptic equations of SSH in POP can be expressed as follows:

$$[\nabla \cdot H\nabla - \phi(\tau)]\eta^{n+1} = \psi(\eta^n, \eta^{n-1}, \tau) \tag{1}$$

where H is the depth of the ocean bottom, ϕ is a function of the timestep τ, η^n is the SSH at the n-th time step, and ψ represents a function of previous states.

In POP, equation (1) is discretized on a two dimensional grid using a nine-point stencil, as shown in Fig. 1. $A^0_{i,j}$, $A^N_{i,j}$, $A^E_{i,j}$ and $A^{NE}_{i,j}$ are symmetrical coefficients between grid point (i, j) and its neighbors, and are determined by H, τ and the grid lengths. The stencil confined to grid point (i, j) is

$$\begin{aligned} A^0_{i,j}\eta_{i,j} + A^E_{i,j}\eta_{i+1,j} + A^N_{i,j}\eta_{i,j+1} + A^{NE}_{i,j}\eta_{i+1,j+1} + A^{NE}_{i-1,j}\eta_{i-1,j+1} \\ + A^E_{i-1,j}\eta_{i-1,j} + A^{NE}_{i-1,j-1}\eta_{i-1,j-1} + A^N_{i,j-1}\eta_{i,j-1} + A^{NE}_{i+1,j-1}\eta_{i,j-1} = \psi_{i,j} \end{aligned} \tag{2}$$

In the global domain, the stencil becomes $Ax = b$. A is a block tridiagonal matrix composed of coefficients A^0, A^N, A^E and A^{NE}, provided that the grid points are ordered along latitude and longitude. Equation (2) shows that A has only nine nonzero elements in each row. For massive parallelism, POP divides the global domain into blocks and distributes them to processes. Each process only computes the evolution procedures related to the grids in its own block, and maintains a halo region to update data with its neighbors.

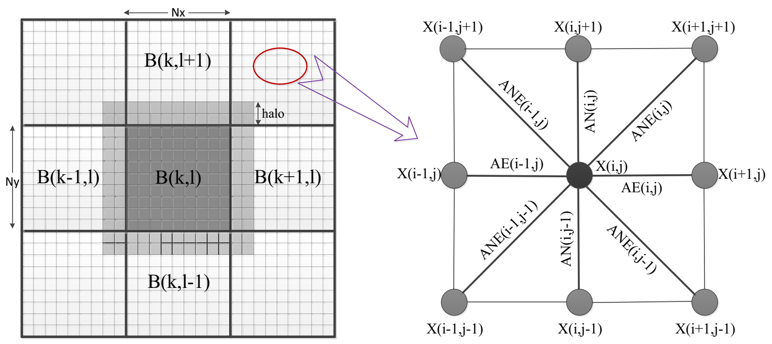


Fig. 1. Grid domain decomposition of POP

For simplicity, in the remainder of this paper, we assume that the global domain size is $N \times N$, and that it is divided into $m \times m$ blocks with size of $n \times n$ ($n = N/m$). Let $\tilde{A}$ be the coefficient matrix associated with the block $B(k, l)$. $\tilde{A}$ is a diagonal block matrix of A with a size of $n^2 \times n^2$ and has at most nine nonzero elements in each row. Thus, matrix-vector multiplication of $\tilde{A}\tilde{x}$ has $9n^2$ times of float multiplication operations rather than $n^2 \times n^2$ operations.

2.1 PCG Solver

The classical conjugate gradient method with a diagonal preconditioner $M = \Lambda(A)$ is used as the default barotropic solver in POP because of its efficiency in small-scale parallelism. The procedure of PCG is shown in Algorithm 1.

Algorithm 1. Preconditioned Conjugate Gradient solver

```
Require: Coefficient matrix $\tilde{\mathbf{A}}$, initial guess $\mathbf{x}_0$ and $\mathbf{b}$ associated with grid block $B_{i,j}$
    //    do in parallel with all processes
 1: $\mathbf{r}_0 = \mathbf{b} - \tilde{\mathbf{A}}\mathbf{x}_0$, $\mathbf{s}_0 = 0$;   $\beta_0 = 1$, $k = 0$;
 2: while $k \leq k_{max}$ do
 3:     $k = k + 1$;   $\mathbf{r}'_{k-1} = \mathbf{M}^{-1}\mathbf{r}_{k-1}$;                /* diagonal preconditioning */
 4:     $\tilde{\beta_k} = \mathbf{r}^T_{k-1}\mathbf{r}'_{k-1}$;   $\beta_k = global\_sum(\tilde{\beta_k})$;        /* global reduction */
 5:     $\mathbf{s}_k = \mathbf{r}'_{k-1} + (\beta_k/\beta_{k-1})\mathbf{s}_{k-1}$;   $\mathbf{s}'_k = \tilde{\mathbf{A}}\mathbf{s}_k$;   /* matrix-vector multiplication */
 6:     $update\_halo(\mathbf{s}'_k)$;                          /* boundary communication */
 7:     $\tilde{\alpha_k} = \mathbf{s}^T_k\mathbf{s}'_k$;   $\alpha_k = \beta_k/global\_sum(\tilde{\alpha_k})$;      /* global reduction */
 8:     $\mathbf{x}_k = \tilde{\mathbf{x}}_{k-1} + \alpha_k\mathbf{s}_k$;   $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_k\mathbf{s}'_k$;
 9:     if $k\%n_c == 0$ then check convergence;
10: end while
```

As shown in Algorithm 1, the PCG solver mainly contains three parts: computation, boundary updating, and global reduction. Computation involves matrix-vector and vector-vector multiplication and vector scaling. Boundary communication is needed to update the halo area after the matrix-vector multiplication. The time-consuming global reduction process occurs during the inner product operations.

2.2 PCG Performance Model

Assume that $P = m^2$ processes are used in the barotropic phase and that each process has exactly one grid block. The total time of the barotropic mode is equal to the execution time of the PCG solver on any block $B(i,j)$. Set T_c, T_b and T_g to be the execution time of the computation, boundary updating and global reduction, respectively, in one solver iteration. In Algorithm 1, the computation involves four vector scaling operations in steps 3, 5 and 8, two vector-vector multiplication operations of the inner products in steps 4 and 7, and one matrix-vector multiplication operation in step 5. Thus, $T_c = \Theta(4n^2 + 2n^2 + 9n^2) = \Theta(15n^2) = \Theta(15\frac{N^2}{P})$. It is obvious that T_c decreases as the number of processes increases and has a lower limit of zero.

Boundary updating occurs only between neighbors for each process, and its time depends on network delay and the volume of the halo regions. The default halo size is 2; thus, the volume in one boundary communication is $2n$ and decreases as the number of processes grows. Each process has to exchange data with its four neighbors, so the updating time in one iteration is $T_b = 2 \times 4T_{delay} + \Theta(2 \times 4 \times 2n) = 8T_{delay} + \Theta(\frac{16N}{\sqrt{P}})$. The updating time also decreases as the number of processes increases but has a lower bound of $8T_{delay}$.

The global reduction in one inner product sums up only one number from each process, so the data transmission time is negligible compared with the time of the global reduction itself. The reduction time, including the initiation delay and network blocking, satisfies $T_g = T_{init} + c_g \cdot \mathcal{G}(P)$. Here, T_{init} and c_g are constants associated with the parallel environment, and $\mathcal{G}(\cdot)$ is a function related to the network topology in the given architecture. For example, $\mathcal{G}(\cdot)$ is logarithmic in an ideal hypercubic network. In all, T_g increases monotonically with the number of processes P.

Let T_0 be the time unit of one floating-point operation and B be the number of floating-point numbers transmitted by the network per second from process to process. The execution time of one PCG iteration can be expressed as:

$$T_{pcg} = T_c + T_b + T_g = 15T_0\frac{N^2}{P} + 8T_{delay} + \frac{16N}{B \cdot \sqrt{P}} + T_{init} + c_g\mathcal{G}(P) \qquad (3)$$

The execution time of an entire PCG solver step is $t_{pcg} = K_{pcg} \cdot T_{pcg}$. Here K_{pcg} is the number of iterations in one PCG step and does not change with the number of processes. Equation (3) shows that the time required for computation and boundary updating decreases as the number of processes increases, while in contrast the time required for global reduction increases with increasing numbers of processes. The execution time of the PCG solver will increase when the number of processors exceeds a certain level.

A series of experiments were conducted with the 1 degree POP (360×240 grids) on a mesoscale cluster – the Explore100 at Tsinghua University. The Explore100 cluster is comprised of 740 computing nodes that each containing two 2.93 GHz Intel Xeon X5670 6-core processors and share 24/48 GB of memory. A performance profiling and tracing toolkit of the parallel program –TAU[12] was used to time the three components of the PCG solver. We compared the

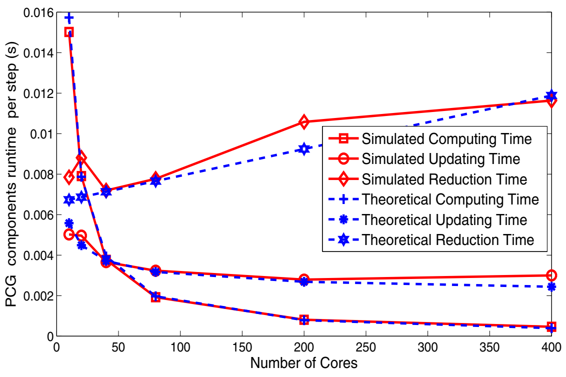


Fig. 2. Time components of one PCG step in 1 degree POP

average execution time of each component for one barotropic step provided by the performance model and the experiments. In the performance model, $\mathcal{G}(\cdot)$ is set as a linear function. c_g is set to 2×10^{-7}, T_0 is 2.5×10^{-9} and B is 5.0×10^9.

As shown in Fig.2, the model results agree well with the outcomes of the actual experiments. The simulated computation time in the barotropic mode is inversely proportional to the number of processes. The simulated updating time remains constant when the number of processes exceeds 100 because the communication volume becomes so small that its transmission time is covered by network latency. The global reduction increases proportionally with the number of processes and becomes dominant in the barotropic phase when more than 100 cores are used. The outcome confirms the theoretical conclusion in equation (3) that the global reduction is the cause of the poor scalability of the PCG solver.

3 Design of the CSI Solver

To address the bottleneck of PCG, the new barotropic solver should involve as few global reductions as possible. Originally less efficient methods, such as Chebyshev iteration, were reconsidered in POP. Chebyshev iteration was revisited by Gutknecht [13] in 2002, and was identified as being suitable for massively parallel computers with high communications costs. Classical Stiefel iteration (CSI) [14] is one kind of Chebyshev iteration methods.

3.1 Algorithm and Evaluation

In contrast to PCG, CSI does not require inner product operations and thus eliminates the bottleneck of global reduction. However, it requires preliminary knowledge about the spectrum of the coefficient matrix A. It is well-known that obtaining the eigenvalues is more complicated than solving a linear equation. Fortunately, for real symmetric and positive definite matrices, such as the coefficient matrix A in POP, only approximations of the largest and smallest eigenvalues λ_{max} and λ_{min} are needed to ensure convergence of CSI. These two extremal eigenvalues can be estimated efficiently by the Lanczos method. The pseudo code of the CSI algorithm designed for POP is shown in Algorithm 2.

Algorithm 2. Classical Stiefel Iteration solver

```
Require: Coefficient matrix $\tilde{\mathbf{A}}$, initial guess $\mathbf{x}_0$ and $\mathbf{b}$ associated with grid block $B_{i,j}$;
    Estimated eigenvalue boundary $[\nu, \mu]$;
    //     do in parallel with all processes
1: $\alpha = \frac{2}{\mu-\nu}$, $\beta = \frac{\mu+\nu}{\mu-\nu}$, $\gamma = \frac{\beta}{\alpha}$, $\omega_0 = \frac{2}{\gamma}$;   $k = 0$;
2: $\mathbf{r}_0 = \mathbf{b} - \tilde{\mathbf{A}}\mathbf{x}_0$; $\mathbf{x}_1 = \mathbf{x}_0 - \gamma^{-1}\mathbf{r}_0$; $\mathbf{r}_1 = \mathbf{b} - \tilde{\mathbf{A}}\mathbf{x}_1$;
3: while $k \le k_{max}$ do
4:     $k = k + 1$;   $\omega_k = 1/(\gamma - \frac{1}{4\alpha^2}\omega_{k-1})$;        /* the iterated function */
5:     $\Delta\mathbf{x}_k = \omega_k\mathbf{r}_{k-1} + (\gamma\omega_k - 1)\Delta\mathbf{x}_{k-1}$;
6:     $\mathbf{x}_k = \mathbf{x}_{k-1} + \Delta\mathbf{x}_{k-1}$;     $\mathbf{r}_k = b - \tilde{\mathbf{A}}\mathbf{x}_k$;
7:     update_halo($\mathbf{r}_k$);        /* boundary communication */
8:     if $k\%n_c == 0$ then check convergence;
9: end while
```

As shown in Algorithm 2, CSI has a similar iteration procedure to PCG but replace the two inner products and their associated vector-vector multiplications with an iterated function of the two extremal eigenvalues of A. The computation time of the CSI solver is $T_c = \Theta(4n^2 + 9n^2) = \Theta(13n^2) = \Theta(\frac{13N^2}{P})$. Because the halo regions are the same in PCG and CSI, the boundary updating time is still $T_b = 8T_{delay} + \Theta(16\frac{N}{\sqrt{P}})$. CSI has no global reduction except for checking convergence; thus, the execution time of one CSI iteration can be expressed as:

$$T_{csi} = T_c + T_b = 11T_0\frac{N^2}{P} + 8T_{delay} + \frac{16N}{B \cdot \sqrt{P}} \tag{4}$$

The execution time of an entire CSI solver step without convergence checking is $t_{csi} = K_{csi} \cdot T_{csi}$. K_{csi} is the number of iterations in one CSI solver step and it is usually larger than K_{pcg} under the same convergence tolerances. As shown in Fig. 3, this model accurately predicts the scalability behavior of CSI. CSI has a slower convergence speed than PCG, and its execution time may be longer than PCG on a small number of cores when global reduction is not a bottleneck. However, as shown in equations (3) and (4), CSI is faster than PCG in a single iteration because of the elimination of the time-consuming global reduction operation. The total execution time of PCG exceeds that of CSI at a threshold number of processes.

3.2 Eigenvalue Estimation

The convergence speed of CSI reaches its theoretical optimum when $\nu = \lambda_{min}$ and $\mu = \lambda_{max}$. Accurate values of λ_{min} and λ_{max} are difficult to obtain. In addition, any transformation of the coefficient matrix A is ill-advised because

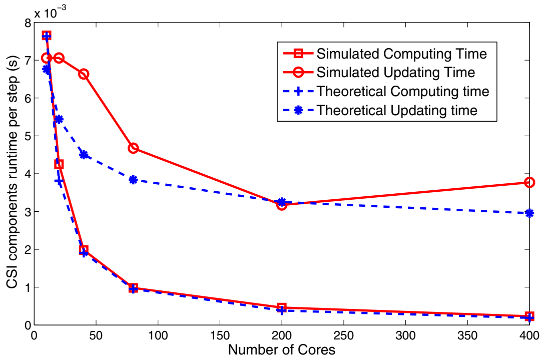


Fig. 3. Time components of one CSI step in 1 degree POP

A was distributed to processes. To utilize the parallism of POP, we employ Lanczos method [15] to construct a series of tridiagonal matrixes $T_m(m = 1, 2, ...)$ whose largest and smallest eigenvalues converge to those of A. The procedure of Lanczos-based eigenvalues estimation is shown in Algorithm 3.

Algorithm 3. Lanczos-based Eigenvalue Estimation

```
Require: Coefficient matrix $\tilde{\mathbf{A}}$ and random vector $\mathbf{r}_0$ associated with grid block $B_{i,j}$;
    // do in parallel with all processes
 1: $\mathbf{q}_1 = \mathbf{r}_0/||\mathbf{r}_0||$;  $\mathbf{q}_0 = \mathbf{0}$;  $T_0 = \emptyset$;  $\beta_0 = 0$;  $\mu_0 = 0$;  $j = 1$;
 2: while $j < k_{max}$ do
 3:     $\mathbf{r}_j = \tilde{\mathbf{A}}\mathbf{q}_j - \beta_{j-1}\mathbf{q}_{j-1}$;  $update\_halo(\mathbf{r}_j)$;
 4:     $\tilde{\alpha}_j = \mathbf{q}_j^T\mathbf{r}_j$;  $\alpha_j = global\_sum(\tilde{\alpha}_j)$;
 5:     $\mathbf{r}_j = \mathbf{r}_j - \alpha_j\mathbf{q}_j$;
 6:     $\tilde{\beta}_j = \mathbf{r}_j^T\mathbf{r}_j$;  $\beta_j = sqrt(global\_sum(\tilde{\beta}_j))$;
 7:     if $\beta_j == 0$ then return
 8:     $\mu_j = max(\mu_{j-1}, \alpha_j + \beta_j + \beta_{j-1})$;        /* Gershgorin circle theorem */
 9:     $T_j = tri\_diag(T_{j-1}, \alpha_j, \beta_j)$;  $\nu_j = eigs(T_j,' smallest')$ ;    /* Tridiagonal */
10:     if $|\frac{\mu_j}{\mu_{j-1}} - 1| < \epsilon$ and $|1 - \frac{\nu_j}{\nu_{j-1}}| < \epsilon$ then return
11:     $\mathbf{q}_{j+1} = \mathbf{r}_j/\beta_j$;  $j = j + 1$;
12: end while
```

In step 9 of Algorithm 3, T_m is a tridiagonal matrix that contains $\alpha_i(i = 1, 2, ..., m)$ as its diagonal entries and $\beta_i(i = 1, 2, ..., m-1)$ as off-diagonal entries.

$$T_m = tridiag\begin{pmatrix} & \beta_1 & \bullet & \beta_{m-1} & \\ \alpha_1 & \alpha_2 & \bullet & & \alpha_m \\ & \beta_1 & \bullet & \beta_{m-1} & \end{pmatrix}$$

Let ξ_{min} and ξ_{max} be the smallest and largest eigenvalues of T_m, respectively. Paige[15] demonstrated that $\lambda_{min} \leq \xi_{min} \leq \lambda_{min} + \delta_1(m)$ and $\lambda_{max} - \delta_2(m) \leq \xi_{max} \leq \lambda_{max}$. Here, $\delta_1(m)$ and $\delta_2(m)$ vanish in most cases as m increases. Thus, the eigenvalue estimation of A is transformed to solve the eigenvalues of T_m. Step 8 in Algorithm 3 employs the Gershgorin circle theorem to estimate the largest eigenvalue of T_m, that is, $\mu = \max_{1 \leq i \leq m} \sum_{j=1}^{m} |T_{ij}| = \max_{1 \leq i \leq m}(\beta_{i-1} + \alpha_i + \beta_i)$.

The efficient QR algorithm [16] with a complexity of $\Theta(m)$ is used to estimate the smallest eigenvalue ν in step 9. As show in Fig. 4, a small number of Lanczos steps will generate favorable eigenvalue estimates of A, and make CSI converges at an optimal speed similar to PCG.

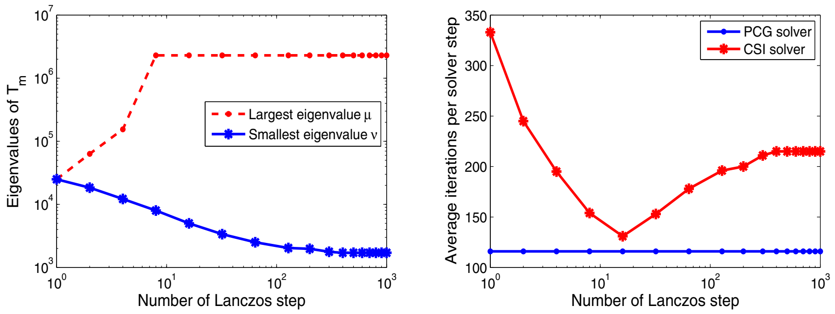


Fig. 4. Relationships between Lanczos steps and eigenvalue estimation

4 Experiments

To ensure that CSI will not introduce inaccuracies into POP, we conducted an experiment with the 1 degree POP on Explore100, which is described in section 2. The calculated SSH of POP versions using PCG and CSI are compared in Table 1. The mean difference between the PCG and CSI versions is small compared with the largest absolute SSH. It is interesting that large differences are only present at coastlines, where the sharp boundary causes instabilities in the difference scheme. The difference between the PCG and CSI versions is mainly due to the turbulence accumulation in the whole ocean model as the simulation period extends, rather than the error in each solver step.

Table 1. The SSH differences between the PCG and CSI versions

Time period	one step	one day	one month	one season
Step number	1	45	14053	40800
Max relative error	1.5016E-3	2.2181E-5	1.2885E-2	1.4114E-1
Mean relative error	3.0223E-6	5.2424E-7	2.6125E-5	7.8872E-4

To test the scalability, we ran the 0.1 degree POP (3600×2400) on the Sunway BlueLight MPP Supercomputer at the National Supercomputing Center in China. BlueLight contains 8,704 SW1600 processors, which are connected by a 40Gb InfiniBand network. Each processor consists of 16 1.1GHz cores that share 16 GB memory.

We tested PCG and CSI on 100 to 15,000 cores and with convergence tolerances that varied from $\epsilon = 10^{-8}$ to $\epsilon = 10^{-16}$. The convergence criterion in

POP is $||r||_2 < \epsilon\bar{a}$, where $\bar{a}$ means the rms of area. A tolerance $\epsilon = 10^{-12}$ is recommended in POP. As shown in Fig. 5, PCG and CSI both scale well on less than 1,000 cores. When more than 1,000 cores are used, the superiority of CSI to PCG becomes clear. When the tolerance is 10^{-12}, the execution time of CSI is 87% that of PCG on 100 cores, and this ratio decreases to 21% on 15,000 cores. PCG is less sensitive to convergence tolerance than CSI. As the convergence tolerance varies from 10^{-8} to 10^{-16}, the number of iterations per PCG step increases from 20 to 281, while the number of iterations in CSI increases from 33 to 1,434. However, the strength in convergence speed makes PCG superior to CSI for the strictest convergence condition($\epsilon = 10^{-16}$) and when fewer than 1,000 cores are used. In other cases, the increased global reduction overhead in PCG can not be matched by the smaller number of iterations. On 15,000 cores, the execution time of PCG is 4.8 times that of CSI, from 0.24s to 0.05s when the convergence tolerance is 10^{-8}, 4.7 times that of CSI, from 1.23s to 0.26s when the convergence tolerance is 10^{-12}, and 3.3 times that of CSI, from 2.41s to 0.72s when the convergence tolerance is 10^{-16}. Due to the global reduction bottleneck, the execution time of PCG increases when more than 1,000 cores are used. In contrast, CSI scales well until 10,000 cores are used.

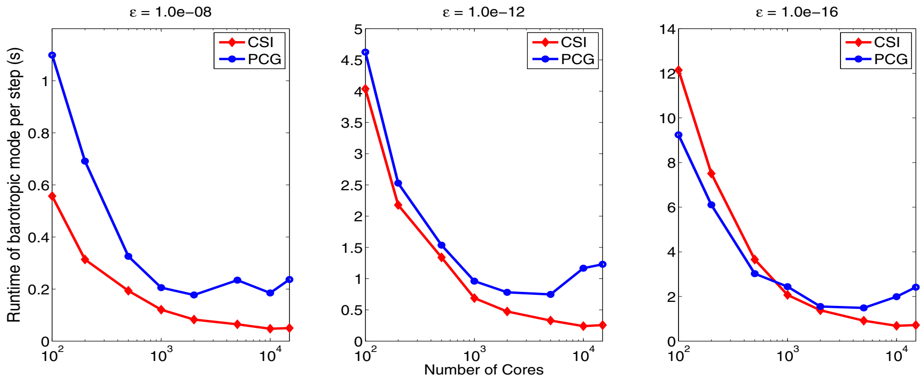


Fig. 5. Scalability of PCG and CSI in the 0.1 degree POP

5 Related Work

The barotropic mode makes up a large proportion of the total execution time of POP, especially when it runs on a large number of cores. Much work has been done on optimizing the performance of the barotropic mode, most of which has been related to decreasing the amount of communication between processes and accelerating the computation of each process. The total costs of global reduction are proportional to the number of processes , and the communications overhead becomes increasingly intolerable as the number of processes increasing. OpenMP parallelism and land elimination are common strategies for reducing the number of processes and the associated MPI overhead. Worley et al. [6] strongly recommended the OpenMP strategy when a large count of cores are needed for the baroclinic phase, but a large number of processes would cause communications difficulties in the barotropic phase. Dennis [10, 11] proposed a load-balancing strategy based on newly developed space-filling curve partitioning algorithms. The strategy not only eliminates land blocks, but also decreases the communications overhead because of the reduced number of processes. The simulation rate on approximately 30,000 processors doubles after applying this strategy. Reducing the frequency of communication also attenuates the overhead in the barotropic mode. As early as 1997, Beare [9] proposed the performance of parallel ocean general circulation models can be improved by increasing the number of extra halos and overlapping the communications with the computation.

Another way to break the bottleneck of the barotropic mode is to improve algorithm and preconditioning of the PCG method. A variant of the standard conjugate gradient method presented by D'Azevedo [8], called the Chronopoulos-Gear algorithm, proposed a way to halve the global communication in PCG. It combines the two separate global reductions into a single global reduction vector by rearranging the conjugate gradient computation procedure, and achieves a one third latency reduction in POP. Preconditioning has been highlighted in the CG method since the 1990s. Many linear systems converge after a few PCG iterations with a suitable preconditioner. Adamidis et al. [17] implemented an incomplete Cholesky preconditioner in the global ocean/sea-ice model MPIOM to improve the scalability and performance of PCG.

The improvement of the methods described above is limited due to the inherent poor data locality and sequential execution of PCG. Some work has been done to accelerate the PCG solver by employing the developing hybrid accelerating devices, such as GPUs[18] and FPGAs[19]. GPUs and FPGAs are helpful in reducing the global overhead. These devices have stronger computational ability and more memory than common CPU, so fewer devices and less communication are needed for the same scale computing job.

6 Conclusion

Much work has been done to improve the barotropic mode in POP. However, most of the methods described above did not eliminate the cause of the poor scalability of the barotropic mode of POP. This paper presents a performance model of the barotropic mode that quantifies the scalability of PCG. The advantage of CSI is demonstrated based on the analysis of this model. CSI is implemented in POP and shows better scalability than PCG. In closing, this paper highlights a promising complement to PCG with elliptic equations.

Acknowlegements. This research was sponsored by the National High Technology Development Program of China (2010AA012401, 2011AA01A203).

References

[1] McClean, J., Poulain, P., Pelton, J., Maltrud, M.: Eulerian and Lagrangian statistics from surface drifters and a high-resolution POP simulation in the north atlantic. Journal of Physical Oceanography 32(9), 2472–2491 (2002)

[2] May, P., Cummings, J., Hogan, T., Rosmond, T., Flatau, M., de Witt, P., Passi, R.: Preliminary results from a global ocean/atmosphere prediction system. In: OCEANS 2002 MTS/IEEE, vol. 2, pp. 667–671 (October 2002)
[3] Smith, R., Jones, P., Briegleb, B., Bryan, F., Danabasoglu, G., Dennis, J., Dukowicz, J., Fox-Kemper, C., Gent, P., Hecht, M., et al.: The parallel ocean program (POP) reference manual ocean component of the Community Climate System Model (CCSM) and Community Earth System Model, CESM (2010)
[4] Jones, P.W., Worley, P., Yoshida, Y., White III, J.B.: Practical performance portability in the parallel ocean program (POP). Concurrency and Computation: Practice and Experience 17, 1317–1327 (2005)
[5] Stone, A., Dennis, J., Strout, M.: The CGPOP miniapp, version 1.0. Technical report, Technical Report CS-11-103, Colorado State University (2011)
[6] Worley, P.H., Mirin, A.A., Craig, A.P., Taylor, M.A., Dennis, J.M., Vertenstein, M.: Performance of the community earth system model. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 54:1–54:11. ACM, New York (2011)
[7] Dennis, J., Vertenstein, M., Worley, P., Mirin, A., Craig, A., Jacob, R., Mickelson, S.: Computational performance of ultra-high-resolution capability in the community earth system model. International Journal of High Performance Computing Applications 26(1), 5–16 (2012)
[8] D'azevedoy, E., Eijkhoutz, V., Rominey, C.: Lapack working note 56 conjugate gradient algorithms with reduced synchronization overhead on distributed memory multiprocessors (1999)
[9] Beare, M., Stevens, D.: Optimisation of a parallel ocean general circulation model. In: Annales Geophysicae, vol. 15, pp. 1369–1377. Springer (1997)
[10] Dennis, J.: Inverse space-filling curve partitioning of a global ocean model. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–10. IEEE (2007)
[11] Dennis, J.M., Tufo, H.M.: Scaling climate simulation applications on the IBM Blue Gene/L system. IBM Journal of Research and Development 52(1.2), 117–126 (2008)
[12] Shende, S., Malony, A.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–311 (2006)
[13] Gutknecht, M., Röllin, S.: The Chebyshev iteration revisited. Parallel Computing 28(2), 263–283 (2002)
[14] Stiefel, E.L.: Kernel polynomials in linear algebra and their numerical applications. ETH (1958)
[15] Paige, C.: Accuracy and effectiveness of the lanczos algorithm for the symmetric eigenproblem. Linear Algebra and its Applications 34, 235–258 (1980)
[16] Ortega, J., Kaiser, H.: The LLT and QR methods for symmetric tridiagonal matrices. The Computer Journal 6(1), 99–101 (1963)
[17] Adamidis, P., Heuveline, V., Wilhelm, F.: A high-efficient scalable solver for the global ocean/sea-ice model MPIOM. KIT (2011)
[18] Cuomo, S., De Michele, P., Farina, R., Chinnici, M.: A PCG implementation of an elliptic kernel in an ocean global circulation model based on GPU libraries. arXiv preprint arXiv:1210.1878 (2012)
[19] Shida, S., Shibata, Y., Oguri, K., Buell, D.: Implementation of a barotropic operator for ocean model simulation using a reconfigurable machine. In: International Conference on Field Programmable Logic and Applications, FPL 2007, pp. 589–592 (August 2007)

Heterogeneous Combinatorial Candidate Generation

Fahad Khalid[1], Zoran Nikoloski[2], Peter Tröger[1], and Andreas Polze[1]

[1] Hasso Plattner Institute for Software Systems Engineering
{fahad.khalid,peter.troeger,andreas.polze}@hpi.uni-potsdam.de
[2] Max Planck Insitute of Molecular Plant Physiology
nikoloski@mpimp-golm.mpg.de

Abstract. Elementary Flux Modes (EFMs) can be used to characterize functional cellular networks and have gained importance in systems biology. Enumeration of EFMs is a compute-intensive problem due to the combinatorial explosion in candidate generation. While there exist parallel implementations for shared-memory SMP and distributed memory architectures, tools supporting heterogeneous platforms have not yet been developed. Here we propose and evaluate a heterogeneous implementation of combinatorial candidate generation that employs GPUs as accelerators. It uses a 3-stage pipeline based method to manage arithmetic intensity. Our implementation results in a 6x speedup over the serial implementation, and a 1.8x speedup over a multithreaded implementation for CPU-only SMP architectures.

1 Introduction

Metabolism is the collection of chemical compounds, called metabolites, transformed via enzymatic reactions to sustain the functions of biochemical systems. The network structure of metabolism can be characterized by a directed weighted hypergraph [1] in which directed hyperedges represent reactions and nodes stand for metabolites. The number of molecules with which a metabolite participates as a substrate and/or product in a reaction specifies the reaction-specific stoichiometry of the metabolite, rendering the hypergraph node-weighted. The concept of a steady state, *i.e.*, equilibrium, whereby there is no change in concentrations of the considered metabolites, is often employed in analyzing the functional behavior of (large-scale) metabolic networks [2].

Interestingly, the steady-state behavior of metabolic networks, described only by the directed weighted hypergraph, can be fully characterized by the minimal subnetworks which operate at equilibrium, referred to as elementary flux modes [3, 4] (EFMs). Due to the minimality condition, an EFM cannot operate in a steady state upon removal of any of its components (*i.e.*, reactions or metabolites). Further, EFMs provide a mathematical definition for the concept of a biochemical pathway. Since EFMs can capture emergent functions of biochemical systems, they have been used to analyze key systemic properties, including robustness and flexibility [5]. However, characterization of a system's behavior by means of EFMs requires their enumeration, which involves systematic evaluation of all possible subnetworks with respect to several constraints/conditions they must satisfy. This process is combinatorial in nature and expensive in terms of both computational and memory requirements, thus, limiting application to systems of small size. Therefore, parallelization of the existing approaches for EFM enumeration is necessary for large-scale networks.

Both shared-memory and distributed-memory parallel approaches have been developed. A parallel out-of-core implementation [6] was one of the first attempts at parallelization. Another implementation, the *efmtool* [1] is targeted towards shared memory SMP architectures. It is based on the state-of-the-art in algorithmic approach for EFM enumeration [7–9], and has been used by scientists other than the developers to report important results [10].

The *ElMo-Comp* tool [11] was designed specifically for distributed memory architectures. Since the first public release, the tool has been extended to handle larger networks. The first extension [12] employs the *Divide and Conquer* strategy, where the complete set of EFMs is partitioned into disjoint subsets that can be processed independently. In the second extension [13], the concept of *Partitioned Global Address Space (PGAS)* is utilized to enable sharing of memory resources across the cluster.

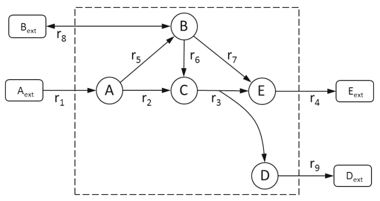
Our study builds on *ElMo-Comp* [11] and extends it to support heterogeneous architectures with GPUs as accelerators. Our efforts are focused on the computational bottleneck, which is the memory-bound part of the algorithm we term *combinatorial candidate generation*.

The paper is organized as follows: Section 1.1, presents the mathematical model and algorithm used for EFM enumeration, including the computational bottleneck. Our approach is presented in Section 2, followed by evaluation in Section 3. Related work is discussed in Section 4, followed by conclusion and future work.

1.1 Enumeration of Elementary Modes

Mathematical Model. Consider the hypergraph representation of a paradigmatic metabolic network presented in Figure 1A (adapted from [14]). Metabolites in the network are represented by nodes and reactions by hyperedges. Metabolites are divided into two groups *internal* to the system and *external*, *i.e.*, in the systems environment, delineated by the dotted line box in Figure 1A. The reactions that involve both external and internal metabolites are termed exchange reactions. Moreover, with respect to directionality, reactions are divided into reversible and irreversible, belonging to the sets Rev and Irr, respectively. In Figure 1A, reaction r_8 is reversible. The information encoded in the directed weighted hypergraph can be captured by the corresponding stoichiometric matrix. For a metabolic network with m metabolites and q reactions, the stoichiometric matrix, S, consists of m rows and q columns. The entry $S_{i,j}$ quantifies the number of molecules with which metabolite i participates in reaction j, and

[1] efmtool - Elementary Flux Mode Tool: `http://www.csb.ethz.ch/tools/efmtool`



$$
\begin{array}{cc}
 & \begin{array}{ccccccccc} r1 & r2 & r3 & r4 & r5 & r6 & r7 & r8 & r9 \end{array} \\
\text{S=}\begin{array}{c} A \\ B \\ C \\ D \\ P \end{array} & \begin{pmatrix} 1 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 & -1 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}
\end{array}
$$

Fig. 1. (A) Metabolic network (B) corresponding stoichiometric matrix

the sign indicates if the metabolite participates as a substrate (negative) or a product (positive), illustrated in Figure 1B.

Reaction rates, called fluxes, quantify the behavior of reactions transforming the metabolites in the network. The steady state of a metabolic network specified by its stoichiometric matrix can be characterized in terms of a q-vector termed flux vector (or distribution), denoted by v. Reaction rates can be used to characterize the change in the concentration of metabolites, since $\frac{dX}{dt} = Sv$, where X is an m-vector gathering the concentrations of metabolites. In a steady state, there is no change in concentrations, and the steady-state flux distribution can be determined by solving the system of linear equations:

$$Sv = 0, \tag{1}$$

whereby v belongs to the nullspace of the stoichiometric matrix S. Since the number of metabolites is usually smaller than the number of reactions, the system in Eq. (1) is underdetermined and usually results in an infinite number of solutions. Note that the system of linear equations is homogeneous if all metabolites are internal; otherwise, the system is inhomogeneous. Moreover, a steady-state flux distribution is further constrained by the reaction directionalities, so that fluxes of irreversible reactions must be non-negative, *i.e.*,

$$v_i \geq 0, \forall i \in Irr \tag{2}$$

By combining the steady-state and directionality constraints imposed by Eq.(1) and Eq.(2), respectively, the solution space forms a convex polyhedral cone, P, defined as [15]:

$$P = \{v \in \mathbb{R}^q \mid Sv = 0, v_i \geq 0, \forall i \in Irr\}, \tag{3}$$

where $\mathbb{R}^q$ is the q-dimensional vector space in real numbers. Clearly, every vector that lies in the cone represents a feasible flux distribution in the metabolic network. With this notation, we need the elementarity constraint to define an EFM. Let $supp(v) = \{i \mid v_i \neq 0\}$ and let E denote the set of all EFMs, then, the following holds:

$$\forall v \in E, \nexists x \in E \mid v \neq x, supp(v) \subseteq supp(x). \tag{4}$$

If the system consists only of irreversible reactions, the solution space forms a pointed polyhedral cone [5]. In this case, the set of elementary modes comprises a unique minimal set of generating vectors for the entire flux space.

The Nullspace Algorithm. All algorithms for elementary mode enumeration are based on the Double Description Method [16] for extreme ray enumeration of a polyhedral cone, well-studied in computational geometry. These algorithms vary primarily in the order in which the steady-state and reaction reversibility constraints are processed. Our study is based on the Nullspace algorithm [17] (see [18] for detailed description). Here, we present a brief sketch, highlighting only the most relevant steps. The Nullspace algorithm is summarized as follows:

1. The stoichiometric matrix S is compressed using methods specified in [5]. Let the compressed matrix be $S'_{m\times q}$. Then the nullspace obtained by solving $S'v = 0$ is denoted by K'. Each row in K' corresponds to a reaction, and each column represents a potential EFM. Let I denote the identity matrix. The compressed nullspace is permuted to obtain the following form:

$$K' = \begin{pmatrix} R^{(1)} \\ R^{(2)} \end{pmatrix} = \begin{pmatrix} I \\ R^{(2)} \end{pmatrix} \tag{5}$$

   where the directionality constraints are already solved for rows in I. Directionality constraints must now be applied to all rows in $R^{(2)}$.
2. For each row in $R^{(2)}$:
   (a) Generate bitwise combinations of selected columns in $R^{(1)}$ to produce candidate bit vectors. Given a threshold τ, a candidate vector, v, $supp(v) > \tau$ is discarded,
   (b) Remove duplicate candidate vectors,
   (c) Verify each candidate for elementarity,
   (d) Generate algebraic combinations on the current row in $R^{(2)}$,
   (e) Convert the current row in $R^{(2)}$ to the corresponding binary representation and move it to $R^{(1)}$,
   (f) Append the generated EFMs as column vectors to the nullspace.

We note that once a row in $R^{(2)}$ is processed, it can be converted to a binary representation and moved to $R^{(1)}$ [5]. Moreover, in *ElMo-Comp*, $R^{(1)}$ is compressed by a factor equal to the machine word length, *i.e.*, 32 or 64 times. Finally, once all reactions have been processed, columns of the kernel matrix represent all EFMs for the given network.

Combinatorial Candidate Generation. The most compute intensive step in the Nullspace algorithm is the generation of combinations in $R^{(1)}$ (see Algorithm 1). We refer to this step as *combinatorial candidate generation*. This step being the computational bottleneck is the primary focus of our work.

Algorithm 1 consists of two core computational operations: a bitwise OR between two columns that results in a candidate vector (Line 3) and a *popcount* on the candidate vector (Line 4). To process a single candidate, we need *two* arithmetic and *four* memory access operations (assuming *popcount()* is available as a hardware instruction). Moreover, two read operations are required to fetch the input columns, and two write operations are required to store the indices corresponding to the input columns. The data type used for both input and output values is 64-bit unsigned integer. As compared to the 32-bit data types, this increases the size of the input and output values, halves the throughput of the two operations, and, thus, results in a very low compute-to-memory-access ratio. Therefore, combinatorial candidate generation can be classified as a memory-bound algorithm with low arithmetic intensity.

Algorithm 1: Serial combinatorial candidate generation. Index vectors contain column indices of the corresponding matrices; OR is the binary bitwise OR operation; τ is a threshold (as described in Section 1.1)

```
Input  : Bit matrices: MatrixA, MatrixB
         Index vectors: IndicesA, IndicesB
         Integer: $\tau$
Output: Candidate column index pairs of the form
        $\{(a, b) \mid a \in IndicesA \text{ and } b \in IndicesB\}$
1 foreach colA: column in MatrixA do
2     foreach colB: column in MatrixB do
3         candidate = MatrixA[colA] OR MatrixB[colA];
4         nonZeros = popcount(candidate);
5         if nonZeros $\leq \tau$ then
6             store index pair (IndicesA[colA], IndicesB[colB]);
7         end
8     end
9 end
```

In massively parallel accelerator architectures, like GPUs, most of the chip area is dedicated to arithmetic and logic units, which results in very small sizes for fast on-chip memory. Therefore, these architectures are ill-suited for algorithms with low arithmetic intensity. Here, we present an approach that renders it possible to exploit the massive parallelism of GPUs, leading to significant speedup despite the memory-bound nature of the combinatorial candidate generation algorithm.

2 Our Approach

In the rest of the document, we assume all index values to be of type 64-bit unsigned integer. We refer to the CPU as *Host*, and the GPU as *Device*.

Algorithm 1 can be decomposed into two parts: (1) generation of a candidate vector followed by popcount (Lines 3,4), which results in a Boolean; and (2) storage of input column indices (Line 6). These two parts can be implemented as distinct phases — *Generate* and *Map*.

Generate Phase. This phase is implemented as the *Device* kernel (see Algorithm 2). As in the serial version, the two input values are fetched to perform the bitwise OR (Line 3) and popcount (Line 4) operations. Since the max-non-zero condition is the final step in this phase, instead of storing two 64-bit integers, the kernel stores a single Boolean value (Line 5). This significantly increases the arithmetic intensity, and hence the kernel performance. Nevertheless, storing a Boolean value against each possible combination has a disadvantage. For two matrices A and B of sizes m and n, respectively, the size of the output Boolean array is $m \times n$. As the number of reactions in the network increases, the size of the output array becomes too large for the *Device* memory. This results in large and frequent *Device*-to-*Host* memory transfers that become a serious bottleneck.

Algorithm 2: GPU kernel for combinatorial candidate generation

```
Input  : Matrix A, Matrix B, $\tau$
Output: Result - bit array
1 for 1 to compressionFactor do
2     compute indexMatA, indexMatB, indexResult ;     // index algebra
3     candidate = MatrixA[indexMatA] OR MatrixB[indexMatB];
4     nonZeros = popcount(candidate);
5     result[indexResult] = (nonZeros $\leq \tau$);
6 end
```

We address this problem by introducing a *compression factor*, which defines the number of result values generated by each *Device* thread. Instead of storing the output in a Boolean variable, we use a single bit. Therefore, a single thread can generate and store 64 values in a single 64-bit unsigned integer. This reduces the size of the output array by *compression factor*, and *Device*-to-*Host* transfers no longer constitute the bottleneck. The data type of the output array is independent of the other data types used, and its size merely indicates the maximum number of results generated by a single thread.

Map Phase. Once kernel execution is complete, the index of each bit in the result array corresponds to a candidate, and the bit value indicates whether the candidate should be considered for further processing. The index of each set bit must then be mapped to the corresponding index pair used to generate the corresponding value (a candidate is identified by the corresponding pair of input columns). The *Map* phase traverses the output bit array, looks for set bits, and maps their indices to the corresponding input index pairs. This phase is highly memory-bound, and thus performs better on the *Host*.

2.1 Concurrent *Host–Device* Processing

The *Generate-Map* strategy with compression factor, results in a significant speedup over both the serial version, and a naïve kernel without compression factor. In the subsections to follow, we show how *pipeline parallelism* [19] can be employed to implement concurrent *Device-Host* processing for larger speedup.

Two-Stage Pipeline: *Device* only Due to the combinatorial nature of the algorithm, a very large number of *Device* threads are required to compute all possible combinations. For NVIDIA GPUs with compute capability up to 2.0, the maximum number of thread blocks is limited to 65535. Therefore, for larger input sizes, multiple grid (batches of threads) executions are required. Between two subsequent grid executions, the result array has to be transferred from *Device* to *Host*, so that enough space is left on the *Device* to hold the result array for the next grid in line for execution. Moreover, the *Map* phase can only begin once the final *Device*-to-*Host* memory transfer is complete. This results in a serial processing of stages as depicted in Figure 2a.

The NVIDIA CUDA programming model provides API that makes it possible to overlap kernel execution and memory transfer. Using this feature, one kernel can be launched after the other without waiting for a *Device*-to-*Host* memory transfer operation to finish. This results in a 2-stage pipeline as depicted in Figure 2b. To ensure that a memory transfer only begins after the corresponding kernel computation is finished, an event notification system is employed. Each kernel executes in its own stream [20], and once the kernel execution is complete, an event notification is sent from the kernel stream to the corresponding memory transfer stream. The memory transfer stream begins operation only after the event notification has been received.

Three-Stage Pipeline: *Device* and *Host* The CUDA stream based event notification system has traditionally been limited to event exchange among *Device* operations only. With the release of CUDA 5.0 however, it is now possible to register *Host* callback functions with *Device* streams. These callbacks make it possible for the *Device* to send stream event notifications to the *Host*. We utilize the callback feature to extend the 2-stage pipeline. The *Map* phase is included as an additional stage, resulting in a 3-stage pipeline that spans both *Host* and *Device* functions. The concept is illustrated in Figure 2c.

The process starts by calculating the total size of the result array, and splitting it into multiple segments. For each segment, one or more kernels are launched in different streams, which we refer to here as grid streams. Events are recorded for each grid stream, on which the asynchronous memory transfer operation waits. Once all grids for the current segment have finished execution, the *Device*-to-*Host* memory transfer begins. At the same time, grids are launched for the next segment. Once the *Device*-to-*Host* memory transfer operation is complete, it

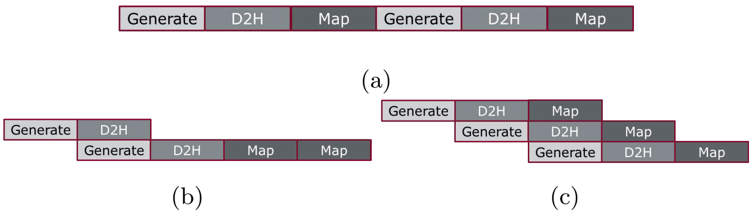


Fig. 2. Illustration of (a) serial, (b) 2-stage, (c) 3-stage processing of phases. D2H is *Device*-to-*Host* memory transfer

calls a *Host* function registered as callback with the memory transfer stream. The callback function sets a flag, indicating that the memory transfer operation for a specific segment is complete. The flag serves as a notification for the *Map* phase, and triggers the map operation on the segments result. To ensure that the *Generate* and *Map* phases execute not just concurrently but in parallel, these operations are executed by two separate *Host* threads that share data structures for the exchange of event notifications.

The memory size on the *Device* is generally smaller than that of the *Host*. Therefore, it is the programmers responsibility to ensure that all datasets that fit into the *Host* memory can also run on the *Device*. For this purpose, we use the concept of *partition*. Maximum partition size depends on the total amount of global memory available on the *Device* and the sizes of the input and result data structures. Large datasets are split into multiple partitions, where each partition consists of multiple segments. The *Device* utilizes the 3-stage pipeline to process one partition at a time. The next partition can start execution only if the required memory resources have been released by the previous partition.

2.2 Overall Architecture

In order to utilize all available processing resources, the candidate generation algorithm is processed using two implementations that execute in parallel (as shown in Figure 3). One is a multithreaded OpenMP based (*Host*-only) implementation that executes solely on the CPU cores. The other is the 3-stage *Device-Host* pipeline as describer in Section 2.1.

A *Host* thread decomposes the input data structures into two parts for distribution amongst the *Host*-only implementation and the 3-stage pipeline. These are passed on to a *parallel harness* that invokes two threads. The first thread invokes the OpenMP based *Host*-only multithreaded implementation. The second invokes the *device harness*. The device harness defines the data structures to be shared amongst the *Generate* and *Map* phases, and instantiates these phases as two OpenMP threads. The *Generate* thread further manages the massively parallel execution on the *Device*. The *Map* thread listens to memory transfer completion events and invokes the bit-to-input-index mapping routine. This routine is also implemented as multithreaded OpenMP code, which helps speedup the mapping process. Once both the *Host*-only code and device harness threads have completed execution, a reduction operation is performed to consolidate the results.

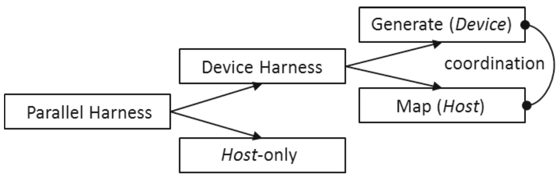


Fig. 3. Thread hierarchy. *Parallel harness* spawns *device harness* and *Host-only* code. *Device harness* spawns the *Generate* and *Map* phases.

3 Evaluation

We present comparative results from three different implementations: the serial Nullspace program available in *ElMo-Comp* [11]; our OpenMP based shared-memory parallel implementation for SMP architectures; and our 3-stage pipeline implementation for heterogeneous architectures. Both our implementations are based on the *ElMo-Comp* code base.

3.1 Test Environment

The machine used for running the reported experiments has 24GB of main memory, and consists of an Intel Xeon E5620 CPU and an NVIDIA Tesla 2050 GPU. The Xeon E5620 processor is based on the Nehalem EX architecture, supporting a 64-bit instruction set with SSE 4.2. It has 4 cores, each supporting 2 hardware threads. The Tesla 2050 GPU is based on the Fermi architecture, with compute capability rating of 2.0. The operating system used is Ubuntu SMP 12.04. The code was compiled using GCC 4.4.3 and NVCC with CUDA 5.0.

3.2 Results

Table 1 summarizes results of three implementations against five real networks of varying sizes, all capturing E.Coli central metabolism. The candidate generation step is performed on compressed networks [5]. Accordingly, network sizes mentioned in the table correspond to compressed networks.

The results show that the utility of the heterogeneous implementation increases with the number of candidate vectors generated during execution. Poor performance of the heterogeneous implementation against small networks is attributed to the overhead incurred by the transfer of data between *Host* and *Device* memory, as well as coordination between the *Generate* and *Map* phases. For larger networks, the incurred overhead is overshadowed by the performance gain. The *Host*-only multithreaded implementation utilizes all 8 threads available on the processor. The heterogeneous implementation dedicates 2 threads to the *Host*-only implementation, and 2 threads to the *Map* phase. Input data is distributed amongst the device harness and the *Host*-only implementation such that only $\frac{1}{8}th$ of the input size is processed by the *Host*-only code, while the rest is processed by the device harness.

It is important to properly tune compression factor and maximum segment size for optimal performance. A higher value of compression factor translates to more work per *Device* thread, and lower *Device* output size. Maximum segment size defines the maximum result array size for which a memory transfer to the *Host* must be initiated. Together these two parameters balance the speed and coordination between the pipeline stages. For the results presented in Table 1, compression factor is 64 and the maximum segment size is 60 MB.

Table 1. Comparative results of three different implementations against five networks. Execution times (in seconds) are presented against each implementation and network; m is the number of metabolites, q is the number of reactions.

Network Size	#candidates	Time (s)		
		Serial	OpenMP	Pipelined
26m × 38q	219743731	1.2	0.38	0.39
26m × 40q	130992739	0.77	0.35	0.35
26m × 41q	752482917	4.1	1.6	1.0
27m × 43q	2616975505	14	5.5	2.5
29m × 45q	122559991284	690	150	110

4 Discussion

Related Work. Recently, major vendors from the hardware and software industries have pointed out the significance of considering arithmetic intensity for decisions concerning suitable hardware. Empirical results [21] were presented to show how Floating Point Operations Per Second (FLOPS) is not an adequate measure for memory-bound algorithms. In addition, the case of Sparse Matrix-vector Multiplication (SpMV) was used to study accelerator (GPU) performance for algorithms with low arithmetic intensity [22]. The authors conclude that with the coming generations of processors, in comparison to GPUs, CPUs are becoming more and more suitable for such problems.

A multitude of scientific applications have been recently designed take advantage of heterogeneous architectures with GPUs as accelerators. These include linear solvers [23], solvers for path problems in graphs [24], as well as applications for simulation science [25], to name a few. Moreover, there have been efforts to increase the arithmetic intensity of certain algorithms [26], so that the processing resources can be utilized effectively.

Conclusions and Future Work. We presented a novel method to utilize GPUs for *combinatorial candidate generation*, a specific memory-bound algorithm, required for EFM enumeration. Our approach focuses on the concurrent, coordinated, *Device-Host* pipelined execution model. This approach is feasible due to the possibility to split the algorithm into two phases, where the phase of a high arithmetic intensity is executed on a GPU, while the other is executed concurrently on the *Host*. The 2-phase computation points to the Map-Reduce Pattern for Parallel Computation [27]. We conjecture that memory-bound algorithms amenable to this pattern may also benefit from the approach presented in this paper.

A large number of important combinatorial algorithms (such as those employed in network analysis on big data [28]) are memory-bound. In order to effectively utilize accelerators for such algorithms, novel methods for managing arithmetic intensity must be developed. The approach presented in this paper is a first step in this direction.

However, despite its effectiveness, the presented approach has certain drawbacks. Only limited functionality is available in the CUDA programming model to facilitate *Device-Host* pipelining; for instance, *Host* memory must be page-locked [20] (which is scarce), and merely a restricted set of operations is permissible within a callback. This complicates *Device-Host* coordination, and requires a greater number of parameters to be tuned for optimal performance.

In the future, we intend to tailor the application for execution on heterogeneous clusters. This requires support for multi-GPU execution. Also, we have only presented acceleration of one of the steps in the Nullspace algorithm. Work is underway to assess the feasibility of heterogeneous implementations for other steps. In addition to acceleration, efficient memory management is a vital factor for processing of large networks. The number of EFMs grows almost exponentially with the input size [29], which puts very high demands on memory. We are currently in the process of devising better compression techniques, as well as strategies for efficient data distribution on distributed memory architectures.

References

1. Klamt, S., Haus, U.U., Theis, F.: Hypergraphs and cellular networks. PLoS Comput. Biol. 5(5), e1000385 (2009)
2. Heinrich, R., Schuster, S.: The Regulation of Cellular Systems. Springer (1996)
3. Schuster, S., Fell, D.A., Dandekar, T.: A general definition of metabolic pathways useful for systematic organization and analysis of complex metabolic networks. Nat. Biotech. 18(3), 326–332
4. Papin, J.A., Price, N.D., Wiback, S.J., Fell, D.A., Palsson, B.O.: Metabolic pathways in the post-genome era. Trends Biochem. Sci. 28(5), 250–258 (2003)
5. Gagneur, J., Klamt, S.: Computation of elementary modes: a unifying framework and the new binary approach. BMC Bioinformatics 5(1), 175 (2004)
6. Samatova, N.F., Geist, A., Ostrouchov, G., Melechko, A.V.: Parallel out-of-core algorithm for genome-scale enumeration of metabolic systemic pathways. In: Proceedings of the 16th International Parallel and Distributed Processing Symposium, IPDPS 2002, p. 249. IEEE Computer Society, Washington, DC (2002)
7. Terzer, M., Stelling, J.: Accelerating the computation of elementary modes using pattern trees. In: Bücher, P., Moret, B.M.E. (eds.) WABI 2006. LNCS (LNBI), vol. 4175, pp. 333–343. Springer, Heidelberg (2006)
8. Terzer, M., Stelling, J.: Large-scale computation of elementary flux modes with bit pattern trees. Bioinformatics 24(19), 2229–2235 (2008)
9. Terzer, M., Stelling, J.: Parallel extreme ray and pathway computation. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part II. LNCS, vol. 6068, pp. 300–309. Springer, Heidelberg (2010)
10. Jungreuthmayer, C., Ruckerbauer, D.E., Zanghellini, J.: Utilizing gene regulatory information to speed up the calculation of elementary flux modes. arXiv:1208.1853 [q-bio.MN]
11. Jevremović, D., Trinh, C.T., Srienc, F., Sosa, C.P., Boley, D.: Parallelization of nullspace algorithm for the computation of metabolic pathways. Parallel Computing 37(6-7), 261–278 (2011)

12. Jevremović, D., Boley, D., Sosa, C.: Divide-and-conquer approach to the parallel computation of elementary flux modes in metabolic networks. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 502–511 (May 2011)
13. Jevremović, D., Boley, D.: Parallel computation of elementary flux modes in metabolic networks using global arrays. In: The 6th Conference on Partitioned Global Address Space Programming Models (2012)
14. Trinh, C.T., Wlaschin, A., Srienc, F.: Elementary mode analysis: a useful metabolic pathway analysis tool for characterizing cellular metabolism. Appl. Microbiol. Biotechnol. 81(5), 813–826 (2009)
15. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1998)
16. Fukuda, K., Prodon, A.: Double description method revisited. In: Deza, M., Euler, R., Manoussakis, I. (eds.) CCS 1995. LNCS, vol. 1120, pp. 91–111. Springer, Heidelberg (1996)
17. Wagner, C.: Nullspace approach to determine the elementary modes of chemical reaction systems. J. Phys. Chem. B 108(7), 2425–2431 (2004)
18. Jevremović, D., Trinh, C.T., Srienc, F., Boley, D.: On algebraic properties of extreme pathways in metabolic networks. J. Comput. Biol. 17(2), 107–119 (2010)
19. Dongarra, J., Foster, I., Fox, G.C., Gropp, W., Kennedy, K., Torczon, L., White, A. (eds.): The Sourcebook of Parallel Computing. Morgan Kaufmann (2002)
20. NVIDIA: CUDA C programming guide. Design Guide PG-02829-001_v5.0 (October 2012)
21. Mora, J.: Do theoretical flops matter for real application performance? In: HPC Advisory Council Spain Workshop (2012)
22. Davis, J.D., Chung, E.S.: Spmv: A memory-bound application on the gpu stuck between a rock and a hard place. Technical report, Microsoft Research Silicon Valley (September 2012)
23. Kurzak, J., Luszczek, P., Faverge, M., Dongarra, J.: LU factorization with partial pivoting for a multicore system with accelerators. IEEE Transactions on Parallel and Distributed Systems PP(99), 1 (2012)
24. Buluç, A., Gilbert, J.R., Budak, C.: Solving path problems on the GPU. Parallel Computing 36(5-6), 241–253 (2010)
25. Domanski, L., Bednarz, T., Gureyev, T., Murray, L., Huang, E., Taylor, J.: Applications of heterogeneous computing in computational and simulation science. In: 2011 Fourth IEEE International Conference on Utility and Cloud Computing (UCC), pp. 382–389 (December 2011)
26. White, B.S., McKee, S.A., de Supinski, B.R., Miller, B., Quinlan, D., Schulz, M.: Improving the computational intensity of unstructured mesh applications. In: Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, pp. 341–350. ACM, New York (2005)
27. Keutzer, K., Massingill, B.L., Mattson, T.G., Sanders, B.A.: A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects (2010)
28. Batagelj, V., Mrvar, A.: Pajek-program for large network analysis. Connections 21, 47–57 (1998)
29. Klamt, S., Stelling, J.: Combinatorial complexity of pathway analysis in metabolic networks. Molecular Biology Reports 29(1), 233–236 (2002)

Solving a Least-Squares Problem with Algorithmic Differentiation and OpenMP

Michael Förster[1] and Uwe Naumann[2]

[1] LuFG Informatik 12 STCE, RWTH Aachen University, Germany
foerster@stce.rwth-aachen.de
[2] LuFG Informatik 12 STCE, RWTH Aachen University, Germany
naumann@stce.rwth-aachen.de

Abstract. Least-squares problems occur often in practice, for example, when a parametrized model is used to describe a behavior of a chemical, physical or an economic application. In this paper, we describe a method for solving least-squares problems that are given as a large system of equations. The solution combines the commonly used methods with algorithmic differentiation and shared-memory multiprocessing. The system of equations contains model functions that are independent from each other. This independence enables the usage of a multiprocessing approach. With help of algorithmic differentiation by source transformation, we obtain the derivative code of the residual function. The advantage of using source transformation is that we can transform the OpenMP pragmas of the input code into corresponding pendants in the derivative code. This is, in particular in the adjoint case, not a straightforward approach. We show the scaling properties of the derivative code and of the optimization process.

Keywords: Least-Squares Problem, Algorithmic Differentiation, Source Transformation, High Performance Computing, OpenMP, Shared-Memory Multiprocessing.

1 Introduction

In science mathematical problems often consist of large systems of equations. These systems often contain hundreds or even thousands of equations. The associated computation of solutions possesses inherent parallelism when the equations do not have dependences among themselves. In other words, a code implementing the system of equations does not have any data dependences between the equations. This lack of dependences can either reflect the real behavior of the model, or it can be a result of a simplification of the underlying mathematical problem. In either case, the resulting parallelism can be exploited by a numerical implementation, which uses parallel programming.

Nowadays, most of the software engineers must take parallelism into account, since the computer architectures consist of a growing number of computational cores. A relative simple way of exploiting parallelism in C/C++ or Fortran code is to use the OpenMP standard [1], which is an API for using shared-memory parallel programming. The software engineer can, with help of OpenMP compiler directives, declare code fragments as parallelizable. The underlying OpenMP enabled compiler transforms this code into a version that takes care of everything that previously was in the responsibility of the code developer. For example, the creation of the threads at the beginning of a parallel region is implicit given by the standard. Furthermore, it is ensured by OpenMP that the sequential execution after the parallel region only continues, when all threads have finished their work (implicit barrier). All these things, and more, have to be implemented by the code developer, when using a common thread standard as, for example, POSIX threads [2]. This abstraction allows the simple development of parallel programs and this is one reason, why OpenMP is nowadays a commonly accepted standard in most of the high performance computing (HPC) cluster environments.

In nonlinear optimization, derivative values play an important role [3, ch. 8]. These derivative values can either be approximated by finite differences or they can be computed with algorithmic differentiation (AD) [4]. The derivative values provided by AD have the advantage of being accurate up to machine precision. The manual differentiation of a given code is error-prone and often infeasible for some reason, for example, because of the code's size. AD methods often are distinguished by two main methods, AD by source transformation on the one hand, and AD by overloading on the other. Both methods have their advantages and drawbacks. The downside of the overloading approach is that valuable information about the code is neglected because this approach is based on overloading floating-point operations but does not take compiler pragmas into account. The source transformation, on the other side, must support a certain range of programming language to cover most of the codes written as a numerical kernel. When a software engineer uses fancy features of a recently appeared language standard, then this will, very likely, not be supported by the source transformation tool. However, since the complexity of the numerical problems continues to increase, the software engineer is interested in new language features that exploit parallelism, for example OpenMP. In the current note we consider AD by source transformation in difference to related work as described in Section 2.

The least-squares problem is often solved by the Gauss-Newton, or the Levenberg-Marquardt method [5,3]. In this work, we will show how these methods can take advantage of AD together with using multiprocessing programming. The main contribution of this work is that we show how valuable a consideration of OpenMP pragmas can be, when using AD by source transformation, not only in terms of runtime complexity, but also in terms of memory complexity.

The structure of this work is as follows. In Section 2, we show related work, Section 3 introduces a least-squares problem where the objective function contains OpenMP directives. The least-squares problem is solved with help of AD, which is introduced in Section 4. Section 5 displays how we apply AD and OpenMP to the introduced problem, whereby Section 6 presents the experimental results, and Section 7 concludes the work.

2 Related Work

An approach where a parallelization of the Levenberg-Marquardt method is introduced, can be found in [6]. The approach recommends two levels of parallelization. On the one hand a parallelization across the data sets can be made when the data sets have no inter-dependences. On the other hand, the approach uses finite differences to approximate the Jacobian. The finite differences computation uses a lot of calls to the objective function that are independent of each other and can therefore also be done in parallel. The downside is that the resulting Jacobian is only an approximation. In the present work, we use AD to get the Jacobian, which is precise up to machine accuracy. This means we need the first-order derivative code.

An early approach in coupling AD with OpenMP can be found in [7]. The approach assumes that a derivative code P' is given, that applies the so-called tangent-linear vector mode to compute the derivative values. In this mode each scalar variable v from the original code P is associated with a vector representing the gradient of v. A scalar assignment in P with v on the left-hand side, leads to a loop in P', where the gradient of v is computed. The authors suggest to put the derivative code inside an OpenMP parallel region, and to distribute the gradient computation among the threads with help of an OpenMP work-sharing construct. The computations from the original code P are also present in the derivative code P'. To avoid that all threads execute these computations, the authors suggest to use the #pragma master directive, which defines that only the master thread should execute these assignments. Since the master construct does not have an implicit barrier, a possibly occurring race condition must be prevented by inserting a barrier directive into P'. This synchronization overhead cannot be avoided unless the task is embarrassingly parallel.

In [8] the above approach was extended to get better scaling properties. The extension is, that the original statements are no longer computed only by the master thread, but also computed in parallel by all the threads. This is realized through additional thread-local memory such that the different computations do not interfere with each other. The above mentioned synchronization is therefore avoided, but paid with more memory consumption. In addition, the last cited note recommends preprocessed loop bounding scheduling, which is possible since all the gradient vectors have the same size n, where n is the size of input variables of the original code P.

The case that the original code P already contains OpenMP directives, is discussed in [9]. Here, the loops for computing the assignment's gradient are preceded by a combined work-sharing construct, as for example #pragma parallel for. Each thread from the group of threads, that executes the top-level parallel region, creates another group of threads, each time, when it encounters such a work-sharing construct. The implicit barrier at the end of each parallel region suggests that this, again, causes a lot of synchronization overhead. Unfortunately, the author does not present any experimental results, only a formula for the potential speedup is presented.

The above work mainly focuses on the tangent-linear vector mode. The vector mode, which associates each variable from the original P with a gradient vector, consumes a lot of memory. The advantage is that at the end of the execution of the derivative code, the whole Jacobian matrix is given. Nevertheless, if the mathematical problem has a large number of input values, this approach is often not feasible due to memory constrains. Therefore, in this work the tangent-linear model and the adjoint model are defined to be projections of the Jacobian. We will not use the way of parallelizing the computations of the derivatives as it was done in the above approaches. Instead, we assume that the original code P is given, containing already an OpenMP parallel region. This parallel region code is differentiated by applying the tangent-linear model or the adjoint model, respectively.

In case that the AD user adjusts the original code with macros, before applying an AD overloading tool to this code, then the fact of an existing parallel region can be exploited. This approach was introduced in [10,11]. As mentioned, the overloading approach cannot exploit OpenMP directives without adjustments, since they are defined to be compiler directives and therefore the information is only retrieved by the processing compiler at compile time and are not accessible at runtime when the overloaded code is executed. Nevertheless, the macro approach can be used to transfer the information of an existing parallel region to the overloaded code, but the user often does not want to adjust the code before applying AD onto it. Therefore, we want to focus on AD by source transformation.

3 A Least-Squares Problem

A typical mathematical problem is the least-squares problem. This optimization problem always occurs whenever observed measurement data should be described by a mathematical model function, but the exact parameters of the model function are unknown. In the following we will introduce a least-squares example. This example should serve as a typical least-squares problem that can be solved with help of AD and OpenMP.

We consider a simple spring-mass system during a period of time. As mentioned in [12], the vibration of this system can mathematically modeled by the differential equation $u'' + \frac{b}{m}u' + \frac{D}{m}u = 0$, where m is the mass, D is the spring constant and b is the damping constant. Solutions of this equation have the form

$$u(t) = x_1 \exp(-x_2 t) \sin(x_3 t + x_4), \tag{1}$$

where the meaning of the parameters x_1, x_2, x_3, x_4 are not of interest here, but any mechanic textbook certainly contains a description. We consider these parameters as unknown, and the goal is to find the parameter values that minimize the discrepancy between the model function and the observed data.

The system of equations that we want to consider, consists of n equations, all having the form (1). This means instead of considering only a single spring-mass system, we want to approximate parameters for n independent spring-mass systems. The reason can be independent measurement data or real independent systems. Each equation has four parameters that should be approximated with help of m measurement values. Hence, the residual function maps $4n$ parameter values to mn measurement values. The objective function $\Phi : \mathbb{R}^{4n} \to \mathbb{R}$ is

$$\Phi(\mathbf{x}) = \frac{1}{2}\|F(x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4}, x_{2,1}, \dots, x_{n,4})\|_2^2 \quad = \frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{m} F_{j,i}(\mathbf{x})^2 \quad ,$$

with the residual function $F : \mathbb{R}^{4n} \to \mathbb{R}^{nm}$, and

$$F_{j,i}(\mathbf{x}) := x_{j,1}e^{-x_{j,2}t_{j,i}} \sin(x_{j,3}t_{j,i} + x_{j,4}) - b_{j,i}, \quad j = 1, \dots, n, i = 1, \dots, m.$$

The observed measurement value of the jth equation at time $t_{j,i}$ is referred to as $b_{j,i}$. The optimization problem is

$$\Phi(\mathbf{x}^*) = \min_{\mathbf{x}\in\mathbb{R}^{4n}} \Phi(\mathbf{x}) \quad , \tag{2}$$

and the gradient of Φ is $\nabla\Phi(\mathbf{x}) = (\nabla F(\mathbf{x}))^T F(\mathbf{x})$. The Hessian of Φ is given by

$$\nabla^2\Phi(\mathbf{x}) = \nabla F(\mathbf{x})^T \nabla F(\mathbf{x}) + \sum_{j=1}^{n}\sum_{i=1}^{m} \nabla^2 F_{j,i}(\mathbf{x})\nabla F_{j,i}(\mathbf{x}) \tag{3}$$

The goal is to find a solution $\mathbf{x}^*$ that satisfies $\nabla\Phi(\mathbf{x}^*) = 0$, and $\nabla^2\Phi(\mathbf{x}^*) \in \mathbb{R}^{4n\times 4n}$ must be symmetric positive definite. A possible implementation of function F may be as shown in Listing 1.1.

```
void F(const int n, const int m,
       double* t, double *x, double* b, double* y)
 // n: number of equations
 // m: number of measurement values per equation
 // t: vector with m*n timestamps
 // x: current x vector with 4*n values
 // b: measurement values taken at timestamps in t
 // y: output vector containing m*n values
{
#pragma omp parallel for
  for(int j=0;j<n;j++)
  {
    int i=0, xbase, mbase;
    double yy, x0, x1, x2, x3, t_i, b_i;
    xbase=j*4;
    x0=x[xbase+0]; x1=x[xbase+1];
    x2=x[xbase+2]; x3=x[xbase+3];
    while(i<m) {
      mbase=j*m;
      t_i=t[mbase+i]; b_i=b[mbase+i];
      yy=x0*exp(0.-x1*t_i)*sin(x2*t_i+x3);
22          y[mbase+i]=yy-b_i;
23          i=i+1;
24        }
25    }
26 }
```

Listing 1.1. Residual function F

The outer loop line 11 iterates through all n equations. After collecting the four values of x for the current equation (lines 16 to 17), the inner loop computes all the residuals for the jth equation (line 18). Since the measurements of each equation are independent of each other, we can parallelize the outer loop. Therefore, we define the parallel pragma together with the loop construct of OpenMP in front of the outer loop (line 10). In Section 5, we show how the optimization problem (2) can be solved with help of AD and the first-order derivative code of F. The next section shows how we obtain the first-order derivative code of F with AD.

4 Algorithmic Differentiation

The following definitions are taken from [13]. Nevertheless, we adjust these definitions to fit our residual function $F : \mathbb{R}^{4n} \to \mathbb{R}^{nm}$ in terms of input and output space dimension. The function $F^{(1)} : \mathbb{R}^{8n} \to \mathbb{R}^{nm}$, defined as

$$\mathbf{y}^{(1)} = F^{(1)}(\mathbf{x}, \mathbf{x}^{(1)}) \equiv \nabla F(\mathbf{x}) \cdot \mathbf{x}^{(1)} \quad , \tag{4}$$

is referred to as the *tangent-linear model* of F. Hence, the tangent-linear model is a projection of the Jacobian $\nabla F(\mathbf{x})$ into the direction $\mathbf{x}^{(1)} \in \mathbb{R}^{4n}$. The superscript of $F^{(1)}$ indicates a first-order tangent-linear component. The *adjoint model* of F is the function $F_{(1)} : \mathbb{R}^{4n+nm} \to \mathbb{R}^{4n}$, defined by

$$\mathbf{x}_{(1)} = F_{(1)}(\mathbf{x}, \mathbf{y}_{(1)}) \equiv (\nabla F(\mathbf{x}))^T \cdot \mathbf{y}_{(1)} \quad , \tag{5}$$

which induces a projection of the transposed Jacobian $\nabla F(\mathbf{x})$ into the direction $y_{(1)} \in \mathbb{R}^{nm}$. The subscript in $F_{(1)}$ displays that this is a first-order adjoint function. As an example, let us consider the assignment

```
y=sin(x2*t_i+x3);    .
```

The right-hand side is a subexpression from line 21 of Listing 1.1. The computation of the right-hand side can be displayed by a directed acyclic graph (DAG), see Figure 1a. We associate each node of the DAG with an auxiliary variable v, as shown in Figure 1b. The DAG in Figure 1b is called linearized, because each edge is labeled with the local partial derivative of the edge's target node with respect to its predecessor. For example, node v_5 represents the value $\sin(v_4)$, and the partial derivative of v_5 with respect to its predecessor v_4 is $\cos(v_4)$. Therefore, the edge from v_4 to v_5 is labeled with $\cos(v_4)$.

When we assume that the derivative code provides the values of the linearized DAG in the variables v0, v1, ..., v5, then the tangent-linear assignment of y=sin (x2*t_i+x3) is

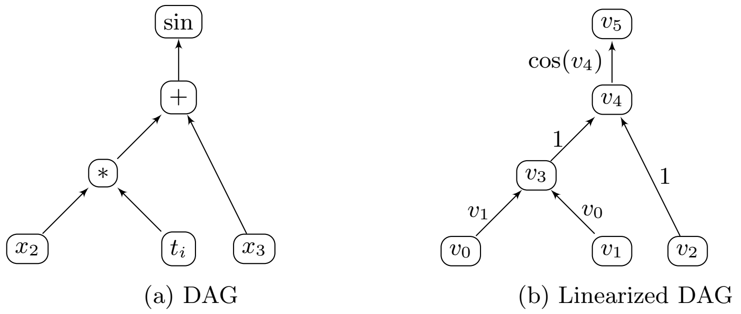


Fig. 1. The expression $\sin(x_2 \cdot t_i + x_3)$ can be represented by a DAG (Figure 1a). The linearized DAG of this expression is shown in Figure 1b.

```
t1_y = t1_x2*(cos(v4)*v1)+t1_t_i*(cos(v4)*v0)+t1_x3*cos(v4) ;
```

where `t1_v` is the tangent-linear component of the variable `v`. The adjoint model looks as

```
a1_x2  += a1_y*(cos(v4)*v1) ;
a1_t_i += a1_y*(cos(v4)*v0) ;
a1_x3  += a1_y*cos(v4) ;
a1_y    = 0;
```

where `a1_v` is the adjoint variable associated with `v`. These listings already indicate that differentiating a given code is error-prone, particularly in the adjoint model case. Fortunately, there are several tools providing AD by source transformation, for example OpenAD[1], ADOL-C[2], Tapenade[3], and `dcc`[4]. Nevertheless, none of these tools provide support for source transformation of OpenMP code. It is ongoing work to enable `dcc` to support OpenMP [14,15,16]. This concludes the AD section. Further details can be found in the textbooks [4,13].

5 Solving Least-Squares with AD and OpenMP

Two generally used methods to solve the least-squares problem are the Gauss-Newton method on the one hand, and the Levenberg-Marquardt method on the other hand. What both methods have in common is the assumption that the impact of the second-order derivatives $\nabla^2 F_{i,j}$ in (3) is small. A good description of these methods can be found in the textbooks [3,12]. Basically, the algorithms solve the optimization problem by solving iteratively a linear system of equations:

$$A\mathrm{s}^k = -\mathbf{b} \quad ,$$

[1] `http://www.mcs.anl.gov/OpenAD/`
[2] `https://projects.coin-or.org/ADOL-C`
[3] `http://tapenade.inria.fr:8080/tapenade/index.jsp`
[4] `http://www.stce.rwth-aachen.de/software/dcc.html`

where $\mathbf{s}^k$ solves the linear system in the kth iteration. The right-hand side is defined by the negative value of the gradient $\mathbf{b} = (\nabla F(\mathbf{x}))^T F(\mathbf{x})$ of the objective function. Matrix $A \in \mathbb{R}^{4n \times 4n}$ consists of an approximation of the second-order derivative of the objective function $A = (\nabla F(\mathbf{x}^k))^T \nabla F(\mathbf{x}^k)$.

We do not compute the whole Jacobian $\nabla F(\mathbf{x}) \in R^{nm \times 4n}$, since the adjoint model (5) provides exactly what we need for acquiring $\mathbf{b}$. Therefore, we compute $\mathbf{b}$ as follows:

$$\begin{aligned} \mathbf{y} &\leftarrow F(\mathbf{x}) \\ \mathbf{b} &\leftarrow F_{(1)}(\mathbf{x}, \mathbf{y}) \end{aligned}$$

We use a combination of the tangent-linear model with the adjoint model to compute the matrix A column by column:

$$\begin{aligned} &\textbf{for } i \in \{1, \ldots, 4n\} : \\ &\quad \mathbf{y}^{(1)} \leftarrow F^{(1)}(\mathbf{x}, \mathbf{e}_i) \\ &\quad A_i \leftarrow F_{(1)}(\mathbf{x}, \mathbf{y}^{(1)}) \quad . \end{aligned}$$

First, we compute the ith column of the Jacobian $\nabla F(\mathbf{x})$ by calling the tangent-linear model with $\mathbf{x}$ and the ith Euclidean basis vector $\mathbf{e}_i$ as arguments. Afterwards, we use this result to obtain the ith column of A by calling the adjoint model. The whole algorithm is presented in Algorithm 1.

Algorithm 1. Gauss-Newton and Levenberg-Marquardt method and using AD to obtain $\mathbf{b}$ and A

```
Require: $\mathbf{x}^0 \in \mathbb{R}^4, \epsilon \in \mathbb{R}, \mu > 0$
Ensure: $\mathbf{x}^* = \min_{\mathbf{x} \in \mathbb{R}^4} \frac{1}{2}\|f(\mathbf{x})\|_2^2$
  $k \leftarrow 0$
  $r \leftarrow 2\epsilon$
  while $r > \epsilon$ do
    $\mathbf{y} \leftarrow F(\mathbf{x}^k)$
    $\mathbf{b} \leftarrow F_{(1)}(\mathbf{x}^k, \mathbf{y})$
    for all $i = 1, \ldots, 4n$ do
      $\mathbf{y}^{(1)} \leftarrow F^{(1)}(\mathbf{x}^k, \mathbf{e}_i)$
      $A_i \leftarrow F_{(1)}(\mathbf{x}^k, \mathbf{y}^{(1)})$
    end for
    if Levenberg-Marquardt method then
      $\mu \leftarrow$ adjust_mu($\mu$)
      $A \leftarrow A + \mu^2 I$
    end if
    Solve linear system $A\mathbf{s}^k = -\mathbf{b}$
    $\mathbf{x}^{k+1} \leftarrow \mathbf{s}^k + \mathbf{x}^k$
    $r \leftarrow \|b\|_2$
    $k \leftarrow k + 1$
  end while
```

Next, we have to clarify why the transformation of the parallel region in the original code can stay a parallel region in the derivative code. This means in terms of multiprocessing that the source transformation does not introduce data dependences that are *critical*. Critical references are references that are written by multiple threads at the same time or references that are written by one thread and simultaneously read by another thread [17].

In case of the tangent-linear source transformation, the data flow of the original code is inherited to the tangent-linear code. The floating-point assignments of the original code are transformed as explained in Section 4. The data flow in Listing 1.1 is first from memory locations, shared by all threads, to thread-local variables (line 16 to line 20). The computation takes place (line 21) with thread-local variables and afterwards the result is written back to a shared memory location (line 22). The store in line 22 is non-critical because each equation has its own set of measurement values.

Each shared or thread-local variable in `F` has its associated tangent-linear component in `t1_F`. The tangent-linear components have the same shared or thread-local status as its original pendant. Hence, the write operation in line 22 to the non-critical reference `y[mbase+i]` leads to a non-critical write access in the tangent-linear code to `t1_y[mbase+i]`.

The adjoint model reverses the data flow of the original code. Without going into details, we have to examine whether or not there are references in the original code that may be read simultaneously by multiple threads. In this case, this leads to a critical reference in the adjoint code and therefore to a race condition. In our example, each equation has its own set of four parameters. For that reason, no shared reference in line 16 to line 20 is read by multiple threads. Therefore, the adjoint source transformation can be executed in parallel as well.

6 Experimental Results

To test the approach we used two systems of the center of computing located at the RWTH Aachen University. On the one hand we used a SUN T5120 node and on the other hand we ran tests on a Bull Coherent System (BCS) system. Both systems provide up to 64 cores. To test the scalability of the derivative codes, we used the Sun T5120 node.

Threads	Original	Tangent-linear	Adjoint
1	304s	686s	1486s
2	154s	340s	735s
4	77s	170s	367s
8	38s	85s	184s
16	22s	43s	106s
32	12s	27s	57s
64	9s	20s	42s

The tangent-linear code is about factor two slower than the original code, whereby the adjoint code needs around five times longer than the original code. These

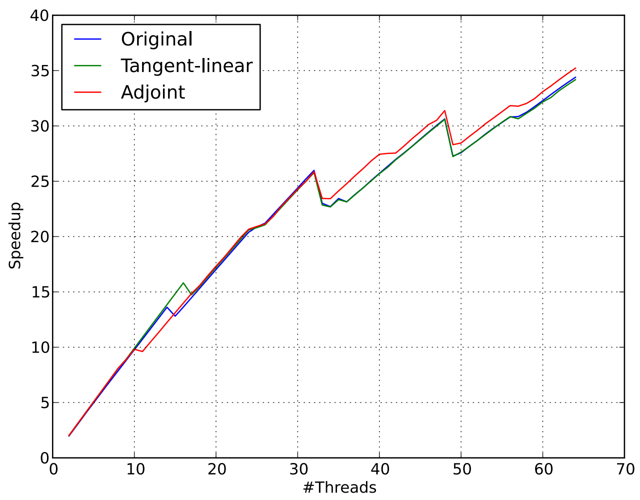


Fig. 2. This figure shows the speedup results of the original code, tangent-linear, and adjoint code. The original code and the derivative codes scale almost linear up to a number of threads of 64.

values are comparable with other results we get from derivative codes generated by source transformation, see for example [13]. Figure 2 indicates that all three codes scale almost linear up to a number of threads of 64. A small decrease can be recognized when the number of threads grows over a number being divisible by 16 (16, 32, 48). This is likely connected to the hardware architecture of the T5120.

Another interesting fact that we could exploit through the knowledge of the parallelism inside the original code is that we form the adjoint code in a way such that a lot of memory can be saved. For the least-squares example the memory usage between the sequential adjoint code without using knowledge about parallelism and the adjoint code where we exploit this fact, this makes a difference of factor 100 when using 10 threads. This factor decreases when using more threads because each thread uses thread-local data for the execution of the adjoint code but this is still an order of magnitude.

Figure 3 shows the speedup results from the least-squares problem where the number of equations is 1000 and the number of measurement points is 21. The results show that the Bull machine provides the best speedup with almost 6 with 12 threads. The Sun node yields the best speedup factor of 14 when using all 64 cores. The speedup on the Bull machine decreases when the number of threads is bigger than 32. This is likely because the node comprises of two connected S6010 machines each equipped with a mainboard with four eight-core CPUs. The connection between these machines is formed by a Bull Coherent Switch (BCS). In case that the number of threads is bigger than 32, there seems to be too much overhead between the two nodes to achieve further performance increase.

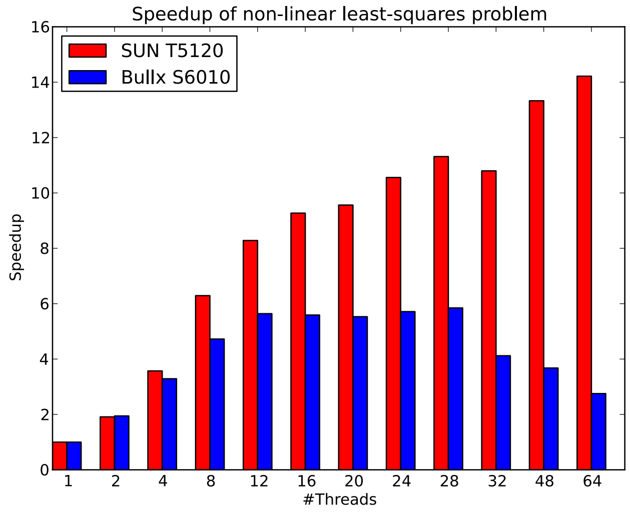


Fig. 3. The speedup results from the least-squares problem with 1000 equations and 21 measurement points

7 Conclusion

We showed how a certain least-squares problem can be solved with help of AD and OpenMP. The usage of OpenMP turns out to be a good way to improve efficiency of the optimization process, not only in terms of runtime but also in terms of memory consumption. Both, the tangent-linear code and the adjoint code reach a speedup value of about 35. The computation of the least-squares solution could be accelerated by factor 6 on a Bull BCS system and by factor 14 on a Sun T5120 machine.

This work showed that pragmas in the original code can provide valuable information for the AD source transformation. These methods are not restricted to OpenMP but rather could be transferred to other pragma based APIs as the OpenACC standard for GPU computation. Further investigations must be made to provide a source transformation of OpenMP parallel regions by a compiler. It is ongoing work to enable our tool `dcc` to support OpenMP source transformation.

References

1. OpenMP Architecture Review Board. OpenMP Application Program Interface. Specification (2011)
2. IEEE. IEEE: Threads Extension for Portable Operating Systems (Draft 6). Specification (1992)
3. Nocedal, J., Wright, S.: Numerical Optimization, 2nd edn. Springer, New York (2006)
4. Griewank, A., Walter, A.: Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation, 2nd edn. SIAM, Philadelphia (2008)

5. Madsen, K., Nielsen, B., Tingleff, O.: Methods for Non-Linear Least-Squares Problems (2004)
6. Cao, J., Novstrup, K., Goyal, A., Midkiff, S., Caruthers, J.: A Parallel Levenberg-Marquardt Algorithm. In: Proceedings of the 23rd International Conference on Supercomputing, ICS 2009, pp. 450–459. ACM, New York (2009)
7. Bücker, M., Lang, B., an Mey, D., Bischof, C.: Bringing together automatic differentiation and OpenMP. In: ICS 2001: Proceedings of the 15th International Conference on Supercomputing, pp. 246–251. ACM, New York (2001)
8. Bücker, M., Lang, B., Rasch, A., Bischof, C., an Mey, D.: Explicit Loop Scheduling in OpenMP for Parallel Automatic Differentiation. In: Annual International Symposium on High Performance Computing Systems and Applications, p. 121 (2002)
9. Bücker, M., Rasch, A., Wolf, A.: A class of OpenMP applications involving nested parallelism. In: SAC 2004: Proceedings of the 2004 ACM Symposium on Applied Computing, pp. 220–224. ACM, New York (2004)
10. Kowarz, A., Walther, A.: Parallel Derivative Computation using ADOL-C. In: Nagel, W.E., Hoffmann, R., Koch, A. (eds.) 9th Workshop on Parallel Systems and Algorithms (PASA) held at the 21st Conference on the Architecture of Computing Systems (ARCS), Dresden, Germany, February 26. LNI, vol. 124, pp. 83–92. GI (2008)
11. Bischof, C., Guertler, N., Kowarz, A., Walther, A.: Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C, pp. 163–173. Springer, Berlin (2008)
12. Dahmen, W., Reusken, A.: Numerik für Ingenieure und Naturwissenschaftler. Springer-Lehrbuch. Springer (2008)
13. Naumann, U.: The Art of Differentiating Computer Programs. Software, Environments and Tools. Society for Industrial and Applied Mathematics (2012)
14. Hannemann, R., Tillack, J., Schmitz, M., Förster, M., Wyes, J., Nöh, K., von Lieres, E., Naumann, E., Wiechert, W., Marquardt, W.: First- and Second-Order Parameter Sensitivities of a Metabolically and Isotopically Non-Stationary Biochemical Network Model. In: Otter, M., Zimmer, D. (eds.) Proceedings of the 9th International Modelica Conference, pp. 641–648. Modelica Association (2012)
15. Schanen, M., Förster, M., Gendler, B., Naumann, U.: Compiler-based Differentiation of Higher-Order Numerical Simulation Codes using Interprocedural Checkpointing. International Journal on Advances in Software 5(1&2), 27–35 (2012)
16. Förster, M., Naumann, U., Utke, J.: Toward Adjoint OpenMP. Technical Report AIB-2011-13, RWTH Aachen (July 2011)
17. Ben-Ari, M.: Principles of Concurrent and Distributed Programming. Prentice-Hall International Series in Computer Science. Addison-Wesley (2006)

Hierarchical Parallel Algorithm for Modularity-Based Community Detection Using GPUs

Chun Yew Cheong[1], Huynh Phung Huynh[1], David Lo[2], and Rick Siow Mong Goh[1]

[1] Institute of High Performance Computing, A*STAR, Singapore
{cheongcy,huynhph,gohsm}@ihpc.a-star.edu.sg
[2] Singapore Management University, Singapore
davidlo@smu.edu.sg

Abstract. This paper describes the design of a hierarchical parallel algorithm for accelerating community detection which involves partitioning a network into communities of densely connected nodes. The algorithm is based on the Louvain method developed at the Université Catholique de Louvain, which uses modularity to measure community quality and has been successfully applied on many different types of networks. The proposed hierarchical parallel algorithm targets three levels of parallelism in the Louvain method and it has been implemented on single-GPU and multi-GPU architectures. Benchmarking results on several large web-based networks and popular social networks show that on top of offering speedups of up to 5x, the single-GPU version is able to find better quality communities. On average, the multi-GPU version provides an additional 2x speedup over the single-GPU version but with a 3% degradation in community quality.

Keywords: Community detection, parallel algorithm, GPU, social networks.

1 Introduction

Detecting community structure has attracted increasing attention [1] with the recent rise of social networks such as Facebook and Twitter. Community detection involves clustering highly connected nodes in a network into communities. For social networks, community detection can be applied to online marketing campaigns such as recommendation systems and viral marketing strategies. Besides social networks, community detection has many applications in other areas such as finding webpages that have the same topic in WWW [2], identifying communities for contact tracing in the event of infectious disease outbreak [3], and identifying a group of friends in a mobile network [4].

Nowadays, networks with hundreds of millions of nodes and links are common and their sizes continue to increase. Community detection on these huge networks will take a large amount of time. This will limit the quality of community information extracted due to significant computational complexity. In order to cope with larger networks as well as perform more complex analysis with faster response time, there is a need to accelerate the performance of the core kernel of the community detection algorithm.

Graphics Processing Unit (GPU) acceleration has been the current trend in the high performance computing community. Each GPU can have up to 2000 physical processing cores running in parallel [5]. On top of these processing cores, tens of thousands of software threads are concurrently executed in interleaved fashion to maximize the performance. With its massively parallel computing power, GPU is potentially a suitable candidate for accelerating community detection algorithms for large networks. However, mapping a community detection algorithm to GPU faces some challenging problems. Firstly, processing a large graph requires a large amount of data communication among different functions of the algorithm but data communications between CPU and GPU or within GPU memory hierarchy are very expensive. Secondly, core computation of the community detection algorithm is quite diversified among nodes and communities but GPU execution uses a Single Instruction Multiple Threads (SIMT) model. In the SIMT model, all threads execute the same instructions at a time step. Finally, there is a need to efficiently utilize the heterogeneous computing power in GPU, multi-GPU, and multi-core systems. To address the above challenges, an efficient implementation of the community detection algorithm using GPU as accelerator is proposed in this paper. The proposed parallel community detection algorithm is based on the highly-cited Louvain method [4]. The key contributions of this paper are summarized:

1. Proposing the first parallel community detection algorithm based on the Louvain method.
2. Accelerating the parallel Louvain version on single GPU platform.
3. Further accelerating the parallel Louvain version on a multi-core with multi-GPU architecture.
4. Comprehensive experiments on different networks: social networks (Facebook [6], Twitter (extracted by Twitter API), Orkut [7], LiveJournal [7], Flickr [8]) and web-based networks (uk-2005 [9], webbase-2001 [10]).

2 Background

2.1 Modularity-Based Community Detection

The community detection problem has been well-studied in the literature. Existing algorithms can be classified into a few major classes: divisive algorithms [11], modularity-based methods [12], dynamic methods [13], and spectral methods [14]. A more complete summary about community detection algorithms can be found in the survey paper [1]. In this paper, the focus is on modularity-based methods which use modularity as a measure of the quality of a community. Modularity, proposed by Newman [12], is the comparison between the actual density of links in a sub-graph (community) and the density one is expected to have in the sub-graph if the vertices of the sub-graph were attached arbitrarily. For a directed graph, the modularity Q is defined as follows:

$$Q = \frac{1}{m}\sum_{i,j \in V}\left(A_{ij} - \frac{k_i^{out}k_j^{in}}{m}\right)\delta(c_i, c_j) \tag{1}$$

where V is the set of nodes in the network, A_{ij} is the weight of the link between nodes i and j, k_i^{out} and k_j^{in} are the sum of the weights of the outgoing and incoming links of i and j, respectively, c_i is the community that node i is in, the δ function $\delta(u, v)$ is 1 if u is equal to v and 0 otherwise, and m is the sum of the weights of all the links in the network. From (1), the modularity of a network ranges between 0 and 1, with a higher value indicating a stronger community structure.

Among modularity-based methods, the Louvain method [4] is well-known to be able to perform fast community detection. It has been successfully applied on many different types of networks [15-17]. Instead of computing modularity for the whole network as in (1), which is computationally expensive, the Louvain method introduced the gain in modularity of moving a node i into a community C, which is computed as follows:

$$\Delta Q = \frac{1}{m}\left(k_{i,C} + k_{C,i} - \frac{k_i^{out}k_C^{in} + k_i^{in}k_C^{out}}{m}\right) \tag{2}$$

where $k_{i,C}$ is the sum of the weights of the links from i to the nodes in C, $k_{C,i}$ is the sum of the weights of the links from the nodes in C to i, and k_C^{in} and k_C^{out} are the sums of the weights of the incoming and outgoing links of all the nodes in C, respectively. k_i^{out}, k_i^{in}, and m are as defined in (1).

2.2 GPU Computing

A GPU supports massive parallelism through a number of streaming multi-processors (SMs), each consisting of a number of physical processing cores running in SIMT mode. Parallel software threads are grouped into thread blocks which run on each of the available SMs. There are typically more software threads than there are physical processing cores. In order to efficiently schedule the large number of software threads on SMs, 32 threads are statically grouped into scheduling units, referred to as warps in the NVIDIA literature. Warps execute in lockstep, and if one or more threads in a warp wait for data to be ready, the entire warp has to wait as well. A hardware scheduler will then select another ready warp for execution. Recently, Soman et al. [18] proposed a community detection algorithm based on label propagation and mapped it to GPU platform using some standard GPU primitives such as Bitonic sort.

3 The Louvain Method for Community Detection

3.1 Description of the Louvain Method

The Louvain method [4] is a greedy optimization method for community detection. It partitions a network into communities by maximizing the modularity of the partition. In essence, the Louvain method consists of two phases, modularity optimization and community aggregation. In the modularity optimization phase, each node of the network is initially assigned to its own community, i.e. the number of communities is equal to the number of nodes. Each node is then considered, in turn, if it would stay in its original community or move to one of its neighboring communities. This is done by removing the node from its original community, computing the gain in modularity if the node were inserted into each of its neighboring communities, and moving the node to the community with the maximum positive gain. Each cycle of this process through all the nodes in the network is referred to as an iteration. The modularity optimization phase terminates when no improvement in modularity can be achieved. At the end of the modularity optimization phase, the network is partitioned into a number of communities. Next, the community aggregation phase involves building a new, but smaller, network by aggregating nodes in the original network that belong to the same community such that the nodes in the new network are the communities at the end of the preceding modularity optimization phase. The weight of the link between two nodes in this new network is the total weight of the links between the nodes of the two corresponding communities in the original network. The links between the nodes of the same community become self-loops of the corresponding node in the new network. With the new network, the modularity optimization phase can then be applied again and the two phases iterate until no improvement in modularity can be achieved. Each application of the modularity optimization phase followed by the community aggregation phase is referred to as a pass.

3.2 Profiling of the Louvain Method

In order to identify the computational bottlenecks that should be targeted when parallelizing the Louvain method to speed up its computation, a profiling of the method was conducted on a web-based network. The network is a sub-network of the .uk domain and it has 16 million nodes and 287 million links. The profiling results are shown in Fig. 1. In order to highlight the contribution of this paper, the profiling results, as well as all subsequent timing results, will only consider the computation time spent on community detection, while I/O times are omitted.

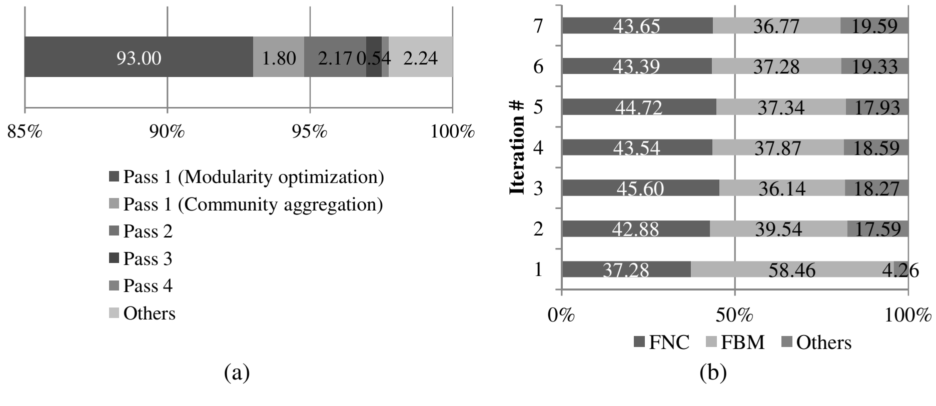


Fig. 1. Profiling results: (a) Percentage of computation time spent in each pass and (b) breakdown of computation time of each iteration of the modularity optimization phase of the first pass

Figure 1(a) shows that 94.8% of the computation time is spent in the first pass of the Louvain method. This result is expected since all the nodes in the network are considered in the first pass, while the other passes process much smaller networks due to the community aggregation phase. Figure 1(a) also shows that 93% of the computation time is spent in the modularity optimization phase of the first pass. It can be seen in Fig. 1(b) that within each of the seven iterations of the modularity optimization phase of the first pass, approximately 80% of the computation time is spent on two main components of the modularity optimization phase. The first component, referred to as Find Neighboring Communities (FNC), involves computing the set of unique neighboring communities for each node i. For each neighboring community C, $k_{i,C}$ and $k_{C,i}$, which are needed for the calculation of the gain in modularity in (2), are also computed at the same time. The other component, referred to as Find Best Move (FBM), computes the gain in modularity of moving each node into each of its neighboring communities and then moving the node to the community that gives the maximum positive gain.

From the profiling results, it is clear that the modularity optimization phase of the first pass of the Louvain method is the main computational bottleneck that should be targeted when parallelizing the algorithm. In order to speed up the modularity optimization phase of the first pass, it is critical to accelerate the computation of FNC and FBM.

4 Hierarchical Parallel Algorithm

The hierarchical parallel algorithm for community detection proposed in this paper targets three levels of parallelism in the Louvain method to speed up its computation.

At the highest level, the original network is partitioned into a number of sub-networks and a set of removed links which consists of the links that join nodes residing in different sub-networks. The Louvain method can then be applied to solve the community detection problem in each of the sub-networks in parallel. After this, the resulting networks are combined into a single network using the removed links, and then the Louvain method is applied once more on this combined network to obtain the final community results. On top of decomposing the original network into sub-networks and processing them in parallel, the effectiveness of this level of parallelism also stems from the fact that the combined network, obtained from the resulting networks after processing the sub-networks in parallel, is typically a few orders of magnitude smaller than the original network due to the community aggregation phase of the Louvain method.

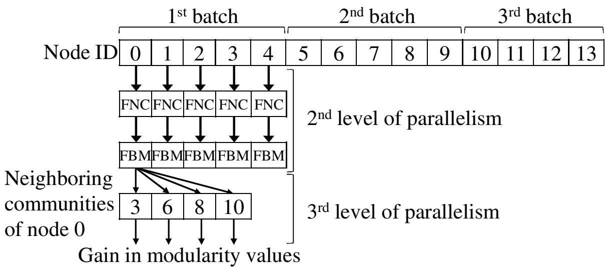


Fig. 2. Illustration of the second and third levels of parallelism

The second level of parallelism involves visiting nodes in parallel during each iteration of the modularity optimization phase. In the original Louvain method, nodes are visited sequentially in each iteration, where each node visit involves applying FNC to obtain the set of neighboring communities of the node, followed by FBM to move the node to the community that results in the maximum positive gain in modularity. As shown in Fig. 2, this level of parallelism suggests the division of nodes in the network into batches, with the nodes in each batch being processed in parallel. It is to be noted that the computations for visiting two nodes in a batch may not be independent since one of the nodes may be in the neighboring community of the other node. While it is possible to find mutually independent batches, it would incur additional computational cost. In this paper, this inaccuracy is allowed but only atomic updates to a node's community status are permitted.

The third and lowest level of parallelism involves computing the gain in modularity of inserting a node into each of its neighboring communities in parallel (See Fig. 2). This level of parallelism is intuitive and would be effective when a node has a large number of neighboring communities.

5 Mapping to GPU

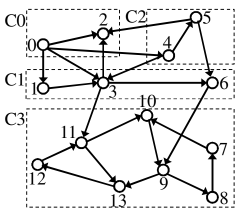
This section describes how the three levels of parallelism proposed in the previous section for the Louvain method are implemented on the GPU.

5.1 Mapping of Find Neighboring Communities to GPU

As described in Section 3.2, Find Neighboring Communities (FNC) performs two functions. It not only finds the set of neighboring communities for each node i but for each neighboring community C of i, it also computes $k_{i,C}$ and $k_{C,i}$ which are needed for the calculation of the gain in modularity as given in (2).

An example to illustrate the mapping of FNC to GPU is shown in Fig. 3. In the figure, the current community status of the network is shown. It can also be seen in the figure that the network is represented by an array of structures. Each structure consists of four elements – the node ID, the neighboring node ID, the outgoing link weight, and the incoming link weight. For example, the first column in the data structure indicates that node 0 has an outgoing link of weight 1 to its neighboring node 1. Only five nodes, i.e. node 0 to node 4, are considered in the data structure in the figure as it is assumed that nodes are processed in batches of five in the second level of parallelism as shown in Fig. 2. The data for the five nodes are combined into a single array and copied to the host memory of the GPU to reduce communication overhead for each memory copy instruction, which can add up to a significant amount due to the sheer size of the network.

GPU kernel 1 performs two functions. Based on the current community status of the network, the assigned GPU thread converts each neighboring node ID in the data structure to its corresponding community ID. The thread also prepares the key for the GPU radix sort in the next step. The GPU radix sort arranges the entire array first in



Community	Nodes
0	0, 2
1	1, 3, 6
2	4, 5
3	7, 8, 9, 10, 11, 12,13

⇩ Data structure

Node ID	0	0	0	0	1	1	2	2	2	3	3	3	3	3	3	4	4	4
Neighboring node ID	1	2	3	4	0	3	0	3	5	0	1	2	4	6	11	0	3	5
Outgoing link weight	1	1	1	1	0	1	0	0	0	0	0	1	0	1	1	0	1	1
Incoming link weight	0	0	0	0	1	0	1	1	1	1	1	0	1	0	0	1	0	0

⇩ GPU kernel 1

Node ID	0	0	0	0	1	1	2	2	2	3	3	3	3	3	3	4	4	4
Neighboring node ID	**1**	**0**	**1**	**2**	**0**	**1**	**0**	**1**	**2**	**0**	**1**	**0**	**2**	**1**	**3**	**0**	**1**	**2**
Outgoing link weight	1	1	1	1	0	1	0	0	0	0	0	1	0	1	1	0	1	1
Incoming link weight	0	0	0	0	1	0	1	1	1	1	1	0	1	0	0	1	0	0

⇩ GPU radix sort

Node ID	**0**	**0**	**0**	**0**	**1**	**1**	**2**	**2**	**2**	**3**	**3**	**3**	**3**	**3**	**3**	**4**	**4**	**4**
Neighboring node ID	**0**	**1**	**1**	**2**	**0**	**1**	**0**	**1**	**2**	**0**	**0**	**1**	**1**	**2**	**3**	**0**	**1**	**2**
Outgoing link weight	1	1	1	1	0	1	0	0	0	0	1	0	1	0	1	0	1	1
Incoming link weight	0	0	0	0	1	0	1	1	1	1	0	1	0	1	0	1	0	0

⇩ GPU kernel 2

Node ID	0	**0**	0		1	1	2	2	2	**3**	**3**	3	3			4	4	4
Neighboring node ID	0	**1**	2		0	1	0	1	2	**0**	**1**	2	3			0	1	2
Outgoing link weight	1	**2**	1		0	1	0	0	0	**1**	**1**	0	1			0	1	1
Incoming link weight	0	**0**	0		1	0	1	1	1	**1**	**1**	1	0			1	0	0

Fig. 3. Example to illustrate mapping of Find Neighboring Communities to GPU

order of increasing node ID and then in order of increasing neighboring community ID for array elements with the same node ID. The radix sort in the Thrust library is used in this paper. With the sorted array, each node is being assigned a GPU thread in GPU kernel 2. The thread goes down the array elements belonging to the node and sums up the weights for adjacent elements with the same neighboring community ID to give the final output of FNC. As can be seen in Fig. 3, node 0 has outgoing links to communities 0, 1, and 2 and the weights of the links are as shown.

5.2 Mapping of Find Best Move to GPU

The second and third levels of parallelism are considered when mapping Find Best Move (FBM) to GPU. As described in Section 4, the second level of parallelism involves dividing the nodes in the network into batches, with the nodes in each batch being processed in parallel. In the GPU implementation, the GPU kernel for FBM assigns a number of threads to each node in a batch. The threads handle the third level of parallelism by computing the gain in modularity of inserting the node into each of its neighboring communities in batches. The first thread of the assigned threads is also responsible for all computations in the second level of parallelism, such as determining which neighboring community offers the maximum positive gain in modularity and moving the node into the community.

5.3 Multi-core with Multi-GPU Implementation

The highest level of parallelism is implemented by using a multi-core with multi-GPU architecture as illustrated in Fig. 4 for a four-partition example. A simple partitioning scheme, which divides the nodes in the network evenly between the sub-networks (SNs) based on their node IDs, is used. The links that join nodes residing in different sub-networks are set aside. The Louvain method is then applied to solve the community detection problem in each sub-network in parallel. The Louvain method incorporates the second and third levels of parallelism as described in the previous two sections. It is to be highlighted that since the profiling results show that most of the computation time is spent in the first pass of the Louvain method, only the FNC and FBM in the first pass is offloaded to the GPU. The parts of the Louvain method that have been offloaded to the GPU are implemented using a multi-GPU architecture, while the serial parts of the Louvain method are implemented as a multi-core architecture. The results obtained for the individual sub-networks are then combined with the removed links into a single network. The serial version of the Louvain method is applied once more on this combined network to obtain the final community results.

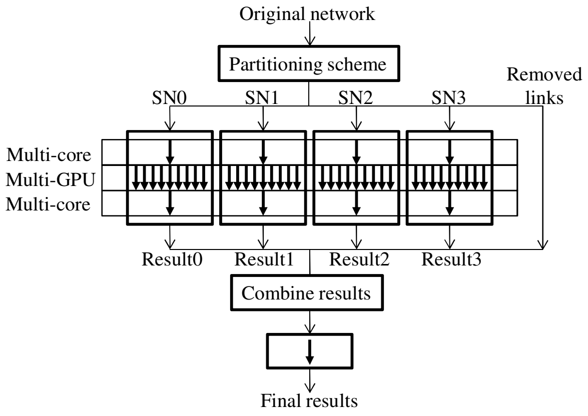


Fig. 4. Illustration of multi-core with multi-GPU implementation

6 Evaluation

The performance of the hierarchical parallel algorithm for community detection proposed in this paper is evaluated using the networks in Table 1. The considered networks include large web-based networks and popular social networks. The results reported in this section were obtained on a server with Intel Xeon E5405 2 GHz processor and four NVIDIA Fermi C2070 GPUs. All executables have been obtained using the '-O3' compiler optimization option. Due to memory limitations of the server, the original Louvain method was not able to process the larger web-based networks, namely uk-2005 and webbase-2001, as a whole. As such, a sub-network of the original network was generated using random sampling.

Table 1. Details of the networks used in the evaluation

Network	Description	# Vertices (M)	# Links (M)	Degree	Reference
uk-2005	Web network (.uk domain)	15.99	287.20	17.96	[9]
webbase-2001	Web network	19.97	138.07	6.91	[10]
twitter	Twitter user follow links	3.26	13.13	4.03	Twitter API
flickr	Flickr follow links	2.30	33.14	14.39	[8]
livejournal	LiveJournal social network	5.20	76.94	14.79	[7]
orkut	Orkut social network	3.07	223.53	72.75	[7]
facebook	Facebook social network	2.94	41.92	14.26	[6]

6.1 GPU Thread Configuration

The performance of the GPU is generally susceptible to its thread configuration. This section seeks to find the optimal thread configuration for the single-GPU implementation of the hierarchical parallel algorithm. This means that only the second and third levels of parallelism are considered. The computation times for the different thread configurations obtained on the uk-2005 network are shown in Table 2.

Table 2. Computation times (in seconds) for different GPU thread configurations on uk-2005

# Blocks per SM	2		4		6		8	
# Threads per block	# Threads per node		# Threads per node		# Threads per node		# Threads per node	
	16	32	16	32	16	32	16	32
256	181.59	243.99	134.71	182.32	122.22	146.72	118.4	132.66
512	133.8	183.02	118.02	134.07	110.89	122.72	107.17	119.02
768	123.6	146.9	112.02	124.37	108.06	118.41	**106.94**	113.62

Table 3. Computation times (in seconds) for uk-2005 using different GPU thread configurations but fixing the number of nodes that is processed in parallel per SM

# Nodes processed in parallel per SM	# Threads per node				Configuration	
	4	8	16	32	# Blocks per SM	# Nodes per block
32	188.82	192.74	179.81	182.32	4	8
32	188.54	186.14	181.59	183.02	2	16
48	150.76	146.96	143.84	146.72	6	8
48	151.34	148.28	146.43	146.9	2	24
64	137.32	134.69	131.86	132.66	8	8
64	137.57	135.47	134.71	134.07	4	16
96	125.76	123.89	122.22	122.72	6	16
96	124.42	124.03	121.73	124.37	4	24

In Table 2, by fixing the number of blocks per SM and the number of threads in each block and then assigning different number of threads to each node, the degrees of parallelism in the second and third levels can be controlled. Assigning a larger number of threads to each node would increase the degree of parallelism in the third level since there are more threads to compute the gain in modularity of inserting the node into its neighboring communities. However, this would lead to a corresponding decrease in the degree of parallelism in the second level with less nodes being able to be processed in parallel. It is clear from Table 2 that assigning 16 threads to each node has a performance advantage over assigning 32 threads to each node. This result is likely due to the fact that the uk-2005 network has a degree of 17.96 and some of the 32 threads assigned to each node would be idling.

To study how the degrees of parallelism in the second and third levels affect computational performance, another set of experiments, whose results are tabulated in Table 3, is performed. In these experiments, the number of nodes that are processed in parallel per SM, which controls the degree of parallelism in the second level, is fixed at 32, 48, 64, and 96. The number of threads assigned to each node, which determines the degree of parallelism in the third level, is set at 4, 8, 16, and 32. It can be seen in Table 3 that by fixing the number of nodes that are processed in parallel per SM, the number of threads assigned to each node has a small, but not negligible, effect on computation times. In all settings, assigning 16 threads to each node resulted in the lowest computation times. The results in Table 3 also show that the GPU configuration, i.e. the number of blocks per SM and the number of nodes per block, has very little effect on computational performance as well. The main parameter that affects performance is the number of nodes that are processed in parallel per SM. The larger the number of nodes processed in parallel per SM, the better the performance.

The best configuration is highlighted in bold in Table 2 and is used to obtain the rest of the results in this paper.

6.2 Comparison of Computation Times

This section assesses the performance, in terms of computation time, of the parallelized Louvain method proposed in this paper. The computation times for three versions of the Louvain method are compared in Table 4. In Table 4, Louvain is the original Louvain method [4], SingleGPU utilizes the second and third levels of parallelism, and MultiGPU considers the highest level of parallelism as well by splitting the original network into four sub-networks. The amount of computational speedups SingleGPU has over Louvain and MultiGPU has over both Louvain and SingleGPU are also shown in the table.

The results in Table 4 show that SingleGPU offers varying degrees (3x on average) of speedup over Louvain. The highest speedups are achieved on the three largest graphs, i.e. uk-2005, webbase-2001, and orkut. On average, MultiGPU offers another 2x speedup over SingleGPU. MultiGPU did not perform as well on facebook, only achieving a speedup of 1.27x over SingleGPU. A detailed examination revealed that MultiGPU suffered a load balancing problem. For facebook, although the nodes in the network were divided evenly between the sub-networks, one of the sub-networks had a disproportionate number of links, resulting in an uneven distribution of the computational load.

Table 4. Computation times (in seconds) for three versions of the Louvain method

Network	Louvain	SingleGPU		MultiGPU		
		Time	Speedup over Louvain	Time	Speedup over Louvain	Speedup over SingleGPU
uk-2005	497.12	109.94	4.52	56.15	8.85	1.96
webbase-2001	419.61	105.82	3.97	52.46	8.00	2.02
twitter	130.54	73.03	1.79	20.97	6.23	3.48
flickr	113.67	66.04	1.72	27.36	4.15	2.41
livejournal	273.1	145.72	1.87	83.84	3.26	1.74
orkut	1683.3	338.13	4.98	100.45	16.76	3.37
facebook	165.76	51.55	3.22	40.55	4.09	1.27

6.3 Comparison of Modularity Values

While the hierarchical parallel algorithm proposed in this paper offers considerable speedups over the original Louvain method, two sources of inaccuracy have been introduced in the parallel versions. In SingleGPU, the second level of parallelism assumes that the computations for visiting any two nodes in a batch are independent, which may not be the case since one of the nodes may be in the neighboring community of the other node. In addition, inaccuracy is introduced when MultiGPU partitions a network into sub-networks, solves the community detection problem in each sub-network independently, and then combines the results. To study the extent to which the inaccuracies affect community detection results, a comparison of the modularity values obtained by the three versions of the Louvain method is provided in Table 5. The best result for each network is highlighted in bold. The percentage differences in results between the two proposed versions and Louvain are also given.

Table 5. Modularity values for three versions of the Louvain method

Network	Louvain	SingleGPU		MultiGPU	
		Q	% difference	Q	% difference
uk-2005	**0.998**	**0.998**	0	**0.998**	0
webbase-2001	**0.998**	**0.998**	0	**0.998**	0
twitter	**0.606**	0.598	-1.32	0.583	-3.8
flickr	0.655	**0.665**	1.53	0.641	-2.14
livejournal	0.734	**0.756**	3	0.701	-4.50
orkut	0.661	**0.682**	3.18	0.600	-9.23
facebook	0.716	**0.730**	1.96	0.709	-0.98

From Table 5, it can be observed that with the exception of the twitter network, the modularity values obtained by SingleGPU are equal, if not better, than those obtained by the original Louvain method. It is clear from these empirical results that the inaccuracy introduced by assuming that the computations for visiting any two nodes in a batch are independent does not have a negative impact on the modularity values.

However, from the modularity values of the solutions obtained by MultiGPU, it can be observed that the inaccuracy introduced by solving the sub-networks rather than solving the original network as a whole has caused an average of 3% degradation in the modularity values. Given the speedups it offers, MultiGPU would still be useful in providing a quick and approximate community detection solution.

7 Conclusions

This paper represents the first attempt to accelerate the Louvain method, or modularity-based methods in general, on the GPU platform. Benchmarking results on several large web-based networks and popular social networks show that on top of offering speedups, the single-GPU version of the proposed hierarchical parallel algorithm is able to find better quality communities. The multi-GPU version provides additional speedups over the single-GPU version but with a slight degradation in community quality.

A future work would be to design a more effective method to partition the network into sub-networks at the highest level of parallelism. The method should address two problems with the current design. Firstly, the computational load for each sub-network needs to be more balanced. Secondly, the degradation in community quality needs to be minimized. The former could be tackled by partitioning the network such that the sub-networks have approximately equal number of links. The latter is more challenging as it requires the network partitioning problem to be treated as a community detection problem so that nodes that would eventually be in the same community are placed in the same sub-network. A bigger challenge lies in integrating the two solutions as their objectives are potentially conflicting in nature.

References

1. Fortunato, S.: Community Detection in Graphs. Physics Reports (2009)
2. Flake, G.W., Lawrence, S., Giles, C.L., Coetzee, F.M.: Self-Organization and Identification of Web Communities. IEEE Computer 35(3), 66–71 (2002)
3. Green, D.M., Werkman, M., Munro, L.A., Kao, R.R., Kiss, I.Z., Danon, L.: Tools to Study Trends in Community Structure: Application to Fish and Livestock Trading Networks. Preventive Veterinary Medicine 99, 225–228 (2011)
4. Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E.: Fast Unfolding of Communities in Large Networks. Journal of Statistical Mechanics: Theory and Experiment, P10008 (2008)
5. NVIDIA Kepler GK110 Architecture Whitepaper, `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`
6. Wilson, C., Boe, B., Sala, A., Puttaswamy, K.P.N., Zhao, B.Y.: User Interactions in Social Networks and Their Implications. In: 2009 EuroSys Conference (2009)
7. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and Analysis of Online Social Networks. In: 5th ACM/Usenix Internet Measurement Conference, IMC (2007)

8. Cha, M., Mislove, A., Gummadi, K.P.: A Measurement-Driven Analysis of Information Propagation in the Flickr Social Network. In: 18th International World Wide Web Conference (2009)
9. Laboratory for Web Algorithmics, `http://law.dsi.unimi.it/`
10. Stanford WebBase Project, `http://dbpubs.stanford.edu:8091/~testbed/doc2/WebBase/`
11. Girvan, M., Newman, M.E.J.: Community Structure in Social and Biological Networks. National Academy of Sciences 99(12), 7821–7826 (2002)
12. Newman, M.E.J., Girvan, M.: Finding and Evaluating Community Structure in Networks. Physical Review E 69(2) (2004)
13. Zhang, Y., Wang, J., Wang, Y., Zhou, L.: Parallel Community Detection on Large Networks with Propinquity Dynamics. In: 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 997–1006 (2009)
14. Eriksen, K.A., Simonsen, I., Maslov, S., Sneppen, K.: Modularity and Extreme Edges of the Internet. Physical Review Letters 90(14) (2003)
15. Pujol, J.M., Erramilli, V., Rodriguez, P.: Divide and Conquer: Partitioning Online Social Networks (2009), `http://arxiv.org/abs/0905.4918v1`
16. Haynes, J., Perisic, I.: Mapping Search Relevance to Social Networks. In: 3rd Workshop on Social Network Mining and Analysis (2010)
17. Hui, P., Sastry, N.: Real World Routing Using Virtual World Information. In: International Conference on Computational Science and Engineering, vol. 4, pp. 1103–1108 (2009)
18. Soman, J., Narang, A.: Fast Community Detection Algorithm with GPUs and Multicore Architectures. In: IEEE International Parallel & Distributed Processing Symposium, pp. 568–579 (2011)

GWAS on GPUs: Streaming Data from HDD for Sustained Performance

Lucas Beyer and Paolo Bientinesi

RWTH Aachen University,
Aachen Institute for advanced study in Computational Engineering Science, Germany
{beyer,pauldj}@aices.rwth-aachen.de

Abstract. In the context of genome-wide association studies (GWAS), one has to solve long sequences of generalized least-squares problems; such a task has two limiting factors: execution time –often in the range of days or weeks– and data management –data sets in the order of Terabytes. We present an algorithm that obviates both issues. By pipelining the computation, and thanks to a sophisticated transfer mechanism, we stream data from hard disk to main memory to GPUs and achieve sustained performance; with respect to a highly-optimized CPU implementation, our algorithm shows a speedup of 2.6x. Moreover, the approach lends itself to multiple GPUs and attains almost perfect scalability. When using 4 GPUs, we observe speedups of 9x over the aforementioned CPU implementation, and 488x over ProbABEL, a widespread biology library.

Keywords: GWAS, generalized least-squares, computational biology, out-of-core computation, high-performance, multiple GPUs, data transfer, multibuffering, streaming, big data.

1 GWAS, Their Importance and Current Implementations

In a nutshell, the goal of a genome-wide association study (GWAS) is to find an association between genetic variants and a specific trait such as a disease [1]. Since there is a tremendous amount of such genetic variants, the computation involved in GWAS takes a long time, ranging from days to weeks and even months [2]. In this paper, we look at *OOC-HP-GWAS*, currently the fastest algorithm available, and show how it is possible to speed it up by exploiting the computational power offered by modern graphics accelerators.

The solution of GWAS boils down to a sequence of generalized least squares (GLS) problems involving huge amounts of data, in the order of Terabytes. The challenge lies in sustaining GPU's performance, avoiding idle time due to data transfers from hard disk (HDD) and main memory. Our solution, *cuGWAS*, combines three ideas: the computation is pipelined through GPU and CPU, the transfers are executed asynchronously, and the data is streamed from HDD to main memory to GPUs by means of a two-level buffering strategy. Combined, these mechanisms allow cuGWAS to attain almost perfect scalability with respect to the number of GPUs; when compared to OOC-HP-GWAS and another widespread GWAS library, ProbABEL, our code is respectively 9 and 488 times faster.

In the first section of this paper, we introduce the reader to GWAS and the computations involved therein. We then give an overview of OOC-HP-GWAS, upon which we build cuGWAS, whose key techniques we explain in Section 3 and which we time in Section 4. We provide some closing remarks in Section 5.

1.1 Biological Introduction to GWAS

The segments of the DNA that contain information about protein synthesis are called *genes*. They encode so-called *traits*, which are features of physical appearance of the organism –like eye or hair color– as well as internal features of the organism –like blood type or resistances to diseases. The hereditary information of a species consists of all the genes in the DNA, and is called *genome*; this can be visualized as a book containing instructions for our body. Following this analogy, the letters in this book are called *nucleotides*, and determining their order is referred to as *sequencing* the genome. Even though the genome sequence of every individual is different, within one species most of it (99.9% for humans) stays the same. When a single nucleotide of the DNA differs between two individuals of the same species, this difference is called a single-nucleotide polymorphism (*SNP*, pronounced "snip") and the two variants of the SNP are referred to as its *alleles*.

Genome-wide association studies compare the DNA of two groups of individuals. All the individuals in the *case group* have a same trait, for example a specific disease, while all the individuals in the *control group* do not have this trait. The SNPs of the individuals in these groups are compared; if one variant of a SNP is more frequent in the case group than in the control group, it is said that the SNP is *associated* with the trait (disease). In contrast with other methods for linking traits to SNPs, such as inheritance studies or genetic association studies, GWAS consider the whole genome [1].

1.2 The Importance of GWAS

We gathered insightful statistics about all published GWAS [3]. Since the first GWAS started to appear in 2005 and 2006, the amount of yearly published studies has constantly increased, reaching more than 2300 studies in 2011. This trend is summarized in the left panel of Fig. 1, showing the median SNP-count of each year's studies along with error-bars for the first and second quartiles. One can observe that while GWA studies started out relatively small, since 2009 the amount of analyzed SNPs is growing tremendously. Besides the number of SNPs, the other parameter relevant to the implementation of an algorithm is the *sample size*, that is the total number of individuals of both the case and the control group. What can be seen in Fig. 1b is that while it has grown at first, in the past four years the median sample size seems to have settled around

10 000 individuals. It is apparent that, in contrast to the SNP count, the growth of the sample size is negligible. This data, as well as discussions with biologists, confirm the need for algorithms and software that can compute a GWAS with even more SNPs, and faster than currently possible.

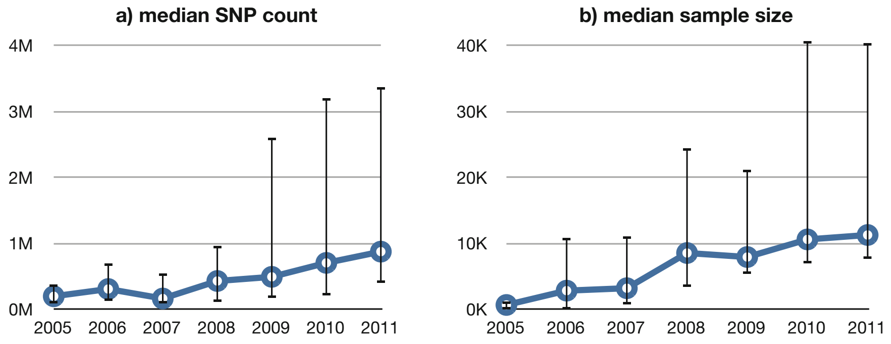


Fig. 1. The median, first and second quartile of a) the SNP-count and b) the sample size of the studies each year

1.3 The Mathematics of GWAS

The GWAS can be expressed as a variance component model [4] whose solution r_i can be formulated as

$$r_i = (X_i^T M^{-1} X_i)^{-1} X_i^T M^{-1} y, \quad i = 1 \ldots m \ , \tag{1}$$

where m is in the millions, and all variables on the right-hand side are known. This sequence of equations is used to compute in r_i the relations between variations in y (the *phenotype*[1]) and variations in X_i (the *genotype*). Each equation is responsible for one SNP, meaning that the number m of equations corresponds to the number of SNPs considered in the study.

Figure 2 captures the dimensions of the objects involved in one such equation. The height n of the matrices X_i and M and of the vector y corresponds to the number of samples, thus each row in the *design-matrix* $X_i \in \mathbb{R}^{n \times p}$ corresponds to a piece of each individual's genetic makeup (i.e. information about one SNP), and each entry in $y \in \mathbb{R}^n$ corresponds to an individual's phenotype.[2] $M \in \mathbb{R}^{n \times n}$ models the relations amongst the individuals, e.g. two individuals being in the same family. Finally, an important feature of the matrices X_i is that they can be partitioned as $(X_L | X_{R_i})$, where X_L contains fixed covariates such as age and sex

[1] A phenotype is the observed value of a certain trait of an individual. For example, if the studied trait was the hair color, the phenotype of an individual would be the one of "blonde", "brown", "black" or "red".

[2] In the example of the body height as a trait, the entries of y would then be the heights of the individuals.

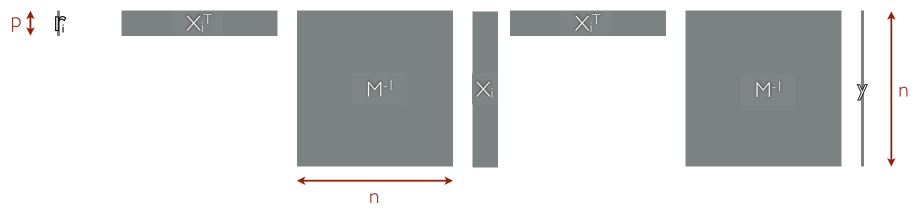


Fig. 2. The dimensions of a single instance of (1)

and thus stays the same for any i, while X_{R_i} is a single column vector containing the genotypes of the i-th SNP of all considered individuals.

Even though (1) has to be computed for every single SNP, only the right part of the design-matrix X_{R_i} changes, while X_L, M and y stay the same.

1.4 The Amount of Data and Computation Involved

We analyze the storage size requirements for the data involved in GWAS. Typical values for p range between 4 and 20, but only one column varies with m. According to our analysis in Section 1.2, we consider $n = 10\,000$ as the size of a study. As of June 2012, the SNP database *dbSNP* lists 187 852 828 known SNPs for humans [5], so we consider $m = 190\,000\,000$. With these numbers, assuming that all data is stored as double precision floating point numbers, [3] the size of y and M is about 80 MB and 800 MB, respectively; both fit in main memory and in the GPU memory. The output r reaches 30 GB, close to the main memory of current high-end systems and too big to fit in a GPU's 6 GB of memory. Weighting in at 14 TB, X is too big to fit into the memory of any system in the foreseeable future and has to be streamed from disk.

In the field of bioinformatics, the ProbABEL [6] library is frequently used for genome-wide association studies. On a Sun Fire X4640 server with an Intel Xeon CPU 5160 (3.00 GHz), the authors report a runtime of almost 4 hours for a problem with $p = 4$, $n = 1500$ and $m = 220\,833$, and estimate the runtime with $m = 2\,500\,000$ to be roughly 43 hours[4] –almost two days. Compared to the current demand, $m = 2.5$ million is a reasonable amount of SNPs, but a population size of only $n = 1500$ individuals is clearly much smaller than the present median (Fig. 1). The authors state that the runtime grows more than linearly with n and, in fact, tripling up the sample size from 500 to 1500 increased their runtime by a factor of 14. Coupling this fact with the median sample size of about 10 000 individuals, the computation time is bound to reach weeks or even months.

[3] Which may or may not be the optimal storage type. More discussion with biologists and analysis of the operations is necessary in order to find out whether `float` is precise enough. If that was the case, the sizes should be halved.

[4] We only consider what the authors called the *linear model* with the `--mmscore` option as this solves the exact problem we tackle.

2 Prior Work: The OOC-HP-GWAS Algorithm

Presently, the fastest available algorithm for solving (1) is OOC-HP-GWAS [4]. Since our work builds upon this CPU-only algorithm, we describe its salient features. Other approaches to GWAS on GPU(s) include [10] and [11].

2.1 Algorithmic Features

OOC-HP-GWAS exploits the the symmetry and the positive definiteness of the matrix M, by decomposing it through a Cholesky factorization $LL^T = M$. Since M does not depend on i, this decomposition can be computed once as a preprocessing step and reused for every instance of (1). Substituting $LL^T = M$ into (1) and rearranging, we obtain

$$r_i = (\underbrace{(L^{-1}X_i)^T}_{\tilde{X}_i} \underbrace{L^{-1}X_i}_{\tilde{X}_i})^{-1} \underbrace{(L^{-1}X_i)^T}_{\tilde{X}_i} \underbrace{L^{-1}y}_{\tilde{y}} \quad \text{for } i = 1 \ldots m \ , \tag{2}$$

effectively replacing the inversion and multiplication of M with the solution of a triangular linear system (`trsv`).

The second problem-specific piece of knowledge that is exploited by OOC-HP-GWAS is the structure of $X = (X_L|X_R)$: X_L stays constant for any i, while X_R varies; plugging $X_i = (X_L|X_{R_i})$ into (2) and moving the constant parts out of the loop leads to an algorithm that takes advantage of the structure of the sequence of GLS shown in Listing 1.1. The acronyms correspond to BLAS calls. A more detailed derivation can be found in [4].

Listing 1.1. Solution of the GWAS-specific sequence of GLS (1)

```
L   ← potrf M                      ($LL^T = M$)
Xl  ← trsm L, Xl                   ($\tilde{X}_L = L^{-1}X_L$)
y   ← trsv L, y                    ($\tilde{y} = L^{-1}y$)
rt  ← gemv Xl, y                   ($\tilde{r}_T = \tilde{X}_L^T\tilde{y}$)
Stl ← syrk Xl                      ($S_{TL} = \tilde{X}_L^T\tilde{X}_L$)
for i in 1..m:
    Xri ← trsv L, Xri              ($\tilde{X}_{R_i} = L^{-1}X_{R_i}$)
    Sbl ← dot Xri, Xl              ($S_{BL_i} = \tilde{X}_{R_i}^T\tilde{X}_L$)
    Sbr ← syrk Xri                 ($S_{BR_i} = \tilde{X}_{R_i}^T\tilde{X}_{R_i}$)
    rb  ← dot Xri, y               ($\tilde{r}_{B_i} = \tilde{X}_{R_i}^T\tilde{y}$)
    r   ← posv S, r                ($r_i = S_i^{-1}\tilde{r}_i$)
```

2.2 Implementation Features

Two implementation features allow OOC-HP-GWAS to attain near-perfect efficiency. First, by packing multiple vectors X_{R_i} into a matrix X_{R_b}, the slow

BLAS-2 routine to solve a triangular linear system (`trsv`) at Line 7 can be transformed into a fast BLAS-3 `trsm`. Second, Listing 1.1 is an *in-core* algorithm that cannot deal with an X_R which does not fit into main memory. This limitation is overcome by turning the algorithm into an *out-of-core* one, in this case using a double-buffering technique: While the CPU is busy computing the block b of X_R in a primary buffer, the next block $b+1$ can already be loaded into a secondary buffer through asynchronous I/O using the POSIX `libaio`. The full OOC-HP-GWAS algorithm is shown in Listing 1.2. This algorithm attains more than 90% efficiency.

Listing 1.2. The full OOC-HP-GWAS algorithm

```
 1  L    ← potrf M                     $(LL^T = M)$
 2  Xl   ← trsm L, Xl                  $(\tilde{X}_L = L^{-1}X_L)$
 3  y    ← trsv L, y                   $(\tilde{y} = L^{-1}y)$
 4  rt   ← gemv Xl, y                  $(\tilde{r}_T = \tilde{X}_L^T\tilde{y})$
 5  Stl ← syrk Xl                      $(S_{TL} = \tilde{X}_L^T\tilde{X}_L)$
 6  aio_read Xr[1]
 7  for b in 1..blockcount:
 8      aio_read Xr[b+1]
 9      aio_wait Xr[b]
10      Xrb ← trsm L, Xrb              $(\tilde{X}_{R_b} = L^{-1}X_{R_b})$
11      for Xri in Xr[b]:
12          Sbl ← gemm Xri, Xl         $(S_{BL_i} = \tilde{X}_{R_i}^T\tilde{X}_L)$
13          Sbr ← syrk Xri             $(S_{BR_i} = \tilde{X}_{R_i}^T\tilde{X}_{R_i})$
14          rb  ← gemv Xri, y          $(\tilde{r}_{B_i} = \tilde{X}_{R_i}^T\tilde{y})$
15          r   ← posv S, r                $(r_i = S_i^{-1}\tilde{r}_i)$
16      aio_wait r[b-1]
17      aio_write r[b]
18  aio_wait r[blockcount]
```

3 Increasing Performance by Using GPUs

While the efficiency of the OOC-HP-GWAS algorithm is satisfactory, the computations can be sped up even more by leveraging multiple GPUs. With the help of a profiler, we determined (confirming the intuition), that the `trsm` at line 10 in Listing 1.2 is the bottleneck. Since cuBLAS provides a high-performance implementation of BLAS-3 routines, `trsm` is the best candidate to be executed on GPUs. In this section, we introduce cuGWAS, an algorithm for a single GPU, and then extend it to an arbitrary number of GPUs.

Before the `trsm` can be executed on a GPU, the algorithm has to transfer the necessary data. Since the size of L is around 800 MB, the matrix can be sent once during the preprocessing step and kept on the GPU throughout the entire computation. Unfortunately, the whole X_R matrix weights in at several TB, way more than the 2 GB per buffer limit of a modern GPU. The same holds true for the result $\tilde{X}_{R_b}$ of the `trsm`, which needs to be sent back to main memory. Thus, there is no other choice than to send X_R in a block-by-block fashion, each block X_{R_b} weighting at most 2 GB minus the size of L.

When profiled, a naïve implementation of the algorithm displays a pattern (Fig. 3) typical for applications in which GPU-offloading is an after-thought: both GPU (green) and CPU (gray) need to wait for the data transfer (orange); furthermore, the CPU is idle while the GPU is busy and vice-versa.

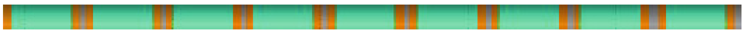


Fig. 3. Profiled timings of the naïve implementation

Our first objective is to make use of the CPU while the GPU computes the `trsm`. Regrettably, all operations following the `trsm` (i.e. the for-loop at Lines 11–15 in Listing 1.2, which we will call the *S-loop*) are dependent on its result and thus cannot be executed in parallel. A way to break out of this dependency is to delay the S-loop by one block, in a pipeline fashion, so that the S-loop relative to the b-th block of X_R is executed on the CPU, while the GPU executes the `trsm` with the $(b+1)$-th block. Thanks to this pipelining, we have broken the dependency and introduced more parallelism, completely removing the gray part of Fig. 3.

3.1 Streaming Data from HDD to GPU

The second problem with the aforementioned naïve implementation is the time wasted due to data transfers. Modern GPUs are capable of *overlapping* data transfers with computation. If properly exploited, this feature allows us to eliminate any overhead, and thus attain sustained peak performance on the GPU.

The major obstacle is that the data is already being double-buffered from the hard-disk to the main memory. A quick analysis shows that when targeting two layers of double-buffering (one layer for disk $\leftrightarrow$ main memory transfers and another layer for main memory $\leftrightarrow$ GPU transfers), two buffers on each layer are not sufficient anymore. The idea here is to have two buffers on the GPU and three buffers on the CPU.

The double-triple buffering can be illustrated from two perspectives: the tasks executed and the buffers involved. The former is presented in Fig. 4; we refer the reader to [7] for a thorough description. Here we only discuss the technique in terms of buffers.

In this single-GPU scenario, the size of the blocks X_{R_b} used in the GPU's computation is equal to that on the CPU. When using multiple GPUs, this will not be the case anymore, as the CPU loads one large block and distributes portions of it to the GPUs.

The GPU's buffers are used in the same way as the CPU's buffers in the simple CPU-only algorithm: While one buffer α is used for the computation, the

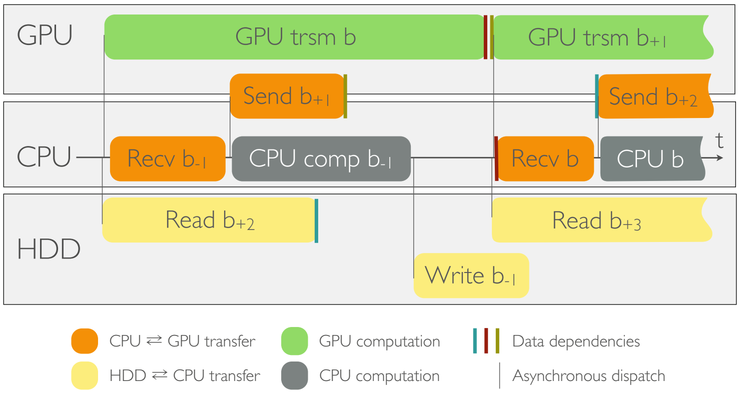


Fig. 4. A task-perspective of the algorithm. Sizes are unrelated to runtime.

data is transferred to and from the other buffer β. But at the CPU's level (i.e. in RAM), three buffers are now necessary. For the sake of simplicity, we avoid the explanation of the initial and final iterations and start with iteration b.

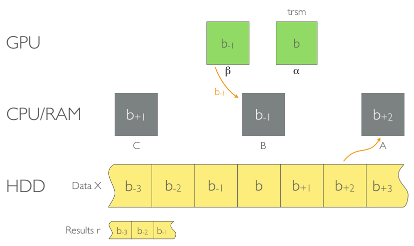
With reference to Fig. 5a, assume that the $(b-1)$-th, b-th and $(b+1)$-th blocks already reside in the GPU buffers β, α, and in the CPU buffer C, respectively. The block $b-1$ (i.e. buffer β) contains the solution of the `trsm` of block $b-1$. At this point, the algorithm proceeds by *dispatching* both the read of the second-next block $b+2$ from disk into buffer A and the computation of the `trsm` on the GPU on buffer α, and by receiving the result from buffer β into buffer B. The first two operations are *dispatched*, i.e. they are executed asynchronously by the memory system and the GPU, while the last one is executed synchronously because these results are needed immediately in the following step.

As soon as the synchronous transfer $\beta \to B$ completes, the transfer of the next block $b+1$ from CPU buffer C to GPU buffer β is *dispatched*, and the S-loop is executed on the CPU for the previous block $b-1$ in buffer B on the CPU (see Fig. 5b).

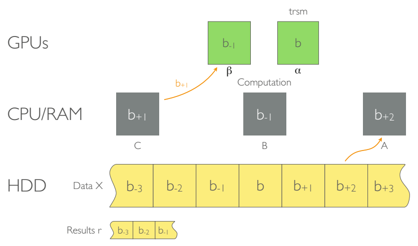
As soon as the CPU is done computing the S-loop, its results are written to disk (Fig. 5c). Finally, once all transfers are done, buffers are rotated (through pointer or index rotations, not copies) according to Fig. 5d, and the loop continues with $b \leftarrow b+1$.

3.2 Using Multiple GPUs

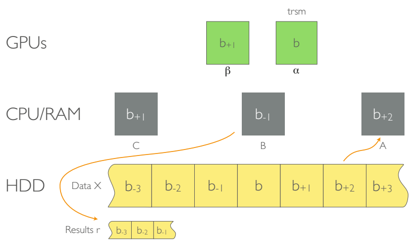
This multi-buffering technique achieves sustained performance on one GPU. Since boards with many GPUs are becoming more and more common in high-performance computing, we explain here how our algorithm is adapted to take advantage of all the available parallelism. The idea is to increase the size of the X_{R_b} blocks by a factor as big as the number of available GPUs, and then split the `trsm` among these GPUs. As long as solving a `trsm` on the GPU takes longer than loading a large enough block X_{R_b} from HDD to CPU, this parallelization



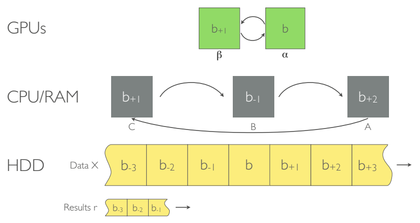
(a) Retrieve the previous result $b-1$ from GPU, and the second-next block $b+2$ of data from disk.



(b) Send the next block $b+1$ from RAM to the GPU, execute the S-loop on $b-1$ on the CPU.



(c) Write the results $b-1$ to disk.



(d) Switch buffers at both levels for the next iteration.

Fig. 5. The multi-buffering algorithm as seen from a buffer perspective

strategy holds up for any number of GPUs. Since in our systems loading the data from HDD was an order of magnitude faster than the computation of the trsm, the algorithm scales up to more GPUs than were available. Listing 1.3 shows the final version of cuGWAS.[5]

Listing 1.3. cuGWAS. The black bullet is a placeholder for "all GPUs".

```
L    ← potrf M                      ($LL^T = M$)
cublas_send L → L_gpu•
Xl   ← trsm L, Xl                   ($\tilde{X}_L = L^{-1}X_L$)
y    ← trsv L, y                    ($\tilde{y} = L^{-1}y$)
rt   ← gemv Xl, y                   ($\tilde{r}_T = \tilde{X}_L^T\tilde{y}$)
Stl  ← syrk Xl                      ($S_{TL} = \tilde{X}_L^T\tilde{X}_L$)
gpubs ← blocksize/ngpus
for b in -1..blockcount+1:
    cu_trsm_wait $\alpha_\bullet$                 (if b in 1..blockcount)
    cu_send_wait C• → $\beta_\bullet$             (if b in 2..blockcount+1)
    $\alpha_\bullet$ ← cu_trsm_async L_gpu•, $\alpha_\bullet$   (if b in 1..blockcount) ↙
                                  ↳ ($\tilde{X}_{R_b} = L^{-1}X_{R_b}$)
    aio_read Xr[b+2] → A           (if b in -1..blockcount-2)
```

[5] The conditions for the first and last pair of iterations are provided in parentheses on the right.

```
13      for gpu in 0..ngpus-1:          (if b in 2..blockcount+1)
14          cu_recv B[gpu*gpubs..(gpu+1)*gpubs] ← $\beta_{gpu}$
15      aio_wait Xr[b+1] → C            (if b in 0..blockcount-1)
16      for gpu in 0..ngpus-1:          (if b in 0..blockcount-1)
17          cu_send_async C[gpu*gpubs..(gpu+1)*gpubs] → $\beta_{gpu}$
18      for Xri in B:                   (if b in 2..blockcount+1)
19          Sbl ← gemm Xri, Xl          ($S_{BL_i} = \tilde{X}_{R_i}^T \tilde{X}_L$)
20          Sbr ← syrk Xri              ($S_{BR_i} = \tilde{X}_{R_i}^T \tilde{X}_{R_i}$)
21          rb  ← gemv Xri, y           ($\tilde{r}_{B_i} = \tilde{X}_{R_i}^T \tilde{y}$)
22          r   ← posv S, r             ($r_i = S_i^{-1} \tilde{r}_i$)
23      aio_wait r[b-2]                 (if b in 1..blockcount+1)
24      aio_write r[b-1]                (if b in 1..blockcount+1)
25      swap_buffers
```

4 Results

In order to show the speedups obtained with a single GPU, we compare the hybrid CPU-GPU algorithm presented in Listing 1.3 using one GPU with the CPU-only OOC-HP-GWAS. Then, to determine the scalability of cuGWAS, we compare its runtimes when leveraging 1, 2, 3 and 4 GPUs.

In all of the timings, the time to initialize the GPU and the preprocessing (Lines 1–7 in Listing 1.3), both in the order of seconds, have not been measured. The GPU usually takes 5 s to fully initialize, and the preprocessing takes a few seconds too, but depends only on n and p. This omission is thus irrelevant for computations that run for hours.

4.1 Single-GPU Results

The experiments with a single-GPU were performed on the *Quadro* cluster at the RWTH Aachen University; the cluster is equipped with two nVidia Quadro 6000 GPUs and two Intel Xeon X5650 CPUs per node. The GPUs, which are powered by Fermi chips, have 6 GB of RAM and a theoretical double-precision computational power of 515 GFlops each. In total, the cluster has a GPU peak of 1.03 TFlops. The CPUs, which have six cores each, amount to a total of 128 GFlops and are supported by 24 GB of RAM. The cost of the combined GPUs is estimated to about \$10 000 while the combined CPUs cost around \$2000.

Figure 6a shows the runtime of OOC-HP-GWAS along with that of cuGWAS, using one GPU. Thanks to our transfer-overlapping strategy, we can leverage the GPU's performance and achieve a 2.6x speedup over a highly-optimized CPU-only implementation. cuBLAS' `trsm` implementation attains about 60 % of the GPU's peak performance, i.e. about 309 GFlops [8]. The peak performance of the CPU in this system amounts to 128 GFlops; if the whole computation were performed on the GPU at `trsm`'s rate, the largest speedup possible would be 2.4. We achieve 2.6 because the computation is pipelined: the S-loop is executed on the CPU, in perfect overlap with the GPU. This means that the performance of cuGWAS is perfectly in line with the theoretical peak.

In addition, the figure indicates that the algorithm (1) has linear runtime in m and (2) allows us to cope with an arbitrary m. The red vertical line in the figure marks the largest value of m for which two blocks of X_R fit into the GPU memory for $n = 10\,000$. Without the presented multi-buffering technique, it would not be possible to compute GWAS with more than $m = 22\,500$ SNPs, while cuGWAS allows the solution of GWAS with any given amount of SNPs.

4.2 Scalability with Multiple GPUs

To experiment with multiple GPUs, we used the *Tesla* cluster at the Universitat Jaume I in Spain, since it is equipped with an nVidia Tesla S2050 which contains four Fermi chips (same model as the *Quadro* system), for a combined GPU compute power of 2.06 TFlops, but with only 3 GB of RAM each. The host CPU is an Intel Xeon E5440 delivering approximately 90 GFlops.

In order to evaluate the scalability of cuGWAS, we solved a GWAS with $p = 4$, $n = 10\,000$, and $m = 100\,000$ on the *Tesla* cluster, varying the number of GPUs. As it can be seen in Fig. 6b, the scalability of the algorithm with respect to the number of GPUs is almost ideal: Doubling the amount of GPUs reduces the runtime by a factor of 1.9.

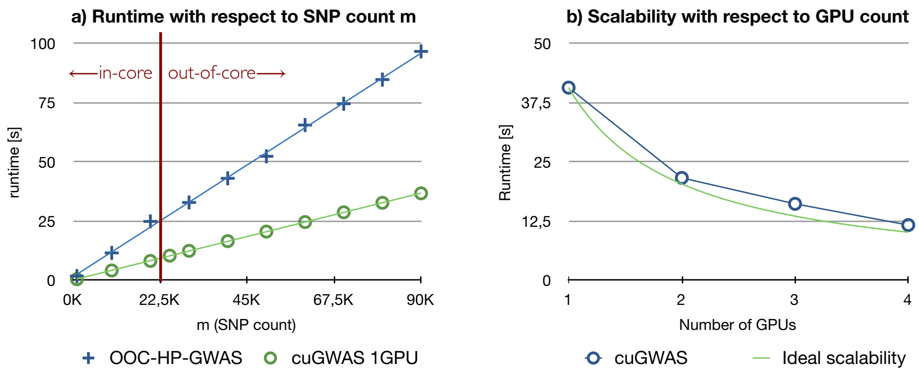


Fig. 6. The runtime of our cuGWAS algorithm a) using one GPU compared to OOC-HP-GWAS (CPU), b) using a varying amount of GPUs

5 Conclusion and Future Work

We have presented a strategy which makes it possible to sustain peak performance on a GPU not only when the data is too big for the GPU's memory, but also for main memory. In addition, we have shown how well this strategy scales to multiple GPUs.

As described by the developers of ProbABEL, the solution of a problem of the size described in Section 1.4 by the GWFGLS algorithm took 4 hours.

In contrast, with cuGWAS we solved the same problem in 2.88 s. Even accounting for about 6 seconds for the initialization and Moore's Law (doubling the runtime as ProbABEL's timings are from 2010), the difference is dramatic. We believe that the contribution of cuGWAS is an important step towards making GWAS practical.

Software. The code implementing the strategy explained in this paper is freely available at `http://github.com/lucasb-eyer/cuGWAS` and `http://lucas-b.eyer.be`.

Acknowledgements. Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged. The authors thank Diego Fabregat-Traver for providing us with the source-code of OOC-HP-GWAS, the Center for Computing and Communication at RWTH Aachen for the resources, Enrique S. Quintana-Ortí for granting us access to the *Tesla* system as well as Yurii S. Aulchenko for introducing us to the computational challenges of GWAS.

References

1. Genome-Wide Association Studies, `http://www.genome.gov/20019523`
2. Fabregat-Traver, D., Bientinesi, P.: Computing Petaflops over Terabytes of Data: The Case of Genome-Wide Association Studies (2012)
3. Catalog of Genome-Wide Association Studies, `http://www.genome.gov/gwastudies`
4. Fabregat-Traver, D., Aulchenko, Y.S., Bientinesi, P.: Solving Sequences of Generalized Least-Squares Problems on Multi-threaded Architectures (2012)
5. `http://www.ncbi.nlm.nih.gov/mailman/pipermail/dbsnp-announce/2012q2/000123.html`
6. Aulchenko, Y.S., Struchalin, M.V., Van Duijn, C.M.: ProbABEL package for genome-wide association analysis of imputed data. BMC Bioinformatics 11, 134 (2010)
7. Beyer, L.: Exploiting Graphics Accelerators for Computational Biology
8. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra
9. Quintana-Ortí, G., Igual, F.D., Marqués, M., Quintana-Ortí, E.S., Van de Geijn, R.A.: A run-time system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures.
10. Yung, L.S., Yang, C., Wan, X., Yu, W.: GBOOST: a GPU-based tool for detecting gene–gene interactions in genome–wide case control studies
11. Lee, S., Kwon, M.-S., Huh, I.-S., Park, T.: CUDA-LR: CUDA-accelerated Logistic Regression Analysis Tool for Gene-Gene Interaction for Genome-Wide Association Study

Topic 15: GPU and Accelerator Computing
(Introduction)

Naoya Maruyama, Leif Kobbelt, Pavan Balaji, Nikola Puzovic, Samuel Thibault, and Kun Zhou

Topic Committee

Computational accelerators such as GPUs, FPGAs and many-core accelerators can dramatically improve the performance of computing systems and catalyze highly demanding applications. Many scientific and commercial applications are beginning to integrate computational accelerators in their code. However, programming accelerators for high performance remains a challenge, resulting from the restricted architectural features of accelerators compared to general purpose CPUs. Moreover, software must conjointly use conventional CPUs with accelerators to support legacy code and benefit from general purpose operating system services. The objective of this topic is to provide a forum for exchanging new ideas and findings in the domain of accelerator-based computing.

This year, seven papers have been accepted for publication in the GPU and accelerator computing track. Besides the important theme of scalable parallelization of applications on accelerator-based systems, the papers in the track explore power/performance optimizations, power profiling, and novel use of accelerators in scientific applications.

Villa, Fatica, Gawande and Tumeo presents a study of power and performance trade-offs in linear algebra solvers using GPUs in "Power/Performance Trade-offs of Small Batched LU Based Solvers on GPUs." Lang and Rünger introduce a new method for profiling power consumption of GPUs at high resolution using low-resolution measurement in "High-Resolution Power Profiling of GPU Functions Using Low-Resolution Measurement." Novalbos, Gonzalez, Otaduy, Lopez-Medrano and Sanchez discuss performance of molecular dynamics code running on multi-GPU accelerator boards in "On-board Multi-GPU Molecular Dynamics." Optimizing convolutions both with GPUs and CPU SSE units is discussed in "Optimizing 3D Convolutions for Wavelet Transforms on CPUs with SSE Units and GPUs" by Videau, Marangozova-Martin, Genovese and Thierry Deutsch. In addition to conventional scientific applications, two papers discuss applications GPUs in data analysis problems. Adinetz, Kraus, Meinke and Pleiter present data clustering using GPUs in "GPUMAFIA: Efficient Subspace Clustering with MAFIA on GPUs." Deveci, Kaya, Ucar and Catalyurek present an acceleration method for graph processing in "GPU Accelerated Maximum Cardinality Matching Algorithms for Bipartite Graphs." Finally, Marques, Paulino, Alexandre and Medeiros discuss a new skeleton-based programming model for GPUs in "Algorithmic Skeleton Framework for the Orchestration of GPU Computations."

We wish to thank all authors who submitted a paper to this topic and all external reviews for delivering quality reviews on time.

High-Resolution Power Profiling of GPU Functions Using Low-Resolution Measurement

Jens Lang and Gudula Rünger

Department of Computer Science, Chemnitz University of Technology, Germany
{jens.lang,gudula.ruenger}@cs.tu-chemnitz.de

Abstract In order to be able to minimise the energy consumption of an application program, information about the specific energy consumption is required. Modern Nvidia graphics processing units (GPUs) measure their current power consumption and the driver makes the value available to the application every 20 ms. However, for evaluating the energy consumption of GPU kernel functions, such a sampling interval might not be sufficient since the kernels may have a shorter execution time.

This article proposes a method for generating high-resolution power profiles, which is the power consumption of a specific function depending on the progress of its execution. The method uses low-resolution measuring instruments offered by GPUs. Power measurements obtained during several executions of the function are combined into a single power profile. The resulting power profile contains power values in intervals which are much shorter than the sampling interval of the hardware driver so that even short-term power changes can be considered, e.g. for calculating the energy consumption of a single function. The article also shows how to extend the approach to an online generation of power profiles. Furthermore, an overview on the power profiles of some important functions, such as BLAS routines, is given.

Keywords: power profiles, power measurement, GPU computing.

1 Introduction

Energy efficiency of codes will become more and more important in scientific computing. Especially general-purpose graphics processing units (GPUs) promise to be a potentially more energy-efficient alternative to CPUs [3,9,21]. Measuring the energy consumption of a hardware is necessary in order to be able to minimise the energy consumption of an application. For quantifying the energy consumption of a piece of code, basically two approaches exist: hardware-based and software-based energy measurement. For hardware-based measurement of the energy consumption, a measuring device is installed between the power supply and the computational device. From the current and the voltage measured, the electrical power P can be computed. By integrating the power over a time interval, the electrical energy consumption E is determined. The observation of influences evoked by short sections of code, requires a device with a high sampling frequency, e.g. 500 Hz [21], 1 kHz [12] or even 50 kHz [7]. For the software-based measurement, modern CPUs, such as Intel's Sandy Bridge or AMD's Bulldozer CPUs, offer an interface for retrieving the value of the energy consumption since some starting time [11, Vol. 3B, Chap. 14.7], [4, Chap. 3.13]. Similarly, the latest GPUs of the manufacturer Nvidia provide the possibility to read the current power consumption by software [19].

The Nvidia Management Library (NVML) provides power measurement [19] for Nvidia GPUs. The power value is updated every 20 ms; this is described in Sect. 3.2. However, such a sampling interval is not sufficient for a precise evaluation of the energy consumption of a GPU kernel function, especially for those with a short execution time. Such kernel functions occur in many areas, such as the solution of linear systems of equations in FEM simulations or Monte Carlo simulations in the field of numerical mathematics as well as, for example, scan algorithms in the field of data processing. Therefore, a method is needed which obtains an accurate, high-resolution power-consumption profile of a GPU function. Furthermore, the availability of high-resolution power profiles are needed to extend approaches which use RAPL to verify energy models, such as [20], to GPUs.

The contribution of this article is to provide such a method for generating high-resolution power profiles of GPU kernel functions, which is based on the measurement interface with a low sampling frequency. These profiles can be used for evaluating and improving the energy consumption of GPU kernel functions. Both, an offline method and an online method for generating the power profiles, are presented. The online method has a low overhead so that it can be used during the actual execution of, e.g., simulation programs which select code paths depending on the energy consumption of specific routines.

The rest of this article is organised as follows: Section 2 presents related work. Section 3 gives an overview on measuring the power consumption on Nvidia GPUs. Section 4 presents the method for generating power profiles from low-resolution measurements, and Sect. 5 gives some sample profiles. The method is extended to an online method in Sect. 6, and Sect. 7 concludes the article.

2 Related Work

Measuring and evaluating the energy consumption of GPUs has been in the focus of research for several years. For example, Huang et al. [9] compare the energy consumption of a biological code for creating an electrostatic potential map on a CPU and on a GPU. Rofouei et al. [21] investigate the energy consumption for some applications from the CUDA SDK executed on a GPU and on a CPU. Both report that the GPU executes the respective codes more efficiently concerning the energy. In contrast, Chen and Singh [5] find no huge difference in the energy consumption for the CPU and the GPU for a document filtering code. Abe et al. [3] measure the effects of frequency and voltage scaling in CPUs and GPUs.

Nagasaka et al. [16] and Chen et al. [6] propose statistical power consumption models for GPUs: Both models execute a set of benchmark functions, measure their power consumption, and record some hardware performance counters of the

GPU. They derive a model which allows the prediction of the power consumption of a kernel function by using the values of the hardware performance counters obtained during its execution. Nagasaki et al. set up this model using linear regression, Chen et al. use a tree-based method.

Collange et al. [7] analyse the influence of computations and memory accesses on the power consumptions of different GPUs. Hong et al. [8] develop an empirical model for predicting the power consumption of a GPU by counting specifics of the code executed, such as the number of memory accesses, the number of streaming multiprocessors used or the use of the computational units. Li et al. [14] extend this model by a possibility to also take streaming multiprocessors executing different workload into account. A similar model is proposed by Kasichayanula et al. [13]: They measure the energy consumption of GPU components, such as global memory, shared memory, floating point unit, etc., and multiply it by the time a given function activates the respective component. Furthermore, they investigate some of the details of the software interface for reading the power consumption of Nvidia GPUs. For the sampling frequency f of the Tesla C2075, they report a value of 62.5 Hz, while the measurements as presented in Sect. 3.2 of this article, resulted in a value of $f = 50$ Hz, possibly due to a different methodology. Weaver et al. from the same research group report a sampling frequency of “roughly 60 Hz” [22].

Hähnel et al. deal with measuring the energy consumption of functions that are shorter than the measuring interval [10]. They measure the energy consumption of the CPU using the Intel RAPL machine-specific registers [11]. They overcome the issue that large sampling intervals might lead to inaccurate results by starting the function exactly at the beginning of a sampling interval of 1 ms and executing a workload with a well-known energy consumption after its return until the end of the sampling interval. Compared to the Nvidia NVML interface, the Intel RAPL has the advantage that the user can retrieve the energy consumption directly and thus does not need to integrate power values. Furthermore, there is no need for clock synchronisation when only measuring on the CPU.

All statistical and empirical models described above may deliver a sufficient temporal resolution for also analysing the energy consumption of short kernel functions. But if such models are used for cases their designers were not aware of, the results might become inaccurate: McCullough et al. [15] indicate that power consumption predictions based on hardware performance counters are inaccurate in complex situations. Therefore, measuring the real power consumption is essential. For all users that do not have access to dedicated measurement hardware, the software-based method proposed in this article might be suitable.

3 Power Consumption of GPUs

Modern GPUs suitable for general-purpose computing implement dynamic frequency scaling in order to save energy in idle mode and to comply with their specified thermal design powers during phases of energy-demanding computations [1,2]. For Nvidia’s GPUs of the latest generation, the Nvidia Management

Library (NVML) [19] offers a software interface for reading the current electrical power consumption.

3.1 Retrieving the Power Consumption

Figure 1 shows the measuring of power consumptions on Nvidia GPUs: There is a specific timer which is triggered in intervals of size T, called the *sampling interval* in the following. At the beginning of each sampling interval, the GPU driver reads the current power consumption P and stores the value. The figure shows the power consumption for the case that the GPU has a power consumption of 70 W in idle mode and of 130 W when executing the user-specified kernel function. However, if the user reads the power value from the GPU, he or she gets data as shown in the line "claimed power consumption", which does not reflect the real power consumption of the GPU.

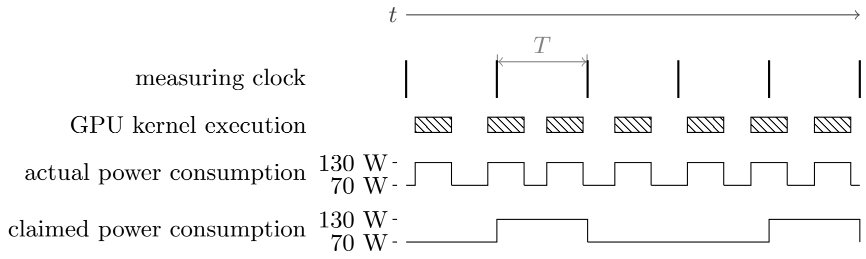


Fig. 1. Repeated execution of a GPU kernel function and measuring its power consumption

The software interface for retrieving the power value is provided by the routine `nvmlDeviceGetPowerUsage(nvmlDevice_t device, unsigned int* power)` of the NVML. The value is measured in milliwatts. It reflects the current electrical power of the whole GPU board including memory, etc. and has an error of ± 5 W [19]. All measurements of this article were conducted on a Nvidia Tesla C2075 GPU [17] built into a machine with two octacore Sandy-Bridge CPUs E5-2650 having a clock speed of 2.0 GHz. The machine runs the Linux kernel version 3.2 with the Nvidia GPU driver version 304.64 and CUDA version 4.2.9.

3.2 Measuring the Sampling Interval *T*

The experiment for measuring the sampling interval T of the GPU makes use of the fact that the power value retrieved is very noisy so that two subsequent measurements normally differ even if the state of the GPU has not changed: The measurement algorithm continually retrieves the current power value P_{curr} from the GPU. If P_{curr} differs from the power value P_{last} retrieved before, it can be inferred that the hardware has updated its power value. Then, the current

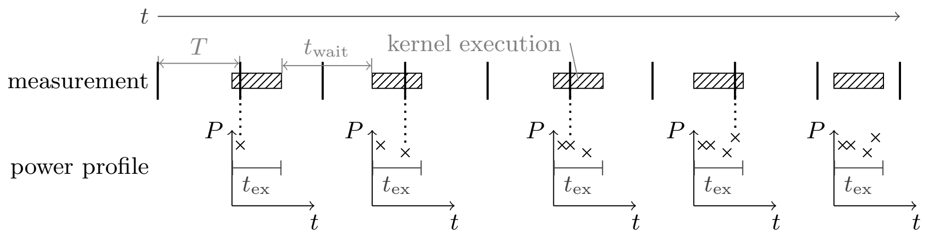


Fig. 2. Illustration of the creation of a power profile from power consumption values measured during repeated GPU kernel execution

time is taken and the time interval elapsed since the last update is emitted. This procedure is repeated for 100 000 times so that one gets 100 000 values for the sampling interval T. In rare cases (0.5 ‰), the power value P_{curr} did not change in two consecutive periods which resulted in a value two or three times as big as the other values. These values have been eliminated. T is then calculated as the arithmetic mean of the remaining values.

For measuring the times, the POSIX function `clock_gettime` is used, which measures the wall-clock time with an accuracy of 1 ns. A value of $T = (20.0082 \pm 0.0008)$ ms, i.e. a relative error of roughly 0.03 ‰, has been determined. The value for the sampling interval of $T \approx 20$ ms results in a sampling frequency of $f = \frac{1}{T} \approx 50$ Hz.

4 The Generation of Power Profiles

For the generation of high-resolution power profiles, i.e. a diagram which shows the electrical power consumption of the GPU during the execution of one kernel function, a statistical method is used to overcome the restrictions imposed by the low sampling frequency of the measurement instrument: The GPU kernel function to be evaluated is executed a large number of times in the range of some hundreds, each time starting at a different phase in the sampling interval. The method is illustrated in Fig. 2: Multiple executions of the kernel function are performed with a random waiting time t_{wait} between them. After starting the execution, the power value is monitored constantly and for each update, this value is emitted together with the corresponding update time relative to the starting time of the function. The two values are used for a step-wise creation of a diagram as shown at the bottom of the figure. If no power value update happens during the execution time of the kernel t_{ex}, no value is emitted as illustrated in the last diagram at the bottom.

Using this approach, the generation of power profiles is performed as shown in Alg. 1: At first, the GPU kernel is executed once in order to determine its execution time t_{ex} (Lines 1 to 5). The algorithm assumes that the execution time does not vary between two calls, i.e. the kernel should always be called with the same parameters. Then, the function is called n_{ex} times. For the experiments

```
1  retrieve current time $t_{\text{start}}$
2  execute GPU kernel asynchronously
3  wait for GPU kernel to finish
4  retrieve current time $t_{\text{ret}}$
5  $t_{\text{ex}} := t_{\text{ret}} - t_{\text{start}}$
6  for $k = 1$ to $n_{\text{ex}}$ do
7      wait a random time $t_{\text{wait}}$
8      retrieve GPU power $P_{\text{last}}$
9      execute GPU kernel asynchronously
10     retrieve current time $t_{\text{start}}$; $t_{\text{curr}} := t_{\text{start}}$
11     while $t_{\text{curr}} < t_{\text{start}} + t_{\text{delay}} + t_{\text{ex}} + T$ do
12         retrieve GPU power $P_{\text{curr}}$
13         if $P_{\text{curr}} \neq P_{\text{last}}$ then
14             retrieve current time $t_{\text{update}}$
15             emit $t_{\text{update}} - t_{\text{start}},\ P_{\text{curr}}$
16             $P_{\text{last}} := P_{\text{curr}}$
17         retrieve current time $t_{\text{curr}}$
```

Algorithm 1: Retrieving power measurements for a GPU kernel function

presented in Sect. 5, $n_{\text{ex}} = 100$ was chosen, which results in an average of 100 values per sampling interval, i.e. 5000 values per second. Before the execution of the GPU kernel, the CPU is halted for a randomly chosen time interval (Line 7) in order to ensure that the power value is determined at a random point of the execution of the function. The waiting time t_{wait} is distributed uniformly in the interval $T \leq t_{\text{wait}} < 2T$ to avoid that the power value of the previous execution is read. Then, the GPU kernel is executed asynchronously (Line 9). Experiments, see Sect. 5, have shown that reading the starting time of the kernel t_{start} right after the return of the routine starting the kernel leads to a constant delay of $t_{\text{delay}} \approx 6$ ms between the starting time and the increase of the measured power. The while loop in Line 11 continually retrieves the power value as long as the GPU kernel is running. Each time the value P_{curr} changes (Line 13), it is emitted along with the time elapsed since the start of the kernel execution (Line 15).

The output of Alg. 1, i.e. a sequence of pairs (P, t), is then processed further to create a diagram. The P values are emitted in milliwatts and are in the range from the *long idle power* of 35 W [13] to the *thermal design power* of 225 W [17]. The t values indicate how many nanoseconds elapsed since the start of the GPU function. They are in the range from 0 to $t_{\text{ex}} + T$. A power profile diagram comprises the points resulting from all (P, t) pairs of one measurement. Results obtained with this method are presented in Sect. 5.

The algorithm presented in this section is only suitable for offline generation as it completely occupies one CPU core in the time interval beginning at the start of the GPU kernel execution until the power update which succeeds the termination of the kernel. However, the continual polling of the power value did not cause any observable effects on the execution of the GPU kernel.

5 Sample Power Profiles

The diagrams in Fig. 3 (a)–(e) show the power consumptions for various call configurations of *x*`gemm`, wich are matrix-matrix multiply BLAS functions from the CUBLAS package [18], and one further function, `vectid`. The `vectid` routine performs arithmetic operations and memory accesses in a way such that the GPU threads have different execution times. Each point in a diagram stands for one *time–power* (P, t) pair emitted by Alg. 1.

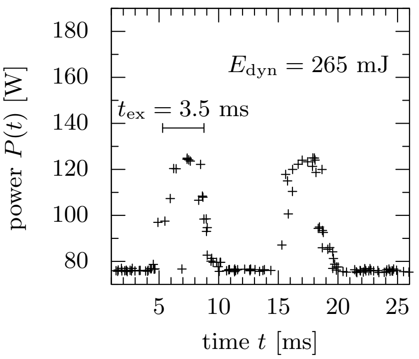
Splitting up the total power P_{tot} into static power P_{stat} and dynamic power P_{dyn} as

$$P_{\text{tot}} = P_{\text{stat}} + P_{\text{dyn}}$$

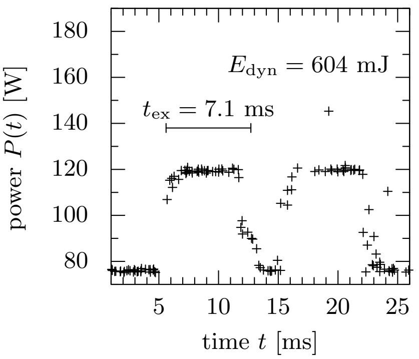
is often suggested, e.g. in [14]. For the static power in the diagrams, a value of $P_{\text{stat}} = 76$ W can be determined. The dynamic power depends on the specific workload to be processed on the GPU. For the `sgemm` call with a matrix size of 1024×1024, one can see that the power consumption increases up to 123 W during the execution of the routine, and then decreases when the routine is finished. After 5 ms, there is a second peak of roughly the same shape. This effect is closely related to the staircase effect that can be seen in the 2048×2048 case: The power consumption increases to 124 W starting at second 6.3 and then again increases to 172 W starting at second 16.3. The behaviour upon the termination of the routine is analogous. As all measurements show a similar effect, it looks as if one half of the power consumption of the GPU might have a delay of 10 ms in the measurement. This means, that integration over the whole interval leads to a correct value for the energy; the actual value for P_{dyn}, however, is twice as large as suggested by the figure during the first 10 ms of the execution of a function. Consequently, to compute the actual power consumption for the two 1024×1024 *x*`gemm` calls, the dynamic power suggested by the figure has to be doubled, i.e. for `sgemm`, the value of P_{tot} is 172 W instead of 124 W with $P_{\text{dyn}} = 96$ W. This corresponds to the 2048×2048 `sgemm` case in which P_{tot} is 172 W as well. For the `dgemm` cases, the dynamic power is $P_{\text{dyn}} = 92$ W with a P_{tot} of 166 W.

The `vectid` test function is a hand-written CUDA function. It mainly consists of a for loop running from 0 to $10 \cdot tid$, where *tid* is the thread id of the CUDA thread in the thread block. There are 32 blocks and 1024 threads per block. Inside the loop, 5 floating point operations are performed. Furthermore, two array entries are read from global memory and one is written. The different execution times of the different threads create imbalance which results in a smoother decrease of the power consumption than for the *x*`gemm` routines. Such an effect would not have been visible with measuring methods that have a coarser resolution.

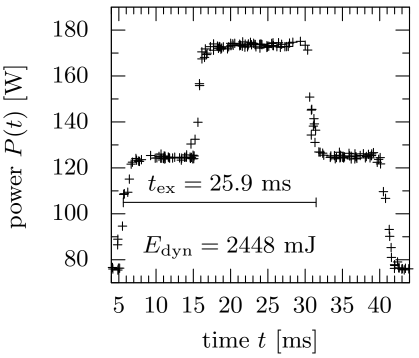
The values for the dynamic energy consumption E_{dyn} shown in Fig. 3 have been obtained by integrating the dynamic power consumption using the trapezoid rule with the values shown in the respective profile. Fig. 3 (f) was obtained by measuring the electric current flowing through the external PCI Express power connectors of the GPU card. The current multiplied by the voltage gives the



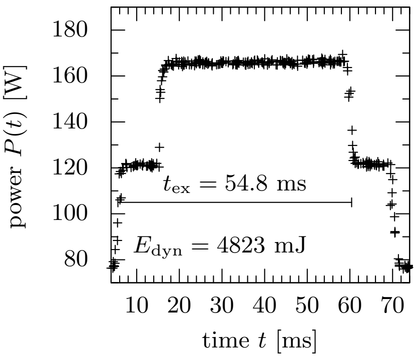
(a) sgemm 1024 × 1024



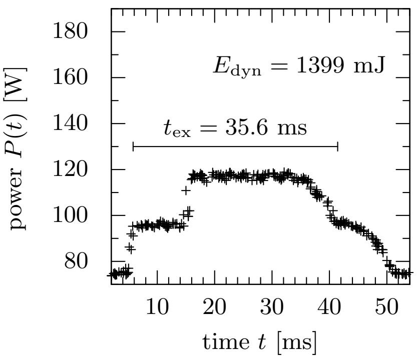
(b) dgemm 1024 × 1024



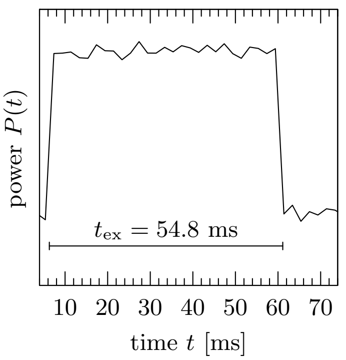
(c) sgemm 2048 × 2048



(d) dgemm 2048 × 2048



(e) vectid



(f) dgemm 2048 × 2048

Fig. 3. (a)–(e) Power profiles of CUBLAS *x*gemm and vectid kernels; (f) Power profile obtained by hardware measurement

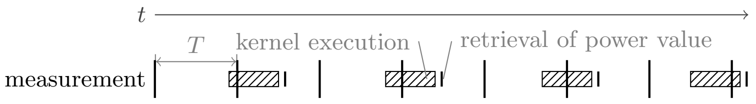


Fig. 4. Creating a power profile using the online method

electric power. Since further current flows through the PCI express socket, the actual values are inaccurate and they have been omitted. Nonetheless, one can see an increase in the power consumption for the duration of the execution of the kernel in the correct time interval.

The thermal design power of 225 W could not be reached with any *x*`gemm` experiment. The CUBLAS `dgemm` with a matrix size of $14 \cdot 2^{10} \times 14 \cdot 2^{10}$, which occupies all available streaming processors, consumed only 163 W, or 171 W if the GPU was pre-heated to roughly 85 °C. This corresponds to the results of Kasichayanula et al. [13] who report a value of 180 W for the average power consumption of the MAGMA `dgemm` kernel with a matrix size of 8192×8192, which is also significantly lower than the thermal design power.

6 Online Generation of Power Profiles

The approach for offline generation of power profiles as presented in Sect. 4 is adapted in order to allow the online generation of power profiles. This online method has less CPU usage and does not extend the execution time of the GPU kernel. The method is illustrated in Fig. 4. If the GPU kernel to be evaluated has an execution time lower than T, it is sufficient to retrieve one value of the power P_{curr} immediately after the termination of the GPU kernel. One can calculate the time of the last update of the power value precisely from the base time t_{sync} of the sampling interval and its period T. If the power value was updated during the execution of the GPU function, this value can be used for the power profile. This approach works on the condition that the machine has very precise clocks so that the times can be calculated exactly.

Experiments, however, have shown that the CPU and the GPU clocks slightly drift apart: This results in power profiles showing no clear peak unless a re-synchronisation is performed at least every 0.2 s. If this synchronisation is performed during the execution of a GPU kernel, not too much overhead is added as often the CPU is not fully occupied during GPU execution. By predicting the beginning of the next sampling interval and the termination time of the GPU kernel, one can estimate if the next update of the power value occurs during the execution.

The online generation of power consumption profiles is conducted as shown in Alg. 2: The function `syncClocks` (Lines 1 to 8) waits for the update of the power value and then returns the current time. The function `main` represents the structure of a general application algorithm of which the GPU energy consumption is to be evaluated. The while loop in Line 19 repeatedly calls the GPU kernel (Line 22). Before the while loop is started, the time t_{sync} at which the power value is updated, is determined (Line 16).

```
1  function syncClocks()
2      retrieve GPU power $P_{\text{curr}}$
3      $P_{\text{last}} := P_{\text{curr}}$
4      while $P_{\text{last}} = P_{\text{curr}}$ do
5          $P_{\text{last}} := P_{\text{curr}}$
6          retrieve GPU power $P_{\text{curr}}$
7      retrieve current time $t_{\text{curr}}$
8      return $t_{\text{curr}}$

9  function syncIfPossible()
10     $t_{\text{expected\_finish}} := t_{\text{start}} + \beta t_{\text{ex}}$
11     $t_{\text{next\_update}} := \lfloor \frac{t_{\text{start}} - t_{\text{sync}}}{\Delta t} + 1 \rfloor \Delta t$
12     if $t_{\text{expected\_finish}} > t_{\text{next\_update}}$ then
13         $t_{\text{sync}} :=$ syncClocks()
14         $n_{\text{last\_sync}} := n_{\text{call}}$

15 function main()
       // main algorithm
16     $t_{\text{sync}} :=$ syncClocks()
17     $n_{\text{call}} := 0$
18     $n_{\text{last\_sync}} := 0$
19     while ... do
           // main algorithm
20         $n_{\text{call}} := n_{\text{call}} + 1$
21         retrieve current time $t_{\text{start}}$
22         call GPU kernel
23         syncIfPossible()
24         wait for GPU kernel to finish
25         retrieve current time $t_{\text{finish}}$
26         retrieve GPU power $P_{\text{curr}}$
27         $t_{\varphi} := t_{\text{next\_update}} - t_{\text{start}}$
28         emit $t_{\varphi}, P_{\text{curr}}$
29         if $n_{\text{call}} - n_{\text{last\_sync}} > \gamma$ then
30             $t_{\text{sync}} :=$ syncClocks()
31             $n_{\text{last\_sync}} := n_{\text{call}}$
           // main algorithm
```

Algorithm 2: Online generation of a power profile

The starting time t_{start} of the GPU kernel is taken in Line 21 and the kernel is executed asynchronously in Line 22. In the function `syncIfPossible`, the clock synchronisation takes place concurrently to the GPU kernel execution: If the expected termination time of the GPU kernel $t_{\text{expected\_finish}}$ is greater than the expected next update of the power value at time $t_{\text{next\_update}}$, the actual time of the update is determined by the function `syncClocks`. For β in Line 10, a value of $\frac{4}{5}$ has been used in order to avoid a synchronisation in the late phase of the GPU kernel execution. After the termination of the GPU function (Line 24), the current power value P_{curr} is retrieved (Line 26). In Line 27, t_{φ}, i.e. the phase of the sampling interval in which the kernel function has been started, is calculated. Next, the values t_{φ} and P_{curr} are emitted.

As mentioned above, the clocks drift apart too much after roughly 0.2 s. Therefore, a re-synchronisation is forced in Lines 29 to 31 if there has been no synchronisation in the function `syncIfPossible` for a while. The value of γ should be set in a way that the re-synchronisation takes place at least every 0.2 s.

The overhead introduced by the measurement, i.e. the extension of the total execution time of the algorithm on the CPU, was not above 10 % in the test cases. A profile created using online generation can be seen in Fig. 5. The 1024×1024 case shows one peak similar to that in Fig. 3(a). Its power consumption corresponds to that measured in Sect. 5. However, the second peak is not visible, possibly it occurs too late after the execution of the function. Due to this effect, currently only GPU functions with an execution time less than 10 ms can be properly evaluated using the online method.

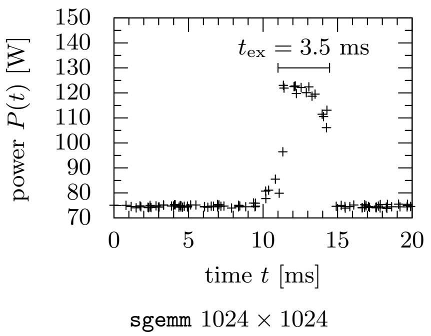


Fig. 5. Online-generated power profile

7 Conclusion

The article has presented a method which can generate high-resolution power profiles for GPU functions even if there are only measurement instruments with a low temporal resolution available. The sampling interval of the measurement offered by the NVML has been determined to be 20 ms. The method presented allows the generation of high-resolution power profiles of GPU functions without requiring any additional hardware. Such power profiles can for example be used by developers to optimise the power consumption of their code. By integrating the power values, the energy consumption of a specific function can be calculated, which can, e.g., be used for auto-tuning its energy consumption.

The offline method for generating power profiles works very accurately, so that some interesting effects could be demonstrated at the sample profiles shown, such as the smooth decrease of the power consumption for unbalanced loads. The method has been extended to an online method in order to enable the generation of power profiles during the execution of simulations etc. There is only a low overhead. Such profiles can, e.g., be used for auto-tuning at runtime. Restrictions of the online method are that it is currently only suitable for kernels with an execution time of less than 10 ms and that its accuracy is lower than the accuracy of the offline method. In general, the method presented is not restricted to GPUs but can also transferred to other measurement instruments whose sampling interval is too large for the targeted purpose.

Acknowledgements. This work is supported by the cluster of excellence *Energy-Efficient Product and Process Innovation in Production Engineering* (eniPROD) funded by the European Union (ERDF) and the Free State of Saxony.

References

1. Powermizer 8.0: Intelligent power management technology. Tech. rep., Nvidia (June 2008), tB-04051-001_v01
2. AMD PowerTune technology. Whitepaper, AMD (December 2010), `http://www.amd.com/uk/Documents/PowerTune_Technology_Whitepaper.pdf`

3. Abe, Y., Sasaki, H., Peres, M., Inoue, K., Murakami, K., Kato, S.: Power and performance analysis of GPU-accelerated systems. In: Workshop on Power Aware Computing and Systems, HotPower 2012 (2012)
4. Advanced Micro Devices: BIOS and Kernel Developer‘s Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, rev. 3.12 (October 2012)
5. Chen, D., Singh, D.: Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering. In: 22nd Int. Conf. on Field Programmable Logic and Applications (FPL), pp. 5–12 (2012)
6. Chen, J., Li, B., Zhang, Y., Peng, L., Peir, J.K.: Statistical GPU power analysis using tree-based methods. In: Int. Green Computing Conf. and Workshops (IGCC), pp. 1–6 (2011)
7. Collange, S., Defour, D., Tisserand, A.: Power consumption of GPUs from a software perspective. In: 9th Int. Conf. on Computational Science (2009)
8. Hong, S., Kim, H.: An integrated GPU power and performance model. SIGARCH Comput. Archit. News 38(3), 280–289 (2010)
9. Huang, S., Xiao, S., Feng, W.: On the energy efficiency of graphics processing units for scientific computing. In: IEEE Int. Symp. on Parallel Distributed Processing (IPDPS 2009), pp. 1–8 (2009)
10. Hähnel, M., Döbel, B., Völp, M., Härtig, H.: Measuring energy consumption for short code paths using RAPL. In: GREENMETRICS 2012 (2012)
11. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer‘s Manual (May 2012)
12. Isci, C., Martonosi, M.: Runtime power monitoring in high-end processors: Methodology and empirical data. In: 36th IEEE/ACM Int. Symp. on Microarchitecture (2003)
13. Kasichayanula, K., Terpstra, D., Luszczek, P., Tomov, S., Moore, S., Peterson, G.: Power aware computing on GPUs. In: 2012 Symp. on Application Accelerators in High-Performance Computing (2012)
14. Li, D., Byna, S., Chakradhar, S.: Energy-aware workload consolidation on GPU. In: 40th Int. Conf. on Parallel Processing Workshops, pp. 389–398 (2011)
15. McCullough, J.C., Agarwal, Y., Chandrashekar, J., Kuppuswamy, S., Snoeren, A.C., Gupta, R.K.: Evaluating the effectiveness of model-based power characterization. In: 2011 USENIX Conf. (2011)
16. Nagasaka, H., Maruyama, N., Nukada, A., Endo, T., Matsuoka, S.: Statistical power modeling of GPU kernels using performance counters. In: Int. Green Computing Conf. (IGCC), pp. 115–122 (2010)
17. Nvidia: Tesla C2075 Computing Processor Board. Board Specification (2011), http://www.nvidia.com/docs/IO/43395/BD-05880-001_v02.pdf, bD-05880-001_v02
18. Nvidia: CUBLAS Library – User Guide, version 5.0 (October 2012), http://docs.nvidia.com/cuda/pdf/CUDA_CUBLAS_Users_Guide.pdf
19. Nvidia: NVML API Reference Manual, ver. 3.295.45 (2012), http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/NVML/nvml.pdf
20. Rauber, T., Rünger, G.: Towards an energy model for modular parallel scientific applications. In: Green Computing and Communications (GreenCom), pp. 523–532 (2012)
21. Rofouei, M., Stathopoulos, T., Ryffel, S., Kaiser, W., Sarrafzadeh, M.: Energy-aware high performance computing with graphic processing units. In: Workshop on Power Aware Computing and Systems, HotPower 2008 (2008)
22. Weaver, V., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., Moore, S.: Measuring energy and power with PAPI. In: Int. Workshop on Power-Aware Systems and Architectures, PASA 2012 (2012)

Power/Performance Trade-Offs of Small Batched LU Based Solvers on GPUs

Oreste Villa[1], Massimiliano Fatica[2], Nitin Gawande[1], and Antonino Tumeo[1]

[1] Pacific Northwest National Laboratory, Richland, WA, USA
{oreste.villa,nitin.gawande,antonino.tumeo}@pnnl.gov
[2] NVIDIA, Santa Clara, CA, USA
mfatica@nvidia.com

Abstract. In this paper we propose and analyze a set of batched linear solvers for small matrices on Graphic Processing Units (GPUs), evaluating the various alternatives depending on the size of the systems to solve. We discuss three different solutions that operate with different levels of parallelization and GPU features. The first, exploiting the CUBLAS library, manages matrices of size up to 32x32 and employs Warp level (one matrix, one Warp) parallelism and shared memory. The second works at Thread-block level parallelism (one matrix, one Thread-block), still exploiting shared memory but managing matrices up to 76x76. The third is Thread level parallel (one matrix, one thread) and can reach sizes up to 128x128, but it does not exploit shared memory and only relies on the high memory bandwidth of the GPU. The first and second solutions only support partial pivoting, the third one easily supports partial and full pivoting, making it attractive to problems that require greater numerical stability. We analyze the trade-offs in terms of performance and power consumption as function of the size of the linear systems that are simultaneously solved. We execute the three implementations on a Tesla M2090 (Fermi) and on a Tesla K20 (Kepler)[1].

1 Introduction

Many computer simulations used in hydrology, combustions and atmospheric modeling require the use of solvers that operate on a large amount of small independent systems of equations. These models typically operate by computing at each time step of the simulation the flow, and then the chemical reactions of fluids and solids in other elements over a large number of locations (or physical grid nodes). The chemical reactions are described through a set of non-linear equations. Profiling of typical codes shows that these models spend over 95% of the time on computing the chemical reactions [7]. A typical simulation involves from few tens to hundreds of chemical reactions in millions of uniform or non-uniform grid locations, depending on the geometry and resolution of the problem solved. A typical method for obtaining a solution for such systems of non linear equations is the Newton-Raphson technique. The technique involves computing

[1] We thank Norbert Juffa (NVIDIA) for the help with the Thread-block level implementation and the useful discussions on the power/performance trade-offs.

a Jacobian matrix and a residual vector for each set of equations representing the reactions for a grid location. The linearized systems are solved iteratively, until convergence is reached, by performing Gaussian elimination with LU factorization. The LU factorization is performed either using partial or full pivoting depending on the numerical characteristics of the problem, time-step of the simulation and ultimately accuracy of the result. Since the Jacobian matrices are generated starting from the chemical reactions, their sizes is typically a square function of the number of equations involved in the process. For example, simulation of kinetic chemical reactions in combustion modeling [3] typically involves matrices up to $\approx 40x40$ in sizes, and is usually numerically stable by just using partial pivoting for the LU decomposition. Reactive transport models for fluids through the Earth's crust over multiple phases, instead, require matrices with sizes up to $\approx 100x100$ and traditionally use LU decomposition with full pivoting to increase numerical stability. STOMP [9], HydroGeoChem [10], PRFLOTRAN [2], and TOUGH [11] use some of these models.

General Purpose computing on Graphic Processing Units (GPGPU) is a very effective approach for implementing linear solvers [8]. However, GPUs are more efficient when the number of operations to perform is much larger than the amount of data involved (flop/byte ratio). Thus, they perform particularly well with matrices of large size. Conventional solvers, such as MAGMA [1] or those provided by the CUDA library [5], target large matrices with several thousand of elements per dimension, achieving speedups of one order of magnitude when compared to CPUs. They exploit parallelism at the level of a single matrix solver, in some cases exploiting the CPU-GPU interaction by assigning the diagonal blocks and interchange of row and columns to CPU cores and reduction and scaling of large submatrices to GPUs [6]. Only recently a combination of the increased parallelism exploited by GPUs, together with their high bandwidth, made them more attractive for operations on small matrices. Indeed, the latest version of the CUBLAS library [4] includes support for a batched LU factorization that can be employed to construct solvers operating on small independent matrices. The batched factorization exploits Warp-level parallelism (a Warp, composed of 32 threads, operates in parallel on a matrix) and is therefore limited to matrices of size 32x32.

In this paper we present a set of solutions for batched solvers that operate on large amounts of small matrices ranging from size 2x2 to 128x128 on Nvidia GPUs. We initially consider the CUBLAS batched LU factorization to construct a solver (up to size 32x32) and then we propose two further implementations. One, exploiting Thread-block level parallelism (a Thread-block works in parallel on a matrix) and shared memory, can manage matrices up to 76x76 in size. By supporting partial pivoting, it appears well fit for solvers in combustion modeling. The other, exploiting thread-level parallelism (a thread for each matrix), can support both partial and full pivoting and manage matrices up to 128x128 in size. Its characteristics make it more amenable for subsurface flow transport applications over multiple phases. We present the details of the implementations, discussing the various programming techniques adopted to maximize the performance depending on the problem constraints. We evaluate all these three implementations from the point of view of performance and power consumption as function of the size of the matrices that are being simultaneously solved. We discuss trade-offs, suggesting which solution better fits different requirements. We execute the three implementations on both a Tesla M2090 (Fermi GPU) and a Tesla K20 (Kepler GPU).

The remainder of this paper is organized as follows. Section 2 provides some preliminaries on the linear solvers and summarizes the significant features of the Fermi and Kepler GPU architectures. Section 3 presents the implementation details of the three solvers. Section 4 discusses the experimental evaluation of the three solvers, also providing power analysis. Finally, Section 5 concludes the paper.

2 Preliminaries

Solver: A solver is a procedure that given a system of linear equations described in a matricial form as $\mathbf{Ax} = \mathbf{b}$ finds the solution vector $\mathbf{x}$. For dense and semi-dense matrices the most efficient method involves finding a decomposition of the matrix $\mathbf{A}$ such that the solution is then obtained by back substitution. The most common method is based on the LU decomposition (also called LU factorization) of the matrix $\mathbf{A}$ which is decomposed in a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ such that $\mathbf{LU} = \mathbf{A}$. However, rounding errors can arise if the magnitude of the elements significantly differs, causing one or more elements on the diagonal to become zero (singular matrix). A solution to this problem is to interchange the rows and columns of $\mathbf{A}$ to avoid zero and unstable pivot elements. These interchanges do not affect the solution of the system as long as the permutations are logged and taken into account during the substitution process. The choice of pivot elements is referred as pivot strategy. There is not an optimal pivot strategy, but two common heuristics are *partial pivoting* and *complete pivoting.*

When using partial pivoting, the factorization produces matrices $\mathbf{L}$ and $\mathbf{U}$, which satisfy the equation $\mathbf{LU} = \mathbf{PA}$, where $\mathbf{P}$ is a permutation matrix. Initially, $\mathbf{P}$ is initialized to $\mathbf{I}$, then each row interchange that occurs during the decomposition of $\mathbf{A}$ causes a corresponding row swap in $\mathbf{P}$. Starting from the linear system of equations $\mathbf{Ax} = \mathbf{b}$ and pre-multiplying both sides by $\mathbf{P}$, we obtain $\mathbf{PAx} = \mathbf{Pb}$. Substituting $\mathbf{PA}$ with $\mathbf{LU}$, we obtain $\mathbf{LUx} = \mathbf{Pb}$. Thus, we can achieve a solution for $\mathbf{A}$ by the sequential solution of two triangular systems: $\mathbf{Lc} = \mathbf{y}$, $\mathbf{Ux} = \mathbf{c}$, where $\mathbf{y} = \mathbf{Pb}$.

When using complete pivoting, factorization produces matrices $\mathbf{L}$ and $\mathbf{U}$,which satisfy the equation $\mathbf{LU} = \mathbf{PAQ}$, where $\mathbf{P}$ is a row permutation matrix and $\mathbf{Q}$ is a column permutation matrix. $\mathbf{Q}$ is derived from column interchanges in the same way $\mathbf{P}$ is derived from row interchanges. The linear system of equations $\mathbf{Ax} = \mathbf{b}$ can be solved by the sequential solution of two triangular systems: $\mathbf{Lc} = \mathbf{y}$, $\mathbf{Uz} = \mathbf{c}$, with $\mathbf{y} = \mathbf{Pb}$ and $\mathbf{x} = \mathbf{Qz}$.

The computational complexity of the LU factorization is $\mathbf{O(2/3 * n^3)}$. Partial pivoting contributes for a further $\mathbf{O((n^2 + n)/2)}$ while full pivoting adds $\mathbf{O(2/3 * n^3 + 1/2 * n^2 + 1/6 * n)}$. Once the matrix is decomposed, each triangular solver has computational complexity $\mathbf{O(n^2)}$. Asymptomatically, a solver with partial pivoting has computational complexity of $\mathbf{O(2/3 * n^3)}$, while with full pivoting complexity is $\mathbf{O(4/3 * n^3)}$. For this reason, there are significant trade-offs between the stability of the solution and the time to solution, depending on the use of the solver and on the data ranges in which it operates. The discussions of these trade-offs are beyond the scope of the paper, because they are tightly coupled with the applications in which the solvers are used.

GPU Architectures: for this work, we used two different GPU architectures. The first is the Fermi architecture, at the base of the Tesla M20 boards. The second is the new Kepler architecture, with the GK110 design, integrated in the Tesla K20 boards.

The Fermi architecture exploits a set of Streaming Multiprocessors (SMs) that include 32 Streaming Processors (SPs), 4 Super Function Units (SFUs), 16 Load/-Store Units and 64 KB of on-chip memory configurable either as 48 KB of L1 cache and 16 KB of shared memory or as 16 KB of L1 cache and 48 KB of shared memory. A Fermi's SM can simultaneously execute two single precision Warps (group of 32 threads) and a one double precision Warp in a minimum of 2 clock cycles. Thus, peak double precision is half of the single precision. Each SM includes a total of 32,768 registers and can maintain up to 1536 threads in-flight. All the SMs in a chip interface to a L2 cache of 768 KB. The SMs access the global memory through a crossbar connected to several 64 bits memory controllers. In Fermi, the SMs run at higher clocks (double) than the rest of the chip.

The Kepler design is radically different. A SM in Kepler, now called SMX, includes 192 single precision SPs, 64 double precision SPs, 32 SFUs, 32 Load/Store Units. NVIDIA respectively incremented the number of threads and of registers per SMX to 2048 and 65536. Kepler can dispatch 8 instructions (2 independent instructions from 4 Warps) simultaneously and can pair double precision instructions with other instructions. Each SMX still has 64 KB of configurable shared memory, which now supports a 32/32 KB split. This results in a higher number of Warps competing for the same shared memory. An SMX also includes a new 48 KB cache for read-only data. Kepler doubles the L2 cache both in terms of size (1536 KB) and bandwidth with respect to Fermi. Kepler includes further new functions such as Hyper-Q and Dynamic Parallelism that however we do not exploit in our evaluation.

For this work, we used a Tesla M2090 board, which include the Fermi T20a GPU, with 16 SMs (a total of 512 SPs) at 1.3 GHz and 6 GB of GDDR5 at 1.85 GHz, connected through a 384-bit interface. The peak memory bandwidth is 177 GB/s. Regarding Kepler, we used a Tesla K20 board with a GPU that implements 13 SMXes (2496 SPs). The GPU works at 706 MHz, and the board includes 5 GB of GDDR5 at 2.6 GHz, connected through a 320-bit bus with 5 memory controllers. The peak memory bandwidth is 208 GB/s.

3 Solvers' Implementation

In this section we present the three different solver implementations. We initially present the Warp level solution (CUBLAS), then present the Thread-block level implementation and finally discuss the Thread level solution. The high-level interface exposed by all the implementations is a function *dsolve_batch()* for solving a batch of N different systems with double-precision. All systems must be of the same dimension n. Besides the batch size and the matrix dimension, the functions expect pointers to an array of matrices, an array of right hand sides, and an array of solution vectors. All arrays are assumed to be stored contiguously. By using a *define* it is possible to support column or row major layouts.

3.1 Warp Level Parallelism (CUBLAS Based)

This implementation exploits the batched interfaces of the CUBLAS library [4], provided in CUDA 5.0. It involves four GPU kernel calls for all the matrices as follows: 1) LU decomposition of A ($PA = LU$); 2) permutation of array b with the array of pivots P ($y = Pb$); 3) solution of the triangular lower system ($Lc = y$), and 4) solution of the upper system to obtain the final solution ($Ux = c$). Three kernel calls are directly provided by two library's function calls in the form of *cublasDgetrfBatched* (i.e. $PA = LU$), *cublasDtrsmBatched* (i.e. $Lc = y, Ux = c$), while we implemented a simple kernel that performs the permutation of the array b. Applying the permutation to the array b has negligible execution time when compared to the LU decomposition and to the solution of the triangular systems.

The batched functions inside the CUBLAS library assign a single Warp (32 threads) to each matrix, and it is limited at most to a size of 32x32. This implementation heavily relies on shared memory. However, shared memory content is not preserved across different kernel calls, thus extra work has to be made to re-cache data in shared memory by each kernel. An advantage of this implementation is that it relies on a NVIDIA's well maintained CUDA library, which is constantly updated as new GPUs become available.

3.2 Thread-Block Level Parallelism

This is a custom implementation (now available on the NVIDIA developer site) that relies on three mutually exclusive kernels, selected depending on the size of the input matrices. The high-level *dsolve_batch()* API functions call into a single templatized function that is parameterized by data type and architecture. For performance reasons, each system is loaded into shared memory in its entirety. This means that the maximum dimension of the matrix that can be handled is limited by the available shared memory. For GPU architectures up to the Fermi and Kepler, this implementation can handle systems up to dimensions 76x76 (double precision). This solver has been finely tuned for the Fermi architecture.

When loaded into shared memory, the matrix is augmented on the right with the right hand side vector, allowing both to be manipulated in parallel. The two-dimensional shared memory layout of the matrix uses padding to minimize bank conflicts. The amount of padding is optimized for each matrix size via the configuration class. The number of Thread-blocks in the launch configuration is identical to the batch size. Therefore, each Thread-block solves a single system. Two-dimensional Thread-blocks are used, where the x-dimension is configured for optimal performance by the template class, and the y-dimension is identical to the number of columns of the augmented matrix. This allows each thread row to handle one row of the augmented matrix in parallel during the solve.

The three kernels used are gauss_jordan1 (used for dimensions 2 through 9), gauss_jordan2 (used for dimension 10), and gauss2 (for dimensions 10 through 76). The switch-over points were determined empirically. The first two kernels implement the standard Gauss-Jordan algorithm with partial pivoting, and the third implements straight Gauss elimination with partial pivoting. Experimentally we can see that the absence of a separate back substitution step in Gauss-Jordan outweighs the smaller count of floating-point operation of Gauss elimination for small matrices. The two Gauss-Jordan kernels differ in that for gauss_jordan1 the number of thread rows is identical to the number of rows in the matrix, i.e. each thread handles exactly one element of the augmented matrix, whereas in gauss_jordan2 the number of threads is less than the number of matrix rows, so each thread handles more than one matrix element. The former approach eliminates some overhead for iteration over the rows of the matrix.

The maximum search in partial pivoting is implemented as a two-stage process. In the first stage, a small number of threads search for a maximum in their respective subset of column elements. In a second stage, these partial results are reduced to an overall maximum by a single thread. This approach was found to be more efficient than the traditional binary reduction process. The number of search threads is fixed at two for the two Gauss-Jordan kernels, but is configurable for optimal performance via the template class for the Gauss elimination kernel. The number of search threads is generally a small, single digit number. The row swapping is implemented by physical exchange. This provides higher performance than an approach that uses an index vector for virtual row reordering, because it incurs in significant overhead for indexing arithmetic and consumes additional shared memory.

3.3 Thread Level Parallelism

This is the only implementation that allows using matrices of size 128x128. Although it can work on larger matrices, we will limit our discussion to the dimensions relevant to the problem of interest, as discussed in the Introduction. At a first glance, this implementation violates basic GPU programming principles, because it assigns different "tasks" to different threads inside a Warp, which could result in poor performance due to thread divergence within the Warp.

However, the threads mostly perform the same operations, each on its assigned matrix. The only source of divergence is the discovery of pivot elements. The key observation is that, when the matrix is larger than 32x32, the cost of these operations is much smaller than the cost of updating the lower matrix and back substituting in the triangular systems. Vice-versa, when the matrix is smaller than 32x32, the cost of pivoting and row interchange is comparable to the cost of updating the matrices and performing back-substitution, resulting in possible thread divergence. However this is true with any other implementation, because with matrices smaller than 32x32 Warps are not fully utilized. Another important issue of this approach is that the input matrices A and vectors b and x are stored as arrays of structures, meaning that big arrays contain all the elements of the different matrices and vectors. If each thread is accessing its own matrix, the threads in a Warp are accessing elements that are strides of the number of elements in the matrix. This results in un-coalesced accesses to memory, which is a main cause of performance degradation. To alleviate this problem, we perform a transformation of the matrix A and of the array b and x before and after the solver phase, such that the resulting data structure is a structure of arrays. This operation is quite expensive, especially for the matrices, as the elements need to be accessed at least once in an un-coalesced manner. However, this procedure has cost $O(n^2)$, while the entire computational complexity of the algorithm is $O(n^3)$.

Since we want to preserve the original matrices A (without re-transforming back after the solver completes), we need to store the transformed matrix in a temporary space. Unfortunately, the shared memory space is not enough for this purpose, because we need at least space for a number of matrices equal to the block size (the effective minimum is 32, as the size of the Warp). For this reason, we exploit another portion of GPU memory that is allocated and deallocated on a Thread-block basis by a single thread in the block. We perform these allocations on a heap space, set during initialization of the device by using the CUDA library call *cudaThreadSetLimit(cudaLimitMallocHeapSize, bytes).*

We perform allocations and de-allocations inside the heap space with *_malloc/_free*, which wraps the standard malloc/free and align data to 128 bytes inside in the heap. We do not need to have a heap space as large as the total dataset, because a Thread-block can reuse the same heap space released by a previous Thread-block. For our experiments we set the heap space to 1GB regardless of the dataset. The pseudo-code of the solver is the following:

```
#define T(id)   ( threadIdx.x + blockDim.x * (id))
#define P(x,y)  ( y * n + x)

__global__ void solver(double *A, double *B, double *X, int n, int num) {
  int m = blockIdx.x * blockDim.x + threadIdx.x;
  if ( m >= num) return;
  int i, j, k;
  __shared__  double * tA, *tB, *tX;
  __shared__  char    * P;

  if(threadIdx.x == 0) {
    tA=(double*) _malloc(blockDim.x * n * n * sizeof(double));
    tB=(double*) _malloc(blockDim.x * n * sizeof(double));
    tX=(double*) _malloc(blockDim.x * n * sizeof(double));
    P=(char*) _malloc(blockDim.x * n * sizeof(char));
  }
  __syncthreads();

  /* coalesce A and B */
  for (j = 0; j < n; j++) {
    tB[T(j)] = B[ m * n + j];
    for (i = 0; i < n; i++)
      tA[T(P(j,i))] = A[ m * n * n + P(j,i) ];
  }

  /* perform LU decomposition and triangular solver
     as in serial code using T(id) and P(x,y) macros */
  single_thread_solver(tA, tB, tX, n);

  /* uncoalesce X */
  for (j = 0; j < n; j++)
    X[ m * n + j] = X[T(j)];
  __syncthreads();

  /* free tmp memory */
  if(threadIdx.x == 0) {
    _free(tA); _free(tB); _free(tX); _free(P);
  }
  return;
}
```

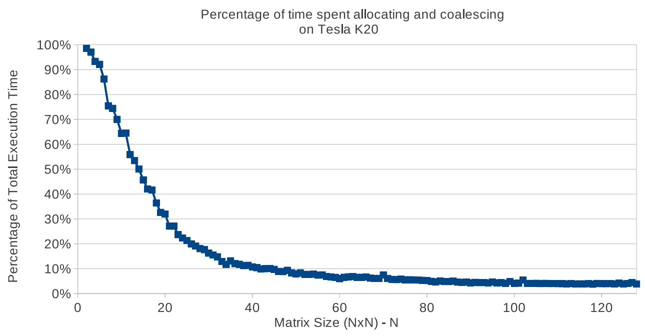


Fig. 1. Percentage of time spent allocating and coalescing on the Tesla K20 with respect to the total execution time

We note that allocating/de-allocating the temporary memory space, coalescing A and b and de-coalescing x is part of the total kernel execution, because it is not performed off-line. Figure 1 shows the cost of this operation on the Kepler GPU in percentage of the total time as the matrix size increases. Because each linear system is solved by a single thread, implementing full pivoting is trivial. In fact, it just adds a few lines of code to the partial pivoting implementation. With the other implementations, parallelism within the Warp or Thread-block is disrupted when both rows and columns are interchanged.

4 Experimental Evaluation

In this section we present the experimental evaluation of the three solver implementations. We initially present the performance analysis, and then proceed with the discussion of the power implications.

4.1 Performance Analysis

Table 1 shows the main characteristics of the three different implementations while increasing the size of the matrices. We extracted these data with the NVIDIA profiler on the Tesla K20, but they are similar for the Tesla M2090. The numbers show that the main limitation for the Thread level implementation is the number of registers, which is the highest and keeps growing while increasing the size of the matrices. To balance with the utilization, however, we set the maximum number of registers per thread to 128. We also see how shared memory utilization progressively becomes the constraining factor for the Warp and the Thread Block level implementations, consequently limiting the maximum size of manageable matrices. The Thread level implementation only uses the few bytes required to store the shared pointers in the heap space. For the Thread level implementation, DRAM utilization keeps increasing with the size

Table 1. Profiling of the three implementations

Dimensions		4	8	16	32	64	76	128
Registers/ Thread	Warp	34	34	34	34			
	Thread block	26	26	23	23	23	23	
	Thread	40	62	70	103	128	128	128
Shared Memory/ Block	Warp	5.25KB	8.5KB	21KB	43.7KB			
	Thread block	184	608	2.5KB	9KB	35KB	45KB	
	Thread	32	32	32	32	32	32	32
DRAM Utilization	Warp	12.36%	15.18%	9.86%	4.44%			
	(GB/s)	24.53	30.10	19.63	8.90			
	Thread block	2.8%	4.1%	6.6%	3.2%	1%	0.9%	
	(GB/s)	5.61	8.12	13.7	6.36	2	1.8	
	Thread	6.9%	18.7%	49.3%	58.7%	61.9%	62.1%	61.2%
	(GB/s)	13.75	37.03	97.79	116.51	122.72	123.17	121.42
Branch Divergence Overhead	Warp	78.49%	72.16%	55.24%	21.30%			
	Thread block	70.38%	62.25%	49.59%	43.04%	28.34%	20.60%	
	Thread	86.20%	67.40%	40.20%	9.00%	1.60%	1.00%	0.07%
SM utilization Achieved	Warp	89.6/95%	69.8/72.13%	24.59/25%	7.69/7.8%			
	Thread block	24.4/25%	74/75%	49.6/50%	31.2/31.2%	17.2/17.2%	15.6/15.6%	
	Thread	21.3/25%	20.8/25%	22.2/25%	24/25%	24.3/25%	24.2/25%	24.2/25%
Memory (MB)	Warp	97	97	127	244			
	Thread block	131	139	171	303	771	1031	
	Thread	1179	1187	1219	1341	1819	2079	3675

of the matrices and stabilizes at around 62% of the available bandwidth. For the other implementations, instead, it remains pretty low because all systems are solved in the shared memory, and only the initial and the final values are read and written to memory. For the Warp and Thread level implementation, branch divergence starts high and remains over 20% even for the highest dimensions. These implementations have threads of the same Warp or Thread-blocks that simultaneously operate on different elements of a matrix, and when searching for the pivots they need to be coordinated. The Thread level implementation instead shows high divergence with small matrices, which progressively reduces with bigger dimensions. The reason is that the loop searching for the pivots has an *if* statement that selects the largest element in the column, so threads of the same Warp diverge. The time spent searching the pivot ($O(n^2)$) decreases with respect to the rest of the algorithm ($O(n^3)$) as the matrix size increases. The SM utilization for the Warp and Thread Block level implementations is limited by shared memory occupation, and decreases with larger matrices (because each Warp/Thread-block uses more shared memory). For the Thread level implementation, instead, it is constrained by the number of registers. The overall memory occupation of the Thread level solver is the highest, because it employs a 1 GB dynamically allocated heap and stores original and transposed matrices to maintain memory access coalescence.

For the performance analysis, we simultaneously solve 20,000 systems of linear equations of different dimensions. Using 20,000 matrices allows exposing the full parallelism of the GPU, even in the Thread level implementation. Figure 2 shows the performance of the three solver implementations (Warp level parallel, Thread-block level parallel and Thread level parallel) on a Tesla M2090 (Fermi). The Thread level parallel implementation is the slowest, but it can scale to matrices up to 128x128 in size. Full pivoting is slower than partial pivoting because of the higher computational complexity, but it follows the same behavior. The performance also appears more stable: the execution time increases almost linearly with the size of the systems to solve. The main limitation for such implemen-

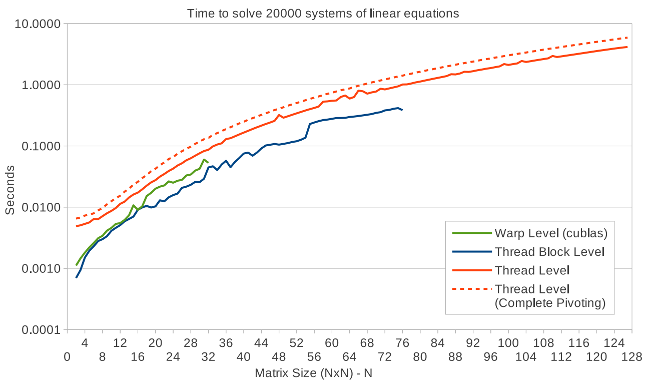


Fig. 2. Performance of the three implementations on a Tesla M2090 when solving 20,000 systems of linear equations while increasing the number of double precision elements in each matrix

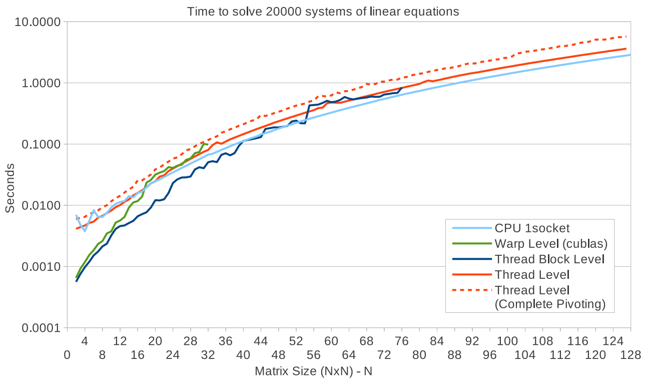


Fig. 3. Performance of the three implementations on a Tesla K20 when solving 20,000 systems of linear equations while increasing the number of double precision elements in each matrix

tation is the number of registers used by each thread, which does not allow full utilization of the SMs. The Warp level implementation with CUBLAS is faster than the thread level implementation. It is comparable to the Thread-block level implementation for matrices up to 16x16 in size, but starts diverging with matrices over 20x20 variables. This happens even if the switch points for the different kernels in the Thread-block level implementation are at lower dimensions (9 and 10 respectively). By exploiting Warp level parallelism, this implementation is only able to manage matrices up to 32 in size. However, it benefits from the higher bandwidth provided by the shared memory. The Thread-block level implementation is the fastest, and manages matrices up to 76x76 elements. However, its performance shows more significant degradation when increasing the size of the matrices, in particular over size 56. The reason is that when increasing the size of the matrices, occupation of the shared memory increases. This in turn reduces the number of simultaneously active Thread-blocks, consequently reducing SM utilization.

Figure 3 proposes the same performance evaluation on the Tesla K20 (Kepler). The Thread level implementation remains the slowest in average, but for certain dimensions now it results faster than both the Warp level and the Thread Block level implementations. The switch points are dimensions of 16x16 for the Warp level implementation and 56x56 for the Thread-block level implementations. The Thread level implementation is near to the memory bandwidth limits and benefits from the increased bandwidth of Kepler. On the other hand, Kepler provides less bandwidth per active Warp to the shared memory, which makes less effective the Warp level and Thread-block level implementations. The performance spread between the Warp level and the Thread-block level implementation with matrices over sizes of 16x16 is more significant. The Thread level implementation with full pivoting is the slowest, but it still follows the behavior of same solution with partial pivoting. This figure also shows that for small matrices the performance of Kepler with the Thread level implementation is comparable to a reference x86 implementation (Xeon X5650 at 2.67GHz, 6 Nehalem cores with 12 threads at 12 MB of L3 cache), while for larger matrices it becomes slower.

4.2 Power Analysis

Figure 4 presents the power consumption for the three different solver implementations on the Tesla M2090. We measured power using the integrated on-chip power monitoring, which is accessible through the NVIDIA System Management Interface (nvidia-smi) utility, executing each benchmark for 30 seconds in a loop and sampling power at the minimum allowed time of 1 second. The power monitoring through nvidia-smi provides the last measured power draw for the entire board, in watts, and the reading is accurate within +/- 5 watts. For the Tesla M2090, the idle power consumption is 42 W. For the Thread level implementation, the power consumption starts at around 100 W for small matrices and stabilizes around 150 W for matrices with sizes over 20x20. Power consumption for the Warp level implementation starts higher than the other implementations at 135 W for matrices 2x2, but becomes lower for matrices bigger than 16x16. The Thread Block level implementation has a more complex behavior. It starts at 120 W, higher than the Thread level but lower than the Warp level implementations. At 8x8 elements, it becomes the lowest power consuming implementation until 20x20 elements, where it crosses the power consumption of the Warp level implementation. Between 20x20 and 36x36 elements its power consumption is very near to the Thread level implementation. It then decreases until reaching the maximum dimensions manageable by the implementation. At 56x56 elements, there is a significant decrease in power consumption, correlated to the drop in performance from Figure 2.

Figure 5 presents the power consumption for the three different solver implementations on the Tesla K20. Idle power (19W) and maximum power (under 120W) consumed by the three implementations are significantly lower on the

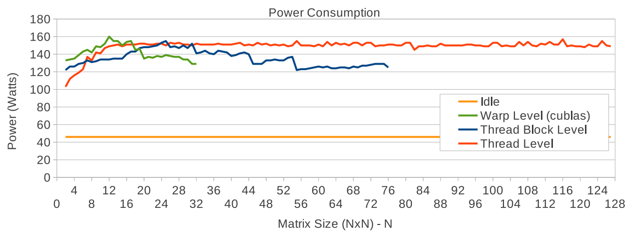


Fig. 4. Power consumption profile of the three implementations when solving 20,000 linear systems on the Tesla M2090 board while increasing the size of the matrices

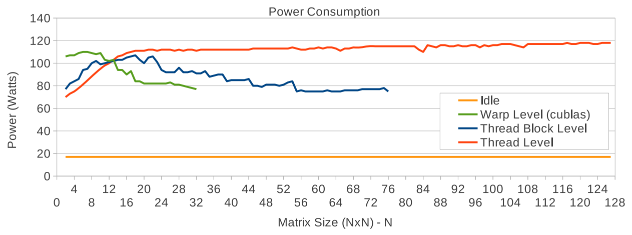


Fig. 5. Power consumption profile of the three implementations when solving 20,000 linear systems on the Tesla K20 board while increasing the size of the matrices

Tesla K20 than on the Tesla M2090. Power consumption of the Thread level implementation is higher than the others implementations starting from matrices of size 12x12. The Thread level implementation stabilizes around 110 W for sizes over 16x16. Again, as memory traffic increases, power consumption increases comparably, until the bandwidth limit of the application is reached. The Warp level implementation starts higher than all the other implementations, but at size 12x12 it becomes the lowest consuming solution. The Thread-block level implementation starts higher than the Thread level implementation, but follows its behavior until size 8x8. At the switch points for the three different kernels in this implementation, power consumption gets lower than the Warp level implementation, but it still keeps slightly increasing, until size of 24x24. It then decreases until the maximum size managed by the implementation.

5 Conclusions

In this paper we presented and evaluated three solvers for large amounts of small linear systems on GPUs. We proposed two solvers, one exploiting Thread level parallelism (one matrix per thread) and one exploiting Thread-block level parallelism (one matrix per Thread-block). The first is able to manage matrices (representing the linear systems) of size up to 128x128 elements. The second manages matrices up to 76x76 elements, because of the limitations connected to the available shared memory. We compared these implementations to a batched solver built with CUBLAS, which exploits Warp-level parallelism and can manage matrices up to 32x32. For each implementation, we evaluated performance and power consumption. We discussed the various performance trade-offs on the Tesla M2090 (Fermi) and the Tesla K20 (Kepler) GPU boards, showing how the Thread level implementation is more general (supports full pivoting for the LU decomposition and can scale to larger matrices) while the Thread-block level parallel solution is generally the fastest and consumes less power.

References

1. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Langou, J., Ltaief, H., Tomov, S.: Lu factorization for accelerator-based systems. In: AICCSA: 9th IEEE/ACS International Conference on Computer Systems and Applications, pp. 217–224 (December 2011)
2. Hammond, G., Lichtner, P., Lu, C., Mills, R.: Pflotran: Reactive flow and transport code for use on laptops to leadership-class supercomputers. In: Groundwater Reactive Transport Models. Bentham Sciene Publishers (2012)
3. Higham, N.: Gaussian elimination. Computational Statistics 3, 230–238 (2011)
4. Nidia corporation. Nidia CUBLAS Library, Version 5.0 (2012)
5. Nidia corporation. Nvidia CUDA c Programming Guide, Version 5.0 (2012)
6. Song, F., Tomov, S., Dongarra, J.: Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. In: ICS 2012: The 26th ACM International Conference on Supercomputing, pp. 365–376 (2012)
7. Tang, G., D'Azevedo, E.F., Zhang, F., Parker, J.C., Watson, D.B., Jardine, P.M.: Application of a hybrid MPI/OPENMP approach for parallel groundwater model calibration using multi-core computers. Computers & Geosciences 36, 1451–1460 (2010)
8. Tomov, S., Nath, R., Ltaief, H., Dongarra, J.: Dense linear algebra solvers for multicore with gpu accelerators. In: IPDPSW 2010: IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, pp. 1–8 (2010)
9. White, M., Oostrom, M.: STOMP Subsurface Transport Over Multiple Phase: User's Guide. Technical report, Pacific Northwest National Laboratory, Richland, WA, USA, PNNL-15782 (2006)
10. Yeh, G., Tripathi, V., Gwo, J., Cheng, H., Chend, J.-R.C., Salvage, K., Li, M., Fang, Y., Li, Y., Sun, J., Zhang, F., Siegel, M.: HYDROGEOCHEM: A coupled model of variably saturated flow, thermal transport, and reactive biogeochemical transport, on laptops to leadership-class supercomputers. In: Groundwater Reactive Transport Models. Bentham Science Publishers (2012)
11. Zhang, K., Wu, Y., Pruess, K.: User's Guide for TOUGH2-MP - A Massively Parallel Version of the TOUGH2 Code. Technical report, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, LBNL-315E (2008)

Optimizing 3D Convolutions for Wavelet Transforms on CPUs with SSE Units and GPUs

Brice Videau[1], Vania Marangozova-Martin[1],
Luigi Genovese[2], and Thierry Deutsch[2]

[1] Nanosim (LIG), France
first.last@imag.fr
[2] L_Sim (CEA - Grenoble), France
first.last@cea.fr

Abstract. Optimizing convolution operators is an important issue as they are used in numerous domains including electromagnetic computations, image processing and nanosimuations. In this paper we present our optimizations for 3D convolutions in the BigDFT nanosimulation software. We focus on processors with vector units and on GPU acceleration and experiment with several architectures. Exploiting the relation between algorithmic specifics and hardware architecture, we obtain performance gains of around x2 on CPU and up to x20 on GPU.

1 Introduction

Obtaining good application performance on a high performance computing (HPC) platform becomes a real challenge. Indeed, the HPC hardware landscape becomes very complex and each architecture brings its performance specifics. As a consequence, it is impossible to obtain satisfactory performances on different architectures with the same implementation and optimization of an algorithm.

In this paper we report on the joint work between the Nanosim team of the LIG laboratory and the L_Sim team of CEA-Grenoble. We study the implementation, code restructuring and optimization of the BigDFT software [5] which simulates the properties of future electronic materials. We focus on BigDFT's key components: three-dimensional (3D) convolutions.

The goal in the presented work is to study the performances and possible optimizations of 3D convolutions on two types of architectures: architectures based on CPUs with vector units and hybrid architectures containing GPUs. In both cases, we coordinate the specifics of the algorithms and the specifics of the hardware architecture. Our major optimization issue concerns data locality as different placements of data in the registers and in the cache may result in very different performance results. In the case of CPUs with vector units, we provide a pioneer work, as 3D convolutions with long filter size have never been vectorized. We explore the algorithmic opportunities, study the resulting performance results and show performance gains of around x2. An important contribution is the definition of locality patterns to be used for memory placement, as well as for computation. We conclude on the cost of development and the need of auto-tuning techniques. In the GPU case, we provided new implementations based on OpenCL which, by providing efficient cache management strategies, succeed in bringing a speedup of up to 20.

The remainder of the paper is organized as follows. Setion 2 presents our use case, BigDFT, and details 3D convolutions. Section 3 explains how our optimization work is different from related projects. Section 4 and Section 5 tackle optimizations on CPUs with vector units and on GPUs respectively. Finally, Section 6 concludes and outlines our future works.

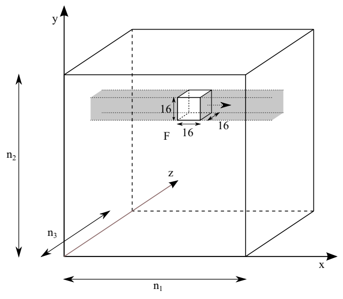
2 BigDFT and 3D Convolutions

The BigDFT application [1] provides a novel approach for electronic structure simulation (nanosimulation) based on the Daubechies wavelets formalism [5][13]. The code is HPC-oriented and hybrid as it uses MPI, OpenMP and GPU programming [6]. Characterized by its high precision, efficiency and flexibility, it has been chosen as an official benchmark for the EU Mont-Blanc project [2].

BigDFT defines nanosimulations in terms of 3D convolution operations. These convolutions consist in applying cubic filters to 3D zones. One such convolution, called MagicFilter, is defined by (1), and illustrated in Fig.1a.

$$\Psi(i_1, i_2, i_3) = \sum_{j_1, j_2, j_3 = -L}^{U} \omega_{j_1}\omega_{j_2}\omega_{j_3} in(i_1 + j_1, i_2 + j_2, i_3 + j_3) \ . \qquad (1)$$

The MagicFilter transforms the three-dimensional array in into an array Ψ. The dimensions of in and Ψ, n_1, n_2 and n_3, give the dimensions of the BigDFT simulation domain. To calculate the output values, the transform uses the ω_k values whose number $U + L + 1$ is equal to the order l of the Daubechie wavelet family. In the BigDFT case, l has been chosen by the physicists to be 16. When the indexes are outside bounds, the data is used in a circular manner (periodic boundary conditions).



(a) Overview

```
const int L = 7, U = 8;
double W[L+U+1] = {W0, W1, ... , W15};
for(i1 = 0; i1 < n1; i1++)
  for(i2 = 0; i2 < n2; i2++)
    for(i3 = 0; i3 < n3; i3++) {
      double temp = 0;
      for(j1 = -L; j1 <= U; j1++)
        for( j2 = -L; j2 <= U; j2++)
          for( j3 = -L; j3 <= U; j3++)
            temp += W[j1+L]*W[j2+L]*W[j3+L]*
                    in[i1+j1][i2+j2][i3+j3];
      psi[i1][i2][i3] = temp;
    }
```

(b) Direct Implementation

Fig. 1. A 3D Convolution Example

A direct implementation of this algorithm contains six nested loops (cf. Fig.1b). The three outer loops scan through the elements of the input array. The three inner loops apply the filter for the computation of element $\Psi(i_1, i_2, i_3)$ The direct algorithm incurs $n_1n_2n_3l^3$ reads, multiplications and additions, as well as $n_1n_2n_3$ writes.

What is interesting about this 3D convolution is that it is separable as it can be expressed as a sequence of three independent 1D convolutions of lesser complexity (cf.Fig. 2a).

$$\begin{aligned} f_1(i_1, i_2, i_3) &= \textstyle\sum_{j=-L}^{U} \omega_j s(i_1 + j, i_2, i_3) \\ f_2(i_1, i_2, i_3) &= \textstyle\sum_{j=-L}^{U} \omega_j f_1(i_1, i_2 + j, i_3) \\ \Psi_r(i_1, i_2, i_3) &= \textstyle\sum_{j=-L}^{U} \omega_j f_2(i_1, i_2, i_3 + j) \end{aligned}$$

(a) Three Separate 1D Convolutions

$$\begin{aligned} F_1(i_2, i_3, i_1) &= \textstyle\sum_{j=-L}^{U} \omega_j s(i_1 + j, i_2, i_3) \\ F_2(i_3, i_1, i_2) &= \textstyle\sum_{j=-L}^{U} \omega_j F_1(i_2 + j, i_3, i_1) \\ \Psi_r(i_1, i_2, i_3) &= \textstyle\sum_{j=-L}^{U} \omega_j F_2(i_3 + j, i_1, i_2) \end{aligned}$$

(b) Transposed 1D Convolutions

Fig. 2. Separated and Transposed 3D Convolutions

In order to optimize memory accesses and data locality, the convolutions may be transposed (cf. Fig. 2b). Indeed, a 1D non-transposed convolution needs $3n_1n_2n_3l$ reads, multiplications and additions, and $3n_1n_2n_3$ writes. However, if data is allocated in column-major order, the first loop will access elements sequentially, while the second and the third loops will need elements at distance of n_1 and of n_1n_2 respectively. Transposing the convolutions [7] prevents such memory jumps and ensures sequential memory access for the reads, the multiplications and the additions. Only writes keep a stride different than 1 (respectively n_2n_3, n_3n_1 and n_1n_2).

3 Related Work

From the algorithmic point of view, 3D convolutions may be optimized using the Discrete Fourier Transform (DFT) or using 1D convolutions and transpositions, as presented in the previous section. In the DFT approach, the 3D convolution is calculated by applying the Fourier Transform to each dimension [8]. It has been used, for example, in the context of the BlueGene machine [12]. The cost of this solution is, however, prohibitive in our case. Indeed, the complexity of the algorithm is $C \ln(n)$ with C depending on the chosen radixes. For powers of 2, C is commonly accepted as being $5/\ln(2)$. So, considering a 128 line length, the cost of the 1D convolution would be about 50 FLOP per element.

When optimizing 3D convolutions using 1D convolutions and transpositions, the possibilities are to consider 1D+2D or 1D+1D+1D. For the considered size of the MagicFilter (i.e. 16x16x16), methods working on 2D convolutions are not applicable as the data size is too big to favour data locality. For this reason, our solution is based on three 1D convolutions. What is different about our work is that we take into account the hardware architecture in our optimization techniques. Our experience has given as the first ideas for auto-tuning strategies.

From the platform point of view, there are many references considering 2D convolutions [4][15] but projects focusing on 3D convolutions are rare. For works considering 3D convolutions with vectorization, we can cite Intel [3] but the filter is very small (3x3x3) and the data is limited to 16bit data. Gaussian 3D filtering has also been ported to SSE [17], but in this case the filters are symmetric and the authors limit the use of buffers. As for 3D convolutions on GPU, Hopf and al. ([9]) use a 2D+1D method but for very small filter sizes.

4 Optimizing 3D Convolutions by Vectorization

After presenting an initial performance study, we discuss convolution vectorization and the obtained performance gains.

4.1 Preliminary Performance Evaluation

The test platform is a Lenovo D20 workstation featuring one quadri-core Intel Xeon X5550 CPU and 8 GiB of RAM. Hyper-threading and turbo boost are deactivated and the processor frequency is set to 2.67GHz. As the Nehalem architecture is capable of providing four double-precision operations per cycle, each core has a peak performance of 10.6 GFlop/s. The operating system is Ubuntu10.10 with 2.6.36 Linux kernel. The compilers used are Intel Compiler Suite version 11.1 and Gnu Compiler Collection version 4.4.5. The used gcc optimization option is `-O2`, as `-O3` proved harmful to performances. Performance counters are obtained using PAPI [11].

We have studied several versions of the 3D convolution algorithm. The first two are the straightforward convolution implementations without (`simple`) and with transposition (`simple_t`). `unrolled` and `unrolled_t` are two versions in which the nested loops for the `simple` and the `simple_t` algorithms are unrolled with a degree of 8 [18]. Finally, the `sse_t` version is the vectorized one.

The PAPI counter values for these versions are given in Table 1. We give the mean for 10 runs for convolving an array of 128x126x130 double precision numbers (16 MiB). We have considered the counters giving the total number of computation cycles (`TOT CYC`) and the number of executed instructions (`TOT INS`). We also present the cache-related counters reflecting the number of data cache accesses (`DCA`) and of data cache misses (`DCM`) for cache levels 1 and 2.

Table 1. Counter Results for different versions of 3D convolutions

version	TOT CYC	TOT INS	INS/CYC	L1 DCA	L1 DCM	L2 DCM	GFlop/s
simple	1500M	2300M	1.53	120M	13.4M	4.68M	0.357
simple_t	1480M	2470M	1.67	211M	7.54M	1.34M	0.361
unrolled	240M	401M	1.67	149M	4.72M	2.06M	2.23
unrolled_t	181M	386M	2.13	137M	2.48M	1.30M	2.96
sse_t	82.2M	175M	2.12	45.4M	3.26M	1.33M	6.52

The Gflop/s measure is computed using the formula $3 * 32 * n/t$. 3 gives the number of dimensions, 32 results from the length of the filter (16) and the number of operations per element (one addition and one multiplication), n is the number of elements and t is the computation time derived from `TOT CYC`.

The first point to note is that, by reducing the number of instructions by a factor of 6, *unrolling* reduces the number of computation cycles. The `unrolled` algorithm gains a factor of 6.25, while `unrolled_t` gains a factor of 8.17.

The second point is that transposed versions of the algorithm present better instruction throughput. This is especially true for the unrolled version ($2.23 \rightarrow 2.96$ i.e. +28%). This can be explained by comparing the data cache miss results: transposed versions incur half the L1 cache misses of the non transposed ones. For L2, this ratio is 3.5 for simple convolutions and 1.6 for unrolled ones.

However, even if the unrolled algorithms are more efficient than the simple ones, their performance achieves respectively only 21% and 28% of the peak performance of the core they use.

The total number of floating point operations needed to compute the whole 3D convolution is: $3 * 32 * n = 3 * 32 * 128 * 126 * 130 = 201M$. If we add to this result the number of data accesses (`L1 DCA`), in the case of the `unrolled_t` version we obtain: $201M + 137M = 338M$ instructions. If we consider the ratio between these values, it indicates that there has been poor register reuse as memory operations take about 2/3 of the arithmetic operations. If we compare this value to the one given in the table (`TOT INS` $386M$), this result leaves about $50M$ of operations that are not accounted for (eg. loop overheads).

4.2 Vectorization Approach

In this section, we present our vectorization approach in order to reduce the number of arithmetic operations even further and to optimize the number of memory accesses through efficient register reuse. We focus on vector instructions used in the x86 architecture, namely the SSE instruction set [16] and its extensions SSE2, SSE3 and SSE4. We have taken into account memory alignment, register pressure, register reuse and memory access patterns.

We have discarded the possibility to reorder data in memory. Indeed, each step of the convolution algorithm (cf. (2a)) needs only 32 arithmetic operations (1 multiplication and 1 addition for 16 elements) per element and data reordering is too costly. Reordering methods are typically useful in the case of general matrix multiply operations with complexity of $O(n)$ per element and with n big enough.

In order to store data and filter values using registers, we have worked by separating data (filters and input array) into 2-element vectors. We have adopted a solution with two filters as illustrated in Fig.3.

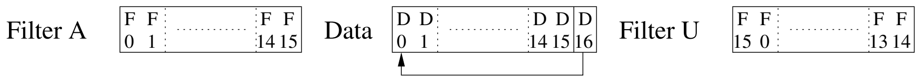


Fig. 3. Two-filter computation where data and filters are loaded into registers

In our first approach, we assume that data is also aligned on 16 byte boundaries and design a computation allowing the reuse of data registers.

To illustrate the idea, let us consider the computation of two successive data, R_0 and R_1, defined by (2) and (3). The D_0 appearing in the equation is the one shown in Fig.3 and aligned on a 16 byte boundary. $\otimes$ is the vectorized multiplication operation which obtains a two-element result by respectively multiplying the first and the second elements of the two-element vector operands.

It can be noted that seven of the eight two-element data vectors can be reused directly. Only the first one needs to be updated with D_{16}. If we compute several values in parallel, filter vectors can also be reused.

$$R_0 = \sum_{i=0}^{15} D_i F_i = V_{0_0} + V_{0_1} \text{ where } \boldsymbol{V_0} = \sum_{i=0}^{7} [D_{2i}, D_{2i+1}] \otimes [FA_{2i}, FA_{2i+1}] \quad (2)$$

$$\begin{aligned} R_1 &= \sum_{i=0}^{15} D_{i+1} F_i = V_{1_0} + V_{1_1} \text{where} \\ \boldsymbol{V_1} &= [D_{16}, D_1] \otimes [FU_0, FU_1] + \sum_{i=1}^{7} [D_{2i}, D_{2i+1}] \otimes [FU_{2i}, FU_{2i+1}] \end{aligned} \quad (3)$$

The register reuse is limited by the number of available registers. Indeed, we need one register per result, half of the registers for data and two registers for the filters. Given this repartition, computing 8 values in parallel consumes 14 registers (8 results, 2 filters, 4 data). As SSE units provide only 16 registers, computing more than 8 values in parallel might create register spilling. On the contrary, using less registers will diminish register reuse and increase the ratio between memory operations and arithmetic operations.

4.3 Performance Study of the Vectorization Approach

In order to experiment with the presented strategy, we have started without transposition (cf. Fig.2) and have generated several patterns for computing data values (cf. Fig.4). The different patterns compute from 2 to 12 values in one pass. For example, 1*2 computes R_0 and R_1, while 1*4 and 2*2 compute R_0, R_1, R_2 and R_3. The vectorization is done using intrinsic operations, the compiler being responsible for register allocation and management.

The benchmarks use arrays of 32 columns of 5040 elements. This size is a multiple of the pattern dimension and fits in the L3 cache of the processor.

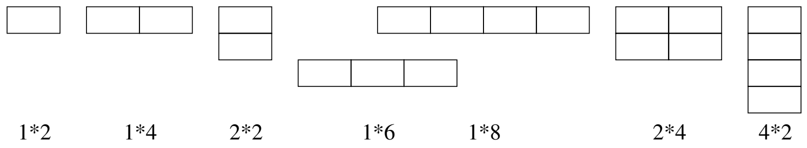


Fig. 4. Some Data Patterns Used for Computing R_0, R_1 ... R_{n-1}

The column size is much longer than what is used in BigDFT but allows the measure of asymptotic performances.

The mean results for 20 runs and for all patterns are presented in Table 2. What we see is that even small patterns present improved performances compared to the non vectorized versions. Indeed, the worst throughput is 7.45 Gflop/s, while the best throughput in Table 1 is 2,96 Gflop/s!

Table 2. Counter results for different versions of convolution kernels

Element computed	2	4	4	6	8	8	8	10	12
pattern column*line	1*2	1*4	2*2	1*6	1*8	2*4	4*2	1*10	1*12
GFlop/s	7.45	8.45	8.03	8.57	8.49	8.43	8.49	8.43	7.77

As the data patterns cannot be used directly for 3D convolutions but need transposition first, we have chosen among the ones inducing successive memory writes when transposed i.e 2*2, 2*4, 4*2. The 2*2 pattern has been discarded because of its inferior performance.

To calculate the transposition, we have simply used the available SSE operation for manipulating vectors. The 4*2 pattern proved faster than the 2*4 pattern (7.14 GFlops/s versus 6.32 GFlops/s). For this reason, it is used as a basis for the construction of the final vectorized convolution.

4.4 Designing the Final 3D Vectorized Convolution

In order to build the vectorized version of the BigDFT convolution, the chosen 2*4 pattern had to be adapted so as to take into account boundary conditions. In periodic conditions, column length is always a multiple of 2 and column number a multiple of 4, so alignment is not a concern. The first 4 and last 4 instantiations of the pattern have to be modified to load data from the end and the start of the column respectively. The 1D algorithm is then used 3 times to create the 3D version.

The performances of this implementation are compared to the previous ones in Table 1. The SSE version with transposition reaches 6.52 GFlop/s which is more than twice (factor x2.2) the performances of the unrolled and transposed version. Indeed, the generated code is more efficient as the total number of instructions (`TOT INS`) has been reduced by the same factor while instruction throughput is the same.

5 Optimizing 3D Convolutions on GPU

Our convolution optimization work has consisted in implementing from scratch and optimizing OpenCL [10] versions of 3D convolutions for both NVIDIA and AMD architectures. In the following we start by presenting the principles of the OpenCL model. We then discuss the algorithmic details for the GPU implementation and conclude with an analysis of the performance of our implementations.

5.1 OpenCL GPU Architecture

The GPU device in OpenCL is made of several address spaces and a set of multiprocessors. OpenCL is aimed at data parallel tasks and describes the computation in terms of workgroups composed of work-items. When executing an OpenCL function (also called *kernel*), work-items execute the same code.

The difference between work-items from different workgroups is the visibility of address spaces. The four address spaces are *global*, *local*, *private* and *constant*. The *global* address space is usually larger and with a higher latency than the *local* address space which is private to a workgroup. The first corresponds to the on-board RAM while the later corresponds to a user managed cache. The *private* memory corresponds to the registers of a work-item.

5.2 Convolution Implementation on GPU

In order to optimize GPU memory accesses, *global* memory accesses should be done in parallel. The easiest way to achieve this goal while transposing is to use a padded square buffer in *local* memory [14].

In BigDFT, data reads are coalesced and, as the GPU processes the last dimension first, the result is stored transposed in the buffer. As the filter is 16 elements long, each column of a 16*16 buffer needs 15 more elements to be convoluted. In order for threads to execute the same code, each work-item loads 2 elements in a 32*16 buffer padded to 33*16. Work-items then compute a filtered value in *private* memory. The calculated value is finally stored in the result buffer in *global* memory. Fig.5 presents an example of work items assignation for a 4*4 block processing a filter of length 5. Buffer is in column major order. Threads 0,i 1,i ... k,i are processed simultaneously.

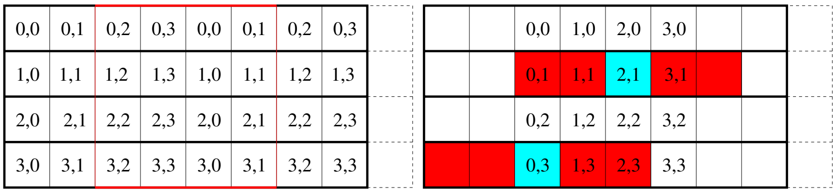


Fig. 5. Thread allocation: example of a 4*4 block and filter of length 5

It has to be noted that this allocation allows memory accesses to be free from bank conflicts. This is true not only during the transposition, but also during the computation of the filtered values. Indeed, consecutive threads access consecutive elements in the buffer. As using *constant* memory proved harmful to performances, filter values are directly inserted into the code.

On more complex kernels found in BigDFT the order of the filtering operations also had an impact that could not be neglected. Indeed, the order in which threads access the elements in the buffer has no real impact on performances, once the filter has been loaded in *local* memory. However, trying to use identical filter coefficients in a grouped manner proved beneficial.

From the GPU parallelism point of view, there is a set of N independent convolutions to be computed. Each of the lines of n elements to be transformed is split in chunks of size N_e. Each multiprocessor computes a group of N_ℓ different chunks and parallelizes the computation on its units. After the calculation of the convolution values, these $N_e N_\ell$ elements are copied in the corresponding part of the output array, which is transposed with relation to the input.

The size of the data fed to each block is identical and chosen to prevent block dependencies. When N and n are not multiples of N_ℓ and N_e, some data treated by different blocks may overlap. This fact has no double-counting effect since the overlap is reproduced also in the output array. Fig.6 shows the data distribution on the grid of blocks during the transposition.

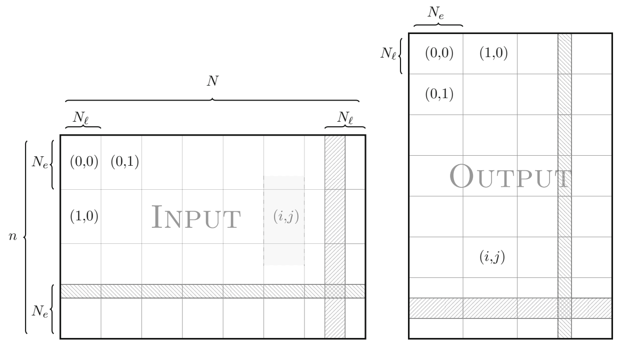


Fig. 6. Data Distribution for 1D Convolution+Transposition on the GPU. Input data (left panel) is ordered along the N-axis, while output data (right panel) is ordered in n-axis direction (cf. Section 2b). When executing GPU convolution kernel, each block of the execution grid (i,j) is associated to a set of N_ℓ (N-axis) times N_e (n-axis) elements. The filled patterns in the figure indicate the overlap region, i.e. data which are associated to more than one block. Behind the (i,j) label, in light gray, is indicated the portion of data which should be copied to the local memory to treat the data in the block, which contains also the buffers needed for computing the convolution.

The most recent AMD architecture (Radeon HD 69xx) is based on vector operations of length 2 in double precision. To account for this, the filter multiply has been vectorized.

5.3 Performance Evaluation of 3D Convolutions on GPU

GPU performance is evaluated on two test computers. Each runs an Ubuntu 11.04 with an unmodified 2.6.38-11 kernel, compiles with Intel 11.1 Compiler Suite and has a X5550 Xeon CPU with 4 cores and 8GiB of RAM. The setups differ only in their GPU and graphic drivers. The first one uses an NVIDIA TESLA C2070 using 270.40 drivers, while the second one has a RADEON HD6970 using 11.6 drivers.

The 3D convolution equals 3 applications of the 1D convolution. The problem dimensions are 124 x 17160 for 1D problems and 124 x 132 x 130 for 3D problems. Results are presented in table 3. For the used algorithms and problem sizes, we observe a slight performance advantage for AMD (10%). This difference is mainly due to the vectorization of the computation loop.

Table 3. Convolution Performance on NVIDIA and AMD GPU Architectures

Dimension	GPU	GFlop/s	GPU	GFlop/s
1D	NVIDIA	93	AMD	105
3D		91		101

5.4 Global BigDFT Performance

In this section we consider different configurations of the BigDFT software. We execute on two hybrid CPU+GPU architectures, varying the number of sequential processes (MPI) and the GPU architecture (NVIDIA or AMD). We focus on the execution times of the core loop, as initialization and finalization are roughly equivalent between NVIDIA and ATI (Table 4). It can be noted that due to OpenCL's portability, the same binary is used on both machines.

Table 4. Performance of several configurations of BigDFT, using CPUs + GPUs

MPI+NVIDIA/AMD	Execution Time (s)	Speedup
1 core	6020	1
4 cores	1660	3.6
1 core + NVIDIA	300	20
4 cores + NVIDIA	160	38
1 core + AMD	347	17
4 cores + AMD	197	30
(4 cores + NV) + (4 cores + AMD)	109	55

The first important thing to note is that BigDFT is not scaling very well in MPI, due to bandwidth limitation. In our case GPUs bring not only their computing power, but also their impressive memory bandwidth. The second interesting trend is that NVIDIA GPU offer better (though comparable) performances than AMD on this problem. The situation is reversed compared to the unit tests presented in the previous section. As BigDFT is synchronous between MPI processes, the hybrid case performance is limited by the AMD node. Speedup is of 1.8 comparing 4 MPI + AMD and (4 MPI + NV) + (4 MPI + AMD). In this case, GPUs increase by a factor of about 10 the performance of the respective node .

For both architectures (GPU and CPU with small vector engines) the only way to increase the computation over memory access ratio and alleviate the memory bandwidth problem would be to merge different convolutions. This would yield longer filters but reduce the number of transposition needed. Strategies presented here can be adapted to fit those conditions, especially since later GPU OpenCL devices have more *local memory*. For the CPU our algorithm is oblivious to the filter length.

6 Conclusions and Future Work

In this paper we focus on optimizing the basic computational blocks, 3D convolutions, in the BigDFT application. We have considered optimizing 3D convolutions for two types of architectures: CPU architectures using SSE units, as well as GPU-based architectures. We have optimized BigDFT on three platforms, featuring respectively CPUs with SSE units, AMD GPUs and NVIDIA GPUs. For the first, we have explored vectorization techniques, while for the second we have used intelligent cache management strategies. For both types of work, the major principle has been *data locality*. In the case of CPUs, we have defined access patterns that define different locality groups used to optimize memory accesses, as well as computation. For GPUs, we have defined blocks of data for aggressive cache usage. All techniques are reusable in other contexts different from convolutions.

A pending challenge, raised by the experiments on vectorized convolutions, is the remaining parameter space to explore in order to find the best convolution version. The different configurations are meaningful not only from the computational point of view but also for physicists. Indeed, filters of different sizes have different properties in terms of convergence and accuracy. For now, the filter size is fixed, but we could imagine changing this parameter in order to improve convergence speed and balance this gain with the computational performance impact.

Our future work will be to build a convolution generator that is able to produce a library of parametrized convolutions to be used for BigDFT's performance evaluation. This tool would allow a broader coverage of SSE versions as those are tedious to produce *by hand*, error-prone and difficult to maintain. The generator could also produce OpenCL kernels and allow better performance studies. These performance evaluations, studies and experimentations are a necessary step in understanding the behavior of HPC applications and in being able, one day, to produce a prediction performance model depending on the target architecture.

References

1. The BigDFT Scientific Application (2012), `http://inac.cea.fr/LSim/BigDFT/`
2. The Mont-Blanc Project (2012), `http://www.montblanc-project.eu`
3. Danovich, Z.: 16bit 3D Convolution: SSE4+OpenMP implementation on Penryn CPU, `http://software.intel.com/en-us/articles/16bit-3d-convolution-sse4openmp-implementation-on-penryn-cpu/`
4. Fialka, O., Cadik, M.: FFT and Convolution Performance in Image Filtering on GPU. In: Tenth International Conference on Information Visualization, IV 2006, pp. 609–614 (July 2006)

5. Genovese, L., Neelov, A., Goedecker, S., Deutsch, T., Ghasemi, S., Willand, A., Caliste, D., Zilberberg, O., Rayson, M., Bergman, A., et al.: Daubechies Wavelets as a Basis Set for Density Functional Pseudopotential Calculations. The Journal of Chemical Physics 129, 014109 (2008)
6. Genovese, L., Ospici, M., Deutsch, T., Méhaut, J., Neelov, A., Goedecker, S.: Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. The Journal of Chemical Physics 131, 034103 (2009)
7. Goedecker, S.: Rotating a three-dimensional array in an optimal position for vector processing: case study for a three-dimensional fast fourier transform. Computer Physics Communications 76(3), 294–300 (1993)
8. Goedecker, S., Boulet, M., Deutsch, T.: An efficient 3-dim fft for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes. Computer Physics Communications 154(2), 105–110 (2003)
9. Hopf, M., Ertl, T.: Accelerating 3D Convolution Using Graphics Hardware (Case Study). In: Proceedings of the Conference on Visualization 1999: Celebrating Ten Years, VIS 1999, pp. 471–474. IEEE Computer Society Press, Los Alamitos (1999), `http://dl.acm.org/citation.cfm?id=319351.319457`
10. Khronos OpenCL consortium: OpenCL: Open Computing Language, `http://www.khronos.org/opencl/`
11. Mucci, P., Browne, S., Deane, C., Ho, G.: PAPI: A Portable Interface to Hardware Performance Counters. In: Proc. Dept. of Defense HPCMP Users Group Conference, pp. 7–10 (1999)
12. Nukada, A., Hourai, Y., Nishida, A., Akiyama, Y.: High Performance 3D Convolution for Protein Docking on IBM Blue Gene. In: Stojmenovic, I., Thulasiram, R.K., Yang, L.T., Jia, W., Guo, M., de Mello, R.F. (eds.) ISPA 2007. LNCS, vol. 4742, pp. 958–969. Springer, Heidelberg (2007)
13. Nussbaumer, H.: Fast Fourier Transform and Convolution Algorithms, vol. 2. Springer, Berlin (1982)
14. nVidia, C.: OpenCL Best Practices Guide (2010), `http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf`
15. Podlozhnyuk, V.: Image Convolution with CUDA. NVIDIA Corporation White Paper 2097(3) (June 2007)
16. Thakkur, S., Huff, T.: Internet Streaming SIMD Extensions. Computer 32(12), 26–34 (1999)
17. Vaško, A., Šrámek, M.: Optimizing Gaussian Filtering of Volumetric Data Using SSE. Concurrency and Computation: Practice and Experience 23(1), 100–116 (2011), `http://dx.doi.org/10.1002/cpe.1620`
18. Wolf, M., Lam, M.: A Loop Transformation Theory and an Algorithm to Maximize Parallelism. IEEE Transactions on Parallel and Distributed Systems 2(4), 452–471 (1991)

GPUMAFIA: Efficient Subspace Clustering with MAFIA on GPUs

Andrew Adinetz[1],[2] Jiri Kraus[3], Jan Meinke[1], and Dirk Pleiter[1]
(NVIDIA Application Lab at Forschungszentrum Jülich)

[1] JSC, Forschungszentrum Jülich, 52425 Jülich, Germany
[2] Research Computing Center, Lomonosov Moscow State University
[3] NVIDIA GmbH, Germany

Abstract. Clustering, i.e., the identification of regions of similar objects in a multi-dimensional data set, is a standard method of data analytics with a large variety of applications. For high-dimensional data, subspace clustering can be used to find clusters among a certain subset of data point dimensions and alleviate the curse of dimensionality.

In this paper we focus on the MAFIA subspace clustering algorithm and on using GPUs to accelerate the algorithm. We first present a number of algorithmic changes and estimate their effect on computational complexity of the algorithm. These changes improve the computational complexity of the algorithm and accelerate the sequential version by 1–2 orders of magnitude on practical datasets while providing exactly the same output. We then present the GPU version of the algorithm, which for typical datasets provides a further 1–2 orders of magnitude speedup over a single CPU core or about an order of magnitude over a typical multi-core CPU. We believe that our faster implementation widens the applicability of MAFIA and subspace clustering.

1 Introduction

Cluster analysis is a valuable data mining tool. With high-dimensional data occurring in many real applications, traditional all-attribute clustering algorithms encounter problems. Often data points form a cluster only in some dimensions, called *significant dimensions*, while their coordinates in other dimensions show no correlation. Sets of significant dimensions may differ for different clusters. This is part of what is commonly referred to as the curse of dimensionality [1].

Subspace clustering algorithms attempt to find clusters that exist only in subsets of dimensions of the original data, i.e., in subspaces. As they discard insignificant dimensions, they are especially useful in applications where analysis of high-dimensional data is required. In biological research, subspace clustering is used to study gene expression, e.g., to find groups of genes with similar function or to find individuals with similar gene expression. Customer recommendation systems use subspace clustering to find groups of individuals with similar preferences. It is also used to find groups of thematically related documents [2].

Here we consider an application to Monte Carlo simulations of protein folding. Proteins are long chains of amino acids that usually adopt a unique three-dimensional shape. This shape is necessary to perform their particular function, e.g., enable an important chemical reaction at body temperature. How these chain molecules are able to fold reliably into these unique shapes (conformations) remains an open question. During simulations millions of conformations are generated and similar conformations have to be grouped into clusters to determine the relative sizes of the clusters. The relative weights relate directly to the free energy, an important physical quantity that tells us, which is the conformation most likely seen in experiments. If the largest clusters are of similar size, a protein might transition between these different states.

While distance-based clustering works, the complexity of the algorithm makes it infeasible to do an exhaustive clustering of all data points. Subspace clustering offers a chance to find the relevant clusters using a d-dimensional state vector that can be calculated with linear complexity. Nevertheless, these algorithms can be computationally expensive. For a d-dimensional dataset, there are $2^d - 1$ possible axis-parallel subspaces. This may result in long run-times. (In practice, the number of subspaces to consider is often much lower.)

Here we focus on MAFIA (Merging of Adaptive Finite IntervAls) [3], a subspace clustering algorithm which applies an adaptive grid method. This reduces the computational requirements while providing similar quality of cluster search. We improve MAFIA by, first, using a number of algorithmic techniques, and second, providing a GPU implementation.

The contributions of this paper are as follows:

- We present a number of algorithmic improvements to the MAFIA subspace clustering algorithm, which gives 1–2 orders-of-magnitude performance increase for the use cases considered here while producing the same output.
- We present a GPU port of MAFIA, which gives an additional 1–2 orders-of-magnitude improvement over a single CPU core, or an order-of-magnitude improvement over a typical multi-core CPU-only system. To our knowledge, this is the first implementation of a subspace clustering algorithm on GPU.
- We present performance analysis, which enables us to justify the algorithmic improvements and parallelization for CPU and GPU architectures.

This paper is organized as follows. Section 2 presents a brief overview of subspace clustering algorithms together with some performance data. It also provides a summary of existing GPU implementations of clustering algorithms. MAFIA is described in section 3 together with an analysis of the operation count. We present the details of algorithmic improvements and GPU implementation in section 4. Finally, we analyse and discuss performance improvements in section 5 and present our conclusions in section 6.

2 Related Work

Subspace clustering is a relatively new field of research. The first two algorithms for finding clusters in subspaces, CLIQUE [4] and PROCLUS [5] were proposed in

1998 and 1999, respectively. MAFIA [3] was introduced shortly thereafter. For a parallel verion, pMAFIA, see [6]. Alternative subspace clustering algorithms are: SeqClus [7], LCM-nCluster [8], Maxncluster [9], and DiSH (Detecting Subspace cluster Hierarchies) [10]. The latter scales super-linearly with the number of points. A good survey of existing subspace clustering algorithms can be found in [2] which, however, does not include a comparison of performance or quality. This has been done elsewhere. For example, in [11] MAFIA and FINDIT are compared. In 2004, both required around 10 minutes to process a 20-dimensional 4-million point set with 5 hidden 5-dimensional clusters. [12] indicates that run-times of SUBCLU can be on the order of several hours even for relatively small dataset. However, SUBCLU is also better at finding subspace clusters, compared to CLIQUE. In [9] Maxncluster is compared to MAFIA and STATPC. Another good survey of subspace clustering performance is [13], which, however, excludes MAFIA.

As clustering algorithms are generally computationally intensive and parallelizeable, they are prime candidates for implementation on GPUs. K-means is perhaps the most widely implemented GPU clustering algorithm [14, 15, 16]. CAMPAIGN, an open-source clustering library [17], comprises an implementation of K-means and other algorithms. Other clustering algorithms implemented on GPUs include fuzzy clustering [18, 19], multi-level clustering [20] and density-based clustering [21]. A distinguishing characteristic of MAFIA is that most of its time is spent in operations on sets of data points, as opposed to the points themselves. To the best of our knowledge, there is no publicly available GPU implementation of a subspace clustering algorithm.

3 Algorithm Analysis

MAFIA starts with breaking each dimension into bins, counting points in each bin, and building *windows* as part of an adaptive grid approach. Initially, each window is a *candidate dense unit* of dimensionality 1 (1-CDU). CDUs that are dense enough become *dense units* (DUs). MAFIA then builds CDUs of increasing dimensionality; an a-CDU is build by merging two $(a-1)$-DUs which are the same in $(a-2)$ dimensions. $(a-1)$-DUs which were not joined into a-DUs are then added to the list of *terminal DUs*. Finally, connected groups of DUs of the same dimensionality are merged into clusters. For a high-level pseudo-code version of the algorithm see Alg. 1. Note that the collections used are arrays, not sets, and may therefore contain duplicate elements.

The algorithm can be broken into three phases: the loop over the dimension of CDUs is the *middle phase*, and what precedes and follows are the *initial* and *final* phases, respectively. The main kernel of the initial phase is the histogram construction **histogram**. During the same phase also the adaptive grid is built (**windows**). The kernels of the middle phase are CDU generation (**gen**), CDU deduplication (**dedup**), finding dense CDUs or point counting (**pcount**), and check for unjoined DUs (**unjoin**). For **gen**, **dedup** and **unjoin**, an $O(N^2)$ algorithm is used, while for **pcount**, each point is just checked against bounds of

Algorithm 1. High-level pseudo-code for MAFIA algorithm

```
$n$ — number of points, $d$ — number of dimensions
$p_{ij}$ — point coordinates, $1 \leq i \leq n$, $1 \leq j \leq d$
$\alpha$ — threshold parameter, $N_b$ — min. #bins, $N_M$ — max. # windows, $N_{uw}$ — #windows for uniform dimensions
for $j = 1 \to d$ do
    $D_j \leftarrow \max_{1 \leq i \leq n} p_{ij} - \min_{1 \leq i \leq n} p_{ij}$
    $h_j \leftarrow histogram(p, j, N_b)$
    $W \leftarrow W \cup adaptiveGrid(h_j, N_{uw}, N_M)$
end for
$w \in W | t_w \leftarrow \frac{\alpha n(r_w - l_w)}{D_{j_w}}$
$CDUs \leftarrow \{\{w\} | w \in ws\}$, $a \leftarrow 1$
while $DUs \neq \emptyset \vee a = 1$ do
    if $a > 1$ then
        $CDUs \leftarrow \{u_1 \cup u_2 | (u_1, u_2) \in pairs(DUs) \wedge canMerge(u_1, u_2)\}$
    end if
    $CDUs \leftarrow dedup(CDUs)$                                    ▷ deduplication
    $u \in CDUs | ns_u \leftarrow \|\{i \in 1 \to n | \forall w \in u : l_w \leq p_{ij_w} < r_w\}\|$        ▷ point counting
    $newDUs \leftarrow \{u \in CDUs | \forall w \in u : ns_u \geq t_w\}$
    $termDUs \leftarrow termDUs \cup \{u \in DUs | \nexists u_1 \in newDUs : u \in u_1\}$ ▷ unjoined check
    $a \leftarrow a + 1$, $DUs \leftarrow newDUs$
end while
$G \leftarrow \{termDUs, \{(u_1, u_2) \in pairs(termDU) | haveCommonFace(u_1, u_2)\}\}$
$cs \leftarrow connectedComponents(graph)$
$c \in cs | inds_c \leftarrow \{i \in 1 \to n | p_i \in c\}$                        ▷ index lists
```

each window belonging to each CDU. The final phase consists of building the DU graph and finding connected components (**graph**), as well as building lists of points belonging to given clusters (**list**). The **graph** and **windows** kernels do not process large amounts of data, and are ignored in our performance analysis.

We now estimate the computational complexity of the kernels in a special case. We assume that the dataset consists of n d-dimensional points, from which a fraction f belongs to m clusters, all of dimensionality k. Furthermore, to simplify analysis, we assume that clusters are arranged in such a way that $(a-1)$-DUs belonging to different clusters do not merge, i.e., they do not have a common $(a-2)$-sub-DU. If the overlap does occur, the complexity of the middle phase becomes even higher, and so does the gain from using GPUs.

Table 1 lists the operation counts for each kernel for fixed DU dimensionality a and the total costs which is obtained by summing over $a = 1, \ldots, k$. We dropped components with lesser order-of-magnitude and used Stirling's approximation. When a-CDUs are being generated, there are $m\binom{k}{a-1}$ $(a-1)$-DUs. The number of operations to check any pair for merging is a, which gives the cost of generating a-CDUs. There are $m\binom{k}{a}$ a-CDUs generated. Each a-CDU is generated $a(a+1)/2$ times, as this is the number of $(a-1)$-DU pairs which merge into this a-CDU, which gives the cost of deduplication. Points must be counted for each a-CDU, and a dimensions must be checked for each point, which gives the cost of counting

Table 1. Operation count for fixed dimensionality a as well as the total costs

Kernel	**Costs for fixed** a	**Total costs**
histogram	—	$O(nd)$
gen	$\frac{a}{2}m\binom{k}{a-1}(m\binom{k}{a-1}-1)$	$O(m^2\sqrt{k}4^k)$
dedup	$\frac{a}{2}\frac{ma(a+1)}{2}\binom{k}{a}(\frac{ma(a+1)}{2}\binom{k}{a}-1)$	$O(m^2k^4\sqrt{k}4^k)$
pcount	$m\,a\,n\binom{k}{a}$	$m\,n\,k\,2^{k-1}$
unjoin	$am^2\binom{k}{a-1}\binom{k}{a}$	$O(m^2\sqrt{k}4^k)$
list	—	$O(nf)$

points. For unjoined check, each $(a-1)$-DU should be checked against all a-DUs, and the cost of a single check is a.

4 Optimization

We first introduce a number of algorithmic improvements. Kernel costs in Table 1 show that there are two possible performance bottlenecks:

- If the number of points is large and the cluster dimensionality is small then the **pcount** kernel will dominate.
- For small to average numbers of points and big cluster dimensionality, kernels with operation counts independent of the number of points, i.e., **gen**, **unjoin**, and, most importantly, **dedup** will start to dominate.

We first optimize the **dedup** kernel by replacing the originally used $O(N^2)$ algorithm by $O(N \log N)$ set deduplication, where the set is implemented as a tree. (N is the number of CDUs at current iteration.) CDU order is defined as lexicographic order of sequences of dimension and window numbers. We also merge the kernels **dedup** and **gen**: A newly generated CDU will be added to the set only if there has not been one previously generated.

We then consider the kernels **gen** and **unjoin**. For **gen**, we build a map from $(a-2)$-subsequences to lists of $(k-1)$-DUs containing that subsequence. A $(a-1)$-DU belongs to the list only if it contains the subsequence. For **unjoin**, we similarly build a set of possible $(a-1)$ subsequences of a-DUs and check each $(a-1)$-DU against that set. Both kernels then also have $O(N \log N)$ complexity.

The optimizations described are simple but, to the best of our knowledge, they have never been implemented so far. As will be shown in the next section the improvement can be large.

For **pcount**, no point index optimization is possible due to high data dimensionality. What MAFIA really needs to do is to compute the intersection of window point sets, and then calculate its cardinality. We use bit arrays to represent those sets, and store the sets of dense windows only. The intersection can now be computed using simple bit-wise and operations. The number of points is given by the number of enabled bits. The actual operation count thus reduces by

Table 2. Same as Table 1 but for improved kernels

Kernel	**Costs for fixed** a	**Total costs (optimized)**	**Total costs (unoptimized)**
bitarray	—	$O(mnk)$	—
gen	$(a-1)am\binom{k}{a-1}\log\{m\binom{k}{a-2}\}$	$O(m(k+\log m)k^2 2^k)$	$O(m^2\sqrt{k}4^k)$
dedup	$a\frac{ma(a+1)}{2}\binom{k}{a}\log\{m\binom{k}{a}\}$	$O(m(k+\log m)k^3 2^k)$	$O(m^2 k^4\sqrt{k}4^k)$
pcount	$\frac{man\binom{k}{a}}{32}$	$\frac{m\,n\,k}{64}2^k$	$\frac{m\,n\,k}{2}2^k$
unjoin	$2am(a-1)\binom{k}{a}\log\{m\binom{k}{a-1}\}$	$O(m(k+\log m)k^2 2^k)$	$O(m^2\sqrt{k}4^k)$

about 32× because now 1 operation taking 32-bit operands is sufficient to process 32 points. This also reduces memory bandwidth requirements while upfront costs for building the bit arrays are small.

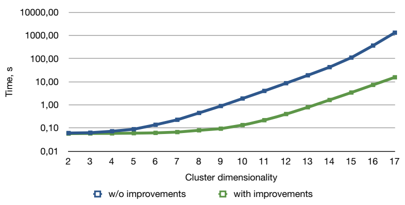
In Table 2 we show how the algorithmic improvements and the use of bit arrays improves the operation count. The costs of **gen**, **dedup** and **unjoin** still grow exponentially with k, but the exponent is reduced from 4 to 2, which makes the algorithm applicable to datasets with higher k. Costs of point counting are cut by a factor of 32×, which makes the algorithm applicable to larger datasets with the same cluster configuration. Note that as a side effect, the algorithmic improvements also reduced dependency on the number of clusters of **gen**, **dedup** and **unjoin** kernels from m^2 to $m \log m$, which is beneficial for real-world applications with datasets containing many clusters.

We now consider parallelization and porting of the most performance critical kernels to the GPU. For CPU parallelization we use OpenMP, GPU kernels are implemented using CUDA.

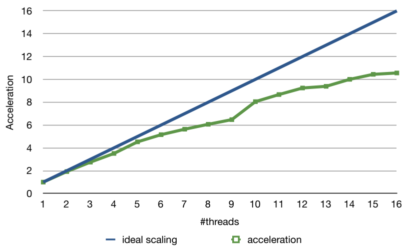
In case of the kernel **histogram** we parallelize both the dimension and point loops. For the latter, CPU threads compute private histograms, which are then summed up. On the GPU shared-memory atomics are used. (Further speed-up may be obtained using warp-synchronous non-atomics operations.)

pcount is a doubly nested loop, the outer over the CDUs and the inner over bit array words. On the CPU both are parallelized. On the GPU, the loops are mapped into different dimensions of the CUDA thread-block grid. Each thread adds up points from several words (128 in the current implementation) of bit arrays using `__popc()`, a CUDA built-in function, to count bits in each word, and global-memory atomics to compute the final point count. We also implemented precomputing of bit array indices in shared memory, and allocated bit arrays using `cudaMallocPitch()`, which cumulatively resulted in a 73% performance improvement over the initial GPU version.

bitarray is also a doubly nested loop, the outer over windows and the inner over words or points. Parallelization is thus similar as for **pcount**. On the GPU global-memory atomics are used to set individual bits. We found the memory access pattern to be better for the point-per-thread approach than for using a separate thread for each word.



(a) Speed-up due to algorithmic improvements as a function of the cluster dimensionality k.



(b) Scaling of MAFIA as a function of the number of OpenMP threads.

Fig. 1. CPU-only implementation of MAFIA

5 Performance Evaluation and Discussion

We implemented MAFIA as a standalone application. For benchmarking we used a dual-socket server comprising 2 8-core Intel Xeon E5-X2670 CPUs running at 2.60 GHz, plus an NVIDIA Tesla K20X (Kepler) GPU. For the experiments, hyper-threading and frequency scaling on CPU were turned off, and ECC was turned on on the GPU.

We first conducted a CPU-only single-core test to evaluate the algorithmic improvements. For this, we generated a series of datasets, each containing 10^5 30-dimensional points and a single embedded cluster, whose dimensionality k varied from 3 to 17. The results of the evaluation are shown in Fig. 1a. For small k, the cost of building the histograms dominates and optimization has little effect. For increasing k the algorithmic improvements start to have an increasingly large effect when the kernels **pcount** and later **dedup** start to dominate execution time. For $k = 17$ we observe almost two orders of magnitude speed-up. For further discussion, we consider only the version with all algorithmic improvements applied.

Fig. 1b shows scaling of MAFIA as a function of the number of OpenMP threads for 10^5 10-dimensional points and a single cluster with $k = 10$. We observe a large deviation from perfect scaling for increasing number of threads. Performance improvement does not saturate before all the cores are used.

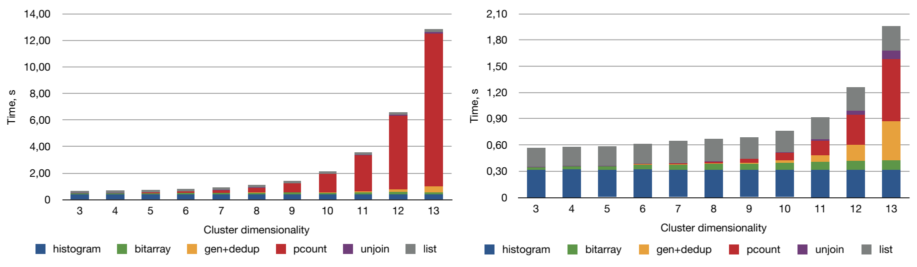
To investigate acceleration on the GPU as well as CPU parallelization we generated datasets comprising 10^7 20-dimensional points and 3 clusters, whose dimensionality k varied from 3 to 16. The different clusters did not intersect in any dimension, which ensures their successful detection by MAFIA. In Fig. 2a and 2b we plot the execution time of each kernel as a function of k for the CPU-only and the GPU-accelerated version, respectively. Runtime of MAFIA grows exponentially with k in all cases, approximately doubling for each successive k, which agrees with operation counts in Table 2. For small k, initial and final phases take significant part of the overall computing time. For higher values of k, the exponentially-growing middle phase, and most importantly for this parameter combination, point counting, dominate execution time. For CPU, **pcount** dominates clearly. With k further increasing, **gen**, **dedup** and **unjoin** start to play a greater role for both architectures, because their operation count grows as $k^4 2^k$. For GPU, times spent in **pcount** and **gen** + **dedup** are already comparable even for mid-range k, as only the former kernel has been ported to GPU. Both CPU and GPU parallelization give considerable performance improvement over sequential version.

The **histogram** part for GPU time breakup mostly consists of transferring initial data to device. As actual histogram computation takes only a small fraction of the **histogram** time, it is not possible to hide data transfer.

For the **pcount** kernel, as the number of memory reads is known exactly, we can estimate effective memory bandwidth achieved by the kernel. For K20X, this is 368 GB/s, much higher than the specification value (250 GB/s) and Stream benchmark value (180 GB/s). However, the kernel is still memory-bound, and the high bandwidth achieved is due to sharing of the same windows, and therefore same bit arrays, between neighboring CDUs, which enables caching to take effect.

Overall execution times are given in Fig. 3. **seq** stands for sequential version, **par4**, **par8** and **par16** are 4-, 8- and 16-thread parallel versions, respectively, and **k20x** is the GPU version. Accelerations for parallel vs. sequential CPU are given in Fig. 4a, while Fig. 4b gives acceleration of GPU vs. sequential CPU version. The GPU-accelerated version does outperform the parallel CPU-only implementation in all the cases. For small k the acceleration is small due to the time needed to transfer data to the GPU. For larger k, as the relative contribution of **pcount** grows, so does the acceleration, peaking at $7\times$ for $k = 14$. As not all systems have 16 CPU cores, we believe that for a typical system, GPU will give at least an order of magnitude acceleration over the parallel CPU version. When compared to a single CPU core, GPU gives more than two orders of magnitude acceleration. However, for $k > 15$ this value starts decreasing, due to increasing role of sequential **gen** + **dedup** kernels.

To test the applicability of MAFIA to protein folding simulations, we used data from a parallel tempering Monte Carlo simulation of the last 16 residues of protein G. This simulation produced 160000 independent conformations. For ease of



(a) 12 x86 CPU cores (b) Kepler K20X GPU

Fig. 2. MAFIA execution time breakdowns in various settings

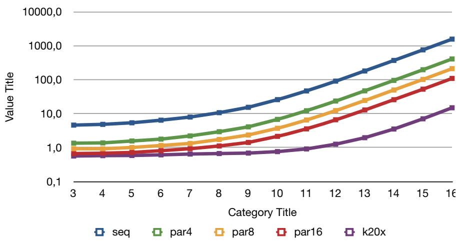
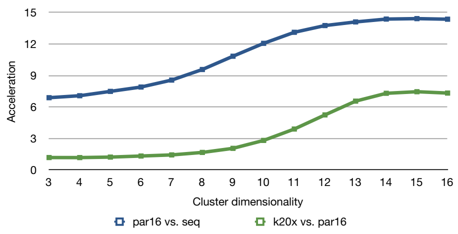
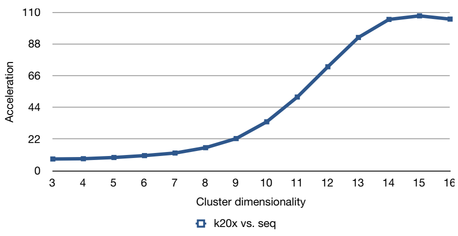


Fig. 3. MAFIA execution times in different settings



(a) 16 vs. 1 CPU threads, K20X vs. 16 CPU threads



(b) K20X vs. 1 CPU core

Fig. 4. MAFIA acceleration in different settings

visualization, we take only 4 properties of the conformation to form the state vector: the temperature at which this confirmation was found, the energy of the conformation, the fraction of residues that is part of a strand called the strand content, and the root-mean-square deviation (RMSD) to the native conformation. This data set is not meant to stress the performance of the algorithm in terms of the run-time (approximately 1 s). Rather, comparing expected and obtained results is an additional check of both algorithm and implementation. Furthermore, it allows to test how the parameters need to be adapted to deal with a real world data. Finally, in combination with our analysis of the algorithm's complexity the example demonstrates that datasets of realistic size could be processed in just $O(1)$ minutes in case of larger cluster dimensionality $k \simeq 10$.

Figure 5 shows the clusters that we obtained using 15 windows and $\alpha = 0.075$. The density threshold needs to be so small to detect the low energy clusters. At $\alpha = 1.5$ MAFIA didn't detect any clusters. We chose 15 windows since there are 15 discrete values for the strand content.

MAFIA identified several low-energy as well as several high-energy clusters. While the separation of structures into different clusters is not obvious, the fact that we retain the original axes makes interpretation of the clusters easier than interpreting clusters obtained from principal component analysis, for example.

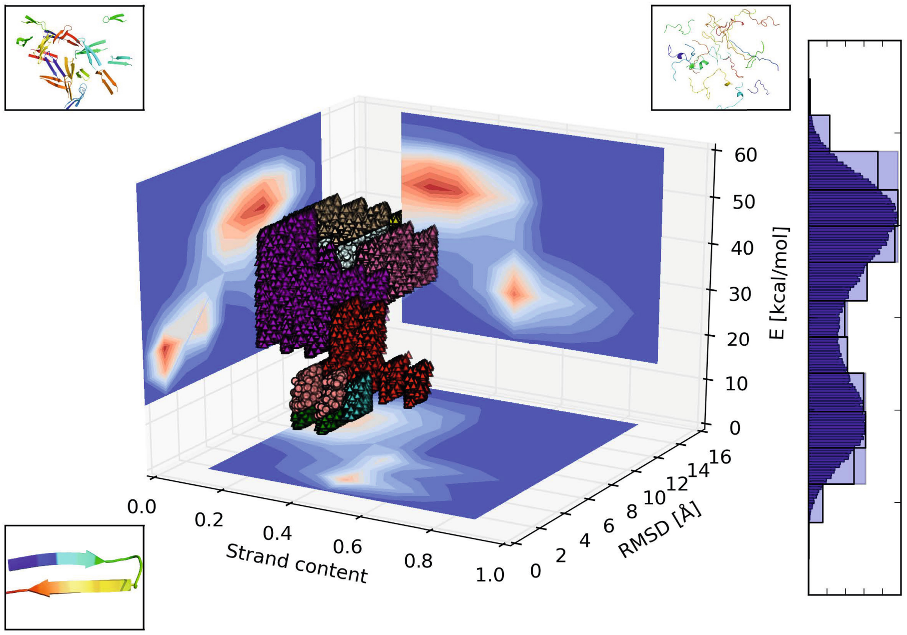


Fig. 5. Clusters of conformations from a Monte Carlo protein folding simulation in the S-R-E subspace including a low-energy and a high-energy 4D cluster (spheres). The images in the corners of the figure show the reference structure (lower left corner) and a random selection of conformations from the low-energy (upper left corner) and the high-energy (upper right corner) 4D clusters. The background contour plots show the 2D histograms in the corresponding planes. On the right is the 1D histogram of energy showing the initial fine bins, the 15 initial windows (lines) and the final merged windows (light blue filled bars).

6 Conclusions

In this paper, we studied porting MAFIA to GPUs. We first performed a number of algorithmic improvements, and then developed a GPU implementation. The algorithmic changes resulted in an almost two-orders-of-magnitude improvement. Porting to GPU accelerated the application by another order of magnitude. Our results put MAFIA subspace clustering well within the limits of interactivity even for large datasets with moderate cluster dimensionality (≤ 15).

Our MAFIA implementation can still be improved. The kernels **gen**, **dedup** and **unjoin** can dominate execution time if the number of points is small. Thus, parallelizing them should be considered, though it might be difficult, as they all access a single set for reading and writing. Porting them to a GPU may also be considered, though this is an even more difficult task. For datasets with larger number of points, parallelization across multiple GPUs or even across cluster nodes is to be considered; as **pcount** performs only per-point operations, this should be simple.

Also, as the run-time of the algorithm is significantly improved, I/O, and more specifically, converting large input text files into data point arrays, which is part of many work-flows involving MAFIA, becomes a bottleneck. One way to remove it is to accelerate string-to-double parsing on GPU, and do this in pipelined fashion. This would enable hiding initial data transfer to GPU, and improve the overall performance without changing the current workflow. Restructuring the implementation into a library accepting points from CPU or GPU memory rather than reading them from a file is another possibility for performance improvement.

Acknowledgements. We would like to thank Wilhelm Homberg for discussions and support of this work.

References

[1] Bellman, R.: Dynamic Programming (Dover Books on Computer Science). Dover Publications (2003)
[2] Kriegel, H.P., Kröger, P., Zimek, A.: Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. ACM Trans. Knowl. Discov. Data 3(1), 1:1–1:58 (2009)
[3] Nagesh, H.S.: High Performance Subspace Clustering for Massive Data Sets. Master's thesis (1999)
[4] Agrawal, R., Gehrke, J., Gunopulos, D., Raghavan, P.: Automatic subspace clustering of high dimensional data for data mining applications. SIGMOD Rec. 27(2), 94–105 (1998)
[5] Aggarwal, C.C., Wolf, J.L., Yu, P.S., Procopiuc, C., Park, J.S.: Fast algorithms for projected clustering. SIGMOD Rec. 28(2), 61–72 (1999)
[6] Nagesh, H., Goil, S., Choudhary, A.: Parallel Algorithms for Clustering High-Dimensional Large-Scale Datasets. Kluwer (2001)
[7] Wang, H., Chu, F., Fan, W., Yu, P.S., Pei, J.: A fast algorithm for subspace clustering by pattern similarity. In: Proceedings of the 16th SSDBM, pp. 51–62 (2004)
[8] Liu, G., Li, J., Sim, K., Wong, L.: Distance based subspace clustering with flexible dimension partitioning. In: IEEE 23rd International Conference on Data Engineering, ICDE 2007, pp. 1250–1254 (April 2007)
[9] Liu, G., Sim, K., Li, J., Wong, L.: Efficient mining of distance-based subspace clusters. Statistical Analysis and Data Mining 2(5-6), 427–444 (2009)
[10] Achtert, E., Böhm, C., Kriegel, H.-P., Kröger, P., Müller-Gorman, I., Zimek, A.: Detection and visualization of subspace cluster hierarchies. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 152–163. Springer, Heidelberg (2007)
[11] Parsons, L.: Evaluating subspace clustering algorithms. In: Workshop on Clustering High Dimensional Data and its Applications, SIAM International Conference on Data Mining (SDM 2004), pp. 48–56 (2004)
[12] Kröger, P., Kriegel, H.P., Kailing, K.: Density-Connected Subspace Clustering for High-Dimensional Data. In: SDM (2004)

[13] Müller, E., Günnemann, S., Assent, I., Seidl, T.: Evaluating clustering in subspace projections of high dimensional data. Proc. VLDB Endow. 2(1), 1270–1281 (2009)
[14] Cao, F., Tung, A.K.H., Zhou, A.: Scalable clustering using graphics processors. In: Yu, J.X., Kitsuregawa, M., Leong, H.-V. (eds.) WAIM 2006. LNCS, vol. 4016, pp. 372–384. Springer, Heidelberg (2006)
[15] Wu, R., Zhang, B., Hsu, M.: Clustering billions of data points using GPUs. In: UCHPC-MAW 2009, pp. 1–6. ACM, New York (2009)
[16] Hong-Tao, B., Li-li, H., Dan-Tong, O., Zhan-Shan, L., He, L.: K-Means on Commodity GPUs with CUDA. In: 2009 WRI World Congress on Computer Science and Information Engineering, March 31-April 2, vol. 3, pp. 651–655 (2009)
[17] Kohlhoff, K.J., Sosnick, M.H., Hsu, W.T., Pande, V.S., Altman, R.B.: CAMPAIGN: An open-source Library of GPU-accelerated Data Clustering Algorithms. Bioinformatics (2011)
[18] Kim, S., Wunsch, D.: A GPU based Parallel Hierarchical Fuzzy ART clustering. In: The 2011 International Joint Conference on Neural Networks (IJCNN), July 31-August 5, pp. 2778–2782 (2011)
[19] Anderson, D., Luke, R., Keller, J.: Speedup of Fuzzy Clustering Through Stream Processing on Graphics Processing Units. IEEE Transactions on Fuzzy Systems 16(4), 1101–1106 (2008)
[20] Chiosa, I., Kolb, A.: GPU-Based Multilevel Clustering. IEEE Transactions on Visualization and Computer Graphics 17(2), 132–145 (2011)
[21] Böhm, C., Noll, R., Plant, C., Wackersreuther, B.: Density-based clustering using graphics processors. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, pp. 661–670. ACM, New York (2009)

GPU Accelerated Maximum Cardinality Matching Algorithms for Bipartite Graphs

Mehmet Deveci[1,2], Kamer Kaya[1], Bora Uçar[3], and Ümit V. Çatalyürek[1,4]

[1] Dept. of Biomedical Informatics, The Ohio State University
({mdeveci,kamer,umit}@bmi.osu.edu)
[2] Dept. of Computer Science & Engineering, The Ohio State University
[3] CNRS and LIP, ENS Lyon, France (bora.ucar@ens-lyon.fr)
[4] Dept. of Electrical & Computer Engineering, The Ohio State University

Abstract. We design, implement, and evaluate GPU-based algorithms for the maximum cardinality matching problem in bipartite graphs. Such algorithms have a variety of applications in computer science, scientific computing, bioinformatics, and other areas. To the best of our knowledge, ours is the first study which focuses on the GPU implementation of the maximum cardinality matching algorithms. We compare the proposed algorithms with serial and multicore implementations from the literature on a large set of real-life problems where in majority of the cases one of our GPU-accelerated algorithms is demonstrated to be faster than both the sequential and multicore implementations.

Keywords: GPU, maximum cardinality matchings, bipartite graphs, breadth-first search.

1 Introduction

Bipartite graph matching is one of the fundamental problems in graph theory and combinatorial optimization. The problem asks for a maximum set of vertex disjoint edges in a given bipartite graph. It has many applications in a variety of fields such as image processing [18], chemical structure analysis [16], and bioinformatics [2] (see also another two discussed by Burkard et al. [4, Section 3.8]). Our motivating application lies in solving sparse linear systems of equations, as algorithms for computing a maximum cardinality bipartite matching are run routinely in the related solvers. In this setting, bipartite matching algorithms are used to see if the associated coefficient matrix is reducible; if so, substantial savings in computational requirements can be achieved [7, Chapter 6].

Achieving good parallel performance on graph algorithms is challenging: they are memory bounded; there are poor localities of the memory accesses; and the dependencies among the computations are irregular. Algorithms for the matching problem are no exception. There have been recent studies that aim to improve the performance of matching algorithms on multicore and manycore architectures. For example, Vasconcelos and Rosenhahn [19] propose a GPU implementation of an algorithm for the maximum weighted matching problem on bipartite graphs.

Fagginger Auer and Bisseling [10] study an implementation of a greedy graph matching on GPU. Halappanavar et al. [13] also study approximate matching on GPU. Çatalyürek et al. [5] propose different greedy graph matching algorithms for multicore architectures. Azad et al. [1] introduce several multicore implementations of maximum cardinality matching algorithms on bipartite graphs.

We propose GPU implementations of two maximum cardinality matching algorithms. We analyze their performance and employ further improvements. We thoroughly evaluate their performance with a rigorous set of experiments on many bipartite graphs from different applications. The experimental results conclude that one of the proposed GPU-based implementation is faster than its existing multicore counterparts.

The rest of this paper is organized as follows. The background material, some related work, and a summary of contributions are presented in Section 2. Section 3 describes the proposed GPU algorithms. The comparison of the proposed GPU-based implementations with the existing sequential and multicore implementations is given in Section 4. Section 5 concludes the paper.

2 Background and Contributions

A bipartite graph $G = (V_1 \cup V_2, E)$ consists of a set of vertices $V_1 \cup V_2$ where $V_1 \cap V_2 = \emptyset$, and a set of edges E such that for each edge, one of the endpoints is in V_1 and other is in V_2. Since our motivation lies in the sparse matrix domain, we will refer to the vertices in the two classes as row and column vertices.

A matching $\mathcal{M}$ in a graph G is a subset of edges E where a vertex in $V_1 \cup V_2$ is in at most one edge in $\mathcal{M}$. Given a matching $\mathcal{M}$, a vertex v is said to be matched by $\mathcal{M}$ if v is in an edge of $\mathcal{M}$, otherwise v is called unmatched. The cardinality of a matching $\mathcal{M}$, denoted by $|\mathcal{M}|$, is the number of edges in $\mathcal{M}$. A matching $\mathcal{M}$ is called maximum, if no other matching $\mathcal{M}'$ with $|\mathcal{M}'| > |\mathcal{M}|$ exists. For a matching $\mathcal{M}$, a path $\mathcal{P}$ in G is called an $\mathcal{M}$-alternating if its edges are alternately in $\mathcal{M}$ and not in $\mathcal{M}$. An $\mathcal{M}$-alternating path $\mathcal{P}$ is called M-augmenting if the start and end vertices of $\mathcal{P}$ are both unmatched.

There are three main classes of algorithms for finding the maximum cardinality matchings in bipartite graphs. The first class of algorithms is based on augmenting paths (see a detailed summary by Duff et al. [8]). Push-relabel-based algorithms form a second class [12]. A third class, pseudoflow algorithms, is based on a more recent work [14]. There are $\mathcal{O}(\sqrt{n}\tau)$ algorithms in the first two classes (e.g., Hopcroft-Karp algorithm [15] and a variant of the push relabel algorithm [11]), where n is the number of vertices and τ is the number of edges in the given bipartite graph. This is asymptotically best bound for practical algorithms. Most of the other known algorithms in the first two classes and the ones in the third class have the runtime complexity of $\mathcal{O}(n\tau)$. These three classes of algorithms are described and compared in a recent study [17]. It has been demonstrated experimentally that the champions of the first two families are comparable in performance and better than that of the third family. Since we investigate GPU acceleration of augmenting-path-based algorithms, a brief description of them is given below (the reader is invited to two recent papers [8,17] and the original resources cited in those papers for other algorithms).

Augmenting-path-based algorithms follow the same common pattern: given an initial matching $\mathcal{M}$ (possibly empty), they search for an $\mathcal{M}$-augmenting path $\mathcal{P}$. If none exists, then $\mathcal{M}$ is maximum by a theorem of Berge [3]. Otherwise, $\mathcal{P}$ is used to increase the cardinality of $\mathcal{M}$ by setting $\mathcal{M} = \mathcal{M} \oplus E(\mathcal{P})$ where $E(\mathcal{P})$ is the edge set of the path $\mathcal{P}$, and $\mathcal{M} \oplus E(\mathcal{P}) = (\mathcal{M} \cup E(\mathcal{P})) \setminus (\mathcal{M} \cap E(\mathcal{P}))$ is the symmetric difference. This inverts the membership in $\mathcal{M}$ for all edges of $\mathcal{P}$. Since both the first and the last edges of $\mathcal{P}$ were unmatched in $\mathcal{M}$, we have $|\mathcal{M} \oplus E(\mathcal{P})| = |\mathcal{M}| + 1$. The augmenting-path-based algorithms differ in the way the augmenting paths are found and the associated augmentations are realized. They mainly use either breadth-first-search (BFS), or depth-first-search (DFS), or a combination of them to locate and perform the augmenting paths.

Multicore counterparts of a number of augmenting-path based algorithms are proposed in a recent work [1]. The parallelization of these algorithms is achieved by using atomic operations at BFS and/or DFS steps of the algorithm. Although atomic operations might not harm the performance on a multicore machine, they might cause a significant performance degradation on a GPU due to the possible serialization of very large number of concurrent thread executions.

As a reasonably efficient DFS is not feasible with GPUs, we accelerate two BFS-based algorithms, called HK [15] and HKDW [9]. HK has the best known worst-case runtime complexity of $O(\sqrt{n}\tau)$ for a bipartite graph with n vertices and τ edges. HKDW is a variant of HK and incorporates techniques to improve the practical runtime while having the same time complexity. Both of these algorithms use BFS to locate the shortest augmenting paths from unmatched columns, and then use DFS-based searches restricted to a certain part of the input graph to augment along a maximal set of disjoint augmenting paths. HKDW performs another set of DFS-based searches to augment using the remaining unmatched rows. As is clear, the DFS-based searches will be a big obstacle to achieve efficiency. In order to overcome this hurdle, we propose a scheme which alternates the edges of a number of augmenting paths with a parallel scheme that resembles to a breadth expansion in BFS. The proposed scheme offers a high degree of parallelism but does not guarantee a maximal set of augmentations, potentially increasing the worst case time complexity to $O(n\tau)$ on a sequential machine. In other words, we trade theoretical worst case time complexity with a higher degree of parallelism to achieve better practical runtime with a GPU.

3 Methods

We propose two GPU-based algorithms which find the augmenting paths via BFS, speculatively perform some of them, and fix any inconsistencies that can be resulting from speculative augmentations. The proposed algorithms exploit the GPU's vast number of thread parallelisms by assigning each vertex to a thread. Then, the threads concurrently perform independent operations for vertices in each kernel call, even though actual work is done for a portion of the vertices.

Therefore, the GPU algorithm differs from a multi-core algorithm in which a shared data structure is used with atomic operations.

The overall structure of the first GPU-based algorithm is given in Algorithm 1, APsB. It largely follows the common structure of most of the existing sequential algorithms and corresponds to HK. It performs a combined BFS starting from all unmatched columns to find unmatched rows, thus locating augmenting paths. Some of those augmentations are then realized using a function called Alternate (will be described later). The parallelism is exploited inside the InitBfsArray, BFS, Alternate, and FixMatching functions. Algorithm 1 is given the adjacency list of the bipartite graph with its number of rows and columns. Any prior matching is given in *rmatch* and *cmatch* arrays as follows: $rmatch[r] = c$ and $cmatch[c] = r$, if the row r is matched to the column c; $rmatch[r] = -1$, if r is unmatched; $cmatch[c] = -1$, if c is unmatched.

Algorithm 1: Shortest augmenting paths (APsB)

```
   Data: $cxadj, cadj, nc, nr, rmatch, cmatch$
1  $augmenting\_path\_found \leftarrow$ true;
2  while $augmenting\_path\_found$ do
3      $bfs\_level \leftarrow L_0$;
4      InitBfsArray($bfs\_array, cmatch, L_0$);
5      $vertex\_inserted \leftarrow$ true;
6      while $vertex\_inserted$ do
7          $predecessor \leftarrow$Bfs($bfs\_level, bfs\_array, cxadj, cadj, nc, rmatch,$
8                          $vertex\_inserted, augmenting\_path\_found$);
9          if $augmenting\_path\_found$ then
10             $break$;
11         $bfs\_level \leftarrow bfs\_level + 1$;
12     $\langle cmatch, rmatch\rangle \leftarrow$ Alternate ($cmatch, rmatch, nc, predecessor$);
13     $\langle cmatch, rmatch\rangle \leftarrow$ FixMatching ($cmatch, rmatch$);
```

The outer loop of Algorithm 1 iterates until no more augmenting paths are found, thereby guaranteeing a maximum matching. The inner loop is responsible from completing the breadth-first-search of the augmenting paths. A single iteration of this loop corresponds to a level of BFS. The inner loop iterates until all shortest augmenting paths are found. Then, the edges in these shortest augmenting paths are alternated inside Alternate function. Unlike the sequential HK algorithm, APsB does not find a maximal set of augmenting paths.

By removing the lines 9 and 10 of Algorithm 1, another matching algorithm is obtained. This method will continue with the BFSs until all possible unmatched rows are found; it can be therefore considered as the GPU implementation of the HKDW algorithm. This variant is called APFB.

We propose two implementations of the BFS kernel. Algorithm 2 is the first one. The BFS kernel is responsible from a single level BFS expansion. That is, it takes the set of vertices at a BFS level and adds the union of the unvisited neighbors of those vertices as the next level of vertices. Initially, the input bfs_array

Algorithm 2: BFS Kernel Function-1 (GPUBFS)

```
Data: $bfs\_level, bfs\_array, cxadj, cadj, nc, rmatch,$
      $vertex\_inserted, augmenting\_path\_found$
1  $process\_cnt \leftarrow getProcessCount(nc)$;
2  for $i$ from 0 to $process\_cnt - 1$ do
3      $col\_vertex \leftarrow i \times tot\_thread\_num + tid$;
4      if $bfs\_array[col\_vertex] = bfs\_level$ then
5          for $j$ from $cxadj[col\_vertex]$ to $cxadj[col\_vertex + 1]$ do
6              $neighbor\_row \leftarrow cadj[j]$;
7              $col\_match \leftarrow rmatch[neighbor\_row]$;
8              if $col\_match > -1$ then
9                  if $bfs\_array[col\_match] = L_0\ -1$ then
10                     $vertex\_inserted \leftarrow$ true;
11                     $bfs\_array[col\_match] \leftarrow bfs\_level + 1$;
12                     $predeccesor[neighbor\_row] \leftarrow col\_vertex$;
13             else
14                 if $col\_match{=}{-1}$ then
15                     $rmatch[neighbor\_row] \leftarrow -2$;
16                     $predeccesor[neighbor\_row] \leftarrow col\_vertex$;
17                     $augmenting\_path\_found \leftarrow$ true;
```

filled with $bfs\_array[c] = L_0 - 1$ if $cmatch[c] > -1$ and $bfs\_array[c] = L_0$ if $cmatch[c] = -1$ by a simple InitBfsArray kernel (L_0 denotes BFS start level).

The GPU threads partition the column vertices in a single dimension. Each thread with id *tid* is assigned a number of columns which is obtained via the following function:

$$getProcessCount(nc) = \begin{cases} \left\lceil \frac{nc}{tot\_thread\_num} \right\rceil & \text{if } tid < nc \bmod tot\_thread\_num, \\ \left\lfloor \frac{nc}{tot\_thread\_num} \right\rfloor & \text{otherwise.} \end{cases}$$

Once the number of columns are obtained, the threads traverse their first assigned column vertex. The indices of the columns assigned to a thread differ by tot_thread_num to allow coalesced global memory accesses. Threads traverse the neighboring row vertices of the current column, if its BFS level is equal to the current bfs_level. If a thread encounters a matched row during the traversal, its matching column is retrieved. If the column is not traversed yet, its bfs_level is marked on bfs_array. On the other hand, when a thread encounters an unmatched row, an augmenting path is found. In this case, the match of the neighbor row is set to -2, and this information is used by Alternate later.

Algorithm 3 gives the description of the Alternate function. This kernel alternates the matched edges with the unmatched edges of the augmenting paths found; some of those paths end up being augmenting ones and some are only partially alternated. Here, each thread is assigned a number of rows. Since *rmatch* of an unmatched row (that is also an endpoint of an augmenting path) has been set to -2 in the BFS kernel, only the threads whose row vertices' matches are

Algorithm 3: Alternate

```
Data: $cmatch, rmatch, nc, nr, predecessor$
1  $process\_vcnt \leftarrow getProcessCount(nr)$;
2  for $i$ from 0 to $process\_vcnt - 1$ do
3      $row\_vertex \leftarrow i \times tot\_thread\_num + tid$;
4      if $rmatch[row\_vertex] = -2$ then
5          while $row\_vertex \neq -1$ do
6              $matched\_col \leftarrow predecessor[row\_vertex]$;
7              $matched\_row \leftarrow cmatch[matched\_col]$ ;
8              if $predecessor[matched\_row] = matched\_col$ then
9                  $break$;
10             $cmatch[matched\_col] \leftarrow row\_vertex$;
11             $rmatch[row\_vertex] \leftarrow matched\_col$;
12             $row\_vertex \leftarrow matched\_row$;
```

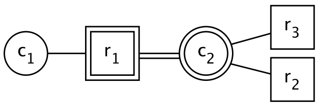


Fig. 1. Vertices r_1 and c_2 are matched; others are not. Two augmenting paths starting from c_1 are possible.

-2 start Alternate. Since there might be several augmenting paths for an unmatched column, race conditions while writing on *cmatch* and *rmatch* arrays are possible. Such a race condition might cause infinite loops (inner while loop) or inconsistencies, if care is not taken. We prevent these by checking the predecessor of a matched row (line-8). For example, in Fig. 1, two different augmenting paths that end with r_2 and r_3 are found for c_1. If the thread of r_2 starts before the thread of r_3 in Alternate, the match of c_2 will be updated to r_2 (line-10). Then, r_3's thread will read *matched_row* of c_2 as r_2 (line-7). This would cause an infinite loop without the check at line-8. Inconsistencies may occur when the threads of r_2 and r_3 are in the same warp. In this case, the if-check will not hold for both threads, and their row vertices will be written on *cmatch* (line-10). Since only one thread will be successful at writing, this will cause an inconsistency. Such inconsistencies are fixed by FixMatching kernel which implements: $rmatch[r] \leftarrow -1$ for any r satisfying $cmatch[rmatch[r]] \neq r$.

Algorithm 4 gives the description of a slightly different BFS kernel function. This function takes a *root* array as an extra argument. Initially, the root array is filled with $root[c] = 0$ if $cmatch[c] > -1$, and $root[c] = c$ if $cmatch[c] = -1$. This array holds the root (as the index of the column vertex) of an augmenting path, and this information is transferred down during BFS. Whenever an augmenting path is found, the entry in bfs_array for the root of the augmenting path is set to $L_0 - 2$. This information is used at the beginning of the BFS kernel. No more BFS traversals is done if an augmenting path is found for the root of the traversed column vertex. Therefore, while the method increases the global memory accesses by introducing an extra array, it provides an early exit mechanism for BFS.

Algorithm 4: BFS Kernel Function-2 (GPUBFS-WR)

```
Data: $bfs\_level, bfs\_array, cxadj, cadj, nc, rmatch, root$
      $vertex\_inserted, augmenting\_path\_found$
1  $process\_cnt \leftarrow getProcessCount(nc)$;
2  for $i$ from 0 to $process\_cnt - 1$ do
3      $col\_vertex \leftarrow i \times tot\_thread\_num + tid$;
4      if $bfs\_array[col\_vertex] = bfs\_level$ then
5          $myRoot \leftarrow root[col\_vertex]$;
6          if $bfs\_array[myRoot] < L_0 - 1$ then
7              $continue$;
8          for $j$ from $cxadj[col\_vertex]$ to $cxadj[col\_vertex + 1]$ do
9              $neighbor\_row \leftarrow cadj[j]$;
10             $col\_match \leftarrow rmatch[neighbor\_row]$;
11             if $col\_match > -1$ then
12                 if $bfs\_array[col\_match] = L_0\ -1$ then
13                     $vertex\_inserted \leftarrow$ true;
14                     $bfs\_array[col\_match] \leftarrow bfs\_level + 1$;
15                     $root[col\_match] \leftarrow myRoot$;
16                     $predeccesor[neighbor\_row] \leftarrow col\_vertex$;
17             else
18                 if $col\_match{=}{-1}$ then
19                     $bfs\_array[myRoot] \leftarrow L_0 - 2$;
20                     $rmatch[neighbor\_row] \leftarrow -2$;
21                     $predeccesor[neighbor\_row] \leftarrow col\_vertex$;
22                     $augmenting\_path\_found \leftarrow$ true;
```

We further improve GPUBFS-WR by making use of the arrays *root* and *bfs_array*. BFS kernels might find several rows to match with the same unmatched column, and set $rmatch[\cdot]$ to -2 for each. These cause Alternate to start from several rows that can be matched with the same unmatched column. Therefore, it may perform unnecessary alternations, until these augmenting paths intersect. Conflicts may occur at these intersection points (which are then resolved with FixMatching function). By choosing L_0 as 2, we can limit the range of the values that *bfs_array* takes to positive numbers. Therefore, by setting the *bfs_array* to $-(neighbor\_row)$ at line 19 of Algorithm 4, we can provide more information to the Alternate function. With this, Alternate can determine the beginning and the end of an augmenting path, and it can alternate only among the correct augmenting paths. APsB-GPUBFS-WR (and Alternate function used together) includes these improvements. However, they are not included in APFB-GPUBFS-WR since they do not improve its performance.

4 Experiments

The proposed implementations are compared against the sequential HK and PFP implementations [8], and against the multicore implementations P-PFP, P-DBFS, and P-HK obtained from [1]. The CPU implementations are tested on a computer with 2.27GHz dual quad-core Intel Xeon CPUs with 2-way hyper-threading and 48GB main memory. The algorithms are implemented in C++ and OpenMP. The GPU implementations are tested on NVIDIA Tesla C2050 with usable 2.6GB of global memory. C2050 is equipped with 14 multiprocessors each containing 32 CUDA cores, totaling 448 CUDA cores. The implementations are compiled with gcc-4.4.4, cuda-4.2.9 and -O2 optimization flag. For the multicore algorithms, 8 threads are used. A standard heuristic (called the cheap matching, see [8]) is used to initialize all algorithms. We compare the runtime of the matching algorithms after this common initialization. The execution times of the GPU algorithms exclude memory copy time. But including memory copy time decreases the reported mean speedups across all data set by at most 6%.

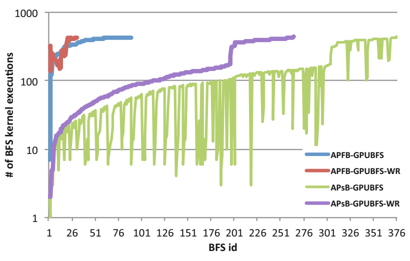
The two main algorithms APFB and APsB can use different BFS kernel functions (GPUBFS and GPUBFS-WR). Moreover, these algorithms can have two versions (i) CT: uses a constant number of threads with fixed number of grid and block size (256×256) and assigns multiple vertices to each thread; (ii) MT: tries to assign one vertex to each thread. The number of threads used in the second version is chosen as MT $= \min(nc, \#threads)$ where nc is the number of columns, and $\#threads$ is the maximum number of threads of the architecture. Therefore, we have eight GPU-based algorithms.

We used 70 different matrices from variety of classes at UFL matrix collection [6]. We also permuted the matrices randomly by rows and columns and included them as a second set (labeled RCP). These permutations usually render the problems harder for the augmenting-path-based algorithms [8]. For both sets, we report the performance for a smaller subset which contains those matrices in which at least one of the sequential algorithms took more than one second. We call these sets O_S1 (28 matrices) and RCP_S1 (50 matrices). We also have another two subsets called O_Hardest20 and RCP_Hardest20 that contain the set of 20 matrices on which the sequential algorithms required the longest runtime.

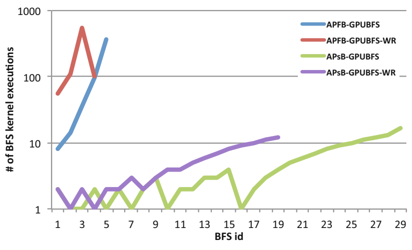
Table 1. Geometric mean of the runtime (in seconds) of the GPU algorithms on different sets of instances

	APFB				APsB			
	GPUBFS		GPUBFS-WR		GPUBFS		GPUBFS-WR	
	MT	CT	MT	CT	MT	CT	MT	CT
O_S1	2.96	1.89	2.12	**1.34**	3.68	2.88	2.98	2.27
O_Hardest20	4.28	2.70	3.21	**1.93**	5.23	4.14	4.20	3.13
RCP_S1	3.66	3.24	1.13	**1.05**	3.52	3.33	2.22	2.14
RCP_Hardest20	7.27	5.79	3.37	**2.85**	12.06	10.75	8.17	7.41

Table 1 compares the proposed GPU implementations on different sets. As we see from the table, using a constant number of threads (CT) always increases the performance of an algorithm, since it increases the granularity of the work performed by each thread. GPUBFS-WR is always faster than GPUBFS. This is due to the unnecessary BFSs in the GPUBFS algorithm. GPUBFS cannot



(a) Hamrle3



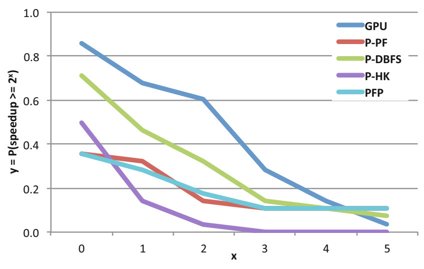
(b) Delanuay_n23

Fig. 2. The BFS ids and the number of kernel executions for each BFS in APsB and APFB variants for two graphs. The x axis shows the id of the **while** iteration at line 2 of APsB. The y axis shows the number of the **while** iterations at line 6 of APsB.

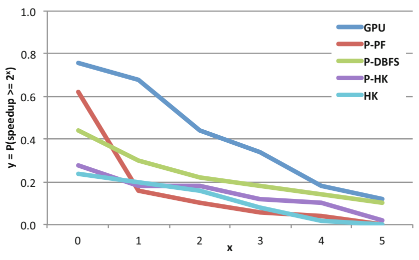
determine whether an augmenting path has already been found for an unmatched column, therefore it will continue to explore. This unnecessary BFSs not only increase the time, but also reduce the likelihood of finding an augmenting path for other unmatched columns. Moreover, the ALTERNATE scheme turns out to be more suitable for APFB than APsB, in which case it can augment along more paths (there is a larger set of possibilities). For example, Figs. 2(a) and 2(b) show the number of BFS iterations and the number of BFS levels in each iteration for, respectively, Hamrle3 and Delanuay_n23. As clearly seen from the figures, APFB variants converge in smaller number of iterations than APsB variants; and for most of the graphs, the total number of BFS kernel calls are less for APFB (as in Fig. 2(a)). However, for a small set of graphs, although the augmenting path exploration of APsB converges in larger number of iterations, the numbers of the BFS levels in the iterations are much less than APFB (as in Fig. 2(b)). Unlike the general case, APsB outperforms APFB in such cases. Since APFB using GPUBFS-WR and CT is almost always the best algorithm, we only compare the performance of this algorithm with other implementations in the following.

Figures 3(a) and 3(b) give the log-scaled speedup profiles of the best GPU and multicore algorithms on the original and permuted graphs. The speedups are calculated with respect to the fastest of the sequential algorithms PFP and HK (on the original graphs HK was faster; on the permuted ones PFP was faster). A point (x, y) in the plots corresponds to the probability of obtaining at least 2^x speedup is y. As the plots show, the GPU algorithm has the best overall speedup. It is faster than the sequential HK algorithm for 86% of the original graphs, while it is faster than PFP on 76% of the permuted graphs. P-DBFS obtains the best performance among the multicore algorithms. However, its performance degrades on permuted graphs. Although P-PFP is more robust than P-DBFS to permutations, its overall performance is inferior to that of P-DBFS. P-HK is outperformed by the other algorithms in both sets.

Figures 4(a) and 4(b) show the performance profiles of the GPU and multicore algorithms. A point (x, y) in the plots means that with y probability,

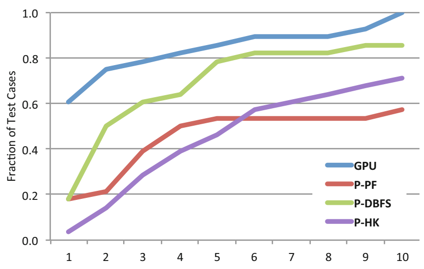


(a) Original graphs

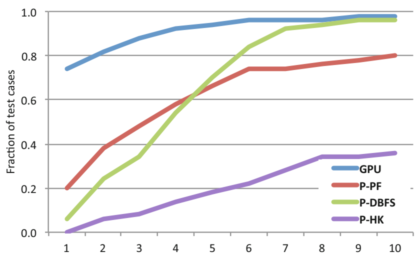


(b) Permuted graphs

Fig. 3. Log-scaled speedup profiles



(a) Original graphs



(b) Permuted graphs

Fig. 4. Performance profiles

the corresponding algorithm obtains a performance that is at most x times worse than the best runtime. The plots clearly mark the GPU algorithm as the fastest in most cases, especially for the original graphs and for $x \leq 7$ for the permuted ones. In particular, the GPU algorithm obtains the best performance in 61% of the original graphs, while this ratio increases to 74% for the permuted ones.

Figure 5 gives the overall speedups. The proposed GPU algorithm obtains average speedup values of at least 3.61 and 3.54 on, respectively, original and permuted graphs. The speedups increase for the hardest instances, where the GPU algorithm achieves 3.96 and 9.29 speedup, respectively, on original and permuted graphs. Moreover, the runtimes obtained by the GPU algorithm are robust among the repeated executions on a graph instance. We calculated the standard deviation of the execution times for different repetitions. For the graphs in O_S1 set, the ratios of the standard deviations to the average time are observed to be less than 10%, 18%, and 47% for 20, 5, and 3 graphs, respectively.

Table 2 gives the actual runtime for O_Hardest20 set for the best GPU and multicore algorithms, together with the sequential algorithms. The compete set of runtimes can be found at `http://bmi.osu.edu/hpc/software/matchmaker2/maxCardMatch.html`. As table shows, except six instances among the original graphs and another two among the permuted graphs, the GPU algorithm is faster than the best sequential algorithm. It is also faster than the multicore ones in all, except five original graphs.

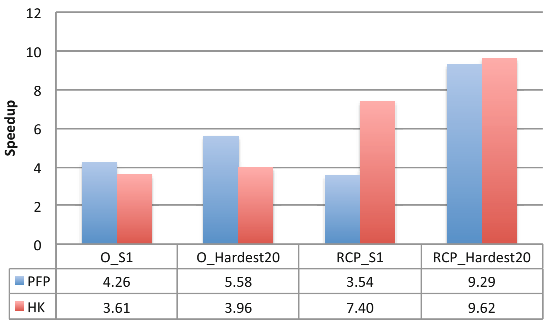


Fig. 5. Overall speedup of the proposed GPU algorithm w.r.t. PFP (left bars) and HK (right bars) algorithms

Table 2. The actual runtime of each algorithm for the O_Hardest20 set

	Original graphs				Permuted graphs			
Matrix name	GPU	P-DBFS	PFP	HK	GPU	P-DBFS	PFP	HK
roadNet-CA	**0.34**	0.53	0.95	2.48	**0.39**	1.88	3.05	4.89
delaunay_n23	**0.96**	1.26	2.68	1.11	**0.90**	5.56	3.27	14.34
coPapersDBLP	**0.42**	6.27	3.11	1.62	0.38	1.25	**0.29**	1.26
kron_g500-logn21	**0.99**	1.50	5.37	4.73	**3.71**	4.01	64.29	16.08
amazon-2008	**0.11**	0.18	6.11	1.85	**0.41**	1.37	61.32	4.69
delaunay_n24	**1.98**	2.41	6.43	2.22	**1.86**	12.84	6.92	35.24
as-Skitter	**0.49**	1.89	7.79	3.56	**3.27**	5.74	472.63	29.63
amazon0505	**0.18**	22.70	9.05	1.87	**0.24**	15.23	17.59	2.23
wikipedia-20070206	**1.09**	5.24	11.98	6.52	**1.05**	5.99	9.74	5.73
Hamrle3	1.36	2.70	**0.04**	12.61	**3.85**	7.39	37.71	57.00
hugetrace-00020	**7.90**	393.13	15.95	15.02	**1.52**	9.97	8.68	38.27
hugebubbles-00000	13.16	**3.55**	19.81	5.56	**1.80**	10.91	10.03	38.97
wb-edu	33.82	8.61	**3.38**	20.35	17.43	20.10	**9.49**	51.14
rgg_n_2_24_s0	3.68	2.25	25.40	**0.12**	**2.20**	12.50	5.72	31.78
patents	0.88	**0.84**	92.03	16.18	**0.91**	0.97	101.76	18.30
italy_osm	5.86	1.20	**1.02**	122.00	**0.70**	3.97	6.24	18.34
soc-LiveJournal1	**3.32**	14.35	243.91	21.16	**3.73**	7.14	343.94	20.71
ljournal-2008	**2.37**	10.30	360.31	17.66	**6.90**	7.58	176.69	23.45
europe_osm	57.53	**11.21**	14.15	1911.56	**7.21**	37.93	68.18	197.03
com-livejournal	**4.58**	22.46	2879.36	34.28	**5.88**	17.19	165.32	29.40

5 Concluding Remarks

We proposed a parallel GPU implementation of a BFS-based maximum cardinality matching algorithm for bipartite graphs. We compared the performance of the proposed implementation against sequential and multicore algorithms on various datasets. The experiments showed that the GPU implementation is faster than the existing multicore implementations. The speedups achieved with respect to well-known sequential implementations varied from 0.03 to 629.19, averaging 9.29 w.r.t. the fastest sequential algorithm on a set of 20 hardest problems. A GPU is a restricted memory device. Although, an out-of-core or distributed-memory type algorithm is amenable when the graph does not fit into the device, a direct implementation of these algorithms will surely not be efficient. We plan to investigate matching algorithms for extreme-scale bipartite graphs on GPUs.

References

1. Azad, A., Halappanavar, M., Rajamanickam, S., Boman, E.G., Khan, A., Pothen, A.: Multithreaded algorithms for maximum matching in bipartite graphs. In: 26th IPDPS, pp. 860–872. IEEE (2012)
2. Azad, A., Pothen, A.: Multithreaded algorithms for matching in graphs with application to data analysis in flow cytometry. In: 26th IPDPS Workshops & PhD Forum, pp. 2494–2497. IEEE (2012)
3. Berge, C.: Two theorems in graph theory. P. Natl. Acad. Sci. USA 43, 842–844 (1957)
4. Burkard, R., Dell'Amico, M., Martello, S.: Assignment Problems. SIAM, Philadelphia (2009)
5. Çatalyürek, Ü.V., Deveci, M., Kaya, K., Uçar, B.: Multithreaded clustering for multi-level hypergraph partitioning. In: 26th IPDPS, pp. 848–859. IEEE (2012)
6. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM T. Math. Software 38(1) 1:1–1:25 (2011)
7. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, London (1986)
8. Duff, I.S., Kaya, K., Uçar, B.: Design, implementation, and analysis of maximum transversal algorithms. ACM T. Math. Software 38(2), 13 (2011)
9. Duff, I.S., Wiberg, T.: Remarks on implementation of $O(n^{1/2}\tau)$ assignment algorithms. ACM T. Math. Software 14(3), 267–287 (1988)
10. Fagginger Auer, B.O., Bisseling, R.H.: A GPU algorithm for greedy graph matching. In: Keller, R., Kramer, D., Weiss, J.-P. (eds.) Facing Multicore-Challenge II. LNCS, vol. 7174, pp. 108–119. Springer, Heidelberg (2012)
11. Goldberg, A., Kennedy, R.: Global price updates help. SIAM J. Discrete Math. 10(4), 551–572 (1997)
12. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum flow problem. J. Assoc. Comput. Mach. 35, 921–940 (1988)
13. Halappanavar, M., Feo, J., Villa, O., Tumeo, A., Pothen, A.: Approximate weighted matching on emerging manycore and multithreaded architectures. Int. J. High Perform. C. 26(4), 413–430 (2012)
14. Hochbaum, D.S.: The pseudoflow algorithm and the pseudoflow-based simplex for the maximum flow problem. In: Bixby, R.E., Boyd, E.A., Ríos-Mercado, R.Z. (eds.) IPCO 1998. LNCS, vol. 1412, pp. 325–337. Springer, Heidelberg (1998)
15. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Comput. 2(4), 225–231 (1973)
16. John, P.E., Sachs, H., Zheng, M.: Kekulé patterns and Clar patterns in bipartite plane graphs. J. Chem. Inf. Comp. Sci. 35(6), 1019–1021 (1995)
17. Kaya, K., Langguth, J., Manne, F., Uçar, B.: Push-relabel based algorithms for the maximum transversal problem. Comput. Oper. Res. 40(5), 1266–1275 (2012)
18. Kim, W.Y., Kak, A.C.: 3-D object recognition using bipartite matching embedded in discrete relaxation. IEEE T. Pattern Anal. 13(3), 224–251 (1991)
19. Vasconcelos, C.N., Rosenhahn, B.: Bipartite graph matching computation on GPU. In: Cremers, D., Boykov, Y., Blake, A., Schmidt, F.R. (eds.) EMMCVPR 2009. LNCS, vol. 5681, pp. 42–55. Springer, Heidelberg (2009)

On-Board Multi-GPU Molecular Dynamics

Marcos Novalbos[1], Jaime Gonzalez[2], Miguel Angel Otaduy[1], Alvaro Lopez-Medrano[2], and Alberto Sanchez[1]

[1] URJC Madrid
{marcos.novalbos,miguel.otaduy,alberto.sanchez}@urjc.es
[2] Plebiotic S.L.
{jaime.gonzalez,alvaro.lopez-medrano}@plebiotic.com

Abstract. Molecular dynamics simulations allow us to study the behavior of complex biomolecular systems. These simulations suffer a large computational complexity that leads to simulation times of several weeks in order to recreate just a few microseconds of a molecule's motion even on high-performance computing platforms. In recent years, state-of-the-art molecular dynamics algorithms have benefited from the parallel computing capabilities of multicore systems, as well as GPUs used as co-processors. In this paper we present a parallel molecular dynamics algorithm for on-board multi-GPU architectures. We parallelize a state-of-the-art molecular dynamics algorithm at two levels. We employ a spatial partitioning approach to simulate the dynamics of one portion of a molecular system on each GPU, and we take advantage of direct communication between GPUs to transfer data among portions. We also parallelize the simulation algorithm to exploit the multi-processor computing model of GPUs. Most importantly, we present novel parallel algorithms to update the spatial partitioning and set up transfer data packages on each GPU. We demonstrate the feasibility and scalability of our proposal through a comparative study with NAMD, a well known parallel molecular dynamics implementation.

1 Introduction

Molecular dynamics simulations [20] are computational approaches for studying the behavior of complex biomolecular systems at the atom level, estimating their dynamic and equilibrium properties which can not be solved analytically. Their most direct applications are related to identifying and predicting the structure of proteins, but they also provide a tool for drug or material design. These simulations recreate the movements of atoms and molecules due to their interactions for a given period of time. However, they are limited by size and computational time due to the current available computational resources. For instance, simulating just one nanosecond of the motion of a well known system with 92 224 atoms (ApoA1 benchmark) using only one processor takes up to 14 days [13].

The simulation times of molecular dynamics can be reduced thanks to algorithms that update atoms in a parallel way. Such algorithms were initially implemented on multi-CPU architectures, such as multicore processors or computer clusters with several computing nodes connected by a local area network

(LAN) [2, 6, 14]. More recent alternatives have used hybrid GPU-CPU architectures to provide parallelism [22], taking advantage of the massive parallel capabilities of GPUs. This approach interconnects several computing nodes, each one with one or more GPUs serving as co-processors of the CPUs [5,9]. The compute power of this approach is bounded by the cost to transfer data between CPUs and GPUs and between compute nodes.

Novel GPU architectures support direct communication between GPUs, even mounted on the same board [21]. These features enable the use of GPUs as the central compute nodes of parallel molecular dynamics algorithms, and not just as mere co-processors, thereby reducing the communication load. In this paper, we present a parallel molecular dynamics algorithm for on-board multi-GPU architectures. We parallelize molecular dynamics simulations at two levels. At the high level, we present a spatial partitioning approach to assign one portion of a molecular system to each GPU. At the low level, we parallelize on each GPU the simulation of its corresponding portion. Most notably, we present algorithms for the massively parallel update of the spatial partitions and for the setup of data packages to be transferred to other GPUs. We demonstrate our approach on a multi-GPU on-board architecture, using PCIe for direct GPU-GPU communication. We show speed-ups and improved scalability over NAMD, a state-of-the-art multi-CPU-GPU simulation algorithm that uses GPUs as co-processors.

2 Related Work

In computer-driven molecular dynamics simulations, atoms are contained in a virtual 3D coordinate system that models the real environment inside a specific volume. In biological systems, the molecules are surrounded by water molecules, and *periodic boundary conditions* are imposed on the simulation volume, i.e., the simulation volume is implicitly replicated infinite times. The simulation time is divided into steps of very small size, in the order of $1\text{fs} = 10^{-15}\text{s}$. Given atom positions X_i and velocities V_i at time T_i, the simulation algorithm evaluates the interaction forces and integrates them to obtain positions X_{i+1} and velocities V_{i+1} at time T_{i+1}. The interaction forces can be divided in:

i) Bonded forces, which can be of several types depending on the number of atoms involved: single bond, angle, proper and improper dihedral forces. *ii*) Non-bonded short-range forces, composed of Van der Waals forces and electrostatic interactions between atoms closer than a cutoff radius (R_c). *iii*) Non-bonded long-range forces, consisting of electrostatic interactions between atoms separated by a distance greater than R_c. In the remaining of this paper, we account only for bonded and short-range non-bonded forces.

The different dynamics of the various types of forces suggest the use of Multiple Time Stepping integrators (MTS) [8, 23], which use a smaller time step for bonded forces. Several MTS integrators have been proposed, such as Verlet-I/r-RESPA [3], MOLLY [19] and LN [1]. Despite the use of more time steps for bonded forces, most of the integrator's time is spent calculating non-bonded forces. The computation of short-range non-bonded forces can be accelerated using a regular grid, which is updated at a lower rate than the simulation. This method is known as *cell list* [4,12,16]. A more comprehensive description of the basics of molecular dynamics can be found in [20].

Several authors have proposed ways to parallelize molecular dynamics algorithms on hybrid CPU-GPU architectures [11,25]. NAMD [17] defines work tasks and then partitions these tasks among the available computing nodes. GPUs are used as massively parallel co-processors to speed-up the tasks that involve computation of non-bonded short-range forces. GROMACS [5] performs a spatial partitioning on the molecular system to distribute it on a multi-core architecture, and then the CPUs may use GPUs as co-processors to speed-up force computations. ACEMD [4] performs GPU-parallel computation of the various forces in a molecular system, and each type of force is handled on a separate GPU. This approach exploits on-board multi-GPU architectures, but its scalability is limited because all communications are handled through the CPU. Very recently, Rustico et al. [18] have proposed a spatial partitioning approach for multi-GPU particle-based fluid simulation, which shares many features with molecular dynamics. Our work addresses massively parallel data management and communication, not discussed by Rustico et al.

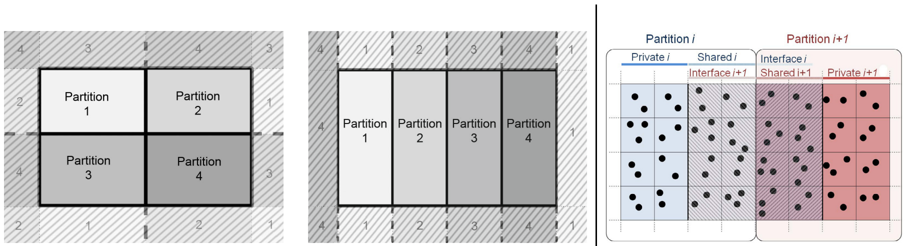
3 Algorithm Overview

In contrast to previous parallel molecular dynamics algorithms, we propose a two-level algorithm that partitions the molecular system, and each GPU handles in a parallel manner the computation and update of its corresponding portion, as well as the communications with other GPUs.

To solve the dynamics, we use a generic Verlet/Respa MTS integrator. In our examples, we have used a time step $\Delta t = 2$ fs for short-range non-bonded forces, and we update bonded forces $nStepsBF = 2$ times per time step. We accelerate short-range non-bonded forces using the cell-list method, with a grid resolution of $R_c/2$. The cell-list data structure can be updated and visited efficiently on a GPU using the methods in [24].

We partition the molecule using grid-aligned planes, thus minimizing the width of interfaces and simplifying the update of partitions. We partition the simulation domain only once at the beginning of the simulation, and then update the partitions by transferring atoms that cross borders. We have tested two partitioning approaches with different advantages:

- Binary partition (Figure 1a): we recursively halve molecule portions using planes orthogonal to their largest dimension. Each portion may have up to 26 neighbors in 3D.
- Linear partition (Figure 1b): we divide the molecular system into regular portions using planes orthogonal to the largest dimension of the full simulation volume. With this method, each portion has only 2 neighbors, but the interfaces are larger; therefore, it trades fewer communication messages for more expensive partition updates.



(a) Binary Partition (b) Linear Partition (c) Cells at the Interface

Fig. 1. On the left and center, comparison of binary **(a)** vs. linear spatial partitioning **(b)**. The striped regions represent the periodicity of the simulation volume. On the right, the different types of cells at the interface between two portions of the simulation volume.

Based on our cell-based partition strategy, each GPU contains three types of cells as shown in Figure 1c:

– *Private cells* that are exclusively assigned to one GPU.
– *Shared cells* that contain atoms updated by a certain GPU, but whose data needs to be shared with neighboring portions.
– *Interface cells* that contain atoms owned by another GPU, and used for force computations in the given GPU.

Algorithm 1 shows the pseudo-code of our proposed multi-GPU MTS integrator, highlighting in blue with a star the differences w.r.t. a single-GPU version. These differences can be grouped in two tasks: update partitions and synchronize dynamics of neighboring portions. Once every ten time steps, we update the partitions in two steps. First, we identify atoms that need to be updated, i.e., atoms that enter shared cells of a new portion. And second, we transfer the positions and velocities of these atoms. To synchronize dynamics, we transfer forces of all shared atoms, and then each GPU integrates the velocities and positions of its private and shared atoms, but also its interface atoms. Again, we carry out the synchronization in two steps. First, after the cell-list data structure is updated, we identify the complete set of shared atoms. And second, we transfer the forces of shared atoms as soon as they are computed.

4 Parallel Partition Update and Synchronization

As outlined above, each GPU stores one portion of the complete molecular system and simulates this subsystem using standard parallel algorithms [24]. In this section, we describe massively parallel algorithms to update the partitions and to transfer interface forces to ensure proper synchronization of dynamics between subsystems. We propose algorithms that separate the identification of atoms whose data needs to be transferred from the setup of the transfer packages. In this way, we can reuse data structures and algorithms both in partition updates and force transfers. Data transfers are issued directly between GPUs, thereby minimizing communication overheads.

Algorithm 1. Multi-GPU Verlet/r-Respa MTS integrator. The modifications w.r.t. the single-GPU version are highlighted in blue with a star.

```
1: procedure STEP($currentStep$)
2:     if $currentStep$ mod 10 = 0 then
3:         $* identifyUpdateAtomIds()$
4:         $* transferUpdatePositionsAndVelocities()$
5:         $updateCellList()$
6:         $* identifySharedAtomIds()$
7:     end if
8:     $integrateTemporaryPosition(0.5 \cdot \Delta t)$
9:     $computeShortRangeForces()$
10:    $* transferSharedShortRangeForces()$
11:    for $nStepsBF$ do
12:        $integratePosition(0.5 \cdot \Delta t/nStepsBF)$
13:        $computeBondedForces()$
14:        $* transferSharedBondedForces()$
15:        $integrateKickVelocity(\Delta t/nStepsBF)$
16:        $integratePosition(0.5 \cdot \Delta t/nStepsBF)$
17:    end for
18:    $currentStep = currentStep + 1$
19: end procedure
```

4.1 Data Structures

The basic molecular dynamics algorithm stores atom data in two arrays:

– *staticAtomData* corresponds to data that does not change during the simulation, such as atom type, bonds, electrostatic and mechanical coefficients, etc. It is sorted according to static atom indexes.
– *dynamicAtomData* that contains position and velocity, a force accumulator, and the atom's cell. It is sorted according to the cell-list structure, and all atoms in the same cell appear in consecutive order.

Both arrays store the identifiers of the corresponding data in the other array to resolve indirections. Each GPU stores a copy of the *staticAtomData* of the whole molecule, and keeps *dynamicAtomData* for its private, shared, and interface cells. The *dynamicAtomData* is resorted in each call to the *updateCellList* procedure, and the atom identifiers are accordingly reset. Atoms that move out of a GPU's portion are simply discarded.

In our multi-GPU algorithm, we extend the *dynamicAtomData*, and store for each atom a list of neighbor portions that it is shared with. We also define two additional arrays on each GPU:

– *cellNeighbors* is a static array that stores, for each cell, a list of neighbor portions.
– *transferIDs* is a helper data structure that stores pairs of neighbor identifiers and dynamic atom identifiers. This data structure is set during atom identification procedures, and it is used for the creation of the transfer packages.

4.2 Identification of Transfer Data

Each GPU contains a $transferIDs$ data structure of size $nNeighbors \cdot nAtoms$, where $nNeighbors$ is the number of neighbor portions, and $nAtoms$ is the number of atoms in its corresponding portion. This data structure is set at two stages of the MTS Algorithm 1, $identifyUpdateAtomIds$ and $identifySharedAtomIds$. In both cases, we initialize the neighbor identifier in the $transferIDs$ data structure to the maximum unsigned integer value. Then, we visit all atoms in parallel in one CUDA kernel, and flag the (atom, neighbor) pairs that actually need to be transferred. We store one flag per neighbor and atom to avoid collisions at write operations. Finally, we sort the $transferIDs$ data structure according to the neighbor identifier, and the (atom, neighbor) pairs that were flagged are considered as valid and are automatically located at the beginning of the array. We have used the highly efficient GPU-based Merge-Sort implementation in the NVidia SDK [15] ($5.3ms$ to sort an unsorted array with one million values on a NVidia GeForce GTX580).

Algorithm 2. Identification of atoms whose data needs to be transferred, along with their target neighbor

```
 1: procedure IdentifyTransferAtomIds($transferIDs$)
 2:     for $atomID$ in $atoms$ do
 3:         for $neighborID$ in $cellNeighbors(dynamicAtomData[atomId].cellID)$ do
 4:             if $MustTransferData(atomID, neighborID)$ then
 5:                 $offset = neighborID \cdot nAtoms + atomID$
 6:                 $transferIDs[offset].atomID = atomID$
 7:                 $transferIDs[offset].neighborID = neighborID$
 8:             end if
 9:         end for
10:     end for
11:     $Sort(transferIDs, neighborID)$
12: end procedure
```

Algorithm 2 shows the general pseudo-code for the identification of transfer data. The actual implementation of the $MustTransferData$ procedure depends on the actual data to be transferred. For partition updates, an atom needs to be transferred to a certain neighbor portion if it is not yet present in its list of neighbors. For force synchronization, an atom needs to be transferred to a certain neighbor portion if it is included in its list of neighbors.We also update the list of neighbors of every atom as part of the $identifyUpdateAtomIds$ procedure.

4.3 Data Transfer

For data transfers, we set in each GPU a buffer containing the output data and the static atom identifiers. To set the buffer, we visit all valid entries of the $transferIDs$ array in parallel in one CUDA kernel, and fetch the transfer data

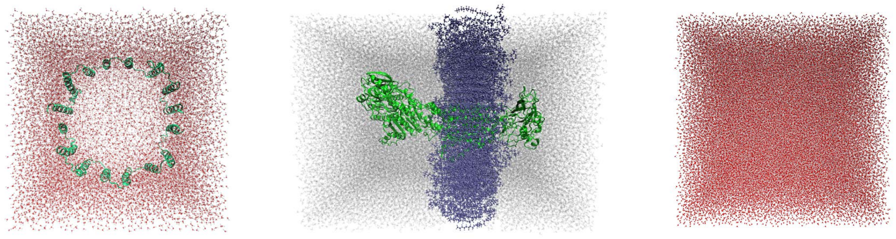

(a) ApoA1 in water (b) C206 in water (c) 400K

Fig. 2. Benchmark molecules

using the dynamic atom identifier. The particular transfer data may consist of forces or positions and velocities, depending on the specific step in the MTS Algorithm 1.

Transfer data for all neighbor GPUs is stored in one unique buffer; therefore, we set an additional array with begin and end indexes for each neighbor's chunk. This small array is copied to the CPU, and the CPU invokes one asynchronous copy function to transfer data between each GPU and one of its neighbors. We use NVidia's driver for unified memory access (Unified Virtual Addressing, UVA) [21] to perform direct memory copy operations between GPUs.

Upon reception of positions and velocities during the update of the partitions, each GPU appends new entries of *dynamicAtomData* at the end of the array. These entries will be automatically sorted as part of the update of the cell-list. Upon reception of forces during force synchronization, each GPU writes the force values to the force accumulator in the *dynamicAtomData*. The received data contains the target atoms' static identifiers, which are used to indirectly access their dynamic identifiers.

5 Evaluation

In order to validate our proposal, we carried out our experiments on a machine outfitted with Ubuntu GNU/Linux 10.04, two Intel Xeon Quad Core 2.40GHz CPUs with hyperthreading, 32 GB of RAM and four NVidia GTX580 GPUs connected to PCIe 2.0 slots in an Intel 5520 IOH Chipset of a Tyan S7025 motherboard. The system's PCIe 2.0 bus bandwidth for peer-to-peer throughputs via IOH chip was 9GB/s full duplex, and 3.9 GB/s for GPUs on different IOHs [10]. The IOH does not support non-contiguous byte enables from PCI Express for remote peer-to-peer MMIO transactions [7]. The complete deployment of our testbed architecture is depicted in Figure 3. Direct GPU-GPU communication can be performed only for GPUs connected to the same IOH. For GPUs connected through QPI, the driver performs the communication using CPU RAM [10].

Given our testbed architecture, we have tested the scalability of our proposal by measuring computation and transmission times for 1, 2, and 4 partitions running on different GPUs. We have estimated scalability further by estimating transmission times for 8 and 16 partitions using the bandwidth obtained with 4 GPUs and the actual data size of 8 and 16 partitions respectively.

We have used three molecular systems as benchmarks (see Figure 2):

- ApoA1 (92,224 atoms) is a well known high density lipoprotein (HDL) in human plasma. It is often used in performance tests with NAMD.
- C206 (256,436 atoms) is a complex system formed by a protein, a ligand and a membrane. It presents load balancing challenges for molecular dynamics simulations.
- 400K (399,150 atoms) is a well-balanced system of 133,050 molecules of water designed synthetically for simulation purposes.

All our test simulations were executed using MTS Algorithm 1, with a time step of 2 fs for short-range non-bonded forces, and 1 fs ($nStepsBF = 2$) for bonded forces. In all our tests, we measured averaged statistics for 2000 simulation steps, i.e., a total duration of 4ps ($4 \cdot 10^{-12}$s).

5.1 Comparison of Partition Strategies

To evaluate our two partition strategies described in Section 3, we have compared their performance on the C206 molecule. We have selected C206 due to its higher complexity and data size. Figure 4a indicates that, as expected, the percentage of interface cells grows faster for the linear partition. Note that with 2 partitions the size of the interface is identical with both strategies because the partitions are actually the same. With 16 partitions, all cells become interface cells for the linear partition strategy, showing the limited scalability of this approach. Figure 4b shows that, on the other hand, the linear partition strategy exhibits a higher transmission bandwidth. Again, this result was expected, as the number of neighbor partitions is smaller with this strategy.

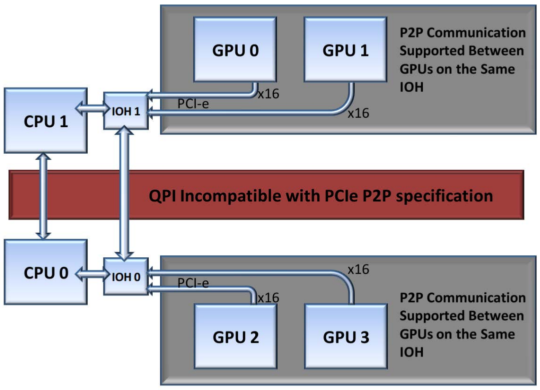
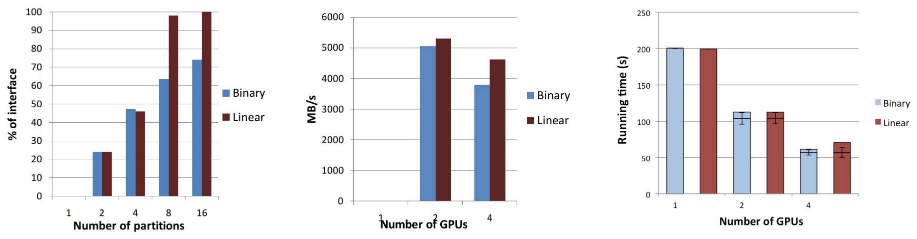

GPU 0
GPU 1
P2P Communication Supported Between GPUs on the Same IOH
x16
CPU 1
IOH 1
PCI-e
x16
QPI Incompatible with PCIe P2P specification
CPU 0
IOH 0
x16
PCI-e
x16
GPU 2
GPU 3
P2P Communication Supported Between GPUs on the Same IOH

Fig. 3. PCIe configuration of our testbed



(a) Interface size (b) Bandwidth (c) Sim. times (2000 steps)

Fig. 4. Performance comparison of binary and linear partition strategies on C206

All in all, Figure 4c compares the actual simulation time for both partition strategies. This time includes the transmission time plus the computation time of the slowest partition. For the C206 benchmark, the binary partition strategy exhibits better scalability, and the reason is that the linear strategy suffers a high load imbalance, as depicted by the plot of standard deviation across GPUs.

Figure 5 shows how the total simulation time is split between computation and transmission times for the binary partition strategy. Note again that the transmission times for 8 and 16 partitions are estimated, not measured. Up to 4 partitions, the cost is dominated by computations, and this explains the improved performance with the binary strategy despite its worse bandwidth.

The optimal choice of partition strategy appears to be dependent of the underlying architecture, but also of the specific molecule, its size, and its spatial atom distribution.

5.2 Scalability Analysis and Comparison with NAMD

Figure 6a shows the total speedup for the three benchmark molecules using our proposal (with a binary partition strategy). Note again that speedups for 8 and 16 GPUs, shown in dotted lines, are estimated based on the bandwidth with 4 GPUs. The results show that the implementation makes the most out of the molecule's size by sharing the workload among different GPUs. The speedup of APOA1 is lower because it is the smallest molecule and the simulation is soon limited by communication times.

Figure 6b evaluates our combined results in comparison with a well-known parallel molecular dynamics implementation, NAMD. Performance is measured in terms of the nanoseconds that can be simulated in one day. The three benchmark molecules were simulated on NAMD using the same settings as on our implementation. Recall that NAMD distributes work tasks among CPU cores and uses GPUs as co-processors, in contrast to our fully GPU-based approach. We could not estimate performance for NAMD with 8 and 16 GPUs, as we could not separate computation and transmission times. All in all, the results show that our proposal clearly outperforms NAMD for all molecules by a factor of approximately 4×.

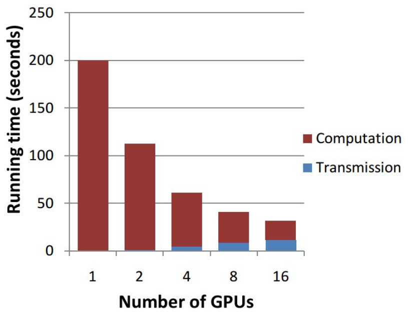
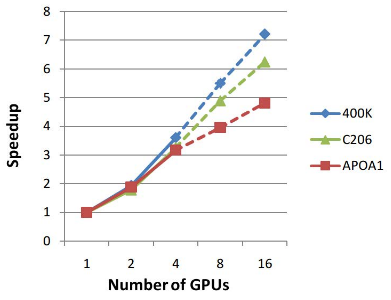
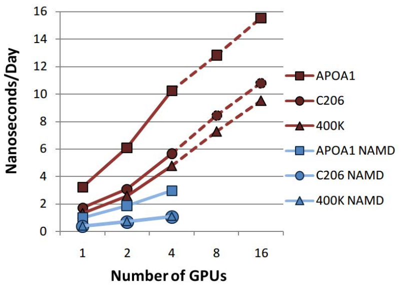


Fig. 5. Running time (2000 steps) for the binary partition strategy on C206



(a) Speedup



(b) Comparison vs. NAMD

Fig. 6. Scalability (left) and performance comparison with NAMD (right), measured in terms of simulated nanoseconds per day

In terms of memory scalability, our approach suffers the limitation that each partition stores static data for the full molecule. From our measurements, the static data occupies on average 78MB for 100K atoms, which means that modern GPUs with 2GB of RAM could store molecules with up to 2.5 million atoms. In the dynamic data, there are additional memory overheads due to the storage of interface cells and sorting lists, but these data structures become smaller as the number of partitions grows. In addition, interface cells grow at a lower rate than private cells as the size of the molecule grows.

6 Conclusions and Future Work

This article presents a parallel molecular dynamics algorithm for on-board multi-GPU architectures. Our approach builds on parallel multiple time stepping integrators, but achieves further speed-ups through spatial partitioning and direct GPU-GPU communication. Our results show the benefits of using GPUs as central compute nodes instead of mere co-processors.

Our approach presents several limitations that could motivate future work. Most importantly, our spatial partitioning supports only bonded and short-range non-bonded forces, but many complex molecular systems require also the computation of long-range non-bonded forces. Our current solution relies on a static partitioning, which does not guarantee even load balancing across GPUs. The experiments indicate that practical molecular systems maintain rather even atom distributions, but dynamic load balancing might be necessary for finer partitions. Last, our work could be complemented with more advanced protocols and architectures to optimize communications between GPUs. Indeed, there are already architectures that outperform the Intel IOH/QPI interface for the PCIe bridge used in the experiments.

Acknowledgments. The authors thank Roldán Martínez from Plebiotic for his helpful advice, and Alberto Hernández and Sergio Quintana for their help with early versions of this work.

References

1. Barth, E., Schlick, T.: Extrapolation versus impulse in multiple-timestepping schemes. II. Linear analysis and applications to Newtonian and Langevin dynamics. Chemical Physics 109, 1633–1642 (1998)
2. Clark, T.W., McCammon, J.A.: Parallelization of a molecular dynamics non-bonded force algorithm for MIMD architecture. Computers & Chemistry, 219–224 (1990)
3. Grubmüller, H.: Dynamiksimulation sehr großer Makromoleküle auf einem Parallelrechner. Diplomarbeit, Technische Universität München, Physik-Department, T 30, James-Franck-Straße, 8046 Garching (1989)
4. Harvey, M.J., Giupponi, G., Fabritiis, G.D.: ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale. Journal of Chemical Theory and Computation 5(6), 1632–1639 (2009)
5. Hess, B., Kutzner, C., van der Spoel, D., Lindahl, E.: GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. Journal of Chemical Theory and Computation 4(3), 435–447 (2008)
6. Hwang, Y.S., Das, R., Saltz, J., Brooks, B., Scek, M.H.: Parallelizing molecular dynamics programs for distributed memory machines: An application of the chaos runtime support library (1994)
7. Intel: Intel 5520 Chipset: Datasheet (March 2009), `http://www.intel.com/content/dam/doc/datasheet/5520-5500-chipset-ioh-datasheet.pdf`
8. Izaguirre, J.A., Matthey, T., Willcock, J., Ma, Q., Moore, B., Slabach, T., Viamontes, G.: A tutorial on the prototyping of multiple time stepping integrators for molecular dynamics (2001)
9. Kalé, L., Skeel, R., Bhandarkar, M., Brunner, R., Gursoy, A., Krawetz, N., Phillips, J., Shinozaki, A., Varadarajan, K., Schulten, K.: NAMD2: Greater scalability for parallel molecular dynamics. Journal of Computational Physics 151(1), 283–312 (1999)
10. Koehler, A.: Scalable Cluster Computing with NVIDIA GPUs (2012), `http://www.hpcadvisorycouncil.com/events/2012/Switzerland-Workshop/Presentations/Day_3/3_NVIDIA.pdf`

11. Kupka, S.: Molecular dynamics on graphics accelerators (2006)
12. van Meel, J., Arnold, A., Frenkel, D., Portegies Zwart, S., Belleman, R.: Harvesting graphics power for md simulations. Molecular Simulation 34(3), 259–266 (2008)
13. NAMD on Biowulf GPU nodes, `http://biowulf.nih.gov/apps/namd-gpu.html` (accessed February 2013)
14. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Journal of Computational Physics 117, 1–19 (1995)
15. Podlozhnyuk, V.: CUDA Samples:: CUDA Toolkit documentation - NVidia's GPU Merge-sort implementations, `http://docs.nvidia.com/cuda/cuda-samples/index.html` (accessed February 2013)
16. Rapaport, D.: Large-scale Molecular Dynamics Simulation Using Vector and Parallel Computers. North-Holland (1988)
17. Rodrigues, C.I., Hardy, D.J., Stone, J.E., Schulten, K., Hwu, W.M.W.: GPU acceleration of cutoff pair potentials for molecular modeling applications. In: Proceedings of the 5th Conference on Computing Frontiers, CF 2008, pp. 273–282 (2008)
18. Rustico, E., Bilotta, G., Gallo, G., Herault, A., Negro, C.D.: Smoothed particle hydrodynamics simulations on multi-GPU systems. In: Euromicro International Conference on Parallel, Distributed and Network-Based Processing (2012)
19. Sanz-Serna, J.M.: Mollified impulse methods for highly oscillatory differential equations. SIAM J. Numer. Anal. 46(2), 1040–1059 (2008)
20. Schlick, T.: Molecular Modeling and Simulation: An Interdisciplinary Guide. Springer-Verlag New York, Inc., Secaucus (2002)
21. Schroeder, T.C.: Peer-to-Peer & Unified Virtual Addressing (2011), `http://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf`
22. Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., Schulten, K.: Accelerating molecular modeling applications with graphics processors. Journal of Computational Chemistry 28(16), 2618–2640 (2007)
23. Streett, W., Tildesley, D., Saville, G.: Multiple time-step methods in molecular dynamics. Molecular Physics 35(3), 639–648 (1978)
24. Wang, P.: Short-Range Molecular Dynamics on GPU (GTC 2010) (September 2010)
25. Yang, J., Wang, Y., Chen, Y.: GPU accelerated molecular dynamics simulation of thermal conductivities. Journal of Computational Physics 221(2), 799–804 (2007)

Algorithmic Skeleton Framework for the Orchestration of GPU Computations*

Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros

CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
2829-516 Caparica, Portugal
herve.paulino@fct.unl.pt

Abstract. The Graphics Processing Unit (GPU) is gaining popularity as a co-processor to the Central Processing Unit (CPU). However, harnessing its capabilities is a non-trivial exercise that requires good knowledge of parallel programming, more so when the complexity of these applications is increasingly rising. Languages such as StreamIt [1] and Lime [2] have addressed the offloading of composed computations to GPUs. However, to the best of our knowledge, no support exists at library level. To this extent, we propose Marrow, an algorithmic skeleton framework for the orchestration of OpenCL computations. Marrow expands the set of skeletons currently available for GPU computing, and enables their combination, through nesting, into complex structures. Moreover, it introduces optimizations that overlap communication and computation, thus conjoining programming simplicity with performance gains in many application scenarios. We evaluated the framework from a performance perspective, comparing it against hand-tuned OpenCL programs. The results are favourable, indicating that Marrow's skeletons are both flexible and efficient in the context of GPU computing.

1 Introduction

The GPU has been maturing into a powerful general processing unit, surpassing even the performance and throughput of multi-core CPUs in some particular classes of applications. The GPU architecture is specially tailored for data-parallel algorithms, where throughput is more important than latency. This makes them particularly interesting for high-performance computing on a broad spectre of application fields [3]. However, the base parallel computing frameworks for General Purpose Computing on GPUs (GPGPU), CUDA [4] and OpenCL [5], require in-depth knowledge of the underlying architecture and of its execution model, such as the disjointness of the host's and the device's addressing spaces, the GPU's memory layout, and so on. Consequently, high-level

* This work was partially funded by FCT-MEC in the framework of the PEst-OE/EEI/UI0527/2011 - Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL) - 2011-2012 and project PTDC/EIA-EIA/102579/2008 - Problem Solving Environment for Materials Structural Characterization via Tomography.

GPU programming is currently a hot research topic that has spawned several interesting proposals, e.g. OpenACC [6], Lime [2], Chapel [7] and StreamIt [1].

Nonetheless, as software developers become increasingly familiarised with GPU computing, and the overall hardware computational power is consistently growing, the complexity of GPU-accelerated applications is tendentiously higher. This *status quo* raises new challenges, namely how to efficiently offload this new class of computations to the GPU without overburdening the programmer. High-level programming frameworks will have to split their focus between the generation of OpenCL or CUDA kernels from higher level constructs, and the generation of the orchestration required on the host side. Languages such as StreamIt and Lime expand the use of the GPU beyond the usual offloading of a single kernel, offering more sophisticated constructs for streaming, pipelining and iterative behaviours. However, the impact of these constructs is restrained by the adoption of a new programming language.

To this extent, our proposal is to provide this expressive power at library level. For that purpose, we build on the concept of algorithmic skeleton to propose a framework for orchestrating the execution of OpenCL kernels that offers a diverse set of compoundable data- and task-parallel skeletons. The generation of OpenCL kernels from source code, namely C++, is orthogonal to this work and can grow from existing tools, such as the Offload compiler [8].

To the best of our knowledge, SkePU [9], SkelCL [10] and Muesli [11] are the sole Algorithmic Skeleton Frameworks (ASkFs) to address GPGPU. Nonetheless, all of them focus on the high-level expression of simple GPU computations, hence supporting only variants of the *map* skeleton that apply a user-defined function.

The contributions of this paper are therefore: 1 - the *Marrow* C++ ASkF for the orchestration of OpenCL computations (Section 3). Marrow pushes the state of the art by extending the set of GPU supported skeletons, introducing ones such as *pipeline*, *stream*, and *loop*, and by allowing these to be nested, thus providing a highly flexible programming model. Moreover, it is optimized for GPU computing, introducing transparent performance optimizations, specifically through a technique known as overlap between communication and computation. 2 - A comparative performance evaluation against OpenCL, whose experimental results attest the quality of our prototype implementation (Section 4).

2 Related Work

Skeletons are a high level parallel programming model that hide the complexity of parallel applications, by implicitly performing all the non-functional aspects regarding parallelization (e.g., synchronization, communication). Additionally, basic skeletons may be combined (nested) into more complex constructs, adding structural flexibility to the application. Skeletons are usually made available as algorithmic skeleton frameworks, analogous to common software libraries.

SkePU [9], SkelCL [10] and Muesli [11] are the only ASkFs to address GPU computing, using C++. They have many common features, focused solely on data-parallel skeletons. A Vector concept is used in all three to abstract data operations, and to introduce implicit optimizations, such as lazy copying. This feature postpones data transfers until needed, and allows different skeletons to access the same data without transferring it back to host memory. None of these ASkFs support skeleton nesting, thus no compound behaviours can be offloaded to the GPU. SkePU offers five skeletons: *map*, *reduce*, *map-reduce*, *map-overlap*, and *map-array*. The programmer can decide which is the target execution platform (CPU or GPU) at compilation time, as SkePU supports both CUDA and OpenCL. Skeleton behavior is specified using a set of predefined macros, although, macro compatibility is not universal among all skeletons. SkelCL supports four basic skeletons: *map*, *reduce*, *zip*, and *scan*. It generates OpenCL code from the aggregation of user-defined functions (supplied as strings) and predefined skeleton code. Muesli is a template library which supports both clusters, multi-core CPUs (OpenMP) and GPUs (CUDA). Among the multiple skeletons it supports, only *fold*, *map*, *scan* and *zip* are allowed on the GPU. More recently, AMD has announced Bolt [12], a C++ template library that provides OpenCL-accelerated *sort*, *transform* (similar to map) and *reduce* patterns. Data is defined through a Vector concept akin to the previous ASkFs, whist user-defined functions are defined through macros encasing a structure with C++ code.

The Lime [2] and StreamIt [13] programming languages provide primitives close to the ones we are proposing in this work, such as the *pipeline*. StreamIt is a stream processing language that enables the creation of disciplined graphs by combining three kinds of constructs: *pipeline*, *split-join* and *feedback-loop*. Recently it has been equipped with the CUDA back-end [1]. All the GPU execution is generated by the compiler, and optimized using a profiling stage. Lime is a language that supports kernel offloading and streaming constructs (akin to StreamIt) using a pipeline operator, while maintaining compatibility with the Java language. Data serialization is required to interface Java with C, presenting a considerable drawback, as data transfers become expensive.

When compared to our proposal, these languages have a narrower scope, as complex GPU applications are limited to the algorithms effectively compatible with the streaming programming model. As Marrow supports the offloading of complex skeletons compositions and streaming constructs, it breaks away from the current ASkF's data-parallel skeletons. This enables the usage of any GPU computation regardless of the applications' remaining purposes.

3 The Marrow Algorithmic Skeleton Framework

Marrow is a dynamic, efficient, and flexible C++ ASkF for GPGPU. We suggest constructs that orchestrate most of the details resulting from the underlying programming model, leaving the developer to focus on the parallel computations (kernels). The latter are common OpenCL kernels orchestrated by the skeletons to achieve a particular parallel application schema.

We are primary interested in skeletons whose execution behaviour is not hampered by the logical, and physical, division between the host and device address spaces. Skeletons whose execution is based on persistent data schemes are particularly attractive, since they do not require data transfers when combining

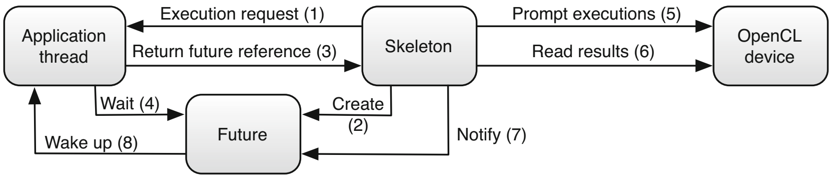


Fig. 1. Marrow's execution model

distinct execution instances. In this way, they avoid the overheads associated to transfers between disjoint memory spaces. Consider, for example, a *pipeline*; in a GPU execution the data produced by a stage i does not have to be transferred back to main memory in order to be available to the subsequent stage $(i+1)$. On the other hand, we target skeletons that provide functionalities that are useful in the usual GPGPU utilization domains. For instance, a *loop* skeleton for iterative computations is of particular interest to the scientific computing field.

We deem as fundamental the nesting of skeletons, as it enables the construction of complex executional structures, possibly containing distinct behaviours, in a simple and efficient manner. This technique is also beneficial performance-wise, in the sense that it is compatible with a disjoint memory layout. An application may apply a successive collection of computations, in the form of skeletons, to an input dataset, and only carry out memory transfers when: writing the input to device memory, and reading the results to main memory. Furthermore, the nesting mechanism embeds the construction of complex structures, allowing the skeletons to focus simply on particular functional behaviours.

Lastly, we seek to introduce transparent performance optimizations by taking advantage of the possibility to overlap communication with computation. By doing so, the skeletons can make better use of the features of modern GPUs, and increase overall efficiency. The transparent application of such optimization enables the development of efficient GPU accelerated applications without a large degree of knowledge of both parallel programming and OpenCL orchestration.

3.1 Execution Model and API

Marrow's execution model, depicted in Figure 1, promotes the decoupling of the skeleton computations from application execution. Given that GPU executions stretch through a somewhat extended period of time, it makes sense to free up the application to perform additional computations while the device carries out its task. This execution model can be regarded as master/slave pattern, on which the application offloads asynchronous executions requests. The submission of such a request to a skeleton (step 1 in the figure) is not immediately relayed to the device . Instead, it is queued, an associated *future* object is created (step 2) and its reference returned to the application (step 3). The *future* allows the application to, not only, query the state of the execution, but also wait until the results are ready (step 4). As soon as the skeleton becomes available to fulfil the execution request, it performs the necessary orchestration to properly carry out the desired computation on the device (step 5). Subsequently, once the results are read to the host memory (step 6) the respective future is notified (7), which, in turn, may wake up the submitting application thread (step 8).

This execution model motivates a rather simple API. Issuing a skeleton execution is accomplished through an asynchronous `write` operation that requests an OpenCL execution, and renders a future object.

3.2 Nesting

A nested skeleton application can be regarded as a composed acyclic graph (composition tree), on which every node shares a general computational domain. Each of these nodes can be categorized according to the interactions with its ancestor/children, the resources it manages, and the types of operations it issues to the device. The categories are: *root node*, *inner node*, and *leaf node*.

The root node is the primary element of the composition tree. It is responsible for processing the application's execution requests, which naturally implies submitting one or more OpenCL executions. Therefore, it must manage most of the resources necessary to accomplish such executions, as well as performing data transfers between host and device memory. Additionally, it prompts executions on its children, parametrizing them with a specific set of resources, e.g. the objects on which they must issue executions, or the memory objects that must use as input and output.

Inner nodes are skeletons whose purpose is to introduce a specific execution pattern/behaviour to their sub-tree. These nodes might not need to allocate resources, since they are encased in a computational context created by the root, but resort to that same context to issue execution requests on their children.

Leaf nodes should not be referred to as skeletons because they export an executional entity, rather than introducing a specific execution pattern. Consequently, they are represented by `KernelWrapper` objects that encapsulate OpenCL kernels, and are used to finalize the construction of the composition tree.

To be compatible with the nesting mechanism, i.e. become an inner-node, a skeleton must be able to perform its execution on pre-initialized device memory, issued by its ancestor. Furthermore, it should be able to share an execution environment with other nodes, even if it adds state (e.g., memory objects, executional resources) to that environment. By contrast, a skeleton whose executional pattern requires the manipulation of input/output data on host memory is incompatible with the nesting mechanism, and thus can only be used as a root node. In any case, a skeleton that supports nesting is also eligible to become the root of a composition tree.

Implementation: One of the main challenges imposed by skeleton nesting is the standardization of distinct computational entities, in a manner that provides a cross-platform execution support. The solution must abstract a single entity from the particularities of others, and yet, provide a simple and well defined invocation mechanism. Furthermore, there are issues intrinsic to GPU computing. For instance, the efficient management of the platform's resources craves the sharing of OpenCL structures - command-queues, memory objects, and the context (upon which skeletons may allocate resources) - between the skeletons that build up a composition tree.

To tackle these issues, skeleton nesting in Marrow is ruled by the `IExecutable` interface that must be implemented by every nestable skeleton, as well as `KernelWrapper`. The interface specifies a set of functionalities that together enable a multi-level nestable execution schema. These include some core requisites, such as the sharing of execution contexts, the convey of inner-skeleton executions, or even the provisioning of executional information to a node's ancestor.

Another challenge is the management of node communication and synchronization. Even though Marrow's execution flow runs downward, from the root to the leafs, nodes at different heights must be synchronized so that their application correctly follows the defined execution pattern. For instance, a node with two children has to ensure that these are prompted in the right order, and that each has its data dependencies solved before being scheduled to execute. Inter-node communication must also be taken into account, since it is vital that the nodes read from, and write to, the appropriate locations. Ergo, both synchronization and communication are seamlessly supported by the nesting mechanism, allowing skeletons to perform these complex tasks in the simplest way possible.

3.3 Overlap between Communication and Computation

Overlap between communication and computation is a technique that takes advantage of the GPU's ability to perform simultaneous bi-directional data transfers between memories (host and device), while executing computations related to one or more kernels. It reduces the GPU's idle time by optimizing the scheduling/issuing of operations associated to distinct kernel executions. However, introducing this optimization in the host's orchestration adds a considerable amount of design complexity, as well as requiring a good knowledge of OpenCL programming. For these reasons, it proves ideal to hide this complexity inside a skeleton, yet letting the developer tweak its parametrization if need be.

Applying concurrency between distinct OpenCL executions is, in itself, a complex exercise. More so, when the mechanism must be general enough to be encapsulated in a skeleton. The scheduling mechanism must optimize the device operation flow. Thus, as the number of skeleton execution requests rises, the framework must be aware, among others: the state of each request (what computation is being done); the resources allocated to the execution, and; how to further advance each execution. However, this by itself does not ensure parallelism. The fine-grain operations (e.g., reads, writes, executions) have to be issued to the OpenCL runtime in a manner that allows their parallel execution. It is not enough simply to launch the operations and expect OpenCL to schedule them in the best way possible. These previous issues gain complexity when the skeleton design includes nesting. Not only must the skeletons support combination between distinct entities, but also, these entities must work together to introduce concurrency to the execution requests. Consequently, every single skeleton must, at least, be supportive of concurrent executions, even if it does not, by itself, provide the overlap optimization.

Finally, the effectiveness of the overlap is directly proportional to where it is applied on the composition tree. The higher it is, the more sub-trees it affects. Hence, in order to maximize performance, it is always applied by the root node.

Implementation: Supporting multiple concurrent device executions implies the coexistence of multiple datasets in device memory. Therefore, a skeleton must allocate a number of memory objects that enables it to issue operations associated to distinct datasets, in an concurrent and independent manner. This strategy is designated as *multiple buffering*. Consider a skeleton s, as well as a kernel k that is parametrized with one buffer as input and another as output. The configuration of s that uses k to concurrently process three datasets at any given moment, requires the allocation of three sets of memory objects, totalling six memory objects.

The issuing of OpenCL operations to the device is performed via command-queues that offer two execution modes: in-order and out-of-order. The latter schedules the operations according to the device's availability, enabling their parallel execution, as our runtime requires. However, we have ascertained that not every OpenCL implementation supports such queues. Therefore, we opt to build our solution on top of in-order queues, one per set of memory objects. The scheduling responsibility is thus transferred to the Marrow runtime, which must enqueue the operations in such a way that they can be overlapped. This scheme can be scaled out as many times as needed, provided that the platform can supply the resources.

3.4 Supported Skeletons

Marrow currently supports the following set of task and data-parallel skeletons:

Pipeline efficiently combines a series of data-dependant serializable tasks, where parallelism is achieved by computing different stages simultaneously on different inputs in an assembly-line like manner. Considering the significant overhead introduced by memory transfers between host and device memories, this skeleton is ideal for GPU execution since the intermediate data does not need to be transferred back to the host in order to become available to next stage. This execution pattern is suitable for an execution that starts with pre-initialized device memory objects, and is fully able to compute in a shared execution environment. Accordingly, `Pipeline` supports nesting.

Loop applies an iterative computation to a given dataset. The result of each iteration is passed as input to the following (without unnecessary data transfers to host memory), until the final iteration is completed and its results provided as output. This construct supports two computational strategies: one where the loop's condition is affected by data external to the execution domain (a `for` loop), and another where the condition is affected by partial results of every iteration (a `while` loop). Analogously to the `Pipeline`, `Loop` fully supports nesting.

Stream defines a computational structure that confers the impression of persistence of the GPU computation. It achieves this by introducing parallelism between device executions associated to distinct datasets by applying the overlap technique (Subsection 3.3). To simplify the overall framework design only `Stream` provides such functionality. If this behaviour is desirable elsewhere, it is obtainable via nesting on a `Stream`, or its direct usage. Given that applying overlap requires direct control over the input data `Stream` is only qualified as a root node, and thus, is not nestable.

Map (and **MapReduce**) apply a computation to every independent partition of a given dataset, followed by an optional reduction of the results. Considering that this construct is designed for GPU computing, its behaviour differs from the general definition of a *map-reduce* skeleton. Firstly, a GPU should be used to process a large amount of data-elements, so as to compensate for its utilization overheads. Therefore, the input dataset is split into partitions, instead of singular data-elements, being the nested composition tree applied dependently to each of them. Secondly, GPUs are not particularly efficient when reducing a full dataset into a single scalar value. Instead, it is preferable to reduce part of the data in the GPU and return N elements to be finally reduced on the CPU, where N is a power of two larger than a certain threshold (that differs between devices). Thereupon, by default, this construct performs a host-side reduction. Nonetheless, it supports the provision of a reduction kernel, which is applied until the previously cited threshold is reached. Overlap may be efficiently applied between the multiple partition executions, yet, since these skeletons requires direct control over both input and output data, they do not support nesting. Therefore, they offer this feature by resorting to `Stream` internally.

3.5 Programming Example

Marrow's programming model comprises three major stages: *skeleton initialization*, *prompting of skeleton executions*, and *skeleton deallocation*. The first stage holds the `KernelWrapper`'s instantiation and appropriate parametrization, for their subsequent utilization as input parameters to the skeleton instantiation process. This may include nesting if desired by the programmer. In turn, the second stage defines how the application issues execution requests. Given the asynchronism of Marrow's execution model the application may adapt its behaviour to accommodate computations that are executed in parallel to the skeleton requests. Finally, the final stage is trivial. It simply requires the deallocation of the root node, since the latter manages all other skeleton related resources (e.g., inner skeletons, `KernelWrappers`).

Listing 1 illustrates an example that builds a three-staged image filter pipeline fed by a stream. Due to space restrictions we confine the presentation the declaration of a single kernelwrapper (line 2), to the nesting application (lines 3 to 7) and the prompting of the execution requests (lines 8 to 13). A `KernelWrapper` receives as input, the source file, the name of the kernel function, info about the input and output parameters, and the work size. `Pipeline` $p1$ is instantiated with the first two `KernelWrappers` - representing the first two stages. Then, `Pipeline`

$p2$ is created and parametrized with $p1$ along with the last `KernelWrapper`. Ultimately, the `Stream` s is instantiated with $p2$. This scheme creates a composition tree represented by $s(p2(p1))$, in which the kernels associated with the innermost skeleton are computed first. As shown, Marrow's skeletons do not distinguish kernels from nestable skeletons, thus standardizing the nesting mechanism. The prompting of a execution request (line 12) requires the specification of the input and output parameters, and returns a `future` object. In this particular application, the image is divided into segments and discretely feed to the `Stream`.

```
 1  // ... instantiate kernel wrappers
 2  unique_ptr<IExecutable> gaussKernel (new KernelWrapper(gaussNoiseSourceFile ,
         gaussNoiseKernelFunction , inputDataInfo , outputDataInfo , workSize));
 3  // instantiate inner skeletons
 4  unique_ptr<IExecutable> p1 (new Pipeline(gaussKernel , solariseKernel));
 5  unique_ptr<IExecutable> p2 (new Pipeline(p1 , mirrorKernel));
 6  // instantiate root skeleton
 7  Stream *s = new Stream(p2, 3); // Overlap with 3 concurrent executions
 8  // request skeleton executions
 9  for(int i = 0; i < numberOfSegments ; i++){
10    inputValues[0] = ... ; // offset in the input image
11    outputValues[0] = ... ; // offset in the output image
12    futures[i] = s->write(inputValues,outputValues);
13  }
14  //  wait for results ; delete s and resources (e.g. the futures)
```

Listing 1. Stream upon an image filter pipeline

4 Evaluation

The purpose of this study is to measure the overhead imposed by the Marrow framework relatively to straight OpenCL orchestrations, and the impact of the overlap optimization on overall performance. For that purpose we implemented four case-studies in OpenCL (without introducing overlap) and Marrow. All measurements were performed on a computer equipped with a Intel Xeon E5506 quad-core processor at 2.13GHz, 12GB of RAM, and a NVIDIA Tesla C2050 with 3 GB VRAM. The operating system is Linux (kernel 2.6.32-41) linked with the NVIDIA CUDA Developer Driver (295.41).

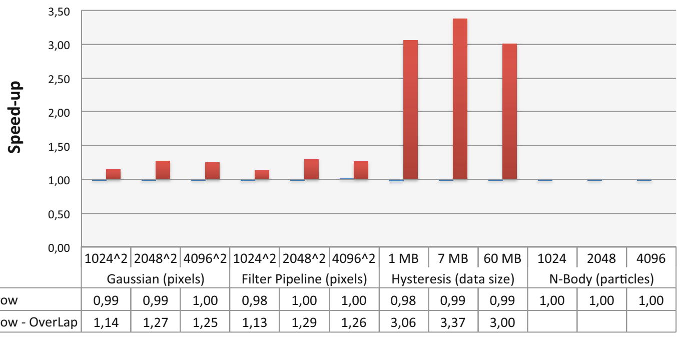
The first case-study applies a Gaussian Noise filter to an image that is split into non-superimposing segments. The OpenCL implementation processes these segments sequentially, whist the Marrow version submits then asynchronously for concurrent processing. Logically, the latter version adopts the `Stream` skeleton.

The second case-study is a pipelined application of image filters, namely Gaussian Noise, Solarise, and Mirror. Once again, we selected filters that are applicable to non-overlapping segments of an image so as to support an overlapped execution. Consequently, the application performs equivalently to the preceding case-study, differing only by applying multiple filters in succession to each slice. Naturally, the Marrow application uses `Pipelines` nested in a `Stream`.

The third case-study applies a tomographic image processing method, denominated as *Hysteresis*, that iteratively eliminates gray voxels from a tomographic image, by determining if they should be altered into white or black ones. The

Table 1. OpenCL case-studies execution times in milliseconds

	Gaussian Noise (pixels)			Filter Pipeline (pixels)			Hysteresis (MBytes)			N-Body (particles)		
Input parameter size	1024^2	2048^2	4096^2	1024^2	2048^2	4096^2	1	8	60	1024	2048	4096
Execution Time (ms)	3.18	11.82	46.36	3.34	12.46	48.95	402.98	2952.98	19742.80	37.77	78.23	174.61



	Gaussian (pixels)			Filter Pipeline (pixels)			Hysteresis (data size)			N-Body (particles)		
	1024^2	2048^2	4096^2	1024^2	2048^2	4096^2	1 MB	7 MB	60 MB	1024	2048	4096
🟦 Marrow	0,99	0,99	1,00	0,98	1,00	1,00	0,98	0,99	0,99	1,00	1,00	1,00
🟥 Marrow - OverLap	1,14	1,27	1,25	1,13	1,29	1,26	3,06	3,37	3,00			

Fig. 2. Speed-up versus OpenCL

algorithm comprises three data-dependent stages, each building upon the results of its predecessor to refine the elimination process. Once more the source images are split into segments, to which the processing performed by each stage is applied independently. However, the size and number of these segments are stage related, and may differ between stages. In any case, at every given stage the computations are iteratively applied to a single segment until the results stabilize. In the OpenCL version, segment processing within each stage is performed sequentially, whist the Marrow version nests a `Loop` into a `Stream` to perform the computation concurrently. Note that the mismatch between corresponding stage related segments prevented us to assemble the `Loops` in a pipelined execution.

The final case-study is an implementation of the particle-particle N-Body simulation algorithm ($O(n^2)$). In contrast with the previous case-studies the algorithm is not compatible with the partitioning of the input dataset. For that reason, the Marrow version resorts to a single `for` variant of the `Loop` skeleton, that makes no use of the overlap facility.

Performance Results: Table 1 presents the execution times of the OpenCL versions. These measurements isolate the time actually spent on the orchestration and execution of the GPU computation, on an input data of a certain grain, excluding, thus, the initialization and deallocation stages. The depicted values reflect the best results obtained by varying the global and local work size configuration parameters. In turn, Figure 2 displays the speed-up obtained by Marrow relatively to the OpenCL baseline, once more for the best work size configuration. The first version, `Marrow`, does not introduce overlap, and hence assesses

Table 2. Productivity comparison between distinct versions

	Gaussian Noise	Filter Pipeline	Hysteresis	N-Body
OpenCL basic/with overlap	61/261	81/281	165/365	98/298
Marrow	50	59	222	79

the framework's overhead, while the second, Marrow - Overlap, presents the best result when tweaking the overlap parameter.

The overhead introduced by the framework is minimum, peeking at 2%. Regarding the performance gains brought by the overlap optimization, the first two case-studies show a very similar behaviour. The balance between computation and communication is optimal for the application of our optimization at the medium grain. The Hysteresis' execution pattern differs from the remainder. Its execution flow dictates that after each loop iteration, which processes a single segment, the Loop reads the results to host memory, and subsequently evaluates them to access its continuity, a process of complexity $O(N)$ where N is the size of a segment. These two processes are computationally heavy and leave the GPU available to execute upon other datasets. Consequently, we assert the existence of a considerable amount of unexplored parallelism between segment executions. On top of that, the speed-up is incremental given that all three stages introduce it. Hence, it is directly propositional to the amount of overlap, consistently increasing with the latter, up to the maximum number of segments per stage.

Programming Model Evaluation: It comes as no surprise that our programming model is simpler, and of higher-level than OpenCL's, since it orchestrates the whole execution. To somehow quantify this judgement, Table 2 presents the number of lines of code for OpenCL, with and without introducing overlap, and Marrow. To introduce overlap in an OpenCL application we estimated a minimum increase of two-hundred lines of code, adding to the design complexity which would surely grow substantially.

Marrow's programming model productivity trumps the *with overlap* OpenCL versions and consistently requires less code per application than the basic ones. The Hysteresis case-study is an exception, requiring roughly more 40% of code than the OpenCL version. This increase in program size comes as a result of: a) the initializion of three Loops nested into three Streams is somewhat verbose, and b) the use of the Loop skeleton requires the derivation of a base class. Joining these two factors adds a considerable amount of lines of code to the application, justifying the discrepancy between OpenCL and Marrow versions.

5 Conclusions

This paper presented Marrow, a ASkF for the orchestration of OpenCL computations. Marrow distinguishes itself from the existing proposals by: (i) enriching the set of skeletons currently available on the GPGPU field; (ii) supporting skeleton nesting, and (iii) empowering the programmer to easily and efficiently exploit the ability of modern GPUs to overlap, in time, the execution of one or more kernels with data transfers between host and device memory.

Compared to the state of the art in ASkFs for GPU computing, Marrow takes a different approach. It focuses on the orchestration of complex OpenCL applications rather than the high level expressiveness of simple data-parallel kernels. This allows for a more flexible and powerful framework, whose kernels are bound only by OpenCL's restrictions, and whose skeleton set is richer and more modular. Naturally, as the programmer must express the parallel (kernels) in OpenCL, Marrow's abstraction of the underlying computing model is less effective than the one offered by the remainder.

The accomplished evaluation attested the effectiveness of these proposals. Compared to hand-tuned OpenCL applications that do not introduce overlap, the `Stream` skeleton consistently boosted performance without compromising the simplicity of the Marrow programming model. In addition, the remainder skeletons supply a set of high-level constructs to develop complex OpenCL based applications with negligible performance penalties.

References

1. Udupa, A., Govindarajan, R., Thazhuthaveetil, M.J.: Software pipelined execution of stream programs on GPUs. In: CGO 2009, pp. 200–209. IEEE Computer Society (2009)
2. Dubach, C., Cheng, P., Rabbah, R.M., Bacon, D.F., Fink, S.J.: Compiling a high-level language for GPUs: (via language support for architectures and compilers). In: PLDI 2012, pp. 1–12. ACM (2012)
3. hgpu.org: High performance computing on graphics processing units - Applications, `http://hgpu.org/?cat=11` (last visited in May 2013)
4. NVIDIA Corporation: NVIDIA CUDA, `http://www.nvidia.com/object/cuda_home_new.html` (last visited in May 2013)
5. Munshi, A., et al.: The OpenCL Specification. Khronos OpenCL Working Group (2009)
6. OpenACC: The OpenACC application programming interface (version 1.0) (2011), `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`
7. Sidelnik, A., Maleki, S., Chamberlain, B.L., Garzarán, M.J., Padua, D.A.: Performance portability with the Chapel language. In: IPDPS 2012, pp. 582–594. IEEE Computer Society (2012)
8. Codeplay Software: Offload compiler, `http://www.codeplay.com/compilers/` (last visited in May 2013)
9. Enmyren, J., Kessler, C.W.: SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: HLPP 2010, pp. 5–14. ACM (2010)
10. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - a portable skeleton library for high-level GPU programming. In: IPDPS 2011 - Workshops, pp. 1176–1182. IEEE (2011)
11. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. IJHPCN 7(2), 129–138 (2012)
12. AMD Corporation: Bolt C++ Template Library, `http://developer.amd.com/tools/heterogeneous-computing/` (last visited in May 2013)
13. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)

Author Index