Roman Wyrzykowski
Jack Dongarra
Ewa Deelman
Konrad Karczewski (Eds.)

LNCS 10778

Parallel Processing and Applied Mathematics

12th International Conference, PPAM 2017
Lublin, Poland, September 10–13, 2017
Revised Selected Papers, Part II

2 Part II

Springer

Roman Wyrzykowski · Jack Dongarra
Ewa Deelman · Konrad Karczewski (Eds.)

Parallel Processing and Applied Mathematics

12th International Conference, PPAM 2017
Lublin, Poland, September 10–13, 2017
Revised Selected Papers, Part II

Springer

Editors
Roman Wyrzykowski
Czestochowa University of Technology
Czestochowa
Poland

Jack Dongarra
University of Tennessee
Knoxville, TN
USA

Ewa Deelman
University of Southern California
Marina Del Rey, CA
USA

Konrad Karczewski
Czestochowa University of Technology
Czestochowa
Poland

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-78053-5 ISBN 978-3-319-78054-2 (eBook)
https://doi.org/10.1007/978-3-319-78054-2

Library of Congress Control Number: 2018937375

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG
part of Springer Nature
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This volume comprises the proceedings of the 12th International Conference on Parallel Processing and Applied Mathematics – PPAM 2017, which was held in Lublin, Poland, September 10–13, 2017. It was organized by the Department of Computer and Information Science of the Czestochowa University of Technology together with Maria Curie-Skłodowska University in Lublin, under the patronage of the Committee of Informatics of the Polish Academy of Sciences, in technical cooperation with the IEEE Computer Society and ICT COST Action IC1305 "Network for Sustainable Ultrascale Computing (NESUS)". The main organizer was Roman Wyrzykowski.

PPAM is a biennial conference. Ten previous events have been held in different places in Poland since 1994. The proceedings of the last six conferences have been published by Springer in the *Lecture Notes in Computer Science* series (Nałęczów, 2001, vol. 2328; Częstochowa, 2003, vol. 3019; Poznań, 2005, vol. 3911; Gdańsk, 2007, vol. 4967; Wrocław, 2009, vols. 6067 and 6068; Toruń, 2011, vols. 7203 and 7204; Warsaw, 2013, vols. 8384 and 8385; Kraków, 2015, vols. 9573 and 9574).

The PPAM conferences have become an international forum for the exchange of ideas between researchers involved in parallel and distributed computing, including theory and applications, as well as applied and computational mathematics. The focus of PPAM 2017 was on models, algorithms, and software tools that facilitate efficient and convenient utilization of modern parallel and distributed computing architectures, as well as on large-scale applications, including big data and machine learning problems.

This meeting gathered more than 170 participants from 25 countries. A strict review process resulted in the acceptance of 100 contributed papers for publication in the conference proceedings, while approximately 42% of the submissions were rejected. For regular tracks of the conference, 49 papers were selected from 98 submissions, giving an acceptance rate of 50%.

The regular tracks covered such important fields of parallel/distributed/cloud computing and applied mathematics as:

- Numerical algorithms and parallel scientific computing, including parallel matrix factorizations and particle methods in simulations
- Task-based paradigm of parallel computing
- GPU computing
- Parallel non-numerical algorithms
- Performance evaluation of parallel algorithms and applications
- Environments and frameworks for parallel/distributed/cloud computing
- Applications of parallel computing
- Soft computing with applications

The invited talks were presented by:

– Rosa Badia from the Barcelona Supercomputing Center (Spain)
– Franck Cappello from the Argonne National Laboratory (USA)
– Cris Cecka from NVIDIA and Stanford University (USA)
– Jack Dongarra from the University of Tennessee and ORNL (USA)
– Thomas Fahringer from the University of Innsbruck (Austria)
– Dominik Göddeke from the University of Stuttgart (Germany)
– William Gropp from the University of Illinois Urbana-Champaign (USA)
– Georg Hager from the University of Erlangen-Nurnberg (Germany)
– Alexey Lastovetsky from the University College Dublin (Ireland)
– Satoshi Matsuoka from the Tokyo Institute of Technology (Japan)
– Karlheinz Meier from the University of Heidelberg (Germany)
– Manish Parashar from Rutgers University (USA)
– Jean-Marc Pierson from the University Paul Sabatier (France)
– Uwe Schwiegelshohn from TU Dortmund (Germany)
– Bronis R. de Supinski from the Lawrence Livermore National Laboratory (USA)
– Boleslaw K. Szymanski from the Rensselaer Polytechnic Institute (USA)
– Michela Taufer from the University of Delaware (USA)
– Andrei Tchernykh from the CICESE Research Center (Mexico)
– Jeffrey Vetter from the Oak Ridge National Laboratory and Georgia Institute of Technology (USA)

Important and integral parts of the PPAM 2017 conference were the workshops:

– Workshop on Models, Algorithms, and Methodologies for Hierarchical Parallelism in New HPC Systems organized by Giulliano Laccetti and Marco Lapegna from the University of Naples Federico II (Italy), and Raffaele Montella from the University of Naples Parthenope (Italy)
– Workshop on Power and Energy Aspects of Computation — PEAC 2017 organized by Ariel Oleksiak from the Poznan Supercomputing and Networking Center (Poland) and Laurent Lefevre from Inria (France)
– Workshop on Scheduling for Parallel Computing — SPC 2017 organized by Maciej Drozdowski from the Poznań University of Technology (Poland)
– The 7th Workshop on Language-Based Parallel Programming Models — WLPP 2017 organized by Ami Marowka from Bar-Ilan University (Israel)
– Workshop on PGAS Programming organized by Piotr Bała from Warsaw University (Poland)
– Special Session on Parallel Matrix Factorizations organized by Marian Vajtersic from the University of Salzburg (Austria) and Slovak Academy of Sciences
– Minisymposium on HPC Applications in Physical Sciences organized by Grzegorz Kamieniarz and Wojciech Florek from the A. Mickiewicz University in Poznań (Poland)
– Minisymposium on High-Performance Computing Interval Methods organized by Bartłomiej J. Kubica from Warsaw University of Technology (Poland)
– Workshop on Complex Collective Systems organized by Paweł Topa and Jarosław Wąs from the AGH University of Science and Technology in Kraków (Poland)

The PPAM 2017 meeting began with three tutorials:

– Scientific Computing with GPUs, by Dominik Göddeke from the University of Stuttgart (Germany) and Robert Strzodka from Heidelberg University (Germany)
– Advanced OpenMP Tutorial, by Dirk Schmidl from RWTH Aachen University (Germany)
– Parallel Computing in Java, by Piotr Bała from Warsaw University (Poland), and Marek Nowicki from the Nicolaus Copernicus University in Toruń (Poland)

A new topic at PPAM 2017 was “Particle Methods in Simulations.” Particle-based and Lagrangian formulations are all-time classics in supercomputing and have been wrestling with classic mesh-based approaches such as finite elements for quite a while now, in terms of computational expressiveness and efficiency. Computationally, particle formalisms benefit from very costly inter-particle interactions. These interactions with high arithmetic intensity make them reasonably “low-hanging” fruits in supercomputing with its notoriously limited bandwidth and high concurrency.

Surprisingly, PPAM 2017 was shaped by articles that give up on expensive particle–particle interactions: discrete element methods (DEM) study rigid bodies which interact only rarely once they are in contact, while particle-in-cell (PIC) methods use the physical expressiveness of Lagrangian descriptions but make the particles interact solely locally with a surrounding grid. It is obvious that the lack of direct long-range particle–particle interaction increases the concurrency of the algorithms. Yet, it comes at a price. With low arithmetic intensity, all data structures have to be extremely fine-tuned to perform on modern hardware, and load-balancing has to be lightweight. Codes cannot afford to resort data inefficiently all the time, move around too much data, or work with data structures that are ill-suited for vector processing, while notably the algorithmic parts with limited vectorization potential have to be revisited and maybe rewritten for emerging processors tailored toward stream processing.

The new session “Particle Methods in Simulations” provided a platform for some presentations with interesting and significant contributions addressing these challenges:

– Contact problems are rephrased as continuous minimization problems coupled with a posteriori validity checks, which allows codes to vectorize at least the first step aggressively (by K. Krestenitis, T. Weinzierl, and T. Koziara)
– Classic PIC is recasted into a single-touch algorithm with only few synchronization points, which releases pressure from the memory subsystem (by Y. Barsamian, A. Chargueraud, and A. Ketterlin)
– Cell-based shared memory parallelization of PIC is revised from a scheduling point of view and tailored parallelization schemes are developed, which anticipate the enormous per-cell load imbalances resulting from clustered particles (by A. Larin et al.)
– Particle sorting algorithms are revisited that make the particles be stored in memory in the way they are later accessed by the algorithm even though the particles tend to move through the domain quickly (by A. Dorobisz et al).

Another new topic at PPAM 2017 was “Task-Based Paradigm of Parallel Computing.” Task-based parallel programming models have appeared in the recent years as an alternative to traditional parallel programming models, both for fine-grain and coarse-grain parallelism. In this paradigm, the task is the unit of execution and traditionally a data-dependency graph of the application tasks represents the application. From this graph, the potential parallelism of the application is exploited, enabling an asynchronous execution of the tasks that do not require explicit fork-join structures.

Research topics in the area are multiple, from the specification of the syntax or programming interfaces, the definition of new scheduling and resource management algorithms that take into account different metrics, the design of the interfaces with the actual infrastructure, or new algorithms specified in this parallel paradigm. As an example of the success of this paradigm, the OpenMP standard has adopted this paradigm in its latest releases.

This topic was presented at PPAM 2017 in the form of a session that consisted of several presentations from various topics:

- "A Proposal for a Unified Interface for Task-Based Programming Models That Enables the Execution of Applications in Multiple Parallel Environments" (by A. Zafari)
- "A Comparison of Time and Energy Oriented Scheduling for Task-Based Programs, Which Is Based on Real Measured Data for the Tasks Leading to Diverse Effects Concerning Time, Energy, and Power Consumption" (by T. Rauber and G. Rünger)
- "A Study of a Set of Experiments with the Sparse Cholesky Decomposition on Multicore Platforms, Using a Parametrized Task Graph Implementation" (by I. Duff and F. Lopez)
- "A Task-Based Algorithm for Reordering the Eigenvalues of a Matrix in Real Schur Form, Which Is Realized on Top of the StarPU Runtime System" (by M. Myllykoski)

A new topic at PPAM 2017 was the "Special Session on Parallel Matrix Factorizations." Nowadays, in order to meet demands of high-performance computing, it is necessary to pay serious attention to the development of fast, reliable, and communication-efficient algorithms for solving kernel linear algebra problems. Tasks that lead to matrix decomposition computations are undoubtedly some of the most frequent problems encountered in this field. Therefore, the aim of the special session was to present new results from parallel linear algebra with an emphasis on methods and algorithms for factorizations and decompositions of large sparse and dense matrices. Both theoretical aspects and software issues related to this problem area were considered for submission.

The topics of the special session focused on: (a) efficient algorithms for the EVD/SVD/NMF decompositions of large matrices, their design and analysis; (b) implementation of parallel matrix factorization algorithms on parallel CPU and GPU systems; (c) usage of parallel matrix factorizations for solving problems arising in scientific and technical applications. Seven papers were accepted for presentation, which covered the session topics. Geographically, the authors were dispersed among two continents and five countries. The individual themes of the contributions included:

- "New Preconditioning for the One-Sided Block-Jacobi Singular Value Decomposition Algorithm" (by M. Bečka, G. Okša, and E. Vidličková)

- "Using the Cholesky QR Method in the Full-Blocked One-Sided Jacobi Algorithm" (by S. Kudo and Y. Yamamoto)
- "Parallel Divide-and-Conquer Algorithm for Solving Tridiagonal Eigenvalue Problems on Manycore Systems" (by Y. Hirota and I. Toshiyuki)
- "Structure-Preserving Technique in the Block SS-Hankel Method for Solving Hermitian Generalized Eigenvalue Problems" (by A. Imakura, Y. Futamura, and T. Sakurai)
- "Parallel Inverse of Non-Hermitian Block Tridiagonal Matrices" (by L. Spellacy and D. Golden)
- "Tunability of a New Hessenberg Reduction Algorithm Using Parallel Cache Assignment" (by M. Eljammaly, L. Karlsson, and B. Kågström)
- "Convergence and Parallelization of Nonnegative Matrix Factorization (NMF) with Newton Iteration" (by R. Kutil, M. Flatz, and M. Vajtersic).

The organizers are indebted to the PPAM 2017 sponsors, whose support was vital for the success of the conference. The main sponsor was the Intel Corporation. Another important sponsor was Lenovo. We thank all the members of the international Program Committee and additional reviewers for their diligent work in refereeing the submitted papers. Finally, we thank all the local organizers from the Częstochowa University of Technology, and Maria Curie-Skłodowska University in Lublin, who helped us run the event very smoothly. We are especially indebted to Grażyna Kołakowska, Urszula Kroczewska, Łukasz Kuczyński, Adam Tomaś, and Marcin Woźniak from the Częstochowa University of Technology; and to Przemysław Stpiczyński and Beata Bylina from Maria Curie-Skłodowska University. Also, Paweł Gepner from Intel offered great help in organizing social events for PPAM 2017, including the excursion to the Zamoyski Palace in Kozłówka and the concert of the youth accordion orchestra "Arti Sentemo" at the Royal Castle in Lublin.

We hope that this volume will be useful to you. We would like everyone who reads it to feel invited to the next conference, PPAM 2019, which will be held during September 8–11, 2019, in Białystok, the largest city in northeastern Poland, located close to the world-famous Białowieża Forest.

January 2018

Roman Wyrzykowski
Jack Dongarra
Ewa Deelman
Konrad Karczewski

Organization

Program Committee

Jan Węglarz (Honorary Chair)	Poznań University of Technology, Poland
Roman Wyrzykowski (Program Chair)	Częstochowa University of Technology, Poland
Ewa Deelman (Program Co-chair)	University of Southern California, USA
Pedro Alonso	Universidad Politecnica de Valencia, Spain
Hartwig Anzt	University of Tennessee, USA
Peter Arbenz	ETH, Zurich, Switzerland
Cevdet Aykanat	Bilkent University, Ankara, Turkey
Marc Baboulin	University of Paris-Sud, France
David A. Bader	Georgia Institute of Technology, USA
Michael Bader	TU München, Germany
Piotr Bała	Warsaw University, Poland
Krzysztof Banaś	AGH University of Science and Technology, Poland
Olivier Beaumont	Inria Bordeaux, France
Włodzimierz Bielecki	West Pomeranian University of Technology, Poland
Paolo Bientinesi	RWTH Aachen, Germany
Radim Blaheta	Czech Academy of Sciences, Czech Republic
Jacek Błażewicz	Poznań University of Technology, Poland
Pascal Bouvry	University of Luxembourg
Jerzy Brzeziński	Poznań University of Technology, Poland
Marian Bubak	AGH Kraków, Poland and University of Amsterdam, The Netherlands
Tadeusz Burczyński	Polish Academy of Sciences, Warsaw, Poland
Christopher Carothers	Rensselaer Polytechnic Institute, USA
Jesus Carretero	Universidad Carlos III de Madrid, Spain
Raimondas Čiegis	Vilnius Gediminas Technical University, Lithuania
Andrea Clematis	IMATI-CNR, Italy
Zbigniew Czech	Silesia University of Technology, Poland
Pawel Czarnul	Gdańsk University of Technology, Poland
Jack Dongarra	University of Tennessee and ORNL, USA
Maciej Drozdowski	Poznań University of Technology, Poland
Mariusz Flasiński	Jagiellonian University, Poland
Tomas Fryza	Brno University of Technology, Czech Republic
Jose Daniel Garcia	Universidad Carlos III de Madrid, Spain
Pawel Gepner	Intel Corporation, Poland
Shamsollah Ghanbari	Universiti Putra, Malaysia

Domingo Gimenez	University of Murcia, Spain
Jacek Gondzio	University of Edinburgh, Scotland, UK
Andrzej Gościński	Deakin University, Australia
Laura Grigori	Inria, France
Inge Gutheil	Forschungszentrum Juelich, Germany
Georg Hager	University of Erlangen-Nuremberg, Germany
José R. Herrero	Universitat Politecnica de Catalunya, Barcelona, Spain
Ladislav Hluchy	Slovak Academy of Sciences, Bratislava, Slovakia
Sasha Hunold	Vienna University of Technology, Austria
Aleksandar Ilic	Technical University of Lisbon, Portugal
Florin Isaila	Universidad Carlos III de Madrid, Spain
Ondrej Jakl	Institute of Geonics, Czech Academy of Sciences, Czech Republic
Emmanuel Jeannot	Inria, France
Bo Kagstrom	Umea University, Sweden
Grzegorz Kamieniarz	A. Mickiewicz University in Poznań, Poland
Eleni Karatza	Aristotle University of Thessaloniki, Greece
Ayse Kiper	Middle East Technical University, Turkey
Jacek Kitowski	Institute of Computer Science, AGH, Poland
Joanna Kołodziej	Cracow University of Technology, Poland
Jozef Korbicz	University of Zielona Góra, Poland
Stanislaw Kozielski	Silesia University of Technology, Poland
Tomas Kozubek	Technical University of Ostrava, Czech Republic
Dieter Kranzlmueller	Ludwig-Maximillian University, Munich and Leibniz Supercomputing Centre, Germany
Henryk Krawczyk	Gdańsk University of Technology, Poland
Piotr Krzyżanowski	University of Warsaw, Poland
Krzysztof Kurowski	PSNC, Poznań, Poland
Jan Kwiatkowski	Wrocław University of Technology, Poland
Giulliano Laccetti	University of Naples Federico II, Italy
Marco Lapegna	University of Naples Federico II, Italy
Alexey Lastovetsky	University College Dublin, Ireland
Laurent Lefevre	Inria and University of Lyon, France
Joao Lourenco	University Nova of Lisbon, Portugal
Tze Meng Low	Carnegie Mellon University, USA
Hatem Ltaief	KAUST, Saudi Arabia
Emilio Luque	Universitat Autonoma de Barcelona, Spain
Piotr Luszczek	University of Tennessee, USA
Victor E. Malyshkin	Siberian Branch, Russian Academy of Sciences, Russian Federation
Pierre Manneback	University of Mons, Belgium
Tomas Margalef	Universitat Autonoma de Barcelona, Spain
Svetozar Margenov	Bulgarian Academy of Sciences, Sofia
Ami Marowka	Bar-Ilan University, Israel
Norbert Meyer	PSNC, Poznań, Poland

Marek Tudruj	Polish Academy of Sciences and Polish-Japanese Academy of Information Technology, Warsaw, Poland
Pavel Tvrdik	Czech Technical University, Prague, Czech Republic
Bora Ucar	Ecole Normale Superieure de Lyon, France
Marian Vajtersic	Salzburg University, Austria, and Slovak Academy of Sciences, Slovakia
Vladimir Voevodin	Moscow State University, Russian Federation
Kazimierz Wiatr	Academic Computer Center CYFRONET AGH, Poland
Bogdan Wiszniewski	Gdańsk University of Technology, Poland
Roel Wuyts	IMEC, Belgium
Andrzej Wyszogrodzki	Institute of Meteorology and Water Management, Warsaw, Poland
Ramin Yahyapour	University of Göttingen/GWDG, Germany
Jiangtao Yin	University of Massachusetts Amherst, USA
Krzysztof Zielinski	Institute of Computer Science, AGH, Poland
Julius Žilinskas	Vilnius University, Lithuania
Jarosław Żola	University of Buffalo, USA

Steering Committee

Jack Dongarra	University of Tennessee and ORNL, USA
Leszek Rutkowski	Częstochowa University of Technology, Poland
Boleslaw Szymanski	Rensselaer Polytechnic Institute, USA

Contents – Part II

Contents – Part I

Workshop on Models, Algorithms and Methodologies for Hybrid Parallelism in New HPC Systems

An Experience Report on (Auto-)tuning of Mesh-Based PDE Solvers on Shared Memory Systems

Dominic E. Charrier(✉) and Tobias Weinzierl

Department of Computer Science, Durham University, Durham, Great Britain
{dominic.e.charrier,tobias.weinzierl}@durham.ac.uk

Abstract. With the advent of manycore systems, shared memory parallelisation has gained importance in high performance computing. Once a code is decomposed into tasks or parallel regions, it becomes crucial to identify reasonable grain sizes, i.e. minimum problem sizes per task that make the algorithm expose a high concurrency at low overhead. Many papers do not detail what reasonable task sizes are, and consider their findings craftsmanship not worth discussion. We have implemented an autotuning algorithm, a machine learning approach, for a project developing a hyperbolic equation system solver. Autotuning here is important as the grid and task workload are multifaceted and change frequently during runtime. In this paper, we summarise our lessons learned. We infer tweaks and idioms for general autotuning algorithms and we clarify that such a approach does not free users completely from grain size awareness.

Keywords: Autotuning · Shared memory · Grain size
Machine learning

1 Introduction

Whenever a code is decomposed into parallel regions or tasks, the number of tasks determines the concurrency level and hence the code's potential to scale. It is common knowledge, however, that tasks must be reasonably computationally intense. Otherwise, the system spends precious time in administering the concurrency [5, p. 197]. Thus, modern parallelisation paradigms allow users to prescribe a *grain size*, a minimal subproblem size for parallel loops, while task-based approaches group logical tasks into one physical task if separate tasks were

This work received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671698 (ExaHyPE). It made use of the facilities of the Hamilton HPC Service of Durham University. The authors furthermore gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (www.lrz.de). All software is freely available from www.exahype.eu.

too lightweight. For plain bulk synchronous processing and non-nested tasks, finding grain sizes is often done manually via trial-and-error since developers assume that the size vs. performance curve is convex ([9, p. 37] or [6]).

Today, nested parallel loops perform efficiently—older OpenMP versions sometimes fail to deliver performance here—but yield a high-dimensional grain size optimisation problem. With the advent of manycores and hierarchical parallelisation, manual search becomes inappropriate. Sophisticated coherence protocols, performance fluctuations, and cache effects invalidate the convexity assumption to some degree. Task formalisms with inhomogeneous execution patterns gain importance. Machine learning (autotuning) which determines both the cost function and well-suited grain sizes becomes necessary. This manuscript discusses an autotuning approach that yields reasonable grain sizes in the ExaHyPE project [2], which combines dynamically adaptive Cartesian grids [12] with ADER-DG plus local limiting [3]. Support of interacting solvers with varying polynomial order (arithmetic intensity), inhomogeneous memory access characteristics and hierarchical hardware [11] render the use of autotuning mandatory. Our goal is two-fold: To present the algorithmic concept and rationale, and to document experiences on how this algorithm is made efficient and used efficiently. Our hypothesis is that autotuning never is a pure black box but that users have to have empirical knowledge to allow autotuning to integrate into software projects successfully and perform economically. Naïve coding of autotuning software is often ill-suited for HPC. Both goals interact.

We briefly sketch ADER-DG [3] in Sect. 2. Its task formulation is straightforward. However, the tasks differ significantly in arithmetic intensity, and some may have largely varying runtime. We then present our autotuning concept (Sect. 3). It tackles the grain size integer optimisation problem [7] parameterised by real-time measurements via randomised directional search. Emphasis is put on implementation pitfalls, e.g. the identification of valid real-time measurements. In Sect. 4, we discuss the algorithm's impact on the simulation workflow, before we present numerical results and close the discussion.

2 Use Case: An ADER-DG Solver

In the underlying ExaHyPE project, we solve hyperbolic PDEs

$$\frac{\partial Q}{\partial t} + \boldsymbol{\nabla} \cdot \mathbf{F}(Q) = 0 \qquad \text{on } \Omega \subset \mathbb{R}^{\mathrm{d}},\ \mathrm{d} = 2, 3 \tag{1}$$

subject to appropriate initial and boundary conditions. Q is the solution, $\mathbf{F}$ the conservative flux, d is the space dimension, $\boldsymbol{\nabla} \cdot (\cdot)$ denotes the tensor divergence, while $\boldsymbol{\nabla}(\cdot)$ is the vector gradient. We solve (1) on a dynamically adaptive Cartesian grid [12] with ADER-DG [3]. In its simplest form, used here, there are three phases per time step (Fig. 1).

Per grid cell K and time step interval $[t_a, t_b]$, we first implicitly solve

$$\int_K \int_{t_a}^{t_b} \theta_h \, \frac{\partial q_h}{\partial t} \, \mathrm{d}\boldsymbol{x}\mathrm{d}t + \int_K \int_{t_a}^{t_b} \theta_h \, \boldsymbol{\nabla} \cdot \mathbf{F}(q_h) \, \mathrm{d}\boldsymbol{x}\mathrm{d}t = 0. \tag{2}$$

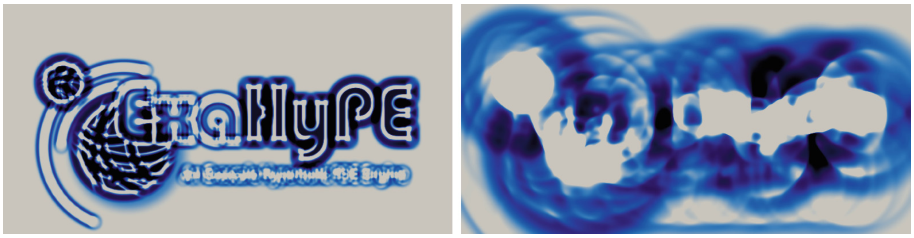


Fig. 1. Two snapshots from a $d = 2$ simulation of the Euler equations applied to an setup where the initial system energy (density) is determined by the project logo.

The space-time predictor q_h and the space-time test functions θ_h are constructed using tensor products of Lagrange polynomials over Gauss-Legendre points. Following Discontinuous Galerkin, they have compact support on each cell. Equation (2) yields a discrete fixed-point problem solved by Picard iterations [3]. All cell operations are independent of each other. The concurrent solves of (2) yield jumps along the cell faces in the solution q_h and its derivatives determining $\mathbf{F}$.

The second phase traverses all faces of the grid and computes a numerical normal flux G using q_h and $\mathbf{F}$ from both adjacent cells. We use a Rusanov Riemann solver. The solves are embarrassingly parallel with low arithmetic intensity.

In the third algorithmic phase, we traverse the cells again and solve

$$\int_K v_h \,\Delta q_h \,\mathrm{d}\boldsymbol{x} = -\int_K \int_{t_a}^{t_b} \boldsymbol{\nabla} v_h : \mathbf{F}(q_h)\,\mathrm{d}\boldsymbol{x}\mathrm{d}t + \int_{\partial K}\int_{t_a}^{t_b} v_h \,G\,\mathrm{d}s\mathrm{d}t \tag{3}$$

for $\Delta q_h = q_h(t_b) - q_h(t_a)$. The time step (3) is derived from spatially testing and partially integrating (1). It can be easily inverted given that the ansatz and test space yield a diagonal mass matrix, is evaluated per cell, and, hence, parallel.

ADER-DG describes three types of parallel tasks corresponding to phases. One is computationally heavy while two are lightweight. In our implementation, we either fuse the three task types within one grid sweep through a task formalism—one task then comprises a triad of predictor, Riemann solve and time step—or run through the grid three times and launch them through parallel force. The runtime of the heavy tasks can typically not been predicted due to the Picard iteration. There is no single grain size well-suited for all steps.

3 Programming an Autotuning Algorithm

Our autotuning approach picks up concepts from Intel's TBB [9]. There is a central instance, a singleton [4] which is notified by the overall algorithm regarding which algorithmic phase is to be run next. We call this instance `Oracle` [6].

Our code runs through the dynamically adaptive Cartesian grid. Whenever it enters a code section which has a multithreaded implementation using tasks or contains parallel for loops, it passes the maximum problem size N of the current subproblem, and an identifier for the current code section to the `Oracle`.

The `Oracle` then returns a `GrainSize` instance. The latter holds information on the grain size to be used and the number of logical tasks which can be grouped into one physical task. After the code exits the code section, the `GrainSize` object is destroyed again.

The `GrainSize` object can also be configured to measure the time which has elapsed since its creation. The measured time is then reported back to the `Oracle` at destruction. Proper move constructors ensure this is only done once.

3.1 Algorithmic Idea

The `Oracle` manages a database which stores, per entry, a code section, the algorithmic step, and further:

N_{max}	the maximum problem size w.r.t. code section and algorithmic step.
g	the grain size used for this problem; $g = N_{\mathrm{max}}$ indicates that parallelisation of this code section does not pay off.
Δg	the delta from g to the previously studied grain size with $g + \Delta g \leq N_{\mathrm{max}}$.
S_{old}	the speedup obtained with this previous grain size $g + \Delta g$.
t_{s}	the time per problem entity needed without parallelisation.
t_g	the time per problem entity needed if grain size g is used.

If no entry for these settings exists or $N > N_{\mathrm{max}}$, a new database entry with $(N_{\mathrm{max}} = N, g = C \cdot N, \Delta g = N - C \cdot N, S_{\mathrm{old}} = \infty, \ldots)$ is created. $C \in \{0.5, \frac{1}{p}\}$ for p threads are convenient choices as detailed later. The `Oracle` then determines a well-suited grain size for the calling code section: For $N > g$, the invoking code is instructed to use g as grain size. Otherwise, it runs serially.

Our algorithm realises interval halving similar to [6]: We start with relatively large g and compare the multithreaded performance to a serial setting. If the serial version is faster, we deactivate the parallelisation, i.e. we set $g = N_{\mathrm{max}}$. Otherwise, we successively shrink g with steps Δg until the resulting runtime starts rising again. Once we observe that g decrements make the runtime rise, we fall back to the previous choice of g and continue the descending search with $\Delta g/2$.

3.2 Implementation Pitfalls

Whilst our approach is realised straightforwardly and similar concepts have been proposed, we identified tiny details which decide whether it is successful. One important detail hereby is the notion of a "valid" timing. We do normalise all timings w.r.t. time per problem item: if a `GrainSize` for a problem of size N measures that the corresponding code lasts t, it reports back a time of t/N to the `Oracle`. Working with `GrainSize` instances ensures that overlapping parallelised code regions can be handled. Yet, all timings are subject to noise and, more importantly, any timing is only a characteristic sample if the underlying work per problem item is not constant. The latter is the case for our nonlinear equation system solves. Our `Oracle` thus tracks accumulated times and the number of measurements. The resulting average time is declared valid by an additional Boolean flag once a new measurement does not change the average by more than ϵ anymore. It is not evaluated for decision making before. *Linux system timers yield useless data if all code regions are paced simultaneously.* Timer invocations come along with an overhead which quickly pollutes all timings. Our solution is to introduce a global flag that determines for which code part a timer is enabled at all. After each grid sweep, this flag is randomly set to another parallel code fragment known. This way, only one code segment at a time is surveyed.

If we start to determine t_s first, the algorithm requires a long time to enable any parallelism at all. As all timings have to converge subject to ϵ, our simulation runs in serial for a while if the `Oracle` first determines the t_s entries in the database. This is not acceptable in HPC. Therefore, our `Oracle` randomises the grain size selection whenever it is invoked for a code fragment for which timings should be made. For one out of $N_{\max}/g$ samples, it instructs the invoking code to run serially and to report back the serial runtime. Otherwise, g shall be used and the parallel runtime t_g is updated. With shrinking grain sizes, i.e. longer simulation runtimes, fewer serial samples are taken. The sliding t_s updates anticipate that the serial timings of code parts change if parallel regions are embedded into each other that search for well-suited grain sizes, i.e. have not converged yet.

Proper constants C determine whether the algorithm exploits a reasonable number of cores in the first place. For $C = 0.5$ in the database entry's initialisation, the maximum initial concurrency equals two. In a multicore environment, this is not acceptable. We thus choose $C = 0.5$ for $N < 2p$, i.e. for small problems compared to the thread count p, and otherwise use $C = 1/p$.

The initial ϵ choice should take the runtime distribution into account. While we may expect runtime noise to cancel out for large data sets N and, thus, that those measurements converge quickly, it is particular important to come up with working grain sizes for large subproblems quickly as those dominate the walltime. In our code, we thus scale the initial ϵ with the total serial runtime of a source code fragment. If a code fragment requests a grain size first, we ask it to run serially and to report back the time. We then scale ϵ with this time: the longer a source code fragments runs serially the more relaxed ϵ.

No fixed ϵ works for all parts of the code. Some tasks in our application solve nonlinear equation systems. Furthermore, we have nested parallelism. While a too relaxed choice of ϵ makes the `Oracle` accept garbage measurements and terminate in suboptimal (local) grain size choices, a restrictive ϵ makes measurements for some code parts never yield valid results. We thus apply widening: After each grid sweep, we analyse whether the code fragment currently studied has been supplemented with new timings and whether those timings have switched on the valid flag for our timings. If this is not the case, we widen the admissibility constraint by 10 i.e. multiply ϵ with 1.1.

No fixed ϵ works all the simulation through. We work with large initial ϵ to come up with reasonable grain sizes choices quickly. We thus must accept inaccurate measurements at startup. Furthermore, runtime statistics do vary significantly as long as the grain sizes of embedded, nested parallel sections do vary. We thus half ϵ each time we have found a better grain size g or roll back to the previous grain size. Our `Oracle` increases the reliability of all data successively.

Track good grain sizes per problem size. We have to assume that a good grain size g depends not only on the algorithmic context but also on the problem size N. Our approach so far is N-agnostic. While a linear dependency on N might exist in some cases, we do not assume such a global relation here. Instead, our approach uses binning. We start searching for good grain sizes for $N_{\max} = 2$. If the code requests a grain size for $N > N_{\max}$, we recursively add new database entries for $2N_{\max}$. Per `Oracle` request, the database entry i is chosen for which $N_{\max}(i-1) < N \leq N_{\max}(i)$.

Restart measurements. After each grid sweep, we examine all database entries subject of search. If we observe that new measurements would have been made but all grain sizes belonging to the code fragment of interest are fixed, i.e. all database entries evaluated hold $\Delta g = 0$, we restart the search for these entries in one out of ten cases. This avoids that we stick to local minima always.

4 Using and Integrating Autotuning

Though we use the autotuning as black box, we found that the user has to remain aware of their integration into the simulation workflow: *Context-aware autotuning is mandatory.* We found our code to react sensitively to machine type, core count, and input data sets. Some data sets may perform poorly with autotuning settings derived for other data sets. This is likely an effect of the nonlinear subalgorithms, but certainly holds for many applications. It is thus important to work with independent autotuning searches per problem setup rather than one holistic database.

Autotuning for large data sets is problematic in large-scale compute environments. Autotuning temporarily runs into inefficient parameter choices (if the grain size becomes too small, e.g.), while large single node parameter studies for the many required parameter settings might be deemed unsuitable for supercomputers or not practical. At the same time, it is important to obtain autotuning configurations on the actual target machine that later shall host a large-scale run. We thus augment our binning. Whenever the database can not host an N, a new entry for a new $N_{\max}$ copies over all setting from the next smaller $N_{\max}$, scales them, and continues to work with those parameters. Further, if a valid parameter configuration is found for some $N_{\max}$, our approach extrapolates this to all database entries with larger $N_{\max}$ and then makes those restart their search. This allows us to run small-scale, yet characteristic runs briefly and to automatically extrapolate reasonable grain size to large production runs.

Accuracy improves over time, i.e. the more samples the more reliable the measurement data. It is thus a natural choice to dump and reload autotuning properties. It further is very reasonable to archive them alongside the simulation data.

Simulation re-runs then do not start autotuning searches from scratch but reuse performance knowledge.

We "sacrifice" only one node in a parallel environment. Autotuning introduces overhead. It has to be used carefully in large-scale simulations where all overheads have to be multiplied with the number of nodes used. We thus disable the autotuning's search on all MPI ranks besides one. All others read in the autotuning properties from a file and stick to those. The one rank tracking runtimes dumps all insight into a property file at the end of the simulation from where this knowledge becomes available to all other ranks in the next simulation. More sophisticated techniques may pass the responsibility for measurements from one rank to another throughout the simulation and propagate knowledge on-the-fly.

5 Computational Evidence

We start our computational exercises with the performance model

$$t_g = (1-\hat{f}) \cdot \frac{t_\mathrm{s}}{min\left(\left\lfloor \frac{N}{g} \right\rfloor, p\right)} + \hat{f} \cdot t_\mathrm{s} + h \cdot \left\lceil \frac{N}{g} \right\rceil \text{ with } \hat{f} = f + \frac{N \bmod g}{N}(1-f)$$

which extends Amdahl's law [1] by a task administration overhead h scaling linearly with the number of tasks. $f \in [0,1]$ is the code fraction not benefiting from multithreading at all. It enters the model through $\hat{f}$ which anticipates that problems might not be decomposed exactly.

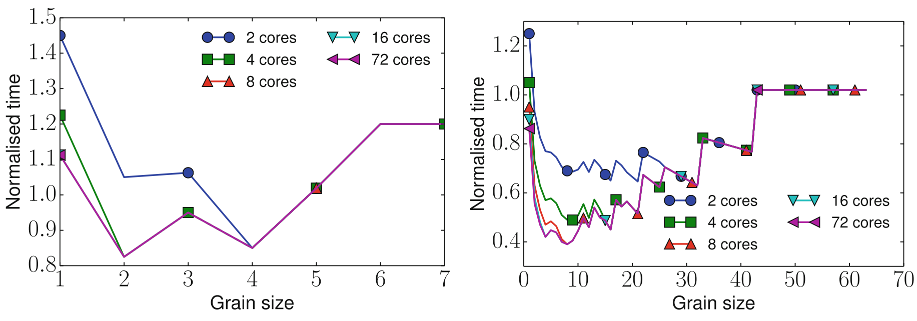


Fig. 2. Normalised time t_g/t_s according to our performance model for $N_\mathrm{max} = 8, f = 0.1, C = 10^{-1}$ (left) and $N_\mathrm{max} = 64, f = 0.2, C = 10^{-2}$ (right).

Our simplistic model relying on invariant t_s illustrates (Fig. 2) that one has to be careful not to choose the grain size too small to avoid overhead, while too large grain sizes do not yield good speedup. This is common knowledge. Different to textbooks [9] our speedups however do not develop smoothly but exhibit a non-convex step pattern. Finally, it might be reasonable not to choose a grain size for small problems that does keep all threads p busy and thus to spare cores.

The performance model motivates our decision to trigger the search for good grain sizes with half the maximum grain size for small problems and $1/p \cdot N_{\max}$ for bigger problems. As the difference between two local minima becomes the smaller the smaller g, it is reasonable to start with rather inaccurate time measurements (noise for large differences can be expected not to pollute any conclusion) and to increase the accuracy successively throughout the search. From our model, we derive that good autotuning searches for a different grain size per core number and problem size: it is reasonable to apply the binning.

Our runtime experiments were run on SuperMUC hosting Haswell Xeon E5-2697 v3 processors with 28 cores and 2.6 GHz base clock. All shared memory tests rely on Intel's TBB [9]. We studied five grain size selection strategies:

`serial` runs provide the measurement baseline and normalise all runtimes.

`dummy` is a choice of grain sizes per code part that does not anticipate the algorithmic context. We manually tuned it to yield good performance in many iterations.

`with-finest-grid` runs the autotuning strategy.

`from-coarse-grid` runs a cascade of autotuning experiments: it starts with a very coarse mesh, runs the autotuning, dumps the grain sizes identified, and then continues with the next finer mesh. We report only on the final run where the finest mesh sizes matches the other setups.

`from-coarse-grid-without-learning` takes the final dump of the cascading autotuning and reruns the test again but switches off the learning, i.e. no time measurements are done and grain sizes remain invariant.

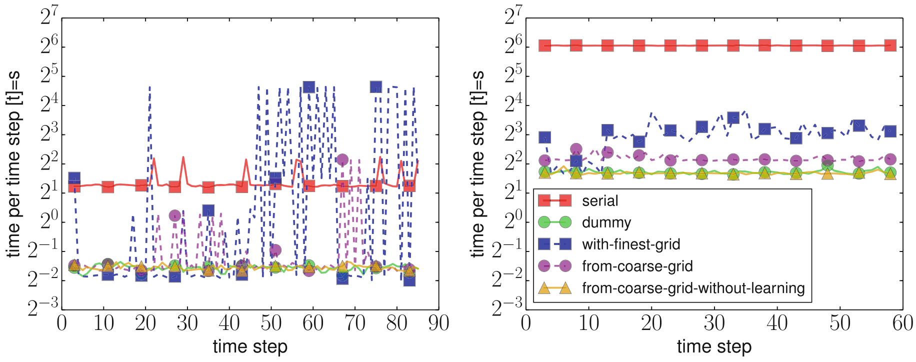


Fig. 3. Cost per time step for $d = 2$ Euler simulations where all three algorithmic steps are fused and we use polynomial order $p = 3$ (left) against a code where the three algorithmic phases are ran after each other with $p = 9$ (right).

Comparing cascading autotuning with the experiment switching off all measurements (Fig. 3) reveals that there is a significant overhead to do real-time measurements, and that there is a price to pay for the sliding updates of t_s. Once this overhead is removed, our autotuning can cope with a manual (and laborious) grain size selection. It thus makes sense to turn off autotuning wherever possible, notably on most MPI ranks.

Autotuning starting on the green field for a large problem does yield some valid grain sizes but the search process suffers from runtime spikes. The spikes result from unfortunate grain size choices that the autotuning tries and then discards. If we start autotuning on a coarse grid and then successively extrapolate the grain sizes to finer grids, we can remove the majority of these peaks.

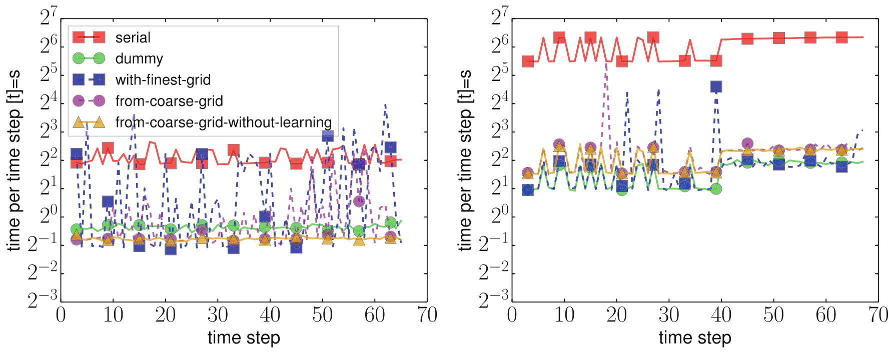


Fig. 4. Cost per time step for a $d = 2$ simulation of a shock where the ADER-DG solution is augmented with a Finite Volume limiter. $p = 3$ (left) vs. $p = 9$ (right) while all phases are fused into one grid sweep.

If we run the three ADER-DG phases consecutively, our autotuning requires longer to identify grain sizes able to compete with a manual optimisation (more than 60 time steps). It particularly struggles for the two arithmetically cheap phases. It is thus advantageous to try to fuse algorithmic phases—which can be read as a task fusion—to end up with computationally heavy individual steps.

We observe that our initial choice of $C \in \{0.5, 1/p\}$ ($C = 1/p$ is the OpenMP default for static partitioning) is reasonable. Already in the first iteration where the autotuning is unaware of $N_{\max}$, we exploit the multicore architecture. Once we switch from ADER-DG to limited ADER-DG (Fig. 4), autotuning becomes particularly important. Here, an additional Finite Volume scheme is interwoven into ADER-DG, eliminating numerical oscillations. As a consequence, the runtimes per cell start to vary greatly and it is hard to find globally valid good grain sizes. Our extrapolating approach is no longer robust and requires appropriate restart mechanisms.

The `Oracle`'s internal decisions are not visible from the plots. It first tries to remove parallelism from the code where parallel overhead increases the walltime. Only afterwards, it starts to tune the grain sizes for the scaling regions. Non-scaling features may significantly perturb the timings of the scaling regions and, thus, the `Oracle`'s decision making.

6 Conclusion

We describe an autotuning algorithm and summarise realisation decisions which made, throughout the development, the difference of whether the autotuning succeeds or not. Though the common perception of a convex runtime curve may be oversimplified, our autotuning yields proper grain size choices.

Our autotuning approach assumes codes which are completely decomposed into tasks and use parallel for loops wherever possible. Our algorithm first switches off parallelism where it does not pay off. Only then, it starts searching for optimal grain sizes for the remaining code sections. Such an approach, assuming omnipresent parallelism, seems to be a reasonable pattern for future code development. In terms of implementation difficulty, we regard it to be favourable to successive automated induction of concurrency.

An interesting next step is to augment the grain size optimisation with an additional constraint w.r.t. employed cores. We see that we can, at little loss of efficiency, for many setups reduce the number of used cores. For codes deploying multiple MPI ranks per node, other ranks then can grab these freed cores [10]. Furthermore, we believe that the proposed implementation pattern and on-the-fly autotuning approach can help with dynamic scheduling on heterogeneous systems. Here, tasks typically have to be reallocated to compute resources which differ in performance per thread and level of parallelism [8].

References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing. In: AFIPS Proceedings of the SJCC, vol. 31, pp. 483–485 (1967)
2. Bader, M., Dumbser, M., Gabriel, A., Igel, H., Rezzolla, L., Weinzierl, T.: ExaHyPE–An Exascale Hyperbolic PDE Engine (2017). http://www.exahype.org
3. Dumbser, M., Zanotti, O., Loubère, R., Diot, S.: A posteriori subcell limiting of the discontinuous Galerkin finite element method for hyperbolic conservation laws. J. Comput. Phys. **278**, 47–75 (2014)
4. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley Longman, Boston (1994)
5. Gerber, R.: The Software Optimization Cookbook-High-performance Recipes for the Intel Architecture. Intel Press, Hillsboro (2002)
6. Nogina, S., Unterweger, K., Weinzierl, T.: Autotuning of adaptive mesh refinement PDE solvers on shared memory architectures. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7203, pp. 671–680. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31464-3_68
7. Papadimitriou, C., Steiglitz, K. (eds.): Combinatorial Optimization: Algorithms and Complexity. Dover Publications Inc., New York (2000)
8. Reano, C., Silla, F., Leslie, M.J.: schedGPU: fine-grain dynamic and adaptative scheduling for GPUs. In: 2016 International Conference on High Performance Computing Simulation (HPCS), pp. 993–997, July 2016
9. Reinders, J.: Intel Threading Building Blocks. O'Reilly, Sebastopol (2007)

10. Schreiber, M., Riesinger, C., Neckel, T., Bungartz, H.J., Breuer, A.: Invasive compute balancing for applications with shared and hybrid parallelization. Int. J. Parallel Prog. **43**(6), 1004–1027 (2015)
11. Wahib, M., Maruyama, N., Aoki, T.: Daino: a high-level framework for parallel and efficient AMR on GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press (2016)
12. Weinzierl, T., Mehl, M.: Peano—a traversal and storage scheme for octree-like adaptive cartesian multiscale grids. SIAM J. Sci. Comput. **33**(5), 2732–2760 (2011)

Using GPGPU Accelerated Interpolation Algorithms for Marine Bathymetry Processing with On-Premises and Cloud Based Computational Resources

Livia Marcellino[1], Raffaele Montella[1](✉), Sokol Kosta[3], Ardelio Galletti[1], Diana Di Luccio[1], Vincenzo Santopietro[1], Mario Ruggieri[1], Marco Lapegna[2], Luisa D'Amore[2], and Giuliano Laccetti[2]

[1] Department of Science and Technologies, University of Napoli Parthenope, Naples, Italy
{livia.marcellino,raffaele.montella,ardelio.galletti,diana.diluccio, vincenzo.santopietro,mario.ruggieri}@uniparthenope.it

[2] Department of Mathematics and Applications, University of Napoli Federico II, Naples, Italy
{marco.lapegna,luisa.damore,giuliano.laccetti}@unina.it

[3] CMI, Aalborg University Copenhagen, Copenhagen, Denmark
sok@cmi.aau.dk

Abstract. Data crowdsourcing is one of most remarkable results of pervasive and internet connected low-power devices making diverse and different "things" as a world wide distributed system. This paper is focused on a vertical application of GPGPU virtualization software exploitation targeted on high performance geographical data interpolation. We present an innovative implementation of the Inverse Distance Weight (IDW) interpolation algorithm leveraging on CUDA GPGPUs. We perform tests in both physical and virtualized environments in order to demonstrate the potential scalability in production. We present an use case related to high resolution bathymetry interpolation in a crowdsource data context.

Keywords: GPGPU · Virtualization · High performance computing Geographic data · Interpolation

1 Introduction

The rise of democratically distributed computing power thanks to the astonishing achievements of low power embedded and mobile devices acted as a spinning wheel effect of the pervasive technology generally known as Internet of Things (IoT) [21]. The first and more touchable result is the increase of data raw availability gathered using ad-hoc sensor networks, sampling campaigns and data crowdsourcing. Focusing on the earth system science, managing spatial data and exploiting hidden knowledge makes the difference between reach the success in human activities development or completely fail it with potential not reversible environmental damages. The indirect sampling techniques available today enabled engineers and scientist to record high resolution land and ocean digital elevation models (DTM) using different instruments but facing the same problem: producing surface models from a finite, but huge, amount of measures irregularly distributed on the spatial domain. The problem size is characterized by a remarkable complexity due to the number of known points (where the measurements are already done and validated) and the new sampled point rate. The management problem became even more complex if data have to be periodically updated from different sources. From the computational point of view, the hierarchical and heterogeneous high performance computing paradigm delivery enough power to process spatial big data leveraging on massive multicore CPUs, general purpose graphic processing units (GPGPUs) and, recently field-programmable generic arrays (FPGAs) and supported by solid state storage skyrocketing the long term memory access performance [30]. While CPU virtualization and elastic storage is a common practice in public, private and hybrid clouds [28], the same techniques are not widely available due to the fact that often the accelerators leverage on closed technologies. In this paper we demonstrate how is possible to democratize the GPGPU resource availability for spatial marine data science leveraging on the GVirtuS [27] GPGPU virtualization service presenting a specific use case related to marine bathymetry processing. In particular we present a CUDA enabled innovative inverse distance weighting (IDW) interpolation algorithm comparing and contrasting the computation performance carried out on both on-premises and cloud computing scenarios both leveraging on GPGPU sharing and multiplexing.

Novelty. GVirtuS has been extended in order to support CUDA ancillary libraries enabling the use of a novel IDW algorithm based on cuBLAS.

Contributions. The rest of the paper is organized as follows: the Sect. 2 is about the related work on the different semantic components of the paper; in the Sect. 3 we show the acceleration infrastructure; the Sect. 4 is about the algorithm description, the design choices and the implementation techniques; the evaluation is carried out in the Sect. 5; finally in the Sect. 6 the conclusions and some future directions.

2 Related Work

About GPGPU virtualization. One of the most prominent solutions related to concurrent remote usage of CUDA-enabled devices in a transparent way is rCUDA [36]. Thanks to the split-driver approach, there is no need to modify and recompile the CUDA-enabled application in order to use it with rCUDA. Indeed, the framework takes care of all the necessary details in order to execute the CUDA kernels on a remote or local GPGPU [32]. The overhead introduced by using a remote GPU is evaluable as about less than 4% when a high performance network fabric is used [33]. At the time of writing, rCUDA delivers high performance CUDA virtualization [26,29] and it is up to date supporting the latest CUDA 8.0 framework and its ancillary libraries.

About CUDA interpolators. The most frequently used spatial interpolation algorithms in geographic information science include the Inverse Distance Weighting (IDW), Kriging, Discrete Smoothing Interpolation, nearest neighbors, etc. [10–12]; see a comparative survey investigated in [16]. As well-known, those interpolation algorithms have a computational cost very high when dealing with large-scale datasets. An effective approach to solve this problem is to perform interpolation algorithms in parallel [31]. There are many research efforts in this field, using different parallel computing architectures [35]. Among them, multicore-cluster approaches, parallel pipeline procedures, domain decomposition strategies. Recently, on the track of current developments of Graphics Processing Units (GPUs) for High Performance Computing (HPC) [9], interpolation algorithms have been accelerated with good results [14,19]. Here we will deal with the IDW interpolation algorithm, who has been parallelized on various platforms [24]. Our parallel implementation starts from [18] and proposes some variants in the algorithm design, in order to exploit the computational power of the NVIDIA cuBLAS library, to perform this basic linear algebra operation.

Algorithm 1. G-IDW

```
Require: locations $p(i)$, known values $z(i)$, query locations $q(j)$, search radius $R$
Ensure: unknown values $z^*(i)$
 1: // initialize $\alpha$
 2: $loc\_q \leftarrow q(tid)$
 3: for each chunk $c$ do
 4:    $i \leftarrow 0$
 5:    $start\_ind \leftarrow tid * stride$
 6:    while $(i < stride)$ and $(i + start\_ind) < size(c)$ do
 7:       // put $p_c(start\_ind + i)$ into shared memory
 8:       $i \leftarrow i + 1$
 9:    end while
10:    // synchronize threads
11:    for $i \leftarrow 1 \ldots size(c)$ do
12:       // $loc\_p \leftarrow p_c(i)$ from shared memory
13:       $d \leftarrow dist(loc\_p, loc\_q)$
14:       if $d \neq 0$ then
15:          if $d_{ij} < R$ then
16:             $\lambda \leftarrow d^{-\alpha}$; $z^*(tid) \leftarrow z^*(tid) + \lambda z_c(i)$; $wsum \leftarrow wsum + \lambda$
17:          end if
18:       else
19:          $z^*(tid) \leftarrow z_c(i)$; $wsum \leftarrow 1$
20:          // break and skip this cycle for the next chunks
21:       end if
22:    end for
23:    // synchronize threads
24: end for
25: // put $z^*(tid)/wsum$ into global memory
```

3 Acceleration Infrastructure

The overall architecture is described with more details, referring to GVirtuS (the GPGPU virtualization and remoting engine) [23,27], designed to be integrated and deeply cooperate in order to accelerate low-power devices.

3.1 GVirtuS GPU Code Offloading

The GPU virtualization architecture is based on a split-driver model [4], involves sharing a physical GPU. Hardware management is left to a privileged domain. A front-end driver runs in the unprivileged VM and forwards calls to the back-end driver in the privileged domain [15]. The back-end driver then takes care of sharing resources among virtual machines. This approach requires special drivers for the guest VM. The split driver model is currently the only GPU virtualization technique that effectively allows sharing the same GPU hardware between several VMs simultaneously [22]. This framework offers virtualization for generic GPU libraries on traditional x86 computers.

3.2 CUDA Ancillary Libraries

It's well known that NVIDIA provides a set of GPU-accelerated libraries containing several highly optimized algorithms for specific problems. For this reason, GVirtuS has been extended by providing the support for several CUDA ancillary libraries, such as cuBLAS. In order to extend the set of CUDA functions supported by GvirtuS, it's necessary to define three main components for each library that are responsible for the communication between the guest and host machine: (i) Front-end Layer; (ii) Back-end Layer; (iii) Function Handler. The first one contains the definitions of the wrapper functions called by the client, with the same signature as the library ones, where the name of the requested routine and the addresses of the input parameters, variables and host/device pointers, are encapsulated in a buffer that is sent to the back-end through a communicator.

4 Algorithm Description

The IDW is a deterministic method for spatial interpolation [34], based on the principle that near points have similar values. Let $p_i \in R^n, i = 1, \dots, N$, be the locations whose the values z_i are known. The interpolated value z_j^* of the j-th query location $q_j \in R^n$ is obtained by computing the weighted average of known value, as follows:

$$z_j^* = \frac{\sum_{i=1}^{N} \lambda_{ji} z_i}{\sum_{i=1}^{N} \lambda_{ji}} \tag{1}$$

where the weights λ_{ji} are defined by the Euclidean distance, as:

$$\lambda_{ji} = \frac{1}{dist(p_i, q_j)^{\alpha}} \tag{2}$$

In most application for each point only a subset of points is chosen with respect to a fixed radius R. Therefore, the weighted average in (1) is computed only for the sub-set $Q_j = \{p_i : d(p_i, q_j) < R\}$, i.e.:

$$z_j^* = \frac{\sum_{p_i \in Q_j} \lambda_{ji} z_i}{\sum_{p_i \in Q_j} \lambda_{ji}} \tag{3}$$

The IDW problem can be re-written as a matrix-vector problem as follows: considering a matrix Λ with M rows (the number of unknown values) and N columns (the number of locations p_i). The j-th row contains the weights λ_{ji} that are required to obtain the unknown value z_j^*. Then, indicated with z the vector that contains the known values, the unknown vector z^*, which contains the unknown values, is the solution of the following problem:

$$z^* = \Lambda z \tag{4}$$

We implemented two strategies for the IDW parallel algorithm on CUDA environment:

G-IDW: Each thread interpolates a different value computing the weight for each known value and updating the weighted mean at the same time. Block threads are synchronized to store dataset points into shared memory before the interpolation phase. For too large datasets, the points are stored into shared memory in different chunks.

G-IDW-MV: The matrix Λ is to compute, where the i-th row contains the weights for the i-th value to be interpolated. Threads are synchronized to store dataset points into shared memory as the first strategy. The i-th thread computes the elements of the i-th row. Λ is multiplied by the vector containing the known values. The i-th element of the result vector is divided by the sum of the weights for the i-th value in order to get the weighted mean. We computed the matrix-vector multiplication using two different approaches: the first one demands a thread for each scalar product; the second one (G-IDW-MVblas) uses the cuBLAS library. For the two strategies, the data transfer, Host-to-Device and vice versa, is based on two fundamental steps: the host sends to device the locations p_i with its related values z_i and the locations q_j corresponding to values to be estimated; the device sends to the host the computed values z_j^*.

The parallel pseudocodes related to the two strategies are shown in Algorithms 1 and 2. We called *tid* the number which uniquely identifies a thread and *stride* the number of dataset points which each thread in a block loads into shared memory. We use the c subscript to indicate a location or value belonging to the c-th chunk.

5 Use Case and Evaluation

Using commercial and leisure vessels as a sensor network for coastal protection and marine area management is an application field that could massively benefit from high-performance tools for big data collection, information processing, and dissemination of the generated metadata [6]. In this operational scenario, FairWind inserts as a smart, cloud-enabled, marine navigation software [25]. Data collected by FairWind from on-board sensor networks deployed in oceans represents a major challenge, as these devices generate huge amounts of geolocated data about the marine coastal environment [1]. By relying on a cloud-based file transfer protocol, collected data could be sent to remote computing facilities for conducting further processing with the aim of calibrating on-board instruments, and enhance depth maps [8]. We made several benchmarks of the proposed approaches, using the GVirtuS GPGPU virtualization service. The client is an Ubuntu based machine characterized by poor computational resources (single-core CPU, 1 GB of RAM), virtualized with KVM. The server machine is characterized by a Xeon E5-2609 v3 CPU, 64 GB of RAM and 2 Nvidia GeForce Titan X GPUs. Table 1 shows the execution times (Fig. 1) obtained by each algorithm, increasing firstly the number of the known values and secondly the number of the query locations. The execution times grow linearly by increasing the number of query locations.

Algorithm 2. G-IDW-MV

```
Require: locations $p(i)$, known values $z(i)$, query locations $q(i)$, search radius $R$
Ensure: unknown values $z^*(i)$
 1: // initialize $\alpha$ and $R$
 2: $loc\_q \leftarrow q(tid)$
 3: for each chunk $c$ do
 4:    for $i \leftarrow 1 \ldots size(c)$ do
 5:       // $loc\_p \leftarrow p_c(i)$ from shared memory
 6:       $d \leftarrow dist(loc\_p, loc\_q)$
 7:       if $d \neq 0$ then
 8:          if $d_{ij} < R$ then
 9:             $\lambda \leftarrow d^{-\alpha}$; $\Lambda(tid, column\,of\,p_c(i)) \leftarrow \lambda$; $wsum \leftarrow wsum + \lambda$
10:          else
11:             $\Lambda(tid, column\,of\,p_c(i)) \leftarrow 0$
12:          end if
13:       else
14:          // put all row values to 0
15:          $\Lambda(tid, column\,of\,p_c(i)) \leftarrow 1$; $wsum \leftarrow 1$
16:          // break and skip this cycle for the next chunks
17:       end if
18:    end for
19:    // synchronize threads
20: end for
21: // use a strategy to compute $\Lambda z$
22: $z^*(tid) \leftarrow z^*(tid)/wsum$
```

The use case is relative to bathymetry dataset extracted from EMODnet Digital Terrain Model (DTM) with original spatial resolution of 1/8

Table 1. Performance results

Known values	Query locations	IDW (s)	G-IDW (s)	G-IDW-MV (s)	G-IDW-MVblas (s)
10^3	10^4	2.365	0.006	0.009	0.009
10^3	10^5	23.711	0.045	0.09	0.089
10^3	$5 \cdot 10^5$	118.296	0.204	0.448	0.411
10^4	10^4	23.734	0.036	0.071	0.064
10^4	10^5	236.800	0.313	0.688	0.560
10^4	$5 \cdot 10^5$	1185.581	1.545	3.462	2.775
$5 \cdot 10^4$	10^4	120.293	0.186	0.406	0.276
$5 \cdot 10^4$	10^5	1183.199	1.757	4.315	2.62
$5 \cdot 10^4$	$5 \cdot 10^5$	5935.798	7.729	19.513	15.227

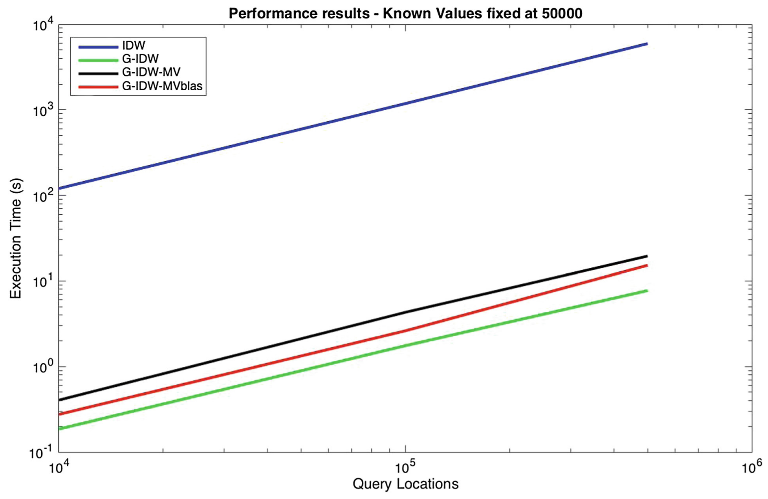


Fig. 1. Execution times of the proposed approaches.

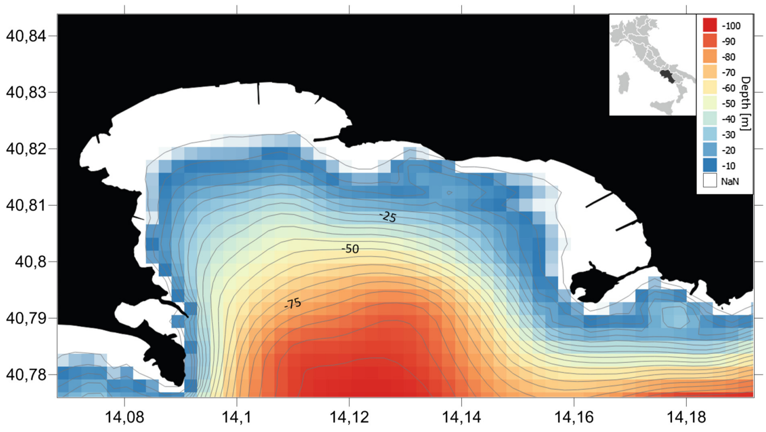


Fig. 2. EMODnet dataset interpolated with G-IDW algorithm on its original spatial resolution computational grid (1/8 * 1/8 arc minutes), fixed R = 400 m.

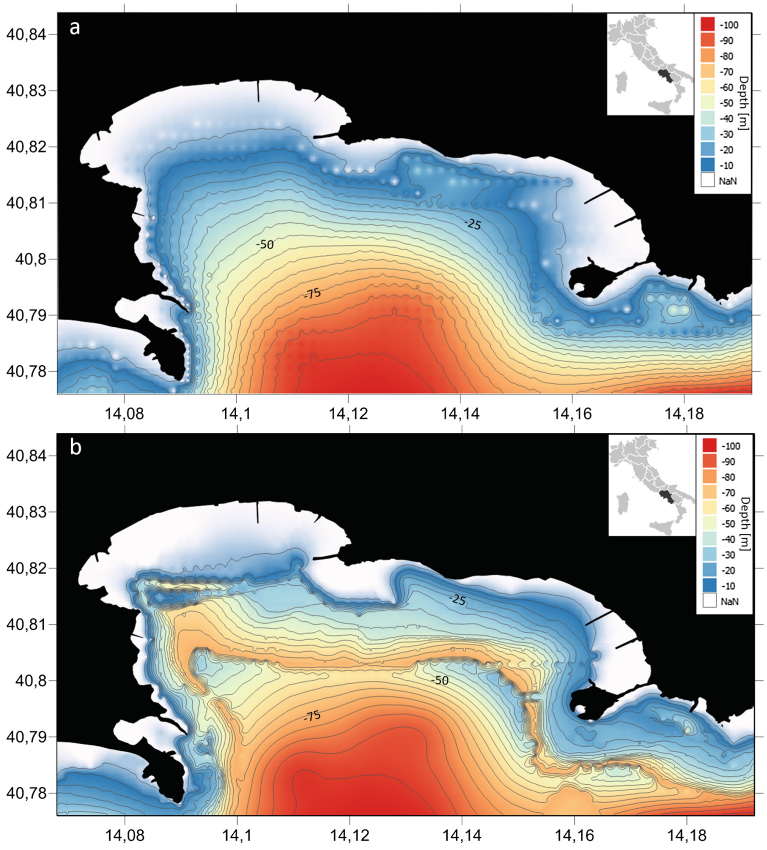


Fig. 3. Detail of bathymetry in Gulf of Pozzuoli (Italy). (a) EDMOnet dataset interpolated with G-IDW algorithm (R = 250 m) on about 25 m spatial resolution grid; (b) EDMOnet dataset added with 100000 crowdsourced punctual depth data interpolated on about 25 m spatial resolution computational grid (R = 250 m).

arc minutes in latitude and longitude. The used dataset ($Lat_{min} = 40.558°$N, $Lat_{max} = 40.84°$N, $Lon_{min} = 13.705°$E, $Lon_{max} = 14.490°$E), consisting of about 206908 points (including land mask points) was interpolated, using G-IDW algorithm, on its original spatial resolution computational grid, fixed R = 400 m. The obtained bathymetry (Fig. 2) has a low accuracy to detect regional sea phenomena, so we interpolated it still on 25 m spatial resolution grid, fixed R = 250 m (Fig. 3a), the new grid (about 2640387 points) is more dense but the depth information in coastal area are still few. To fill this gap, we increase the dataset with a cloud of 100000 points, between 0 and −75 m water depth, collected in marine data-crowdsourcing mode using FairWind. To do this, using the G-IDW algorithm, about 2740387 points have been interpolated on a 25 m spatial resolution lat-lon regular grid (Fig. 3b) to obtain a more accurate seabed morphology.

6 Conclusions and Future Directions

In this paper we demonstrate the performance achieved by our IDW implementation in both regular and virtualized environments. Nevertheless some improvements could be done considering other computational approaches on HPC systems [20]. An hierarchical approach combining distributed memory techniques [17] and performance contracts [7] could better exploit the highly distributed environment in which we set our prototype [5]. In the next steps, we will focused on infrastructure improvement with regard to cloud based data movement protocols in order to implement a reliable mechanism able to move acquired data from the logging equipment to the cloud infrastructure for processing and usage in data assimilation in order to improve the results produced by prediction models [3] with techniques [2] devoted to improve the scalability [13].

Acknowledgments. This research has been supported by the Grant Agreement no. 644312-RAPID-H2020-ICT-2014/H2020-ICT-2014-1 "Heterogeneous Secure Multilevel Remote Acceleration Service for Low-Power Integrated Systems and Devices (RAPID)" and by the project DSTE333 "Modelling mytilus farming System with Enhanced web technologies (MytiluSE)".

References

1. Ajmar, A., Balbo, S., Boccardo, P., Tonolo, G.F., Piras, M., Princic, J.: A low-cost mobile mapping system (LCMMS) for field data acquisition: a potential use to validate aerial/satellite building damage assessment. Int. J. Digit. Earth **6**(Suppl. 2), 103–123 (2013)
2. Arcucci, R., D'Amore, L., Celestino, S., Laccetti, G., Murli, A.: A scalable numerical algorithm for solving tikhonov regularization problems. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 45–54. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_5
3. Arcucci, R., D'Amore, L., Carracciuolo, L.: On the problem-decomposition of scalable 4D-Var data assimilation models. In: 2015 International Conference on High Performance Computing and Simulation (HPCS), pp. 589–594. IEEE (2015)
4. Armand, F., Gien, M., Maigné, G., Mardinian, G.: Shared device driver model for virtualized mobile handsets. In: Proceedings of the First Workshop on Virtualization in Mobile Computing, pp. 12–16. ACM (2008)
5. Boccia, V., Carracciuolo, L., Laccetti, G., Lapegna, M., Mele, V.: HADAB: enabling fault tolerance in parallel applications running in distributed environments. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7203, pp. 700–709. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31464-3_71
6. Van den Broek, A., Neef, R., Hanckmann, P., van Gosliga, S.P., Van Halsema, D.: Improving maritime situational awareness by fusing sensor information and intelligence. In: 2011 Proceedings of the 14th International Conference on Information Fusion (FUSION), pp. 1–8. IEEE (2011)

7. Caruso, P., Laccetti, G., Lapegna, M.: A performance contract system in a grid enabling, component based programming environment. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 982–992. Springer, Heidelberg (2005). https://doi.org/10.1007/11508380_100
8. Chard, K., Pruyne, J., Blaiszik, B., Ananthakrishnan, R., Tuecke, S., Foster, I.: Globus data publication as a service: lowering barriers to reproducible science. In: 2015 IEEE 11th International Conference on e-Science (e-Science), pp. 401–410. IEEE (2015)
9. Cuomo, S., De Michele, P., Galletti, A., Marcellino, L.: A parallel PDE-based numerical algorithm for computing the optical flow in hybrid systems. J. Comput. Sci. **22**, 228–236 (2016)
10. Cuomo, S., Galletti, A., Giunta, G., Marcellino, L.: A class of piecewise interpolating functions based on barycentric coordinates. Ricerche Mat. **63**(1), 87–102 (2014)
11. Cuomo, S., Galletti, A., Giunta, G., Marcellino, L.: A novel triangle-based method for scattered data interpolation. Appl. Math. Sci. **8**(133–136), 6717–6724 (2014)
12. Cuomo, S., Galletti, A., Giunta, G., Marcellino, L.: Piecewise hermite interpolation via barycentric coordinates: in memory of prof. carlo ciliberto. Ricerche Mat. **64**(2), 303–319 (2015)
13. D'Apuzzo, M., Lapegna, M., Murli, A.: Scalability and load balancing in adaptive algorithms for multidimensional integration. Parallel Comput. **23**(8), 1199–1210 (1997)
14. De Ravé, E.G., Jiménez-Hornero, F.J., Ariza-Villaverde, A.B., Gómez-López, J.: Using general-purpose computing on graphics processing units (GPGPU) to accelerate the ordinary kriging algorithm. Comput. Geosci. **64**, 1–6 (2014)
15. Dunlap, G.W., Lucchetti, D.G., Fetterman, M.A., Chen, P.M.: Execution replay of multiprocessor virtual machines. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 121–130. ACM (2008)
16. Falivene, O., Cabrera, L., Tolosana-Delgado, R., Sáez, A.: Interpolation algorithm ranking using cross-validation and the role of smoothing effect. A coal zone example. Comput. Geosci. **36**(4), 512–519 (2010)
17. Gregoretti, F., Laccetti, G., Murli, A., Oliva, G., Scafuri, U.: MGF: a grid-enabled MPI library. Future Gener. Comput. Syst. **24**(2), 158–165 (2008)
18. Henneböhl, K., Appel, M., Pebesma, E.: Spatial interpolation in massively parallel computing environments. In: Proceedings of the 14th AGILE International Conference on Geographic Information Science (AGILE 2011) (2011)
19. Huraj, L., Siládi, V., Siláci, J.: Design and performance evaluation of snow cover computing on GPUs. In: Proceedings of the 14th WSEAS International Conference on Computers: Latest Trends on Computers, pp. 674–677 (2010)
20. Laccetti, G., Lapegna, M., Mele, V., Romano, D., Murli, A.: A double adaptive algorithm for multidimensional integration on multicore based HPC systems. Int. J. Parallel Prog. **40**(4), 397–409 (2012)
21. Laccetti, G., Montella, R., Palmieri, C., Pelliccia, V.: The high performance internet of things: using GVirtuS to share high-end GPUs with ARM based cluster computing nodes. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013. LNCS, vol. 8384, pp. 734–744. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55224-3_69
22. Li, T., Narayana, V.K., El-Araby, E., El-Ghazawi, T.: GPU resource sharing and virtualization on high performance computing systems. In: 2011 International Conference on Parallel Processing (ICPP), pp. 733–742. IEEE (2011)

23. López, L., Nieto, F.J., Velivassaki, T.H., Kosta, S., Hong, C.H., Montella, R., Mavroidis, I., Fernández, C.: Heterogeneous secure multi-level remote acceleration service for low-power integrated systems and devices. Procedia Comput. Sci. **97**, 118–121 (2016)
24. Mei, G., Tian, H.: Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. SpringerPlus **5**(1), 104 (2016)
25. Montella, R., Di Luccio, D., Ferraro, C., Izzo, F., Troiano, P., Giunta, G.: FairWind: a marine data crowdsourcing platform based on internet of things and mobile/cloud computing technologies. In: 8th International Workshop on Modeling the Ocean (IWMO), Bologna, Italy, 7–10 June 2016
26. Montella, R., Giunta, G., Laccetti, G., Lapegna, M., Palmieri, C., Ferraro, C., Pelliccia, V.: Virtualizing CUDA enabled GPGPUs on ARM clusters. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 3–14. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_1. https://www.scopus.com/inward/record.uri?eid=2-s2.0-84964461702&doi=10.1007%2f978-3-319-32152-3_1&partnerID=40&md5=79bc02e92d87e0d0b24026a8c7196967
27. Montella, R., Coviello, G., Giunta, G., Laccetti, G., Isaila, F., Blas, J.G.: A general-purpose virtualization service for HPC on cloud computing: an application to GPUs. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7203, pp. 740–749. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31464-3_75
28. Montella, R., Foster, I.: Using hybrid grid/cloud computing technologies for environmental data elastic storage, processing, and provisioning. In: Furht, B., Escalante, A. (eds.) Handbook of Cloud Computing, pp. 595–618. Springer, Boston (2010). https://doi.org/10.1007/978-1-4419-6524-0_26
29. Montella, R., Giunta, G., Laccetti, G.: Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. Cluster Comput. **17**(1), 139–152 (2014)
30. Montella, R., Giunta, G., Laccetti, G., Lapegna, M., Palmieri, C., Ferraro, C., Pelliccia, V., Hong, C.H., Spence, I., Nikolopoulos, D.S.: On the virtualization of CUDA based GPU remoting on ARM and X86 machines in the GVirtuS framework. Int. J. Parallel Program. **45**(5), 1142–1163 (2017)
31. Murli, A., D'Amore, L., Laccetti, G., Gregoretti, F., Oliva, G.: A multi-grained distributed implementation of the parallel block conjugate gradient algorithm. Concurr. Comput.: Pract. Exp. **22**(15), 2053–2072 (2010)
32. Reaño, C., Silla, F.: A performance comparison of CUDA remote GPU virtualization frameworks. In: 2015 IEEE International Conference on Cluster Computing (CLUSTER), pp. 488–489. IEEE (2015)
33. Reaño, C., Silla, F.: Reducing the performance gap of remote GPU virtualization with InfiniBand Connect-IB. In: 2016 IEEE Symposium on Computers and Communication (ISCC), pp. 920–925. IEEE (2016)
34. Shepard, D.: A two-dimensional interpolation function for irregularly-spaced data. In: Proceedings of the 1968 23rd ACM National Conference, pp. 517–524. ACM (1968)
35. Shi, X., Ye, F.: Kriging interpolation over heterogeneous computer architectures and systems. GISci. Remote Sens. **50**(2), 196–211 (2013)
36. Silla, F., Prades, J., Iserte, S., Reano, C.: Remote GPU virtualization: is it useful? In: 2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), pp. 41–48. IEEE (2016)

Relaxing the Correctness Conditions on Concurrent Data Structures for Multicore CPUs. A Numerical Case Study

Giuliano Laccetti[1], Marco Lapegna[1](✉), Valeria Mele[1], and Raffaele Montella[2]

[1] Department of Mathematics and Applications, Università di Napoli Federico II, Napoli, Italy
{giuliano.laccetti,marco.lapegna,valeria.mele}@unina.it

[2] Department of Science and Technologies, Università di Napoli Parthenope, Napoli, Italy
raffaele.montella@uniparthenope.it

Abstract. The rise of new multicore CPUs introduced new challenges in the process of design of concurrent data structures: in addition to traditional requirements like correctness, linearizability and progress, the scalability is of paramount importance. It is a common opinion that these two demands are partially in conflict each others, so that in these computational environments it is necessary to relax the requirements on the traditional features of the data structures. In this paper we introduce a relaxed approach for the management of heap based priority queues on multicore CPUs, with the aim to realize a tradeoff between efficiency and sequential correctness. The approach is based on a sharing of information among only a small number of cores, so that to improve performance without completely losing the features of the data structure. The results obtained on a numerical algorithm show significant benefits in terms of parallel efficiency.

Keywords: HPC heterogeneous systems
Concurrent data structures · Programming models

1 Introduction

Concurrent data structures are widely used in many software stack levels, ranging from high level parallel scientific applications to low level operating systems. The key issue of these objects is their concurrent use by two or more threads (or processes) in a shared address space with a high risk of data inconsistency caused by a bad interleaving of hardware instructions (the so-called race condition). Such a problem makes the design of these structures much more difficult compared to their sequential counterpart, because of the need of a synchronization protocol to ensure data consistency [24].

Furthermore, the presence of multicore CPUs in last 10 years has dramatically changed the algorithms development methodologies, since these computing devices do not merely represent a new generation of CPUs. More precisely, it is widely recognized that their special architecture (based on several computing units sharing key resources such as memory, caches and buses) is forcing scientists towards new programming models and new requirements imposed to the execution of the algorithms, both in the scientific computing field and in the more general software design field [11,18,31]. Such studies emphasize a tension between the needs of sequential correctness and efficiency of algorithms, and in many cases it is evident the need to rethink the software design and data structure using approaches based on randomization and/or redistribution techniques in order to fully exploit the computational power of the multicore CPUs.

In very recent years, a new research trend is emerging, where the main underlying ideas are the relaxation of the semantic conditions required for the data structures and a non-deterministic execution of the algorithms. At the same time, the lack of a inherently sequential protocol, as one based on the critical sections, makes it possible a higher degree of concurrency [31].

Our work joins the previous research trend and it presents a relaxed approach to the management of a heap-based priority queue in multicore environment, aimed to achieve high scalability relaxing the strong conditions related to a strict replica of the behavior of the sequential data structure.

The rest of the paper is organized as follows: Sect. 2 provides an overview of the computing environment, focusing the attention on the multicore architectures and the model used to assess the performances. Section 3 is aimed to describe the core of the paper: a relaxed approach to heap-based priority queues in multicore environments. Section 4 shows numerical tests about the efficiency and the effectiveness of an algorithm using a relaxed heap described in Sect. 3. Finally, Sect. 5 concludes this paper.

2 The Computing Model

Current general-purpose multicore CPUs can be modeled as a collection of N processing elements (the cores) sharing a common memory. Updated examples of general-purpose multicore CPUs are nowadays the Intel XEON E7 v4 series with 16 cores, the AMD Opteron 6300 series with 16 cores and the IBM POWER8 series with 12 cores. In these CPUs, each core has its own set of processor registers so that the operating system is able to schedule independent threads among them, so that a general purposes shared-memory Single Program Multiple Data programming model can be used. To this end several software tools are available for the threads management, such as the POSIX Thread Library (pthreads) or the OpenMP environment. Even if each core can execute efficiently more than one thread, for our purpose let assume that on the N cores are in execution just N independent threads p_i $(i = 0, .., N-1)$, one for each core. In our computing model, a multi-threaded task is then represented by the spawning of N computing threads at the beginning of the execution, that run independently interacting among them by means of the shared memory, up to the waiting for their completion, according a fork-join model.

Based on this model, we define now $T(k, z)$ as the total elapsed time to complete a task with problem dimension z using k threads. For our purposes we then decompose it as [13]:

$$T(k, z) = T_s + \frac{T_c(z)}{k} + T_o(k) \tag{1}$$

where [13]:

- T_s is the running time for the serial sections of the algorithm. It is assumed that T_s is independent from z and from k;
- $T_c(z)/k$ is the running time for the parallelizable sections of the algorithm. It is assumed that parallelizable sections of the algorithm can be decomposed in k concurrent tasks of equal running time;
- $T_o(k)$ is the synchronization overhead. We assume that $T_o(1) = 0$ and that $T_o(k)$ is a not-decreasing function only depending on k.

Therefore, with these definitions in hand, we can define *scalable* an algorithm when, if the number of threads k and the problem size z are increased by a same factor N, the running time (1) remains the same [12]. When the original number of threads is $k = 1$, a formal definition for the scalability is for example:

$$R(N, z) = \frac{T(1; z)}{T(N; Nz)} \tag{2}$$

The (2) is often said *scaled efficiency* or *weak scalability* and its ideal value is $R(N, z) = 1$ but in practice a small degradation, due to operating system overhead, is acceptable. Furthermore, under the above assumptions 1–3, if $T_c(Nz)/N$ is a not decreasing function of N, it is easy to prove that:

$$R(N, z) \leq \frac{T(1, z)}{T(1, z) + T_o(N)} \leq 1 \tag{3}$$

The previous expression (3) represents an upper limit for the weak scalability when the number of threads N increases, and its strong dependence on the synchronization overhead $T_o(N)$.

3 A Loosely Coordinated Heap

It is very frequent that an application uses set of data not requiring a complete ordering, but only the access to some items tagged with high priority. For example many iterative algorithms attempt to reduce the numerical error accessing only the data with maximum error, or the process scheduling algorithms for real time applications need to access the data with the closest deadline. A priority queue $\mathcal{S}$ is a dynamical data structure where each node $s(k) \in \mathcal{S},\ (k = 1, .., K)$ is tagged with a problem-dependent priority $e(k)$. A very efficient priority queue is a heap, that is a partially ordered binary tree where each node has a priority higher than its children, so that the item $\hat{s}$ with highest priority $\hat{e} = \max_k e(k)$ is in the root (the so called *max-heap property*).

A heap, and more generally all dynamical data structures, is used when there is need to process items produced at run time by the execution of the algorithm, with an ordering depending on the application data and that cannot be envisaged. Therefore, in many scientific applications, the heap is periodically updated in an iterative section as in the following Algorithm 1.

Algorithm 1. Updating heap in an iterative algorithm

```
while (stopping criterion == false) do iteration $j$
  ...
   remove(max_priority_item)
   process data
   produce new items
   insert( new items)
   ...
endwhile
```

Usually, at each iteration j, the item with the highest priority is removed from the root, and it is replaced by two o more new elements. If all threads process data with high priority, a fast convergence of the iterative process is ensured.

In a multicore CPU, an efficient way to reorganize the Algorithm 1 is to process several nodes simultaneously by threads running on different cores, with the aim to share among them the items with the highest priority. In a centralized approach, where all threads access a single shared heap with a global synchronization, all the basic operations on the heap must be carried out in a critical section, with a strong scalability degradation.

In the case of N threads entering M times in a critical section one at a time, the total overhead is $T_o(N) = M(N-1)t_c = O(N)$, where t_c is the time to traverse the critical region, so that from (3) follows $R(N, z) \leq O(N^{-1})$. Because of the linear dependence on $T_o(N)$ on the number of threads N, the scaled efficiency $R(N, z)$ will quickly decreases, so the algorithm is poorly scalable. For these reasons, our first step in the development of a scalable heap, is to remove all global critical sections from the algorithm. To this aim, we give up the idea of a single centralized heap, and we reorganize the heap $\mathcal{S}$ in N separate heaps $\mathcal{S}_i$, one for each thread p_i, each of them accessing its private data structure without synchronizations with other threads. Without global synchronization, we have a pleasantly parallel algorithm (e.g. [12]), where it easy to show that $T_o(N) = const.$

However also this approach has a side effect: because the complete disjunction of the heaps $\mathcal{S}_i$, the N items with the highest priority $\hat{s_i}$ may not be the ones that globally have the highest priority, so that some threads can process unimportant items with no significant progress for the whole application. Therefore it is important to observe that, in case of items with priority very poorly distributed among the heaps, it should be desirable a periodical redistribution strategy for the $\hat{s_i}$ in order to balance the critical items among the threads. Actually the sequence of items with high priority is unpredictable, and it is impossible to distribute them uniformly among the several heaps before the computation, with the aim to process only the items with highest priority at each iteration. In other words we have to deal with the contraposition between algorithms with centralized heap requiring several global synchronizations where all threads always process items with high priority, and algorithms with data structures distributed in several locations without synchronizations but with the risk that some threads process unimportant items. Our approach is aimed to address the above contraposition, by means of a tradeoff based on the periodical reorganization of the items $\hat{s_i}$, only among a small group of threads, with a synchronization overhead that does not depend on the number of threads N. More precisely the N threads p_i are logically organized according a 2-dimensional periodical mesh $\mathcal{M}_2$. This structure is a virtual grid of $\Lambda_0 \times \Lambda_1 = N$ threads, arranged along the points of a 2-dimensional space with integer non negative coordinates, and where a shared buffer between each couple of connected nodes is established. The buffers are used to allow sharing data between two threads according to a producer-consumer protocol. In addition, the corresponding threads on the opposite faces of the mesh are connected too, so that the mesh is periodical. In a 2-dimensional periodical mesh, each thread p_i has 4 neighbors: 2 for each direction. In the horizontal direction ($dir = 0$), we define $p_{i-}^{(0)}$ and $p_{i+}^{(0)}$ respectively the leftmost and the rightmost thread of p_i in $\mathcal{M}_2$. Analogously in the vertical direction ($dir = 1$) we define $p_{i-}^{(1)}$ and $p_{i+}^{(1)}$ the lowermost and the uppermost threads of p_i.

We now define $\mathcal{S}^*$ a *loosely coordinate heap* [21] as a collection of partially ordered binary trees $\mathcal{S}_i \;\; i = 0, .., N-1$ with the max-heap property, where the roots are connected among them according to a given topology. In our loosely coordinated approach, at the iteration j, each thread p_i attempts to share its item $\hat{s}_i \in \mathcal{S}_i$, with highest priority $\hat{e}_i$ only with the next thread $p_{i+}^{(dir)}$ in the direction $dir = mod(j, 2)$ of the mesh $\mathcal{M}_2$, that is alternatively in the two horizontal and vertical directions. More precisely, in a fixed direction dir let $\hat{e}_i$ $\hat{e}_{i+}$ and $\hat{e}_{i-}$ be respectively the highest priority of the items in the heap root of the threads p_i, $p_{i+}^{(dir)}$ and $p_{i-}^{(dir)}$. If $\hat{e}_i > \hat{e}_{i+}$ then the item $\hat{s}_i \in \mathcal{S}_i$ with highest priority $\hat{e}_i$ is moved to the heap $\mathcal{S}_{i+}$ along the direction dir, using a producer-consumer protocol on the shared buffer. In the same way if $\hat{e}_{i-} > \hat{e}_i$ the item $\hat{s}_{i-} \in \mathcal{S}_{i-}$ with highest priority $\hat{e}_{i-}$ is moved to the heap $\mathcal{S}_i$. In this way, the critical items with highest priority are passed from thread in thread, an iteration after the other, through all the nodes of $\mathcal{M}_2$. The loosely coordinated heap management, described for the i-th thread p_i with a Single Program Multiple Data (SPMD) programming model, is then reported in Algorithm 2.

About the scalability, it should be noted that in the proposed data redistribution, at each iteration j, there are not global synchronizations among threads p_i and each of them exchanges data only with the two threads $p_{i+}^{(dir)}$ and $p_{i-}^{(dir)}$, so that the cost of threads synchronization is $T_o(N) = \mathcal{O}(1)$ because it does not depends on the number of threads N. From (3) it follows that $R(N, z) \leq const \leq 1$, so that the resulting algorithm can be considered scalable.

Algorithm 2. SPMD version of Algorithm 1 with a loosely coordinated heap

```
1 Determine $p_{i-}^{(dir)}$ and $p_{i+}^{(dir)}$, $(dir = 0, 1)$
2 initialize $\mathcal{S}_i$
3 **while** (stopping criterion == false) **do** iteration $j$
4     define $dir = mod(j, 2)$
5     share $\hat{e}_i$ with the closest threads $p_{i-}^{(dir)}$ and $p_{i+}^{(dir)}$
6     **if** $(\hat{e}_i > \hat{e}_{i+})$ **then**
7         **remove** $(\hat{s}_i)$ from $\mathcal{S}_i$
8         **produce** $(\hat{s}_i)$ for $p_{i+}^{(dir)}$
9     **endif**
10    **if** $(\hat{e}_{i-} > \hat{e}_i)$ **then**
11        **consume** $(\hat{s}_{i-})$ produced by $p_{i-}^{(dir)}$
12        **insert** $(\hat{s}_{i-})$ in $\mathcal{S}_i$
13    **endif**
14    remove(max_priority_item)
15    process data
16    produce two o more new items
17    insert( new items)
18    ...
19 **endwhile**
```

4 A Numerical Case Study

A scientific computing area where heaps are widely used is the development of adaptive algorithms for numerical computation of multidimensional integrals:

$$I(f) = \int_U f(\underline{x})\, d\underline{x} = \int_U f(x_1, ..., x_d)\, dx_1 \cdots dx_d, \tag{4}$$

In (4) $U = [a_1, b_1] \times \cdots \times [a_d, b_d]$ is a d-dimensional hyper-rectangular region. Because the scientific importance of such problem, since the 1980's several efficient parallel routines have been developed for its solution. Most of them (see for example [3,19,20,22]) are based on adaptive algorithms, that allow high accuracy with a reasonable computational cost.

An adaptive algorithm for the computation of (4) is an iterative procedure processing a family of hyper-rectangular subdomains $s(k)$ $(k = 1, .., K)$ of a partition $\mathcal{P}$ of U with the aim to compute a sequence $Q^{(j)}$ approaching $I(f)$ and a sequence $|E^{(j)}|$ of approximations of the error $|Q^{(j)} - I(f)|$ approaching 0, until a stopping criterion is not satisfied. Since the convergence rate of this procedure depends on the behavior of the integrand function (presence of peaks, oscillations, and so on), in order to reduce as soon as possible the error, at the iteration j, the algorithm splits in two parts, $s(\lambda)$ and $s(\mu)$, the subdomain $\hat{s} \in \mathcal{P}$ with maximum error estimate $\hat{e}$. The two new subdomains take the place of $\hat{s}$ in the partition $\mathcal{P}$, that is: $\mathcal{P} = \mathcal{P} - \{\hat{s}\} \cup \{s(\lambda)\ ,\ s(\mu)\}$. In a similar way the approximations $Q^{(j)}$ and $E^{(j)}$ are updated. A natural implementation of a such procedure can be done with a priority queue with the max-heap property, where the nodes of the heap $\mathcal{S}$ contain the subdomains $s(k)$ of the partition $\mathcal{P}$, and where the priority is represented by the error estimate $e(k)$ in each subdomain. The subdomain $\hat{s}$ to be split at the iteration j, with maximum error estimate $\hat{e}$ is then in the root of the tree. For such a reason, to implement an adaptive in a multicore based computing environment, it is possible to use the loosely synchronous approach to the heap management previously described in Sect. 3. More precisely, after the arrangement of the N threads p_i according a 2-dimensional periodical mesh $\mathcal{M}_2$ at the beginning of the algorithm, the integration domain U is fairly subdivided in N equal subdomains s_i, each of them assigned to a thread p_i. Such a step represents the initialization of $\mathcal{S}_i$ in p_i. Then, each thread p_i of the algorithm repeatedly refines the subdomains in the heap root $\hat{s}_i \in \mathcal{S}_i$, sharing them with the neighboring threads $p_{i-}^{(dir)}$ and $p_{i+}^{(dir)}$, along the 2 directions dir of $\mathcal{M}_2$.

To test the effectiveness of our approach we use a test functions family taken from the Genz's package [14]:

$$f(\underline{x}) = \begin{cases} 0 & \text{if } x_1 > \beta_1 \text{ or } x_2 > \beta 2 \\ \exp(\sum_{i=1,..,d} \alpha_i \, x_i) & \text{otherways} \end{cases}$$

where $U = [0,1]^d$ with $d = 10$ and $\alpha_i < 1$ and $\beta_i < 1$ are positive random value. The values of α_i are scaled according to $d^2 \sum \alpha_i = 100$ in order to control the difficulty of the function. The integrand function has a integrable discontinuity along the edges of the rectangle $[0, \beta_1] \times [0, \beta_2]$. For such a reason, the error estimate procedure will compute a very large error only in those subdomains where $x_1 = \beta_1$ or $x_2 = \beta_2$, that will be managed only by some threads. Without a suitable redistribution of the subdomains with large errors, only one thread will perform an useful job, while the other threads will refine subdomains where the error is already small enough.

For the experiments of this case study we use a computing system based on 2 CPUs Intel Xeon E5-4610 v2 with 8-core at 2.33 GHz, and a shared main memory DDR3 of 256 GB at 133 MHz. The system runs an operating system Scientific Linux 6.2, with GNU C compiler version 4.4 and POSIX Thread Library.

A first set of experiments is aimed to measure the scaled efficiency of the algorithm as defined in (2) with the described test function, with an approach similar to the one described in [10]. In this case the problem size z when $N = 1$ is the total number of evaluations of the integrand function $f(\underline{x})$. The quadrature rule on the basis of the adaptive algorithm is based on the Genz and Malik rule [15], and in $d = 10$ dimensions it requires $m = 1245$ function evaluations, so that the number of iterations, at the basis of the algorithm stopping criterion, can be computed by dividing the total number of function evaluations Nz by $2m$, because in each iteration the algorithm evaluate the quadrature rule in the two subdomains $s_i(\lambda)$ and $s_i(\mu)$. Therefore we run the algorithm in two way:

- **Option (a):** without redistribution procedure (i.e. without rows 4–13 in Algorithm 2): the integration domain U is equally distributed among the

N threads p_i and the calculation goes on without interaction among threads. In this case any difficulties in the integration domain are not shared among the threads.

- **Option (b)** with redistribution procedure (i.e. with rows 4–13 in Algorithm 2): after the same distribution of U among the threads, the computation attempts to balance the work load among the local data structures $\mathcal{S}_i$ of the loosely coordinated heap $\mathcal{S}^*$, as described in Sect. 3. In this case the difficulties in the integration domain are shared among the threads.

To this aim we used 10 different integrand functions with different values of α_i and β_i in order to test different locations and sharpness of the discontinuity.

In Fig. 1 are reported respectively the scaled efficiencies of the algorithm for both tests with Option (a) and Option (b). More precisely are reported the best value, the worst value and average values, over the 10 test functions, of the scaled efficiency as defined in (2) with $N = 16$ threads and $z = 250000$, 500000, 750000, 1000000 integrand function evaluations.

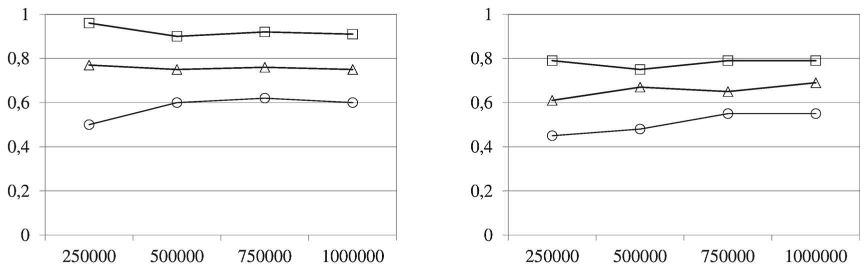


Fig. 1. Scaled efficiency of the adaptive algorithm. Left: 16 threads and Option (a). Right: 16 threads and Option (b). ○ = worst case - △ = average case - □ = best case

Mainly for large values of the number of function evaluations ($z = 750000$ and $z = 1000, 000$) we observe a very small difference between the two cases, confirming, once again, our expectation of a small impact of the redistribution procedure among the threads on the scaled efficiency. In the worst case, the scaled efficiency is about $R(N, z) \simeq 0.6$ with both Options (a) and (b), and it is only 0.1 larger with Option (a) with respect to Option (b) in the best case. The average values differ of only 0.05 between Option (a) and Option (b).

A second set of experiments is aimed, instead, to measure the benefit of the proposed redistribution procedure on the accuracy of the results. This is a critical experiment because it is tested the ability of the loosely coordinated heap $\mathcal{S}^*$ to supply effectively high-priority items to the threads. Also in this case we used the function described for the previous experiment. In Fig. 2 are reported the estimated numerical error, with $N = 1$ thread and $N = 16$ threads. Furthermore, with $N = 16$ threads we executed the Algorithm 3 with Option (a) and Option (b). In all cases we report the numerical error with $z = 250000, 500000, 750000, 1000000$ integrand function evaluations for each thread, so that the total number of function evaluations is Nz. It is possible to observe the great benefit on the numerical error, achieved when the number of function evaluations z in each thread is increased, as well as when are used $N = 16$ threads with Option (b). On the contrary, the execution of the algorithm with $N = 16$ threads and Option (a) produces a poorly significant improvement in numerical accuracy respect to the case with $N = 1$ thread. That means that, in this case, only 1 thread performs useful work.

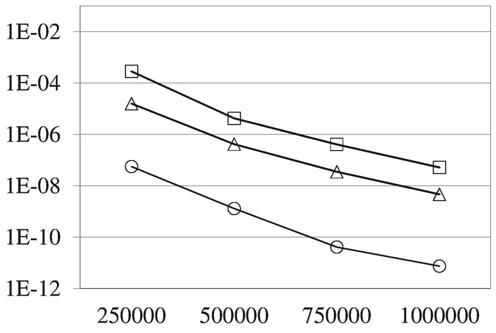


Fig. 2. Numerical error vs number of function evaluations $\circ = 16$ threads with Opt. (b) - $\triangle = 16$ threads with Opt. (a) - $\square = 1$ thread

5 Conclusion

In this paper we proposed a relaxed model for heap-based priority queues in multicore environments. The work is motivated by the need to achieve a balance between two contrasting requirements on the data structure: correctness and scalability. The first one requires global access to the data structure in order to assess traditional issues such as the linearizability [17]; on the other hand, high efficiency can be achieved in parallel environments only if the synchronization cost is independent from the number of processing units. To this end, we have developed an approach based on a distribution of the data structure among the computing units where the synchronization strategy involves only a small (and constant) number of processing units.

Our experiments show that such a strategy is able to realize an effective compromise between the two requirements. More precisely, we compared our algorithm with a pleasantly version of the same algorithm without redistribution of the node in the data structure, and we observed a gain of at least 3 significant digits in accuracy with a loss of efficiency of only 5% in the average case.

In any case it should be noted that modern computing environments are based on hybrid forms of parallelism, where large clusters are connected together by means of grid and/or cloud infrastructures. Therefore, we plan to integrate the resulting software in geographically distributed systems or computing environments (e.g. [6,7]) as we did already for other applications in

[1,2,8,9,16,23,25,26], paying special attention to the techniques developed to enhance the performance [5], the fault tolerance [4], the transparent use of resources [27,28,30] and the load balancing among them [29].

References

1. Arcucci, R., D'Amore, L., Celestino, S., Laccetti, G., Murli, A.: A scalable numerical algorithm for solving tikhonov regularization problems. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 45–54. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_5
2. Arcucci, R., D'Amore, L., Carracciuolo, L.: On the problem-decomposition of scalable 4D-Var data assimilation models. In: Proceedings of the 2015 International Conference on High Performance Computing and Simulation, pp. 589–594 (2015)
3. Berntsen, J., Espelid, T., Genz, A.: Algorithm 698: DCUHRE - an adaptive multidimensional integration routine for a vector of integrals. ACM Trans. Math. Softw. **17**, 452–456 (1991)
4. Boccia, V., Carracciuolo, L., Laccetti, G., Lapegna, M., Mele, V.: HADAB: enabling fault tolerance in parallel applications running in distributed environments. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7203, pp. 700–709. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31464-3_71
5. Caruso, P., Laccetti, G., Lapegna, M.: A performance contract system in a grid enabling, component based programming environment. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 982–992. Springer, Heidelberg (2005). https://doi.org/10.1007/11508380_100
6. D'Ambra, P., Danelutto, M., di Serafino, D., Lapegna, M.: Advanced environments for parallel and distributed applications: a view of current status. Parallel Comput. **28**, 1637–1662 (2002)
7. D'Ambra, P., Danelutto, M., di Serafino, D., Lapegna, M.: Integrating MPI-based numerical software into an advanced parallel computing environment. In: Proceedings of 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing, Euro-PDP 2003, pp. 283–291 (2003)
8. D'Amore, L., Marcellino, L., Mele, V., Romano, D.: Deconvolution of 3D fluorescence microscopy images using graphics processing units. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7203, pp. 690–699. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31464-3_70
9. D'Amore, L., Laccetti, G., Romano, D., Scotti, G., Murli, A.: Towards a parallel component in a GPU-CUDA environment: a case study with the L-BFGS Harwell routine. Int. J. Comput. Math. **92**, 59–76 (2015)
10. D'Apuzzo, M., Lapegna, M., Murli, A.: Scalability and load balancing in adaptive algorithms for multidimensional integration. Parallel Comput. **23**, 1199–1210 (1997)
11. Dongarra, J., Gannon, D., Fox, G., Kennedy, K.: The impact of multicore on computational science software. CTWatch Q. **3**(1), 1–10 (2007)
12. Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A.: Sourcebook of Parallel Computing. Morgan Kaufmann Publishers, Burlington (2003)

13. Flatt, H.P., Kennedy, K.: Performance of parallel processors. Parallel Comput. **12**, 1–20 (1989)
14. Genz, A.: Testing multiple integration software. In: Ford, B., Rault, J.C., Thommaset, F. (eds.) Tools, Methods and Language for Scientific and Engineering Computation. North Holland, New York (1984)
15. Genz, A., Malik, A.: An embedded family of fully symmetric numerical integration rules. SIAM J. Numer. Anal. **20**, 580–588 (1983)
16. Guarracino, M.R., Laccetti, G., Murli, A.: Application oriented brokering in medical imaging: algorithms and software architecture. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 972–981. Springer, Heidelberg (2005). https://doi.org/10.1007/11508380_99
17. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**, 463–492 (1990)
18. Herlihy, M.P., Shavit, N.: The Art of Multiprocessor Programming, Revised 1 edn. Morgan Kaufmann, Burlington (2012)
19. Laccetti, G., Lapegna, M.: PAMIHR. A parallel FORTRAN program for multidimensional quadrature on distributed memory architectures. In: Amestoy, P., Berger, P., Daydé, M., Ruiz, D., Duff, I., Frayssé, V., Giraud, L. (eds.) Euro-Par 1999. LNCS, vol. 1685, pp. 1144–1148. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48311-X_160
20. Laccetti, G., Lapegna, M., Mele, V., Romano, D., Murli, A.: A double adaptive algorithm for multidimensional integration on multicore based HPC systems. Int. J. Parallel Program. **40**, 397–409 (2012)
21. Laccetti, G., Lapegna, M., Mele, V.: A loosely coordinated model for heap-based priority queues in multicore environments. Int. J. Parallel Program. **44**, 901–921 (2016)
22. Lapegna, M.: A global adaptive quadrature for the approximate computation of multidimensional integrals on a distributed memory multiprocessor. Concurr.: Pract. Exp. **4**, 413–426 (1992)
23. Maddalena, L., Petrosino, A., Laccetti, G.: A fusion-based approach to digital movie restoration. Pattern Recogn. **42**, 1485–1495 (2009)
24. Moir, M., Shavit, N.: Concurrent data structures. In: Metha, D., Sahni, S. (eds.) Handbook of Data Structures and Applications, pp. 47-1–47-30. CRC Press, New york (2005)
25. Montella, R., Giunta, G., Riccio, A.: Using grid computing based component in on demand environmental data delivery. In: Proceedings of 2nd workshop on Use of P2P, Grid and Agent for the development of content Networks, pp. 81–86 (2005)
26. Montella, R., Coviello, G., Giunta, G., Laccetti, G., Isaila, F., Blas, J.G.: A general-purpose virtualization service for HPC on cloud computing: an application to GPUs. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7203, pp. 740–749. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31464-3_75
27. Montella, R., Giunta, G., Laccetti, G., Lapegna, M.: Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. Cluster Comput. **17**, 139–152 (2014)
28. Montella, R., Giunta, G., Laccetti, G., Lapegna, M., Palmieri, C., Ferraro, C., Pelliccia, V.: Virtualizing CUDA enabled GPGPUs on ARM clusters. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 3–14. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_1

29. Murli, A., Boccia, V., Carracciuolo, L., D'Amore, L., Laccetti, G., Lapegna, M.: Monitoring and migration of a PETSc-based parallel application for medical imaging in a grid computing PSE. In: Gaffney, P.W., Pool, J.C.T. (eds.) Grid-Based Problem Solving Environments. ITIFIP, vol. 239, pp. 421–432. Springer, Boston, MA (2007). https://doi.org/10.1007/978-0-387-73659-4_25
30. Murli, A., D'Amore, L., Laccetti, G., Gregoretti, F., Oliva, G.: A multi-grained distributed implementation of the parallel Block Conjugate Gradient algorithm. Concurr. Comput. Pract. Exp. **22**, 2053–2072 (2010)
31. Shavit, N.: Data structure in multicore age. Commun. ACM **54**, 76–84 (2011)

Energy Analysis of a 4D Variational Data Assimilation Algorithm and Evaluation on ARM-Based HPC Systems

Rossella Arcucci[1](✉), Davide Basciano[1], Alessandro Cilardo[1], Luisa D'Amore[1], and Filippo Mantovani[2]

[1] University of Naples Federico II, Naples, Italy
rossella.arcucci@unina.it
[2] Barcelona Supercomputing Center (BSC), Barcelona, Spain

Abstract. Driven by the emerging requirements of High Performance Computing (HPC) architectures, the main focus of this work is the interplay of computational and energetic aspects of a Four Dimensional Variational (4DVAR) Data Assimilation algorithm, based on Domain Decomposition (named DD-4DVAR). We report first results on the energy consumption of the DD-4DVAR algorithm on embedded processor and a mathematical analysis of the energy behavior of the algorithm by assuming the architectures characteristics as variable of the model. The main objective is to capture the essential operations of the algorithm exhibiting a direct relationship with the measured energy. The experimental evaluation is carried out on a set of mini-clusters made available by the Barcelona Supercomputing Center.

Keywords: Data assimilation · 4DVar · Domain Decomposition
Embedded processor architectures · Energy consumption

1 Introduction and Motivations

Data assimilation (DA) is an uncertainty quantification technique by which measurements and model predictions are combined to obtain an accurate representation of the state of the modeled system [8,13]. Due to the scale of the forecasting area and the number of state variables used to describe ocean or atmosphere for climate or weather predictions, DA applications are large scale problems that should be solved in near real-time. This mandates to design and develop DA algorithms to be run by exploiting High Performance Computing (HPC) environments till to the heterogeneous ones composed by multiprocessors multicores and graphics accelerators (see for example [10,16,17]).

During the last 20 years, parallel algorithms for DA have been investigated by a number of federal research institutes and universities. Up to now, the main efforts towards the development of parallel 4DVAR DA systems were achieved in numerical weather prediction applications, namely by the ECMWF (European

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 37–47, 2018.
https://doi.org/10.1007/978-3-319-78054-2_4

Center for Medium-Range Weather Forecasts), in Reading (UK) and by the NCAR (National Center for Atmospheric Research), in Colorado (USA). In this paper, we employ a 4DVAR algorithm described in [1,2,9], named DD-4DVAR, based on a Domain Decomposition approach. In [5,15,18,19] are described some different approaches to take full advantage of emerging HPC architectures. In the model we employ, the parallelism is achieved by dividing the global problem into multiple local 4DVAR DA sub-problems solved across processors. The global solution is obtained by collecting the local minimums. The sub-problems are handled by a slightly modified 4DVAR algorithm, custom implemented on an ARM-based low-energy node with the aim of minimizing the overall energy-to-solution experienced by the application.

The performance and energy cost of a parallel algorithm executing on HPC systems have different trade-offs, depending on how many processors the algorithm uses, at what characteristics these processors have, and the structure of the algorithm. Due to the interest of the HPC community towards low-power architectures such as the ones used in smartphone and tablets [20], we report in this paper the first results on the energy consumption of the DD-4DVAR algorithm on embedded processor. Note that our approach addresses the problem in the spirit of scalability analysis of parallel algorithms as distinct from practical performance analysis on specific architecture. We provide a mathematical analysis of the energy behavior of the DD-4DVAR algorithm as function of the architectures characteristics of the platforms where are executed. The main objective is to capture the essential operations in the algorithm exhibiting a direct relationship with the measured energy. Such analysis will enable predicting the energy requirements of the DD-4DVAR code, provided that a set of architecture-dependent parameters are available, as well as understanding its energy breakdown, which may in turn underpin a systematic approach to combined performance/energy optimization. The experimental evaluation is carried out on a set of AMR based platforms made available by the Barcelona Supercomputing Center in the context of the Mont-Blanc European project [21]. The evaluation, aimed at understanding the energy breakdown and the related scalability issues, pointing out the importance of the underplay between parallel performance and energy optimization.

2 The DD-4DVAR Computational Kernel

Hereafter we provide a concise formalization of the DD-4DVAR model we implemented in Algorithm 1 [2].

Let t_k, $k = 0, 1, \ldots, n$ be a sequence of observation times and, for each k, let be

$$x_k \equiv x(t_k) \in \Re^N \tag{1}$$

the vector denoting the state of a sea system such that $x_k = \mathcal{M}_k\left(x_{k-1}\right)$ with $\mathcal{M}_k : \Re^N \mapsto \Re^N$ forecasting model.

At each time step t_k, let be

$$y_k = \mathcal{H}_k(x_k) \in \Re^p \tag{2}$$

the observations vector where $\mathcal{H}_k : \Re^N \mapsto \Re^p$ is a non-linear interpolation operator collecting the observations at time t_k.

The aim of DA problem is to find an optimal tradeoff between the current estimate of the system state (background) defined in (1) and the available observations y_k defined in (2).

Let (3) be an overlapping decomposition of the physical domain Ω such that $\Omega_i \cap \Omega_j = \Omega_{ij} \neq 0$ if Ω_i and Ω_j are adjacent and Ω_{ij} is called *o*verlapping region [2].

$$\Omega = \bigcup_{i=1}^{N_{sub}} \Omega_i \tag{3}$$

For a fixed time $t_k = t_0$, according to this decomposition, the DD-4DVAR computational model is a system of N_{sub} non-linear least square problems described in (4)–(5) where J_i in (5) is called cost-function.

$$x_0^{DA} = \sum_{i=1}^{N_{sub}} \tilde{x}_{0_i}^{DA}, \quad \text{with} \quad \tilde{x}_{0_i}^{DA} = \begin{cases} argmin_{x_0} J_i(x_{0_i}^{DA}) & on \quad \Omega_i \\ 0 & on \quad \Omega - \Omega_i \end{cases} \tag{4}$$

$$J_i(x_{0_i}^{DA}) = \|\mathbf{x}_{0_i}^{DA} - \mathbf{x}_{0_i}^{M}\|_{\mathbf{B_i}}^2 + \sum_{k=0}^{N} \|\mathcal{G}_{\mathbf{k_i}}(\mathbf{x}_{0_i}^{DA}) - \mathbf{y}_i\|_{\mathbf{R_i}}^2 + \|\mathbf{x}_{0_i}^{DA}/\Omega_{ij} - \mathbf{x}_{0_j}^{DA}/\Omega_{ij}\|_{\mathbf{B_{ij}}} \tag{5}$$

where $\mathcal{G}_k = \mathcal{M}_k \circ \mathcal{H}_k$.

x_0^{DA} in (4) is the *analysis* (i.e. the estimation of the vector $x_{0_i}^{DA}$ at time t_0). The variables $\mathbf{x}_{0_i}$ and $\mathbf{y}_{k_i}$ are the same vectors $\mathbf{x}_0$ and $\mathbf{y}_k$ in (1) and (2) defined on the subdomain Ω_i, $\mathbf{R_i}$ and $\mathbf{B_i}$ are the covariance matrices whose elements provide the estimate of the errors on y_{k_i} and on x_{0_i}, respectively.

Let $d = [y_k - \mathcal{H}(x_k)]$ be the *misfit*, by using the linearization of $\mathcal{H}$ such that $\mathcal{H}(x) = \mathcal{H}(x + \delta x) + H\ \delta x$, where H is the matrix obtained by the first order approximation of the Jacobian of $\mathcal{H}$ and, by setting $v_i = V_i^T \delta x_i$, with V_i such that $\mathbf{B_i} = V_i V_i^T$, the cost function in (5) is written as:

$$\begin{aligned} J_i(v_i) = \frac{1}{2} v_i^T v_i + \frac{1}{2} \sum_{k=0}^{N} (G_{k_i} V_i v_i - d_{k_i})^T R_{k_i}^{-1} (G_{k_i} V_i v_i - d_{k_i}) \\ + \frac{1}{2} (V_{ij} v_i^+ - V_{ij} v_i^-)^T (V_{ij} v_i^+ - V_{ij} v_i^-) \end{aligned} \tag{6}$$

The minimum of the cost function J_i in (6) is computed by the L-BFGS method [22] which implements a quasi Newton method. Then we need to compute $\nabla J_i(v_i)$ such that:

$$\nabla J_i(v_i) = v_i + \sum_{k=0}^{N} V_{k_i}^T G_{k_i}^T R_{k_i}^{-1} (G_{k_i} V_i v_i - d_{k_i}) \tag{7}$$

where $G_{k_i}^T$ is the adjoint operator of G_{k_i}.

Algorithm 1. The DD-4DVAR algorithm on each subdomain $\Omega_i \times [t_0, t_n]$

```
1: Input: $\{y_{k_i}\}_{k=0,\dots,m}$ and $x_{0_i}^M$
2: Define $H_{k_i}$
3: Compute $d_{k_i} \leftarrow y_{k_i} - H_{k_i} M_{k_i} \dots M_{1_i} x_{0_i}^M$   % compute the misfit
4: Define $R_{k_i}$ starting from the observed data $y_{k_i}$
5: Define $V_i$ starting from a temporal sequence of hystorical data $\{x_{k_i}^M\}_{k=0,\dots,M}$
6: Define the initial value of $\delta x_i^{DA}$
7: Compute $v_i \leftarrow V_i^T \delta x_i^{DA}$
8: repeat % start of the L-BFGS steps
9: Send and Receive the boundary conditions from the adjacent domains
10: Compute $J_i \leftarrow J_i(v_i)$   % Defined in (6)
11: Compute $gradJ_i \leftarrow \nabla J_i(v_i)$   % Defined in (7)
12: Compute new values for $v_i$
13: until (Convergence on $v_i$ is obtained) % end of the L-BFGS steps
14: Compute $x_i^{DA} \leftarrow x_{0_i}^M + V_i v_i$
```

3 Energy Analysis of the Algorithm

In this section we set a DD-4DVAR algorithm configuration and we perform a mathematical analysis of the energy behaviour of the algorithm.

For the DD-4DVAR algorithm configuration we assume:

– N defined in (1), which is the dimension of the problem, such that

$$N = n_x \times n_y \times n_z = n \times n \times 3$$

as this does not affect the generality, where $n \in \mathcal{N}$, $n > 1$;

– a 2D decomposition along the x-axes and the y-axes such that each subdomain has dimension:

$$N_i = \frac{n}{p} \times \frac{n}{p} \times 3; \tag{8}$$

where $p \in \mathcal{N}$, $p > 1$. Then, N_{sub} the number of subdomain in (3) (which constitutes the domain decomposition) is

$$N_{sub} = p^2. \tag{9}$$

– the algorithm be implemented on a parallel architecture by employing *nproc* processors such that $nproc = N_{sub}$, i.e. from (9), we are assuming

$$nproc = p^2.$$

Concerning the energy model, we assume that [14]:

– the energy consumption is *additive* and it is essentially proportional to the respective activity intensity in each component of the computing architecture, in terms of compute operation count, exchanged messages, memory accesses, *plus* a static energy contribution which is not affected by the activity and only depends on the considered time interval.

Based on the above assumption, we can write the energy breakdown as:

$$E^{HC}(p,n) = E_{comp}(p,n) + E_{mem}(p,n) + E_{mes}(p,n) + E_{static}(p,n) \quad (10)$$

where the superscript HC denotes the dependency on the computing architecture, and

- $E_{comp}(p,n)$ is the energy for computation:
$$E_{comp}(p,n) = E_d \cdot f^2 \cdot \mu_{comp}(p,n), \quad (11)$$
where E_d is a hardware constant [7], $\mu_{comp}(p,n)$ is the number of computations and f is the frequency;
- $E_{mem}(p,n)$ is the energy for memory accesses:
$$E_{mem}(p,n) = E_m \cdot \mu_{mem}(p,n), \quad (12)$$
where E_m is the energy consumed for a single memory access (both read and write) and and $\mu_{mem}(p,n)$ is the number of memory accesses;
- $E_{mes}(p,n)$ is the energy for message transfers:
$$E_{mes}(p,n) = E_t \cdot \mu_{mes}(p,n), \quad (13)$$
where E_t is the energy consumed for a single message transfer between the processors and $\mu_{mes}(p,n)$ is the number of message transfers at all processors;
- $E_{static}(p,n)$ is the static energy:
$$E_{static}(p,n) = E_l \cdot f \cdot T_{active}(p,n). \quad (14)$$
where E_l is a hardware constant [7] and $T_{active}(p,n)$ is the execution time for performing the whole algorithm.

Let

- $N_{L-BFGS,p}$ be the number of L-BFGS steps (see Steps 8–13 of Algorithm 1) which depends on the sub domains dimension (i.e., from (8), it depends on p) [3];
- n_C^{HC} be the maximum size of the allocable problem in the memory cache of the architecture HC.
- p_{max}^{HC} be the maximum number of cores of the architecture HC.

By assuming
$$n \leq n_C^{HC}, \quad p < p_{max}^{HC} \quad (15)$$
and by analyzing the time complexity of Algorithm 1, we can estimate the order of magnitude of the energy consumption by the following result.

Theorem 1. *By assuming (10), (11)–(14) and (15), it holds:*

$$E^{HC}(p,n) = \mathcal{O}\left(\mathcal{C}^{HC}(p) \cdot 9\frac{n^4}{p^2}\right) \quad (16)$$

where $E^{HC}(p,n)$ denotes the energy consumption defined in (10) and where $\mathcal{C}^{HC}(p)$:

$$\mathcal{C}^{HC}(p) = E_d \cdot N_{L-BFGS,p} + E_l \cdot t_{flop} \tag{17}$$

with t_{flop} denotes the unitary time required for the execution in each processor of one floating point operation.

Proof: *Let $S_i(p,n)$ and $V_i(p,n)$ denote the number of floating point exchanges at each algorithm iteration and the floating point computations at each iteration respectively, proportional to surface area and the volume of each subdomain in Algorithm 1:*

$$S_i(p,n) = 12\frac{n}{p} \tag{18}$$

$$V_i(p,n) = 3\frac{n^2}{p^2} \tag{19}$$

then $\mu_{comp}(p,n)$, $\mu_{mem}(n,p)$ and $\mu_{mes}(p,n)$ are such that:

$$\mu_{comp}(p,n) = N_{L-BFGS,p} \cdot p^2 \cdot V_i^2(p,n), \tag{20}$$

$$\mu_{mem}(p,n) = 2N_{L-BFGS,p} \cdot p^2 \cdot V_i(p,n), \tag{21}$$

$$\mu_{mes}(p,n) = N_{L-BFGS,p} \cdot p^2 \cdot S_i(p,n), \tag{22}$$

Also we assume $T_{active}(p,n)$ be the execution time for performing $V_i^2(p,n)$ floating point operations:

$$T_{active}(p,n) = t_{flop} \cdot V_i^2(p,n) \tag{23}$$

Then, from (10), (18)–(19) and (20)–(22), it holds

$$E^{HC}(p,n) = E_d \cdot N_{L-BFGS,p}\ (p^2)\left(3\frac{n^2}{p^2}\right)^2 \cdot f^2 + E_m \cdot 2N_{L-BFGS}\ (p^2)\left(3\frac{n^2}{p^2}\right) + E_t \cdot N_{L-BFGS,p}\ (p^2)\left(6\frac{n}{p} + 6\frac{n}{p}\right) + E_l \cdot t_{flop}(p^2)\left(3\frac{n^2}{p^2}\right)^2 \cdot f \tag{24}$$

As we run in a single computational node (i.e. $p < p_{max}$ as expressed in (15)) this means that we are not implying communications, so the third term can be neglected. From qualitative observations, we can assume that the second term can be neglected because we fit the whole data in cache (as expressed in (15)), therefore a negligible number of access to the main memory are performed. Then the (16) follows.

Definition 1 *(Energy Variation parameter)***.** *We denote with Energy Variation parameter the ratio*

$$VE_{p1,p2} = \frac{E^{HC}(p_1,n)}{E^{HC}(p_2,n)} \tag{25}$$

The following result holds:

Proposition 1. *For a fixed architecture and, under the hypothesis of Theorem 1, it is*

$$VE_{p1,p2} > \frac{p_2^2}{p_1^2} \tag{26}$$

for $p_2 \geq p_1$.

Proof: *From (24) and (16) for a fixed value of n, it is*

$$VE_{p1,p2} = \frac{\mathcal{C}^{HC}(p_1)}{\mathcal{C}^{HC}(p_2)} \frac{p_2^2}{p_1^2} \tag{27}$$

We observe that, from (27), it is

$$\frac{\mathcal{C}^{HC}(p_1)}{\mathcal{C}^{HC}(p_2)} > 1 \Longrightarrow VE_{p1,p2} \geq \frac{p_2^2}{p_1^2}$$

which gives:

$$\mathcal{C}^{HC}(p_1) > \mathcal{C}^{HC}(p_2) \Longrightarrow VE_{p1,p2} > \frac{p_2^2}{p_1^2} \tag{28}$$

From (28) and (17) it is

$$\mathcal{C}^{HC}(p_1) > \mathcal{C}^{HC}(p_2) \Longleftrightarrow E_d \cdot N_{L-BFGS,p_1} + E_l \cdot t_{flop} > E_d \cdot N_{L-BFGS,p_2} + E_l \cdot t_{flop}$$

As for a fixed architecture, the values of E_d, E_l and t_{flop} are also fixed, it is

$$\mathcal{C}^{HC}(p_1) > \mathcal{C}^{HC}(p_2) \Longleftrightarrow N_{L-BFGS,p_1} > N_{L-BFGS,p_2}$$

Due the better conditioning of the smaller problems, it is $N_{L-BFGS,p_1} > N_{L-BFGS,p_2}$ [3]. Then the (26) holds.

Remark 1. *We observe that, if the (15) is not satisfied, then $\mathcal{C}^{HC}(p)$ includes also E_{mes} which increases as the number of processors increases. In that case, for $p_2 > p_1$, it is:*

$$\mathcal{C}^{HC}(p_2) \geq \mathcal{C}^{HC}(p_1) \tag{29}$$

which gives

$$VE_{p1,p2} \leq \frac{p_2^2}{p_1^2} \tag{30}$$

4 Experimental Results

The proposed approach is validated on a case study based on the linear Shallow Water Equation (SWE) for $n = 64$, i.e. we consider a fixed size configuration of the DD-4DVAR algorithm and we discuss results obtained by varying p.

Table 1. Reference architectures details

Specifications	Cavium ThunderX	Nvidia JetsonTx1	Samsung Exynos
Instruction set	ARMv8	ARMv8	ARMv7
Num. of cores/node	$2 \cdot 48$	4	2
Num. of cluster nodes	1	16	16
Clock freq. [GHz]	2.5	1.73	1.7
L2 cache size [MB]	16	2	1

The experiments are been conducted on architectures available at the Barcelona Supercomputing Center (BSC) and the power measurements have been enabled by the Mont-Blanc computing environment [21].

In Table 1 are summarized the reference architectures. $HC = CT$ refers to a single Cavium ThunderX server [23], $HC = JT$ refers to a cluster of 16 nodes of Nvidia JetsonTx1, while $HC = MB$ refers to a partition of 5 nodes of the Mont-Blanc prototype cluster [21] used for this work.

Relying on the potential of the Mont-Blanc computing environment, we were particularly interested in the results in terms of power efficiency and energy-to-solution. Here we provide results in terms of (measured) Energy Variation Parameter defined in (25) and computed using the values of energy consumptions given by $E^{HC}(p, 64) = P_p^{HC} \cdot T_p^{HC}$, where P_p^{HC} and T_p^{HC} are the power and the execution time respectively. We compare the obtained results with the upper and lower bounds provided in (26) and (30).

We observe that, in Table 1, the Cavium ThunderX has 16 Megabyte of memory cache which allows to satisfy condition in (15). In fact[1],

$$n_C^{CT} = 16 \cdot n_{C,1} = 96 > 64 = n, \quad p < p_{max}^{CT} = 2 \cdot 48 = 96.$$

Under condition (15), the (26) holds as confirmed by the results in Table 2.

Table 2. Cavium ThunderX

p^2	P_p^{CT}	T_p^{CT}	$E^{CT}(p, 64)$	$VE_{1,p}$	$p^2/1$
1	125.0 W	906 s	113250.0 J	1.0	1
4	125.5 W	211 s	26480.5 J	4.3	4
16	126.5 W	42 s	5313.0 J	21.3	16

The JetsonTx1 and Mont-Blanc, with 2 Megabyte and 1 Megabyte of cache instead (see Table 1) do not satisfy (15). In fact, $n_C^{JT} = 2 \cdot n_{C,1} = 12$ and

[1] Due the time complexity of the computation, for each Megabyte, the values on n_C which is independent from the computing architecture, is such that: $n_{C,1} = \left\lceil \left(\frac{1048576}{8*3} \right)^{\frac{1}{6}} \right\rceil = 6$, where $\lceil \cdot \rceil$ denotes the integer part.

Table 3. JetsonTx1

$f = 800000$					
p^2	P_p^{JT}	T_p^{JT}	$E^{JT}(p, 64)$	$VE_{1,p}$	$p^2/1$
1	5.3 W	429 s	2273.7 J	1.0	1
4	6.6 W	115 s	759.0 J	3.0	4
16	6.6 W	45 s	297.0 J	7.7	16
$f = 1700000$					
p^2	P_p^{JT}	T_p^{JT}	$E^{JT}(p, 64)$	$VE_{1,p}$	$p^2/1$
1	6.5 W	210 s	1365.4 J	1.0	1
4	10.0 W	86 s	860.6 J	3.1	4
16	10.0 W	21 s	210.0 J	6.5	16

Table 4. Mont-Blanc

$f = 800000$					
p^2	P_p^{MB}	T_p^{MB}	$E^{MB}(p, 64)$	$VE_{1,p}$	$p^2/1$
1	5.4 W	375 s	2025.0 J	1.0	1
4	5.5 W	86 s	473.0 J	4.2	4
16	5.5 W	23 s	126.5 J	16.0	16
$f = 1700000$					
p^2	P_p^{MB}	T_p^{MB}	$E^{MB}(p, 64)$	$VE_{1,p}$	$p^2/1$
1	5.4 W	181 s	977.4 J	1.0	1
4	5.5 W	48 s	264 J	3.7	4
16	5.5 W	13 s	71.5 J	13.7	16

$n_C^{MB} = 1 \cdot n_{C,1} = 6$ for the JT and MB respectively, both smaller than $n = 64$. In these cases, the upper bound in (30) holds as confirmed by the results in Tables 3 and 4.

5 Conclusions

We introduced an energy analysis of the DD-4DVAR algorithm for data assimilation problems. An implementation of the algorithm was evaluated on some prototype ARM-based platforms made available by the Barcelona Supercomputing Center. We performed the analysis of the energy behaviour of the algorithm depending on several architectures characteristics. A preliminary experimental evaluation confirmed the estimations provided by our analysis on a fixed size problem varying the number of processors. As a future development, we aim at scaling up the methodology by demonstrating energy-driven parallelization approaches on production-grade ARM-based HPC clusters.

Future developments could be straightforwardly take into account the expertise of scientists of our workgroup, to face fault-tolerance problems [4,6,18] as well as implementations in cloud and/or distributed environments [11,12], and in heterogeneous ones [15,19].

Acknowledgment. The research has received funding from European Commission under H2020-MSCA-RISE NASDAC project (grant agreement no. 691184) FP7 Mont-Blanc and Mont-Blanc 2 (grant agreements no. 288777 and 610402), H2020-FET Mont-Blanc 3 (grant agreement 671697).

References

1. Arcucci, R., D'Amore, L., Carracciuolo, L., Scotti, G., Laccetti, G.: A decomposition of the tikhonov regularization functional oriented to exploit hybrid multilevel parallelism. Int. J. Parallel Prog. **45**(5), 1214–1235 (2017)
2. Arcucci, R., D'Amore, L., Carracciuolo, L.: On the problem-decomposition of scalable 4D-Var data assimilation models. In: Proceedings of HPCS 2015, pp. 589–594 (2015)
3. Arcucci, R., D'Amore, L., Pistoia, J., Toumi, R., Murli, A.: On the variational data assimilation problem solving and sensitivity analysis. JCPH **335**, 311–326 (2017)
4. Boccia, V., Carracciuolo, L., Laccetti, G., Lapegna, M., Mele, V.: HADAB: enabling fault tolerance in parallel applications running in distributed environments. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7203, pp. 700–709. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31464-3_71
5. Carracciuolo, L., D'Amore, L., Murli, A.: Towards a parallel component for imaging in PETSc programming environment: a case study in 3-D echocardiography. Parallel Comput. **32**, 67–83 (2006)
6. Caruso, P., Laccetti, G., Lapegna, M.: A performance contract system in a grid enabling, component based programming environment. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 982–992. Springer, Heidelberg (2005). https://doi.org/10.1007/11508380_100
7. Chandrakasan, A.P., Sheng, S., Brodersen, R.W.: Low-power CMOS digital design. J. Solid-State Circ. **27**(4) (1992)
8. D'Amore, L., Arcucci, R., Marcellino, L., Murli, A.: HPC computation issues of the incremental 3D variational data assimilation scheme in OceanVar software. JNAIAM **7**(3–4), 91–105 (2012)
9. D'Amore, L., Arcucci, R., Carracciuolo, L., Murli, A.: A scalable approach to variational data assimilation. J. Sci. Comput. **2**, 239–257 (2014)
10. Di Lauro, R., Giannone, F., Ambrosio, L., Montella, R.: Virtualizing general purpose GPUs for high performance cloud computing: an application to a fluid simulator. In: Proceedings of 10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA (2012)
11. Gregoretti, F., Laccetti, G., Murli, A., Oliva, G., Scafuri, U.: MGF: a grid-enabled MPI library. Future Gener. Comput. Syst. (FGCS) **24**(2), 158–165 (2008)
12. Guarracino, M.R., Laccetti, G., Murli, A.: Application oriented brokering in medical imaging: algorithms and software architecture. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 972–981. Springer, Heidelberg (2005). https://doi.org/10.1007/11508380_99

13. Kalnay, E.: Atmospheric Modeling, Data Assimilation and Predictability. Cambridge University Press, Cambridge (2003)
14. Korthikanti, V.A., Agha, G.: Energy-performance trade-off analysis of parallel algorithms. In: Hot Topics in Parallelism (HotPar) (2010)
15. Laccetti, G., Lapegna, M., Mele, V., Romano, D., Murli, A.: A double adaptive algorithm for multidimensional integration on multicore based HPC systems. Int. J. Parallel Program. (IJPP) **40**(4), 397–409 (2012)
16. Montella, R., Giunta, G., Laccetti, G.: Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. Cluster Comput. **17**(1), 139–152 (2014)
17. Montella, R., Giunta, G., Laccetti, G., Lapegna, M., Palmieri, C., Ferraro, C., Pelliccia, V.: Virtualizing CUDA enabled GPGPUs on ARM clusters. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 3–14. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_1
18. Murli, A., Boccia, V., Carracciuolo, L., D'Amore, L., Laccetti, G., Lapegna, M.: Monitoring and migration of a PETSc-based parallel application for medical imaging in a grid computing PSE. In: Gaffney, P.W., Pool, J.C.T. (eds.) Grid-Based Problem Solving Environments. ITIFIP, vol. 239, pp. 421–432. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-73659-4_25
19. Murli, A., D'Amore, L., Laccetti, G., Gregoretti, F., Oliva, G.: A multi-grained distributed implementation of the parallel Block Conjugate Gradient algorithm. Concurr. Comput.: Pract. Exp. **22**(15), 2053–2072 (2010)
20. Rajovic, N., Carpenter, P.M., Gelado, I., Puzovic, N., Ramirez, A., Valero, M.R.: Supercomputing with commodity CPUs: are mobile SoCs ready for HPC? In: International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12 (2013)
21. Rajovic, N., et al.: The Mont-Blanc prototype: an alternative approach for HPC systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Piscataway, NJ, USA, pp. 38:1–38:12 (2016)
22. Nocedal, J., Byrd, R.H., Lu, P., Zhu, C.: L-BFGS-B: fortran subroutines for large-scale bound-constrained optimization. ACM Trans. Math. Softw. **23**(4), 550–560 (1997)
23. http://www.anandtech.com/show/10353/investigating-cavium-thunderx-48-arm-cores/

Performance Assessment of the Incremental Strong Constraints 4DVAR Algorithm in ROMS

Luisa D'Amore[1(✉)], Rossella Arcucci[1], Yi Li[2], Raffaele Montella[3], Andrew Moore[4], Luke Phillipson[2], and Ralf Toumi[2]

[1] University of Naples Federico II, Naples, Italy
{luisa.damore,rossella.arcucci}@unina.it
[2] Imperial College in London, London, UK
{y.li14,l.phillipson14,r.toumi}@imperial.ac.uk
[3] University of Naples Parthenope, Naples, Italy
raffaele.montella@uniparthenope.it
[4] University of Santa Cruz, Santa Cruz, USA
ammoore@ucsc.edu

Abstract. We consider the Incremental Strong constraint 4D VARiational (IS4DVAR) algorithm for data assimilation implemented in ROMS with the aim to study its performance in terms of strong scaling scalability on computing architectures such as a cluster of CPUs. We consider realistic test cases with data collected in enclosed and semi enclosed seas, namely, Caspian sea, West Africa/Angola, as well as data collected into the California bay. The computing architecture we use is currently available at Imperial College London. The analysis allows us to highlight that the ROMS-IS4DVAR performance on emerging architectures depends on a deep relation among the problems size, the domain decomposition approach and the computing architecture characteristics.

Keywords: Data assimilation · 4DVAR algorithm
Performance analysis · Parallel algorithm

1 IS4DVAR Algorithm

The Incremental Strong Constraint 4DVAR (IS4DVAR) Algorithm is one of Data Assimilation modules of the Regional Ocean Modelling System (ROMS) [18–20]. It solves a regularized Non Linear Least Square (NL-LS) problem of the type (see [2–4,8,22] for details):

$$argmin_{\mathbf{u}\in\Re^N} J_{DA}(\mathbf{u}) = argmin_{\mathbf{u}\in\Re^N} \|\mathbf{F}_{DA}(\mathbf{u}, \mathcal{M}^{\Delta\times\Omega}, \mathbf{u}_0^b, \mathbf{R}, \mathbf{B}, \mathbf{v}, \Delta, \Omega)\|,$$

where $\mathcal{M}^{\Delta\times\Omega}$ the predictive model defined in the time-and-space physical domain $\Delta\times\Omega$ with initial condition $\mathbf{u}_0^b$, $\mathbf{R}$, and $\mathbf{B}$ the covariance matrices and $\mathbf{v}$ the vector of the observations.

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 48–57, 2018.
https://doi.org/10.1007/978-3-319-78054-2_5

The common approach for solving NL-LS problems consists in defining a sequence of local approximations of J_{DA} where each member of the sequence is minimized by employing Newton's method or one its variants (such as Gauss-Newton, L-BFGS, Levenberg-Marquardt). See Algorithms 1 and 2. Approximations are obtained by means of truncated Taylor's series, while the minimum is obtained by using second-order sufficient conditions [1,24] (see step 7 of Algorithm 1). In particular, two approaches could be employed:

(a) by truncating Taylor's series expansion of $\mathbf{J}_{DA}$ at the second order such as Newton'methods (including LBFGS and Levenberg-Marquardt) following the Newton's descend direction (see Algorithm 3);
(b) by truncating Taylor's series expansion of $\mathbf{J}_{DA}$ at the first order such as Gauss-Newton's methods (including Truncated Gauss-Newton or Approximated Gauss-Newton) following the steepest descend direction, which is computed solving the normal equations arising from the local Linear Least Squares (LLS) problem (see Algorithm 4).

In ROMS-IS4DVAR the NL-LS problem is solved by using Gauss-Newton's method, where solution of normal equations system is obtained by applying a

Algorithm 1
```
1: procedure IS4DVar($in : \mathcal{M}^{\Delta\times\Omega}, \mathbf{u}_0^b, \mathbf{R}, \mathbf{B}, \mathbf{v}, \Delta, \Omega; out : \mathbf{u^{DA}}$)
2:     %Run $\mathcal{M}^{\Delta\times\Omega}$ with initial condition $\mathbf{u}_0^b$ for computing $\mathbf{u}^b$, in $\Delta\times\Omega$
3:     $\mathbf{u}^b = \mathcal{M}^{\Delta\times\Omega}[\mathbf{u}_0^b]$
4:     $k := 0, \mathbf{u}_{DA}^0 = \mathbf{u}_{DA}^b$
5:     repeat
6:         $k := k+1$
7:         Call NLLS($in : \mathcal{M}^{\Delta\times\Omega}, \mathbf{R}, \mathbf{B}, \mathbf{v}, \mathbf{u}^b, \Delta, \Omega; out : \mathbf{u}_{DA}^k$)
8:     until $\|\mathbf{u}_{DA}^k - \mathbf{u}_{DA}^{k-1}\| < eps$
9: end procedure
```

Algorithm 2
```
1: procedure NLLS($in : \mathcal{M}^{\Delta\times\Omega}, \mathbf{R}, \mathbf{B}, \mathbf{v}, \mathbf{u}^b, \Delta, \Omega; out : \mathbf{u}_{DA}^k$)
2:     Initialize $\mathbf{u}^0 := \mathbf{u}^b$;
3:     Initialize $k = 0$;
4:     repeat
5:         % Compute $\delta\mathbf{u}_{DA}^k = argmin\,\mathbf{J}_{DA}$ by using $QN$ or $LLS$
6:         If (QN) then
7:             Call QN ($in : \mathcal{M}^{\Delta\times\Omega}, \mathbf{R}, \mathbf{B}, \mathbf{v}, \mathbf{u}^b, \Delta, \Omega; out : \mathbf{u}_{DA}^k$ )
8:         ElseIf (LLS) then
9:             Call LLS ($in : \mathcal{M}^{\Delta\times\Omega}, \mathbf{R}, \mathbf{B}, \mathbf{v}, \mathbf{u}^b, \Delta, \Omega; out : \mathbf{u}_{DA}^k$)
10:        EndIf
11:        Update $\mathbf{u}_{DA}^k = \mathbf{u}_{DA}^k + \delta\mathbf{u}_{DA}^k$
12:        Update $k = k+1$
13:    until (convergence is reached)
14: end procedure
```

Algorithm 3

```
1: procedure QN($\mathcal{M}^{\Delta\times\Omega}, \mathbf{R}, \mathbf{B}, \mathbf{v}, \mathbf{u}^b, \Delta, \Omega; out : \mathbf{u}^k_{DA}$)
2:     Initialize $\mathbf{u}^0_{DA} := \mathbf{u}^b_A$;
3:     Initialize $k = 0$;
4:     repeat
5:         %Compute $\delta\mathbf{u}^k_{DA} = argmin\, \mathbf{J}^{QD}_{DA}$, by Newton's method
6:         1.1 Compute $\nabla\mathbf{J}_{DA}(\mathbf{u}^k_{DA}) = \nabla\mathbf{F}^T_{DA}(\mathbf{u}^k_{DA})\nabla\mathbf{F}_{DA}(\mathbf{u}^k_{DA})$
7:         1.2 Compute $\nabla^2\mathbf{J}_{DA}(\mathbf{u}^k) = \nabla\mathbf{F}^T_{DA}(\mathbf{u}^k)\nabla\mathbf{F}_{DA}(\mathbf{u}^k_{DA}) + \mathbf{Q}((\mathbf{u}^k_{DA}))$
8:         1.3 Solve $\nabla^2\mathbf{J}_{DA}(\mathbf{u}^k_{DA})\delta\mathbf{u}^k_{DA} = -\nabla\mathbf{J}_{DA}(\mathbf{u}^k_{DA})$
9:         Update $\mathbf{u}^k_{DA} = \mathbf{u}^k_{DA} + \delta\mathbf{u}^k_{DA}$
10:        Update $k = k + 1$
11:    until (convergence is reached)
12: end procedure
```

Algorithm 4

```
1: procedure LLS($\mathcal{M}^{\Delta\times\Omega}, \mathbf{R}, \mathbf{B}, \mathbf{v}, \mathbf{u}^b, \Delta, \Omega; out : \mathbf{u}^k_{ji}$)
2:     Initialize $\mathbf{u}^0 := \mathbf{u}^b$;
3:     Initialize $k = 0$;
4:     repeat
5:         Compute $\nabla\mathbf{J}_{DA} = \nabla\mathbf{F}^T_{DA}(\mathbf{u}^k_{DA})\nabla\mathbf{F}_{DA}(\mathbf{u}^k_{DA})$
6:         %Compute $\delta\mathbf{u}^k_{DA} = argmin\, \mathbf{J}^{TL}_{DA}$ by solving the normal equations system:
7:         Solve $\nabla\mathbf{F}^T_{DA}(\mathbf{u}^k_{DA})\nabla\mathbf{F}_{DA}(\mathbf{u}_{DA})\delta\mathbf{u}^k_{DA} = -\nabla\mathbf{F}^T_{DA}(\mathbf{u}^k_{DA})\mathbf{F}_{DA}(\mathbf{u}^k_{DA})$
8:         Update $\mathbf{u}^k_{DA} = \mathbf{u}^k_{DA} + \delta\mathbf{u}^k_{DA}$
9:         Update $k = k + 1$
10:    until (convergence is reached)
11: end procedure
```

Krylov subspace iterative method (this task is also referred to as the inner-loop while the steps along the descent direction are called the outer-loop) (see Algorithm 6). IS4DVAR is described in Algorithms 5 and 6 [13]. Finally, in Fig. 1 we report the flowchart of IS4DVAR algorithm as it is implemented in ROMS.

Figure 1 illustrates the IS4DVAR Algorithm as it is implemented in ROMS and in Fig. 2 we describe the software architecture of ROMS. For details see description in [18].

2 Performance Assessment of Parallel IS4DVAR Algorithm

As IS4DVAR is part of the ROMS, the parallelization strategy implemented for the IS4DVAR algorithm takes advantage of the parallelization strategy implemented in ROMS. In other words, each part of the IS4DVAR which depends on the forecasting model (in particular, NLROMS, TLROMS and ADROMS modules) implement the two dimensional DD approach (2D-DD) approach (i.e. a coarse-grain parallelism), while Preconditioner and Lanczos Algorithm modules implement the one dimensional DD (1D-DD) approach (i.e. a fine-grain parallelism). I/O is all happening on the master process unless you specifically ask

Algorithm 5 (IS4DVAR refined)

```
1:  procedure IS4DVAR($\mathcal{M}^{\Delta\times\Omega}, \mathbf{R}, \mathbf{B}, \mathbf{v}, \mathbf{u}^b, \Delta, \Omega; out : \mathbf{u}^k_{DA}$)
2:      Initialize $\mathbf{u}^0 := \mathbf{u}^b = \mathcal{M}^{\Delta\times\Omega}[\mathbf{u}^b_0]$;
3:      Initialize $k = 0$;
4:      repeat
5:          Compute $\mathbf{d}^k = \mathbf{v} - \mathcal{H}(\mathbf{u}^k_{DA})$
6:          Compute $\mathbf{G}$, $\mathbf{V}$
7:          %Solve the normal equations system by using Krylov iterative methods
8:          Call Lanczos-4DVAR ($\mathbf{G}, \mathbf{V}, \mathbf{R}, \mathbf{B}, \mathbf{v}, \mathbf{u}^b, \Delta, \Omega; out : \delta\mathbf{u}^k_{DA}$)
9:          Update $\mathbf{u}^k_{DA} = \mathbf{u}^k_{DA} + \delta\mathbf{u}^k_{DA}$
10:         Update $k = k + 1$
11:     until (convergence is reached)
12: end procedure
```

Algorithm 6

```
1: procedure Lanczos-4DVar($\mathbf{G}, \mathbf{V}, \mathbf{R}, \mathbf{B}, \mathbf{v}, \mathbf{u}^b, \Delta, \Omega; out : \delta\mathbf{u}^k_{DA}$)
2:         Compute $\mathbf{G}, \mathbf{G}^T, \mathbf{R}^{-1}, \mathbf{V}$;
3:         Solve $(\mathbf{I} + \mathbf{G}^T\mathbf{R}^{-1}\mathbf{G}\delta\mathbf{u}^k_{DA} = \mathbf{G}^T\mathbf{R}^{-1}\mathbf{d}$
4:         % by using Lanczos algorithm
5: end procedure
```

it to use MPI-I/O. Concerning the 1D-DD approach, the parallelism in ROMS is introduced (into the step (vi) of Fig. 1) by distributing the data among a 1D processor grid blocked by rows (see the Parallel version of the ARPACK library [15] for details). We observe that this is the most suitable way to reduce communication overheads in the execution of linear algebra operations required by concurrently performing Lanczos algorithms.

Let us briefly model the coarse-and-fine parallelization strategy implemented in IS4DVAR Algorithm.

Definition 1 (1D and 2D Domain Decomposition Strategy). *Let the domain Ω be decomposed in $Ntile$ subdomains (also named* tiles*) with overlap areas, where*

$$Ntile = NtileI \times NtileJ.$$

If

$$size(\Omega) = N = N_1 \times N_2 \times N_3,$$

then in 2D-DD

$$size_{2D-DD}(tile) = \frac{N_1}{NtileI} \times \frac{N_2}{NtileJ} \times N_3;$$

while in 1D-DD, it is

$$size_{1D-DD}(tile) = \frac{N_1}{NtileI} \times N_2 \times N_3.$$

♠

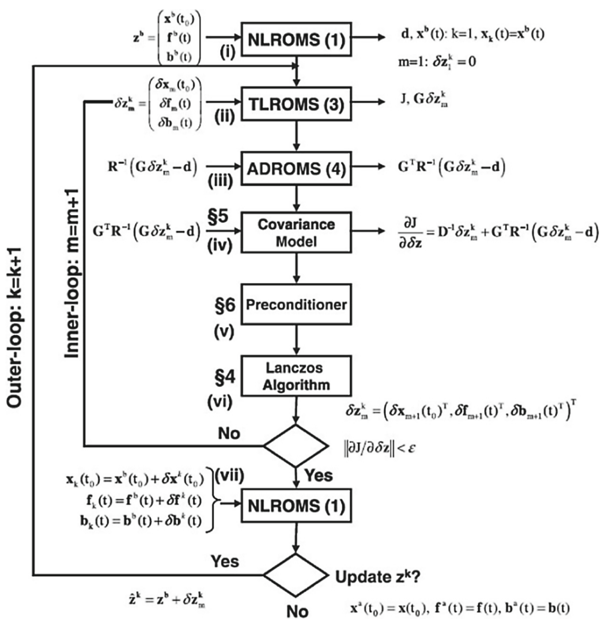


Fig. 1. A flow chart illustrating ROMS-IS4DVar algorithm where NLROMS, TLROMS and ADROMS implement the ROMS non linear model, the tangent linear (First Order Taylor Approximation of ROMS) and the Adjoint (for computing the Adjoint operator of ROMS) [18]. Parameters k and m (where $k \ll m$) are the steps for the linearization (First Order Taylor approximation) and for the minimization algorithms (by using Lanczos algorithm) respectively.

The surface $S(N, Ntile)$ of each 2D-DD tile is

$$S(N, Ntile) = 2\left(\frac{N_1}{NtileI} \times \frac{N_2}{NtileJ}\right) + 2\left(2\frac{N_1}{NtileI} + 2\frac{N_2}{NtileJ}\right) \times N_3 \qquad (1)$$

and the volume is

$$V(N, Ntile) = \frac{N_1}{NtileI} \times \frac{N_2}{NtileJ} \times N_3. \qquad (2)$$

If the 2D-DD is uniform, i.e. if $N_1 = N_2 = N_3 = M$, and $NtileI = NtileJ = p$ then, from (1) and (2) it is

$$S(M, p) = O\left(2\frac{M^2}{p^2} + 2\frac{M^2}{p}\right), \quad V(M, p) = O\left(\frac{M^3}{p^2}\right). \qquad (3)$$

As communication is much slower than computation, we will continue to get slower relative to computation over time, so we address performance of IS4DVAR Algorithm computing an estimate of the communication overhead, let us say Oh_{com}. In particular, we investigate the behavior of the communication overhead, let us denote Oh_{com}, in terms of the surface-to-volume ratio, for the 2D-DD approach.

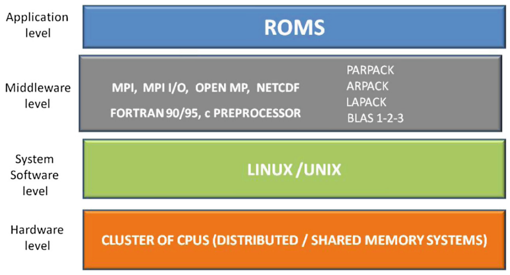


Fig. 2. ROMS software architecture. The version 3.6 of ROMS is been installed. This is the last version available. The ROMS source code is only distributed using Subversion (SVN). Its parallel framework includes both shared-memory (OpenMP) and distributed-memory (MPI) paradigms. The Middleware level includes a copy of ARPACK in the Lib directory which is used for the adjoint-based algorithms. The Lib directory also contains a copy of the Model Coupling Toolkit (MCT) which you will need if you wish to couple ROMS to other models. ROMS-IS4DVAR is written in F90/F95 with dynamic allocation of memory which allows multiple levels of nesting and/or composed grids. Finally, ROMS-IS4DVAR uses extensive C-preprocessing (CPP) to configure its various numerical and physical options. ROMS supports serial, OpenMP, and MPI computations, with the user choosing between them at compile time. Here we focus on the compiling for MPI. Also, details about parallelization strategy and the variables involved are available on www.myroms.org/wiki/Parallelization.

Definition 2 (Surface-to-volume). *The surface-to-volume ratio is a measure of the amount of data exchange (proportional to surface area of domain) per unit operation (proportional to volume of domain).*

Definition 3 (Communication Overhead). *Let T_{com} denote the total communication time and T_{flop} the total computation time, then*

$$Oh_{com} := \frac{T_{com}}{T_{flop}}.$$

Proposition 1. *Let t_{com} be the sustained communication time for sending/receiving one data in IS4DVAR and t_{flop} the sustained execution time of one floating point operation in IS4DVAR, such that*[1]*.*

$$t_{com} = \alpha t_{flop}, \quad \alpha = 10^q, \quad q > 1.$$

For the IS4DVAR Algorithm it holds that

$$Oh_{com} < 1 \Leftrightarrow 0 < k < r - q \quad , q \in]0, r[. \tag{4}$$

[1] Relation between t_{com} and t_{calc} (namely, the value of the parameter q) heavily depends on how the software under consideration is able to efficiently exploit the parallelism of such advanced architectures (the so called sustained performance).

Proof: For each m, from (3) it follows

$$Oh_{com} = \frac{S}{V}\frac{T_{com}}{T_{flop}} = \frac{2+2p\cdot t_{com}}{M\cdot t_{flop}}.$$

We write $N = 10^r$ and $p = 10^k$, then we have

$$Oh_{com} = O\left(\frac{10^q(2+2p)}{10^r}\right) = O\left(10^{q-r}(2+2\cdot 10^k)\right) = O(10^{q-r+k})$$

i.e. the (4).

Expression in (4) states that in order to increase the upper bound on $k = \log(p)$, the problems size should increases, and/or the ratio of the sustained unitary communication time over the sustained computation time (i.e. parameter $a = 10^q$) should decreases. Since the experiments which we consider here use realistic configurations of medium-size, performance results will confirm that the efficiency degrades below 50% for $p > 16$.

3 Experiments

We describe the configurations we have chosen for testing and analysing the performance of IS4DVAR on the California Current System, the Caspian sea and the Angola Basin. All the experiments are carried out on the CX2 (Helen) computing system provided by Imperial College London[2]. For each experiment, we report strong scaling results, in terms of execution time, speed up and efficiency. The variable *proc* on the tables refer to the number of processors involved, T_p refers to the execution time, $Ntile = p$, $S_p = \frac{T_1}{T_p}$, $E_p = \frac{S_p}{p}$. Finally, we use the mapping $proc \leftrightarrow MPI\ process$. The test cases we have chosen refers to:

- TC1: the California Current System (CCS) with 30 km (horizontal) resolution and 30 levels in the vertical direction. The global grid is then:

$$N = 54 \times 53 \times 30 = 8.586 \times 10^4.$$

- TC2: the Caspian Sea with 8 km resolution and 32 vertical layers. The vertical resolution is set with a minimum depth of 5 m. Then, problem dimension in terms of the grid/mesh size consists of

$$N = 90 \times 154 \times 32 = 4.43520 \times 10^5$$

  grid points. A set of sensitivity experiments (not shown) suggests that $k = 1$ and $m = 50$. In each of these experiments, only one assimilation cycle (4 days) is conducted.

[2] Helen is an SGI ICE 8200EX system. The first part of the system is comprised of 122 nodes. Each node has two 4-core 2.93 GHz Intel X5570 (Nehalem) processors and 24 GB of RAM. The processors are hyperthreaded - each physical core appears as two logical processors. The second part of the system consists of two extra ICE 8400EX racks with 179 extra nodes. These nodes have two 6-core 2.93 GHz X5670 (Westmere) processors and 24 GB of RAM. Like the Nehalem processors these are hyperthreaded. Then, the system has a total of 602 processors.

– TC3: the Angola Basin with 10 km of resolution and 40 terrain-following vertical levels. The vertical levels are stretched as so to increase resolution near the surface. The model domain with highlighted bathymetry is shown in Fig. 1. The experiments consist of a 4 day window using IS4DVar assimilating satellite Sea Surface Temperature (SST), in situ T&S profiles and Sea Surface Height (SSH) observations from the 1st to the 5th January 2013 (Table 1).

Table 1. Strong scaling results for TC1, TC2 and TC3 on CX2. As $k = \log(p) = 1.2$ and $r = 4$, experimental results confirm the upper bound in (4).

p	T_p (secs)	S_p	E_p	p	T_p (secs)	S_p	E_p	p	T_p (secs)	S_p	E_p
TC1				TC2				TC3			
1	7088	1	1	1	42224	1	1	1	109905	1	1
2	3859	1.84	0.92	2	24424	1.7	0.8	2	57550	1.91	0.96
4	2348	3.02	0.76	4	15411	2.7	0.7	4	31648	3.47	0.87
8	1704	4.16	0.52	8	10501	4.0	0.5	8	18697	5.88	0.74
16	1770	4.00	0.25	16	9117	4.6	0.3	16	11755	9.35	0.58
32	8022	13.70	0.43					32	8022	13.70	0.43
								64	5814	18.90	0.30

In all the experiments we use realistic configurations of medium-size, so performance results show that the efficiency degrades below 50% for $p > 16$. As $k = \log(p) = 1.2$ and $r = 4$ or $r = 5$ at most, experimental results confirm the upper bound in (4), assuming that for the ROMS implementation of IS4DVAR Algorithm the ratio of the sustained unitary communication time over the sustained computation time is $a = 10^q$ where $q \simeq 2.8$ or $q \simeq 3.8$.

4 Conclusion and Future Work

The analysis showed that the surface-to-volume of the current parallelization strategy of IS4DVAR Algorithm strongly limits the performance of the ROMS software as it does not fulfill the features of the emerging architectures, where the unitary sustained communication time should be comparable to the computation time. In line with these issues, and relying on previous activities of the authors [23], the approach we are going to adopt in the NASDAC research activity meets the following demand: parallelization of IS4DVAR Algorithm has be considered from the beginning, which means on the numerical model [6,9,10].

In the next steps in future direction, we will focused on infrastructure improvement with particular regard to data movement [5,7,11,14,16,17,21] in order to implement a reliable mechanism able to move acquired data for processing, publishing and usage with techniques devoted to improve the scalability on HPC systems [12].

Acknowledgment. The research has received funding from European Commission under H2020-MSCA-RISE NASDAC project (grant agreement n. 691184).

References

1. Antonelli, L., Carracciuolo, L., Ceccarelli, M., D'Amore, L., Murli, A.: Total variation regularization for edge preserving 3D SPECT imaging in high performance computing environments. In: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (eds.) ICCS 2002. LNCS, vol. 2330, pp. 171–180. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46080-2_18
2. Arcucci, R., D'Amore, L., Celestino, S., Laccetti, G., Murli, A.: A scalable numerical algorithm for solving Tikhonov regularization problems. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 45–54. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_5
3. Arcucci, R., D'Amore, L., Carracciuolo, L., Murli, A.: A scalable variational data assimilation. J. Sci. Comput. **61**, 239–257 (2014)
4. Arcucci, R., D'Amore, L., Marcellino, L., Murli, A.: Hpc computation issues of the incremental 3D variational data assimilation scheme in oceanvar software. J. Numer. Anal. Ind. Appl. Math. **7**, 91–105 (2012)
5. Boccia, V., Carracciuolo, L., Laccetti, G., Lapegna, M., Mele, V.: HADAB: enabling fault tolerance in parallel applications running in distributed environments. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7203, pp. 700–709. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31464-3_71
6. Carracciuolo, L., D'Amore, L., Murli, A.: Towards a parallel component for imaging in PETSc programming environment: a case study in 3-D echocardiography. Parallel Comput. **32**, 67–83 (2006)
7. Caruso, P., Laccetti, G., Lapegna, M.: A performance contract system in a grid enabling, component based programming environment. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 982–992. Springer, Heidelberg (2005). https://doi.org/10.1007/11508380_100
8. D'Amore, L., Campagna, R., Galletti, A., Marcellino, L., Murli, A.: A smoothing spline that approximates laplace transform functions only known on measurements on the real axis. Inverse Probl. **28**, 025007 (2012)
9. D'Amore, L., Laccetti, G., Romano, D., Scotti, G., Murli, A.: Towards a parallel component in a GPU-CUDA environment: a case study with the L-BFGS harwell routine. Int. J. Comput. Math. **92**, 59–76 (2015)
10. D'Amore, L., Marcellino, L., Mele, V., Romano, D.: Deconvolution of 3D fluorescence microscopy images using graphics processing units. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7203, pp. 690–699. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31464-3_70
11. Gregoretti, F., Laccetti, G., Murli, A., Oliva, G., Scafuri, U.: MGF: a grid-enabled MPI library. Future Gener. Comput. Syst. (FGCS) **24**, 158–165 (2008)
12. Guarracino, M.R., Laccetti, G., Murli, A.: Application oriented brokering in medical imaging: algorithms and software architecture. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 972–981. Springer, Heidelberg (2005). https://doi.org/10.1007/11508380_99

13. Gurol, S., Weaver, A.T., Moore, A.M., Piacentini, M., Arango, H.G., Gratton, S.: B-preconditioned minimization algorithms for variational data assimilation with the dual formulation. Q. J. Roy. Metereol. Soc. **140**, 539–556 (2014)
14. Laccetti, G., Lapegna, M.: PAMIHR. A parallel FORTRAN program for multidimensional quadrature on distributed memory architectures. In: Amestoy, P., Berger, P., Daydé, M., Ruiz, D., Duff, I., Frayssé, V., Giraud, L. (eds.) Euro-Par 1999. LNCS, vol. 1685, pp. 1144–1148. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48311-X_160
15. Maschhoff, A.J., Sorensen, D.: A portable implementation of ARPACK for distributed memory parallel architectures, vol. 91 (1996)
16. Montella, R., Giunta, G., Laccetti, G.: Virtualizing high-end GPGPUS on ARM clusters for the next generation of high performance cloud computing. Clust. Comput. **17**, 139–152 (2014)
17. Montella, R., Giunta, G., Laccetti, G., Lapegna, M., Palmieri, C., Ferraro, C., Pelliccia, V.: Virtualizing CUDA enabled GPGPUs on ARM clusters. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 3–14. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_1
18. Moore, A.M., Arango, H.G., Broquet, G., Edwards, C., Veneziani, M., Powell, B., Foley, D., Doyle, J.D., Costa, D., Robinson, P.: The Regional Ocean Modeling System (ROMS) 4-dimensional variational data assimilation systems. Part II - performance and application to the California Current System. Prog. Oceanogr. **91**(1), 50–73 (2011)
19. Moore, A.M., Arango, H.G., Broquet, G., Edwards, C., Veneziani, M., Powell, B., Foley, D., Doyle, J.D., Costa, D., Robinson, P.: The Regional Ocean Modeling System (ROMS) 4-dimensional variational data assimilation systems. Part III - observation impact and observation sensitivity in the California Current System. Prog. Oceanogr. **91**(1), 74–94 (2011)
20. Moore, A.M., Arango, H.G., Broquet, G., Powell, B.S., Weaver, A.T., Zavala-Garay, J.: The Regional Ocean Modeling System (ROMS) 4-dimensional variational data assimilation systems. Part I - system overview and formulation. Prog. Oceanogr. **91**(1), 34–49 (2011)
21. Murli, A., Boccia, V., Carracciuolo, L., D'Amore, L., Laccetti, G., Lapegna, M.: Monitoring and migration of a PETSc-based parallel application for medical imaging in a grid computing PSE. In: Gaffney, P.W., Pool, J.C.T. (eds.) Grid-Based Problem Solving Environments. ITIFIP, vol. 239, pp. 421–432. Springer, Boston, MA (2007). https://doi.org/10.1007/978-0-387-73659-4_25
22. Murli, A., Cuomo, S., D'Amore, L., Galletti, A.: Numerical regularization of a real inversion formula based on the Laplace transform's eigenfunction expansion of the inverse function. Inverse Prob. **23**(2), 713–731 (2007)
23. Murli, A., D'Amore, L., Laccetti, G., Gregoretti, F., Oliva, G.: A multi-grained distributed implementation of the parallel block conjugate gradient algorithm. Concurr. Comput.: Pract. Exp. **22**, 2053–2072 (2010)
24. Nocedal, J., Wright, S.: Numerical Optimization. Springer, New York (1999). https://doi.org/10.1007/978-0-387-40065-5

Evaluation of HCM: A New Model to Predict the Execution Time of Regular Parallel Applications on a Heterogeneous Cluster

Thiago Marques Soares, Rodrigo Weber dos Santos, and Marcelo Lobosco(✉)

Graduate Program on Computational Modeling, Federal University of Juiz de Fora, Juiz de Fora, Brazil
thiagomarquesmg@gmail.com, {rodrigo.weber,marcelo.lobosco}@ufjf.edu.br

Abstract. In a previous work we proposed a new model that predicts the execution time of a regular application on a heterogeneous parallel environment. The model considers that a heterogeneous cluster is composed by distinct types of processors, accelerators and networks. This work further details and proposes some modifications to the original model, as well as evaluate it on a heterogeneous cluster environment. The results have show that the worst error in the estimations of the parallel execution time was about 12.7%, and, in many cases, the estimated execution time is equal to or very close to the actual one.

Keywords: Performance modeling · Parallel architectures Heterogeneous clusters · Scheduling · Resource management

1 Introduction

Scientific simulations usually demand the use of high-end computers, which are projected to deal with the processing of large amounts of data. Cluster of computers is perhaps the most representative architecture devoted to deal with this task. Due to the rapid development and adoption of new technologies, this architecture is becoming more heterogeneous, mixing, in a single system, distinct processors, accelerators, such as GPUs, and network connections.

The goal of this paper is to present a parallel model, HCM (Heterogeneous Cluster Model), that estimates the execution time of regular parallel applications running on small heterogeneous clusters. HCM modifies its previous version [8] in order to better estimate the time applications spend doing computation. In this work we also present, in more details, some of the key aspects of HCM, as well as evaluate it using distinct applications. The preliminary results are

The authors would like to thank UFJF, FAPEMIG, CAPES, and CNPq for supporting this research.

promising: the model could predict the parallel execution time with a moderate margin of error (from 0% to 12.7%).

The remaining of this work is organized as follows. Section 2 reviews the model and presents the proposed modifications. The benchmarks used to evaluate HCM are described in Sect. 3. The experimental results are presented in Sect. 4. Finally, Sect. 5 presents our conclusions and plans for future works.

2 HCM: A Model to Predict the Execution Time on Parallel Environments

The objective of HCM is to predict, with a moderate margin of error, the total execution time of a regular parallel application on modern small heterogeneous cluster environments, composed by distinct processors, accelerators and networks. In HCM, the prediction of the execution time takes into account that it is composed by two distinct phases: (a) computation, and (b) communication. So, the model introduces some variables that will be used to model the computation phase, and other variables that will be used to model the communication phase.

In order to estimate the execution time of a regular parallel application, the following steps must be followed: (a) a mathematical model that describes the main computation and communication phases of the application must be written using the set of variables defined in the HCM model; (b) the time spent by a CPU to execute a small number of sequential steps of the application must be collected; and (c) some parameters must be collected from the heterogeneous cluster. The first two steps varies from application to application, while the last one depends only on the hardware used and can be collected and stored for all applications only once, as it will be explained in details below.

2.1 Estimating the Computation Time

The main parameters and variables used in HCM to model the computation time of a regular parallel application are the following: (a) R_P, the relative computing power of the processing units (i.e., CPU cores and accelerators, such as GPUs); (b) the total number of iterations(**I**) that will be executed; and (c) the time (T_s) to execute a small number of sequential iterations of the program (I_s).

Our previous work [8] proposed two ways to measure R_P on each processing unit: (a) running a benchmark to collect the amount of data that can be handled per time, or (b) using the average computation time that a processing unit takes to run some sequential iterations of an application. In the first case, given the size of the problem (**size**) and the amount of data a device can handle per time (R_P), the computation time was given by: $size/R_P$. In the second case, the computation time was given by $R_P \times I$.

In this work we propose a new way to compute R_P. A benchmark or some sequential iterations of an application can be executed to collect any metric: execution time, amount of data processed per time, iterations per second, etc. The execution is done in each processing unit and, after executed, the value obtained by the slowest processing unit is used to normalize the results, so R_P is dimensionless. The following equation is proposed to estimate the computation time of an application:

$$T_{computation} = \frac{I}{I_s} \times \left(\frac{T_s}{\sum_{i=1}^{n} R_{P_i} \times F_r}\right), \tag{1}$$

where I is the total number of iterations of the problem, and I_s is the number of sequential iterations that will be used to predict the computation time of the application. More accurate values for the prediction of the parallel execution time are obtained when larger values of I_s are used. T_s is the time to execute I_s iterations of the application considering the slowest processing unit, which is used as reference because the faster ones must wait for it before finishing a computation step. Even if load balancing is used, the computation time will not be exactly the same in all processing units due to the overhead imposed by the load balancing algorithm or because it may be impossible to split the data unit used in computation, which would be mandatory to obtain a perfect balancing across multiple heterogeneous devices. $\sum_{i=1}^{n} R_{P_i}$ is the sum of the R_p values for all processing units that will be used in the parallel execution, and F_r is a correction factor. The correction factor can be used when a linear speedup is not expected to occur in the computation phase. Constant values equal to, less than or greater than one or even the result of a function can be used to impose a speedup factor distinct from the linear one. For example, this could occur due to the reduction of memory access costs, since more cache space is available with the use of more processors. Additionally, observe that **size** is not used anymore to estimate the computation time.

The use of some sequential steps of the application whose parallel execution time will be estimate rather than the use of a generic benchmark has the advantage of generating precise results, since the set of instructions present in the benchmark may be distinct from the set of instructions used by the application. While a standard benchmark can be executed only once for each processing unit and then used to estimate the computation time of many distinct applications, the execution of some sequential steps of the application is necessary each time a new hardware is included in the parallel environment.

2.2 Estimating the Communication Time

This work proposes the use of a modified version of the LogP [2] model in order to predict the communication time of a parallel application in a heterogeneous environment. The main parameters and variables used in HCM are the following: (a) $\mathbf{L}_d$, that represents an upper bound on the communication latency of a device d; (b) $\mathbf{o}_d$, that represents the overhead in device d, i.e., the time that a processor spends in the transmission or reception of each message and that cannot be used to execute other instructions; (c) $\mathbf{g}_d$, the gap, that represents the minimum time interval between consecutive message transmissions/receptions by a processor in a device d; (d) N_{op}, the number of communication operations per iteration, and (e) $\mathbf{M}$, the message size. The gap can be replaced by the bandwidth (B_d), since one is the inverse of the other.

Some parameters used to estimate the communication time can be acquired only once, using specific benchmarks for this purpose. For example, the values of $\mathbf{L}_d$ and $\mathbf{g}_d$ were obtained in this work using a network benchmark. The benchmark was executed for each type d of network that was available in the cluster. The benchmark collected the values of $\mathbf{L}_d$ and $\mathbf{g}_d$ for distinct message sizes, varying from 0 to 4 MB. Each message size modeled by our model uses their corresponding $\mathbf{L}_d$ and $\mathbf{g}_d$ values found by the benchmark. Also, the value of $\mathbf{o}_d$ was obtained using a benchmark [3] that considers that the overhead varies with the message size. The total overhead in a point-to-point communication operations is given by: $o_d = o_s + o_r$, where o_s and o_r are respectively the send and the receive overheads. Again, each message size modeled by our model uses the corresponding $\mathbf{o}_d$ value found by the benchmark. Benchmarks can be executed once to collect the communication costs, overheads, as well as the relative performance of the processors and accelerators, or each time a new hardware or network is included in the system.

The communication cost depends basically on two factors: the type of message sent (point-to-point or collective), and the message size. The message size determines the costs, and the type of message sent the number of messages exchanged. As a general rule, the cost of a single message is equal to:

$$T_{single} = L_d + \frac{M}{B_d} + o_d. \tag{2}$$

The type and the size of each message exchanged by the application can be found directly in its source code. It is also important to verify how the communication library implements their primitives in a specific network, since some optimizations can be made. The next section will present some formulas that can be used to model the communication costs for distinct communication patterns.

The main differences between HCM and the original LogP model are the following: (a) HCM focus on the prediction of the total execution time; (b) the LogP model assumes an homogeneous environment, while HCM assumes an heterogeneous one; (c) the LogP model assumes that all messages are of the same small size, while HCM makes no assumption about the message size, which is also a parameter for our model; and (d) new parameters and variables, such as R_P, N_{op}, $\mathbf{M}$, I_s, etc., were introduced in order to better estimate both the computation and communication times of an application.

3 Benchmarks and Their Models

This section presents the process of building a mathematical model based on HCM to describe the execution time of an application. Two NAS benchmarks were selected to illustrate the process: IS and CG. Although this section presents only two NAS benchmarks, the next section will present the parallel execution times estimated using HCM for the complete NAS benchmark. Since the NAS benchmarks were developed to execute in a CPU environment, another application, HIS (Human Immune System) [6], was chosen to illustrate how HCM can be used to estimate the execution time of an application on a hybrid CPU/GPU environment.

3.1 IS

Integer Sort (IS) is a kernel that performs sorting operations. This kernel tests both integer computation speed as well as the communication performance [1]. Algorithm 1 presents the pseudocode of this application.

Algorithm 1. Pseudocode of the IS kernel

```
main
  ... generate sequence of rand numbers and subsequent keys on all processors ...
  ... get the bucket size for the entire problem using MPI_Allreduce ...
  ... determine the redistribution of keys ...
  ... redistribute using MPI_AlltoAll ...
  ... send the keys to the respective processors using MPI_Alltoallv ...
  ... determine total # of keys on all other processors ...
end-main
```

Equation 3 models the IS benchmark.

$$T_{total} = T_{computation} + T_{communication}, \tag{3}$$

where

$$T_{communication} = N_{op} \times \{[2 \times (P - 1) \times T_{single}] + \log_2 P \times T_{single}\}. \tag{4}$$

For IS and all the following applications, the value of $T_{computation}$ is given by Eq. 1 and the value of T_{single} is given by Eq. 2.

As one can observe, for each MPI communication primitive in the code, a distinct equation is used to model it. However, some of them can be modeled in the same way, since their implementation are similar. The first part of the equation (inside square brackets) models two MPI primitives: MPI_AlltoAll and MPI_Alltoallv. In the MPI library used, these primitives are implemented in the same way for all network cards used: a process send messages to all other processes, except itself. Distinct MPI implementations can implement these primitives in distinct ways depending on, for example, the network characteristics. The last part of the equation models the MPI_Allreduce primitive.

3.2 CG

The CG kernel uses the conjugate gradient method to compute an approximation to the smallest eigenvalue of a large sparse symmetric positive matrix [1]. This kernel tests irregular long distance communication employing unstructured matrix vector multiplication [1]. Algorithm 2 presents the pseudocode of this application, in which only point-to-point communication primitives are used.

Equation 5 models the CG benchmark.

$$T_{total} = T_{computation} + N_{op} \times T_{single}. \tag{5}$$

Algorithm 2. Pseudocode of the CG kernel

```
main
  ... call to the conjugate gradient routine ...
      ... obtain rho with a sum-reduce using MPI_Send ...
      ... sum the partition submatrix-vec A.z's across rows using MPI_Send ...
      ... exchange pieces of q using MPI_Send ...
  ... normalize z to obtain x ...
end-main
```

3.3 HIS

A three dimensional simulator of the Human Immune System (HIS) [6] was used in the model evaluation. The simulator implements a mathematical model that uses a set of eight Partial Differential Equations (PDEs) to describe how some cells and molecules involved in the innate immune response react to a pathogen. A detailed discussion about the model can be found in previous works [5,6]. The implementation is based on the Finite Difference Method [4] for the spatial discretization and the explicit Euler method for the time evolution. The code was implemented in C and uses CUDA to solve the PDEs simultaneously in CPUs and GPUs. The CPU is also responsible for the communication, due to border exchanges, using MPI for this purpose. As one can observe, this implementation is distinct from the one used to evaluate our model in our previous work [8] because HIS used only GPUs for computation [7], and now both GPUs and CPUs are used. Also, a dynamic load balancing technique (not shown in the algorithm) is used to improve performance. This represents an additional challenge to HCM because the application behavior changes along its execution: the time to execute each iteration is distinct, so this application behaves like a irregular one and HCM was designed to deal with regular applications. Border exchange occurs at the end of each iteration. For this application, the communication between CPU and GPU was not considered. Algorithm 3 gives an overview of the implementation of the HIS simulator.

Algorithm 3. Implementation of the HIS simulator

```
main
  ... define the mesh slice to be computed by each device ...
  ... initialize submeshes according to their initial conditions ...
    for $t$ from 0 to final time do
      ... Computes points on CPUs and GPUs ...
      ... use MPI_Isend and MPI_Receive to exchange boundaries ...
      ... synchronize all machines ...
    end-for
end-main
```

Equation 6 models the HIS benchmark. Point-to-point communication primitives are used to exchange boundaries between distinct machines. The simultaneous use of CPUs and GPUs in the computation does not change the way the $T_{computation}$ is modeled. Although the use of the dynamic load-balancing technique impacts the amount of data exchanged at each iteration, we assumed it remains constant in order to simplify the model.

$$T_{total} = T_{computation} + N_{op} \times T_{single}. \tag{6}$$

4 Results

In this section, we present the values that HCM estimates to the execution time of HIS and all NAS benchmarks. The actual execution time was obtained using the average of 5 executions in order to minimize the standard deviation.

The experiments were executed on a small cluster with 16 machines. Half of these machines have two Intel Xeon $E5620$ processors with 16 GB of main memory, six of these have two Tesla C1060 GPUs (240 CUDA cores and 4 GB of global memory each) and the other two have two Tesla M2050 GPUs (448 CUDA cores and 3 GB of global memory). The other eight machines have two AMD 6272 processors, with 32 GB of main memory, two Tesla M2075 GPUs, each one with 448 CUDA cores and 6 GB of global memory. Linux 2.6.32, CUDA driver version 6.0, OpenMPI version 1.6.2, nvcc release 6.0 and gcc version 4.4.7 were used to run and compile all codes. Two distinct networks are available in the cluster: Intel 82576 Gigabit Ethernet and InfiniBand Mellanox MT26428 with a QDR of 40 Gb/s. The Intel machines are connected by the Gigabit Ethernet card, while the AMD machines are connected by both cards. Both cards have the full-duplex mode, so data can be transmitted and received simultaneously. For this reason, the model for each application considers only half of the number of messages exchanged since they occur in parallel. Although the total number of cores available in each machine is equal to 32 for AMD (2×16) and 8 for Intel (2×4), in all experiments only one core was used per machine.

Two distinct environments were used in the experiments. A homogeneous environment that uses only one type of CPU and a heterogeneous one, that mixes distinct types of CPUs. In the homogeneous environment, composed by AMD processors, we also used distinct types of network cards (Ethernet and Infiniband). In the heterogeneous environment, half of the processors are AMD and half are Intel. The only exception are BT and SP benchmarks that were executed with 9 processors, in which 5 Intel and 4 AMD CPUs were used. Also, we evaluated our model on a homogeneous and a heterogeneous GPU environment. The homogeneous environment is composed only by M2075 GPUs, while the heterogeneous one mixes C1060, M2050 and M2075 GPUs.

4.1 Parameters

For the NAS benchmark, the value of I is given by the problem size. For all benchmarks, the class C was used in the simulations. The exceptions are FT and LU, which used class B in the evaluations. For the HIS benchmark, it is equal to $10,000$ steps. The value of I_s was fixed in a value equal to 10% of I. In all experiments we considered that F_r is equal to one. The value used for R_P is given in Table 1, which was obtained using the HIS execution time as reference to establish a relationship among distinct processing units. For some applications, the N_{op} value is constant. However, for other applications its value depends on P. For this reason, and due to the lack of space, its value will be omitted. The MTU size, 1.5 kbytes, defines the values used for the latency and bandwidth, which are equal to (a) $L_{eth} = 6.9 \times 10^{-5}$ s and $B_{eth} = 93.4$ MB/s for Ethernet and (b) $L_{inf} = 5.1 \times 10^{-6}$ s and $B_{inf} = 1{,}030.3$ MB/s for Infiniband. The value of o_d depends on the message size M.

Table 1. Values of R_P for each processing unit available in the computational platform.

Processing unit	R_P
AMD	1
Intel	1.78
C1060	131.22
M2050	299.34
M2075	333.73
M2090	364.41

4.2 Results

Table 2 shows the result for HIS. HIS is executed simultaneously on GPUs and CPUs, which is a very challenge scenario, since we mix distinct GPUs and CPUs. As one can observe, the errors varied from 0% to 11.8%.

Table 3 presents the results for the NAS benchmark running on a homogeneous CPU environment, but using distinct network cards, Ethernet and Infiniband. The errors varied from 3% to 12.7%. In absolute values, the difference between the estimated execution time and the actual one was very tiny for IS: 0.1 s in InfiniBand and 0.4 s in Ethernet. Table 4 presents the results for the NAS benchmark running on a heterogeneous CPU environment, composed by Intel and AMD processors, and using the Ethernet network card. The errors varied from 3.2% to 11.1%. The difference between the estimated execution time and the actual one, in absolute values, was tiny for IS (0.4 s) and EP (0.9 s).

In general, the execution time estimated by HCM was very close to the actual one for the IS application, both when considering a scenario with heterogeneous processors, as well as in a scenario with distinct network adapters. On the other hand, the worst results were obtained for the estimation of the execution time of SP. The reason is that SP did not scale linearly in our environment: the computation time reduces about 3.2 times (from 414 s to 130 s) when the number of nodes increases 4 times (from 4 to 16). Perhaps a good choice is to choose a function to represent F_r, instead of one, as we used.

Table 2. Results for HIS using both GPUs and CPUs and Ethernet network. All times in seconds. Both absolute and percentage errors are presented. Configuration 1: 2 CPUs (1 AMD and 1 Intel) and 2 GPUs (M2075 and C1060). Configuration 2: 3 CPUs (1 AMDs and 2 Intels) and 3 GPUs (1 M2075 and 2 C1060). Configuration 3: 7 CPUs (3 AMDs and 4 Intels) and 7 GPUs (3 M2075, 2 M2050 and 2 C1060).

Configuration #	Actual	Estimated	Error
1	47.2	51.2	4.0/8.6%
2	57.4	57.4	0.0/0.0%
3	107.8	95.1	12.7/11.8%

Table 3. Results for the NAS benchmark using 8 AMD processors on two distinct network cards. All times are in seconds. Both absolute and percentage errors are presented. BT and SP require a square number of processors, and executed in 9 nodes.

	Ethernet			Infiniband		
	Actual	Estimated	Error	Actual	Estimated	Error
FT	73.8	68.7	5.1/6.9%	23.9	21.7	2.2/9.0%
IS	10.0	9.6	0.4/3.4%	3.4	3.3	0.1/5.4%
CG	150.3	169.2	18.9/12.6%	70.5	77.9	7.4/10.5%
MG	38.2	42.3	4.1/10.6%	23.3	25.1	1.8/7.4%
EP	71.3	74.0	2.7/3.8%	71.2	74.0	2.8/3.9%
LU	77.0	74.7	2.3/3.0%	62.0	57.2	4.8/7.7%
BT*	371.1	340.5	30.6/8.3%	294.7	264.5	30.2/10.2%
SP*	309.0	334.9	25.9/8.4%	238.7	266.5	27.8/12.7%

Table 4. Results for the NAS benchmark using 16 processors (8 Intel and 8 AMD) and Ethernet. All times are in seconds. Both absolute and percentage errors are presented.

	Actual	Estimated	Error
FT	65.7	61.3	4.4/6.7%
IS	4.9	4.5	0.4/7.8%
CG	262.5	253.7	8.8/3.2%
MG	51.8	46.1	5.7/11.1%
EP	28.5	27.6	0.9/3.2%
LU	62.7	57.9	4.8/7.4%
BT	245.8	259.5	13.7/5.5%
SP	343.2	305.1	38.1/11.1%

5 Conclusion and Future Works

This paper evaluated HCM, a new model to predict the execution time of regular parallel applications on a small heterogeneous parallel environments. Some modifications to our previous work were proposed in this paper, basically in the way the computation time is estimated. The results have shown that HCM can predict the total execution time of regular applications with distinct communication characteristics, running on distinct devices and interconnected by different network types. The error found during the estimation of the total execution time stayed below to 12.7% and, in some cases, was equal or very close to the actual execution time. As future work, we plan to use this model to predict the hardware configuration that minimizes the execution time of an application, which is not necessarily the configuration that uses all computational resources available. This can be of great impact in the management of computational resources and scheduling of tasks in a cluster environment.

References

1. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The NAS parallel benchmarks. Int. J. High Perform. Comput. Appl. **5**(3), 63–73 (1991)
2. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: Logp: towards a realistic model of parallel computation. SIGPLAN Not. **28**(7), 1–12 (1993)
3. Doerfler, D., Brightwell, R.: Measuring MPI send and receive overhead and application availability in high performance network interfaces. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) EuroPVM/MPI 2006. LNCS, vol. 4192, pp. 331–338. Springer, Heidelberg (2006). https://doi.org/10.1007/11846802_46
4. LeVeque, R.: Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics Classics in Applied Mathematics). SIAM, Philadelphia (2007)
5. Pigozzo, A.B., Macedo, G.C., Santos, R.W., Lobosco, M.: On the computational modeling of the innate immune system. BMC Bioinform. **14**(Suppl. 6), S7 (2013)
6. Rocha, P.A.F., Xavier, M.P., Pigozzo, A.B., de M. Quintela, B., Macedo, G.C., dos Santos, R.W., Lobosco, M.: A three-dimensional computational model of the innate immune system. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012. LNCS, vol. 7333, pp. 691–706. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31125-3_52
7. Soares, T.M., Xavier, M.P., Pigozzo, A.B., Campos, R.S., dos Santos, R.W., Lobosco, M.: Performance evaluation of a human immune system simulator on a GPU cluster. In: Malyshkin, V. (ed.) PaCT 2015. LNCS, vol. 9251, pp. 458–468. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21909-7_44
8. Soares, T.M., dos Santos, R.W., Lobosco, M.: A parallel model for heterogeneous cluster. In: Carretero, J., et al. (eds.) ICA3PP 2016. LNCS, vol. 10049, pp. 76–90. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49956-7_6

Workshop on Power and Energy Aspects of Computations (PEAC 2017)

Applicability of the Empirical Mode Decomposition for Power Traces of Large-Scale Applications

Gary Lawson[✉], Masha Sosonkina, Tal Ezer, and Yuzhong Shen

Old Dominion University, Norfolk, VA 23529, USA
{glaws003,msosonki,tezer,yshen}@odu.edu

Abstract. Current trends in HPC show that exascale systems will be power capped, prompting their users to determine the best combination of resources to satisfy a power budget. Hence, performance and energy models must interplay and aid users in this resource selection based on the desired application parameters. While existing performance models may predict application execution at a scale, current power models are inadequate for this propose due, in part, to the variability of instantaneous dynamic power and the need to handle large amount of power measurements at the runtime to populate the models. In this paper, the latter challenge is tackled by selecting certain power measurements and applying to them the empirical mode decomposition (EMD) technique, which itself already deals with instantaneous variability of power during the runtime. Specifically, it is proposed here to apply EMD to segments of a power trace to rapidly generate a quadratic model that describes overall time, power, and thus energy simultaneously. The proposed models have been applied to several realistic applications. The error across the proposed models and the measured energy consumption is within 5% for the smaller segments consisting of 2,000 trace samples and is about 2% for the segments of 6,000 samples.

Keywords: Energy savings · Energy modeling · Power traces
Empirical mode decomposition · Power measurements

1 Introduction

Exascale systems of the future will have more strict power requirements for devices in order to meet the 20 MW power cap imposed by the U.S. Department

This work was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, by the U.S. Department of Energy, Office of Advanced Scientific Computing Research, through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC02-07CH11358, and by the U.S. Department of Defense High Performance Computing Modernization Program, through a HASI grant. The authors thank the reviewers for their constructive comments.

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 71–80, 2018.
https://doi.org/10.1007/978-3-319-78054-2_7

of Energy [9] to keep the annual cost of the cluster maintainable since power is expensive, especially when 60% goes toward cooling the system. Thus, it is not far fetched to believe that future systems will come in two flavors; those where hardware devices meet power draw limitations, and those where users must abide by power caps for resource requests. Considering the current trend in power draw, the later is the more likely scenario in the near-future; thus it is important to optimize application performance for power constraints. However, this is quite the endeavor.

One of the many areas of research in HPC is the determination of the optimal configuration settings for an application on a given hardware platform. A user may not be able to run long experiments to test all permutations of admissible configuration options. To alleviate this problem, other methods include auto-tuning an application for common compatible optimizations, which is difficult to perform for real-world applications, and still requires testing all permutations. A less costly option is to run the application fewer times, capture application behavior using time, power, and energy models, and then use the models to predict untested options. Although this is more ideal, it is incredibly difficult to quantify application-hardware interactions in a manner that a model can accurately predict. This work is a step towards predicting application performance on hardware platforms using a novel energy model, based on the Empirical Model Decomposition (EMD) method [6,18].

Empirical Mode Decomposition: EMD has been investigated by the authors to obtain a reliable model for energy [10]. In [10], EMD has been applied to the entire power trace to avoid breaking it into phases corresponding to specific parallel patterns, such as communication or computation, to correlate them with specific power measurements. For the proxy application CoMD, such a phase distinction within a power trace has been achieved by the authors in [11], leading to a construction of a sequence of models corresponding to each phase. For real-world application, this distinction is not always possible; thereby EMD has been adapted in [10] for the power trace.

EMD provides non-parametric non-stationary time-series analysis, which has been already successfully applied in a variety of fields, such as medicine, finance, engineering, and more recently in geosciences. The EMD code used here is based on the code adapted for analysis of sea level data [4] and climate change studies [3]. The main advantage of EMD over standard spectral methods is that it detects oscillating modes with time-dependent amplitudes and frequencies, so it is useful for analyzing irregular data with unknown frequencies. When EMD is applied to a time-series, it produces a sequence of intrinsic mode functions (IMFs) and the resultant trend known as the "residual"; the sum of the residual and IMFs restores the original time-series [6]. Each IMF has an amplitude and frequency, which can be used to estimate the Teager Energy of the time-series. Teager Energy is the amount of physical energy required to generate a signal, computed as the product of amplitude and frequency [14]. To compute the Teager Energy, a power trace in its entirety may be required.

Quadratic-Fit Model for Residual: The EMD-based energy model proposed in [10] describes power draw as a function of time. The model is obtained by first applying EMD to a *complete* power trace and then fitting a quadratic equation ($P(t) = a \cdot t^2 + b \cdot t + c$) to the residual. In this paper, that model is denoted as the *quadratic-fit residual* (QFR) model, an example of which is shown in Fig. 1. Power is shown on the y-axis, and time is the x-axis in Fig. 1. The power model describes the trend in power draw over time where power draw always returns to idle. The coefficients relate to time in maximum power draw as shown in Eqs. (1) and (2). To obtain these definitions, assume that c = 0 and static power draw is removed from the trend; then time may be described as the x-intercept greater than zero, see Eq. (1). Power is defined at the apex, or axis of symmetry [16], as shown in Eq. (2). Notice that power, even defined as a quadratic, has a static and dynamic component, where static power is c. Conversely, the coefficients may then be defined using these definitions; a is shown in Eq. (3) and b is shown in Eq. (4). The QFR shown in Fig. 1 was created for a time of 450 s, static power of 80 W, and dynamic power of 90 W - the coefficients are then $a = -0.0018$, $b = 0.80$, and $c = 80$. Using the QFR to model energy, it has been shown that measured energy for traces longer than 100 s has an error of 10% or less [10]. Power measurements and analysis of these measurements is important; the influence of hardware behavior is critical to the power behavior of the application and must be tested since the hardware behavior is not easily predicted.

The paper is organized as follows: Sect. 2 describes the proposed model and its variant using certain trace segments instead of the entire trace, Sect. 3 discusses the error in the proposed energy modeling, and Sect. 4 concludes and notes on future research directions.

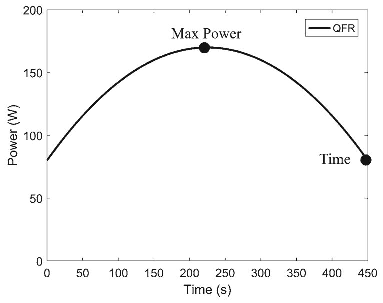


Fig. 1. QFR model of power over time.

$$Time = \frac{-b}{a} \tag{1}$$

$$MaxPower = \frac{-b^2 + 4ac}{4a} \tag{2}$$

$$a = \frac{-b}{Time} \tag{3}$$

$$b = \frac{4 \cdot Pdyn}{Time} \tag{4}$$

1.1 Related Work

Determination of the optimal hardware-application strategy is challenging. Autotuning [7,8] is a method where many different compiled versions of a code are tested for performance. However, all permutations must be tested which is time consuming but is a practical choice. It would be more ideal to use a model, but execution performance is not easily quantified into available models. Some models measure only performance [17]; others require advanced or expert level knowledge of the application, hardware, or both [1,12]. These requirements are not feasible for an end-user of an Exascale system. The method proposed in this work can be used to rapidly generate a model of an application-hardware combination.

2 Segmented Trace Modeling (STM)

STM approximates the QFR using a power trace of only a fraction of total execution time, which speeds up the EMD modeling process and makes the power-trace handling manageable. In particular, modeling with EMD is dominated by the number of times EMD is applied to the trace, which increases non-linearly with the number of samples. For example, a trace with 6,000 samples (i.e., 30-s long at a sampling rate of 5 ms) requires 24 s to perform EMD, and a trace ten times long, with 60,000 samples, requires as much as 443 s. After 30,000 samples (150 s) of a trace in this example, the EMD processing time begins to surpass the trace length, which is detrimental for real-world applications, with typical runtime on the order of hours or days. And a faster sampling rate may only exacerbate the problem. Hence, the major aim of the proposed STM is to reduce the amount of time power must be measured to obtain the EMD residual. The viability of the segment trace modeling lies in the fact that EMD can be applied to a time-series of any length as long as there are enough measurement samples, often as few as few as five [6].

2.1 Segmenting Power Traces

In this work, 2,000 measurement samples are used per segment. EMD is sensitive to the sampling rate; the more fine-grained the samples, the more information EMD can yield. However, the more fine-grained, the more space required to store such a trace. Note that modern systems are also limited in the maximum sampling rate allowed, which is about 1 ms. For a realistically stable sampling rate, a value of 5 ms is used, which leads to 10-s trace segments (given 2,000 samples). A 10-s segment may also fit the experimentally found durations of the pre- and post-execution measurements[1], which span five seconds each.

Figure 2 shows the complete power trace (left), the power trace after EMD has been applied to each 10 s segment (center), and the power trace after every other segment has been removed (right).[2] Notice that when EMD is applied to each segment (center), it closely mimics the trend of each respective segment in the original trace (left). This is because EMD acts similar to a low-pass filter when applied to such short traces. High frequency oscillations are removed from

[1] See [10] for a discussion of the importance of these measurements.

[2] Here, all the traces were collected for the CoMD proxy application on an Intel Xeon E5-2650 v1 with 16 cores; processor clock-rate is varied.

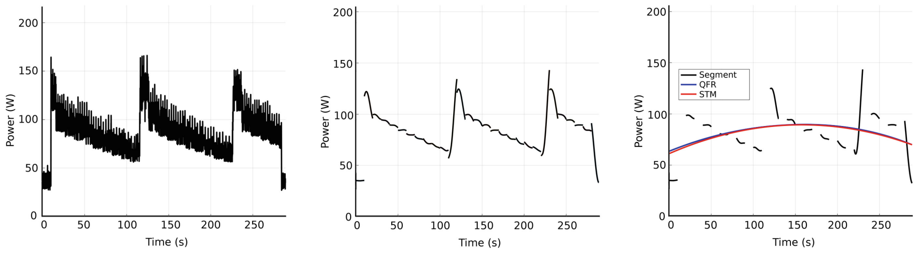


Fig. 2. Segmenting a power trace: the original power trace (left), set of residuals when EMD was applied to each 10-s segment (center), and a comparison of STM and QFR residuals for the trace with missing segments (right).

the trace to reveal the underlying trend, as shown in Fig. 2 (right). A quadratic function may be fit to the segment trends, which provides an estimate of the QFR that is quite accurate, especially when many segments are used to fit the model. In fact, the quadratics for QFR and STM appear as overlapping in Fig. 2 (right). Indeed, the difference of these curves is within 1%, albeit the STM has been calculated using all of the segments shown in Fig. 2 (right), whereas the QFR is calculated using the residual of EMD when applied to the entire power trace. This indicates that STM with missing segments is a good candidate for approximating the QFR.

The STM requires a minimum of three *key* segments, broadly denoted as *start*, *end*, and *workload*. The *start* and *end* segments are required to capture power draw at the start and end of the application with respect to idle power. Generally, an application begins by allocating memory and reading data from the hard-drive; this causes a large spike in power draw which is captured by the *start* segment. Likewise, when the application exits and memory is released, a large drop in power draw is observed which is captured by *end*. The *workload* segment depends on the application; at least one segment must be provided. Applications with large variations in power draw may require additional segments to more accurately estimate workload power draw. In this work, only one *workload* segment is used, for the sake of simplicity of exposition. The number of *workload* segments, however, may depend of the nature of the application power trace, and its determination is left as future work.

2.2 EMD on Partial Trace

Figure 3 presents two examples of the STM applied to complex power traces. The two applications CG [13] and GAMESS [5,15] with class D and 1L2Y inputs, respectively, were chosen because their power traces exhibit rather erratic variability. In particular, CG features two sections of execution with distinctly different power draws, see Fig. 3(a): The first section, ending near 200 s, has much more variability than the remainder of the trace. In Fig. 3(d), GAMESS may be characterized by multiple sections. For example, there is a distinct rise in power draw between 50 s and 100 s. Large trace variability is not typically found in

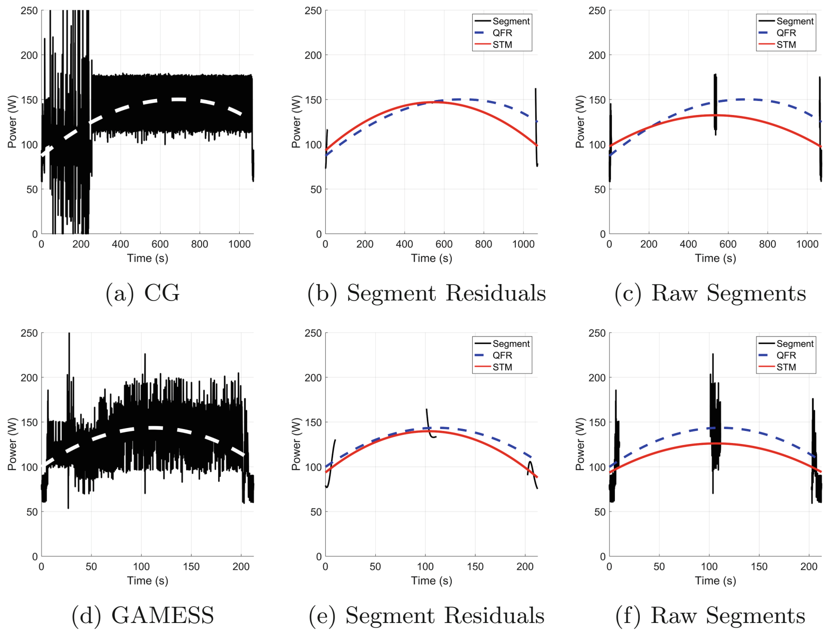


Fig. 3. STM applied to complex power traces. The QFR is shown as a dashed line in each plot (white and blue), the STM is a solid red line, and segments in black. (Color figure online)

most benchmarks, which are designed to test a particular computational feature iterating on data objects, and, hence, present a rather regular power draw. On the other hand, complex real-world applications may exhibit large variability in power draw due to the variability in workload throughout the simulation.

Figure 3(a and d) show the original power trace for CG and GAMESS, respectively, as well as their QFRs (white dashed lines). Figure 3(b, c, e, f) compare the corresponding QFRs and STMs with only three segments, chosen in a certain way. Specifically, the *workload* segment is composed of one 10-s interval taken from the absolute center of the trace; this segment was chosen to keep the resulting STM balanced (peak power in the center). At any other location (assuming only one segment), peak power would be more towards the start or end of execution which impacts the resulting STM. This may be desired to more accurately model the power draw of an application that ends with a higher power draw than that when starting the application (cf. CG in Fig. 3(a)).

Observe the differences between the QFR and STM model curves in Fig. 3(b, c) and (e, f), respectively. When EMD is applied to each of the three chosen segments, the error between QFR and STM is within 5% of the measured energy as shown in Fig. 3(b, e), while the error is greater than 10% when using the raw traces of the three key segments the for the quadratic fit as in Fig. 3(c, f). Hence, STM, which employs EMD on the key segments followed by the quadratic fit, is beneficial.

Next observe that the QFR model more closely mimics the power trend since it is based on the entire trace. On the other hand, the STM accuracy may be improved by adding more *workload* segments, and thus, capturing various trace spikes. Hence, a trade-off between the STM accuracy and speed of processing with EMD may be sought and tailored to the particular needs and resource availability.

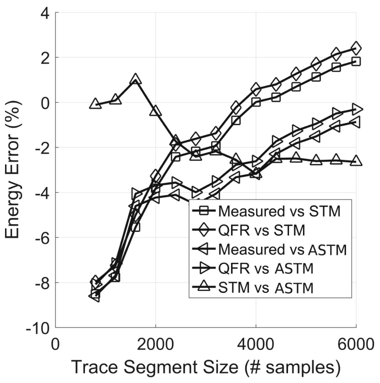
2.3 STM with Segment Approximations

Although STM reduces the amount of measurement samples required to perform the quadratic fit to the final EMD residual, the few remaining measurements must still be made throughout the entire execution. With this requirement, one still has to wait till the execution end to compute the quadratic fit. This may not be desirable for the large-scale applications that may take hours or days to execute. Therefore, a further approximation of the STM is proposed, which relies only on *one* measured segment, at the start of the execution, and assumes that the average power draw and the execution time are known (or may be estimated).

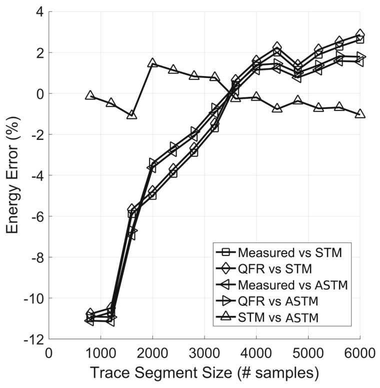
Recall that the STM requires three key segments: *start*, *workload*, and *end*. The *start* segment can be measured easily by the user, since only one segment is needed, and the time for each segment is relatively short compared to the total execution time. Assuming that average power is known, an *artificial* segment, where every sample is equal to the average power, may then be created as substitute for the *workload* segment. The *end* segment may be approximated also, if assumed that the "cool-down" period mirrors the start-up one—corresponding to the *start* segment—with a negative slope. Hence, the *start* segment characteristics may be used in place of those for the *end* with the samples in reverse order (with respect to time). The STM with the *start* segment mirroring and the *artificial* segment creation is denoted henceforth as *approximate* STM (ASTM).

3 Relative Modeling Error

Power traces were collected for several applications (CoMD, NAS parallel benchmarks, and GAMESS). CoMD is a molecular dynamics application developed by the Department of Energy co-design research effort [2] at the Extreme Materials at Extreme Scale (ExMatEx) center. Both force kernels, Lennard-Jones and the Embedded Atom Model are tested. The General Atomic and Molecular Electronic Structure System (GAMESS) [5,15] is a widely used quantum chemistry package capable of performing molecular structure and property calculations by a rich variety of *ab initio* methods finding an (approximate) solution of the Schrödinger equation for a given molecular system. The NAS Parallel Benchmarks [13] is a collection of programs used to evaluate the performance of parallel supercomputers, which was derived from computational fluid dynamics applications.



(a) CG-D



(b) GAMESS-1L2Y

Fig. 4. Energy consumption error for the STM and ASTM applied to complex power traces.

In this work, applications were run on two hardware platforms while varying the number of available cores and problem size. The two hardware platforms, *Borges* and *Rulfo* of Old Dominion University, are single-node systems; Borges is a Sandy-bridge node and Rulfo is a Xeon Phi "Knights Landing" node. Borges has 2x Intel Xeon E5-2650's with a total of 16 cores at 2.0 GHz. Rulfo has an Intel Xeon Phi 7210 with a total of 64 cores at 1.3 GHz.

Figure 4 shows the relative error in energy consumption for three pairs of models—(QFR, STM), (QFR, ASTM), and (STM, ASTM)—and for measured energy vs STM and vs ASTM with the increase in the number of samples used. All but (STM, ASTM) error curves approach zero as the segment size increases, although errors between STM and measured or QFR errors continue to grow beyond 30-s segments (i.e., 6,000 samples). The error in the energy consumption between the STM and ASTM exhibits a horizontal trend, starting from zero and leveling at about 3% of difference. This indicates that the ASTM is a good approximation of the STM. In general, the overall small magnitude of errors demonstrates that the STM and its variant ASTM approximate with an acceptable accuracy the quadratic fit into the entire trace. Note that, although Fig. 4 only depicts CG and GAMESS modeling errors, the errors computed for all the other applications tested were found to be of magnitudes and trends comparable to the ones in Fig. 4. That is to say that each method tested using CoMD, GAMESS, and each benchmark in the NPB converge within 5% error as the number of samples per segment increases.

4 Conclusion

In the proposed trace segmentation and subsequent use of the Empirical Mode Decomposition on each segment, large amounts of power trace data may be now be avoided without sacrificing the accuracy of the EMD energy model QFR, which is modified to use a quadratic fit on the residuals of each segment. The segment length, defined either by the number of samples or by time in seconds, can be used to improve the accuracy of the STM model without requiring additional segments. For segments between 10 and 30 s, error is within 5% of the measured and QFR-modeled energy consumption. Furthermore, the proposed approximations of the two key STM segments lead to as little as 5% of the additional error. The future work includes using the proposed trace approximation modeling techniques for predicting the energy consumption of large-scale applications.

References

1. Choi, J., Mukhan, M., Liu, X., Vuduc, R.: Algorithmic time, energy, and power on candidate HPC compute building blocks. In: 2014 IEEE 28th International Symposium on Parallel Distributed Processing (IPDPS), Arizona, USA, May 2014
2. DOE: Co-design (2013). http://science.energy.gov/ascr/research/scidac/co-design/
3. Ezer, T., Atkinson, L.P., Corlett, W.B., Blanco, J.L.: Gulf stream's induced sea level rise and variability along the U.S. Mid-Atlantic coast. J. Geophys. Res. Oceans **118**(2), 685–697 (2013). https://doi.org/10.1002/jgrc.20091
4. Ezer, T., Corlett, W.: Is sea level rise accelerating in the Chesapeake Bay? A demonstration of a novel new approach for analyzing sea level data. Geophys. Res. Lett. **39**(19) (2012). http://dx.doi.org/10.1029/2012GL053435
5. Gordon, M.S., Schmidt, M.W.: Advances in electronic structure theory: GAMESS a decade later (2005)
6. Huang, N., Shen, Z., Long, S., Wu, M., Shih, H., Zheng, Q., Yen, N., Tung, C.C., Liu, H.: The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis. Proc. Roy. Soc. Lond. A: Math. Phys. Eng. Sci. **454**(1971), 903–995 (1998). http://rspa.royalsocietypublishing.org/content/454/1971/903
7. Jordan, H., Thoman, P., Durillo, J., Pellegrini, S., Gschwandtner, P., Fahringer, T., Moritsch, H.: A multi-objective auto-tuning framework for parallel codes. In: Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 10:1–10:12. IEEE Computer Society Press, Los Alamitos (2012). http://dl.acm.org/citation.cfm?id=2388996.2389010
8. Kamil, S., Chan, C., Oliker, L., Shalf, J., Williams, S.: An auto-tuning framework for parallel multicore stencil computations. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–12, April 2010
9. Kusnezov, D., Binkley, S., Harrod, B., Meisner, B.: DOE exascale initiative (2013). http://www.industry-academia.org/download/20130913-SEAB-DOE-Exascale-Initiative.pdf
10. Lawson, G., Sosonkina, M., Ezer, T., Shen, Y.: Empirical mode decomposition for modeling of parallel applications on Intel Xeon Phi processors. In: Proceedings of 2nd International Workshop on Theoretical Approaches to Performance Evaluation, Modeling and Simulation. TAPEMS 2017 (2017)
11. Lawson, G., Sundriyal, V., Sosonkina, M., Shen, Y.: Modeling performance and energy for applications offloaded to Intel Xeon Phi. In: Proceedings of 2nd International Workshop on Hardware-Software Co-design for High Performance Computing, Co-HPC 2015, pp. 7:1–7:8. ACM, New York (2015). http://doi.acm.org/10.1145/2834899.2834903

12. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., Jouppi, N.P.: McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 469–480, December 2009
13. NASA: NAS parallel benchmarks (2013). http://www.nas.nasa.gov/publications/npb.html
14. Ramakrishna, G., Padmaja, N.: Estimation of teager energy using EMD. In: 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), pp. 6–11, July 2016
15. Schmidt, M.W., Baldridge, K.K., Boatz, J.A., Elbert, S.T., Gordon, M.S., Jensen, J.H., Koseki, S., Matsunaga, N., Nguyen, K.A., Su, S., Windus, T.L., Dupuis, M., Montgomery, J.J.A.: General atomic and molecular electronic structure system. J. Comput. Chem. **14**, 1347–1363 (1993). http://portal.acm.org/citation.cfm?id=163483.163497
16. Wikipedia: Quadratic function (2017). https://en.wikipedia.org/wiki/Quadratic_function
17. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009). http://doi.acm.org/10.1145/1498765.1498785
18. Wu, Z., Huang, N.: Ensemble empirical mode decomposition: a noise-assisted data analysis method. Adv. Adapt. Data Anal. **01**(01), 1–41 (2009). http://www.worldscientific.com/doi/abs/10.1142/S1793536909000047

Efficiency Analysis of Intel, AMD and Nvidia 64-Bit Hardware for Memory-Bound Problems: A Case Study of Ab Initio Calculations with VASP

Vladimir Stegailov[1,2,3](✉) and Vyacheslav Vecher[1,2]

[1] Joint Institute for High Temperatures of RAS, Moscow, Russia
[2] Moscow Institute of Physics and Technology (State University), Dolgoprudny, Russia
vecher@phystech.edu
[3] National Research University Higher School of Economics, Moscow, Russia
v.stegailov@hse.ru

Abstract. Nowadays, the wide spectrum of Intel Xeon processors is available. The new Zen CPU architecture developed by AMD has extended the number of options for x86_64 HPC hardware. Moreover, Nvidia has released a custom 64-bit Denver architecture based on the ARM instruction set. This large number of options makes the optimal CPU choice for perspective HPC systems not a straightforward procedure. Such a co-design procedure should follow the requests from the end-users community. Modern computational materials science studies are among the major consumers of HPC resources worldwide. The VASP code is perhaps the most popular tool for these research. In this work, we discuss the benchmark metric and results based on a VASP test model that give us the possibility to compare different hardware and to distinguish the best options with respect to energy-to-solution criterion.

Keywords: Energy-to-solution · VASP · Broadwell · Zen · Denver

1 Introduction

Computational materials science provides an essential part of the deployment time for high performance computing (HPC) resources worldwide. The VASP code [1–4] is among the most popular programs for electronic structure calculations that gives the possibility to calculate materials properties using the non-empirical (so called *ab initio*) methods. According to the recent estimates, VASP alone consumes up to 15–20% of the world's supercomputing power [5,6]. Such unprecedented popularity justifies the special attention to the optimization of VASP for both existing and novel computer architectures (e.g. see [7]). At the same time, one can ask a question what type of processing units would be the most efficient for VASP calculations.

A large part of HPC resources installed during the last decade is based on Intel CPUs. Novel generations of Intel CPUs present the wide spectrum of multicore processors. The number Xeon CPU types for dual-socket systems is 26 for the Sandy Bridge family, 27 for Ivy Bridge, 22 for Haswell and 23 for Broadwell families. In each family, the processors share the same core type but differ by their frequency, core count, cache sizes, network-on-chip structure etc.

In March 2017, AMD released the first processors based on the novel x86_64 architecture called Zen. It is assumed that the efficiency of this architecture for HPC applications would be comparable to the latest Intel architectures (Broadwell and Skylake).

In March 2017, Nvidia released the Jetson TX2 minicomputer with the Tegra "Parker" SoC. This Tegra SoC consists of four 64-bit ARMv8 Cortex-A57 cores, two custom-made Nvidia Denver cores that use the 64-bit ARMv8 instruction set [8], a GPU unit and other components. These Denver cores represent a new type of 64-bit architecture that could be used in HPC systems in the future.

The diversity of CPU types complicates significantly the choice of the best variant for a particular HPC system. The first criterion is certainly the time-to-solution of a given computational task or a set of different tasks that represents an envisaged workload of a system under development.

Another criterion is the energy efficiency of an HPC system. Energy efficiency becomes one of the most important concerns for the HPC development today and will remain in foreseeable future [9].

The need for clear guiding principles stimulates the development of models for HPC systems performance prediction. However, the capabilities of the idealized models are limited by the complexity of real-life applications. That is why the empirical benchmarks of the real-life examples serve as a complimentary tool for the co-design and optimization of software-hardware combinations.

In this work, we present the efficiency analysis of a limited but representative list of modern Intel, AMD and Nvidia 64-bit systems using a typical VASP workload example.

2 Related Work

HPC systems are notorious for operating at a small fraction of their peak performance and the deployment of multi-core and multi-socket compute nodes further complicates performance optimization. Many attempts have been made to develop a more or less universal framework for algorithms optimization that takes into account essential properties of the hardware (see e.g. [10,11]). The recent work of Stanisic et al. [12] emphasizes many pitfalls encountered when trying to characterize both the network and the memory performance of modern machines.

The increase of power consumption and heat generation of computing platforms is a very significant problem. Measurement and presentation of the results of performance tests of parallel computer systems become more and more often evidence-based [13], including the measurement of energy consumption, which is crucial for the development of exascale supercomputers [14].

The work of Calore et al. [15] discloses some aspects of relations between power consumption and performance using small Nvidia Jetson TK1 minicomputer running the Lattice Boltzmann method algorithms. An energy-aware task management mechanism for the MPDATA algorithms on multicore CPUs was proposed by Rojek et al. [16].

Minicomputers serve as prototypes that allow benchmarking novel technologies without spending a significant budget for purchasing full-scale prototypes. Moreover, minicomputers are considered as a perspective elements of energy-efficient HPC systems [17].

Our previous results on energy consumption for minicomputers running classical MD benchmarks were published previously for Odroid C1 [18] and Nvidia Jetson TK1 and TX1 [19, 20].

3 Hardware and Software

In this work, we consider several Intel Xeon CPUs, the novel AMD Ryzen processor and the novel Nvidia Tegra "Parker" SoC and compare the results with the data [21] for the IBM Power 7. The features of the systems considered are summarized in Table 1.

Table 1. The main features of the systems considered

CPU type	N_{cores}	$N_{mem.ch.}$	LLC (Mb)	CPU_{freq} (GHz)	$DRAM_{freq}$ (MHz)
Single socket, Intel X99 chipset					
Xeon E5-2620v4	8	4	20	2.1	2133
Xeon E5-2660v4	14	4	35	2.0	2400
Single socket, AMD B350 chipset					
Ryzen 1800X	8	2	16	3.6	2400
Nvidia Jetson TX2 (2 Denver + 4 Cortex-A57 cores)					
Tegra "Parker"	$2+4$	4 (32-bit)	$2+2$	2.0	1866
Dual socket, Intel C602 chipset (the MVS10P cluster)					
Xeon E5-2690	8	4	20	2.9	1600
Dual socket, Intel C612 chipset (the MVS1P5 cluster)					
Xeon E5-2697v3	14	4	35	2.6	2133
Dual socket, Intel C612 chipset (the IRUS17 cluster)					
Xeon E5-2698v4	20	4	50	2.2	2400
Quad socket, IBM Power 775 (the Boreasz cluster [21])					
Power 7	8	4	32	3.83	1600

The single socket Intel Broadwell systems benchmarks are performed under Ubuntu ver. 16.04 with Linux kernel ver. 4.4.0. The single socket AMD Ryzen system is benchmarked under Ubuntu ver. 17.04 with Linux kernel ver. 4.10.0. Jetson TX2 is benchmarked under Linux4Tegra Ubuntu 16.04 LTS aarch64 with Linux kernel ver. 4.4.0.

3.1 Test Model in VASP

VASP 5.4.1 is compiled for Intel systems using Intel Fortran, Intel MPI and linked with Intel MKL for BLAS, LAPACK and FFT calls. For the AMD and Nvidia systems, gfortran ver.6.3 is used together with OpenMPI, OpenBLAS and FFTW libraries.

VASP is known to be both a memory-bound and a compute-bound code. In general, VASP execution is dominated by the back and forth FFTs (from/to the real space to/from the Fourier space), and zgemm/dgemm calls [7]. Our test model in VASP is the same as used previously for the benchmarks of the IBM 775 system [21]. The model represents a GaAs crystal consisting of 80 atoms in the supercell. The calculation protocol corresponds to the iterative electron density optimization. We use the time for the first iteration τ_{iter} during this optimization as a target parameter of the performance metric.

The choice of a particular test model has a certain influence on the benchmarking results. However, our preliminary tests of other VASP models show that the main conclusions of this study do not depend significantly on a particular model. In the future, a set of regression tests will be considered.

3.2 Power Consumption Measurement

For the single socket systems considered, the power consumption measurements are performed. For Intel and AMD systems, we use APC Back-UPS Pro BR1500G-RS and the corresponding apcupsd linux driver for digital sampling of power consumed during VASP runs.

For Jetson TX2, we use two SmartPower digital wattmeters with the integrated DC source. Each wattmeter provides voltage in the range from 3 to 5.25 V and measures the current and power consumption every 0.2 s with a nominal error of less than 0.01 V. The wattmeter shows the data on the display in real time and allows to transfer the data via USB to the PC for further analysis. Because both Jetson platforms have nominal voltage values higher than 5.25 V, we connect two SmartPower wattmeters in a sequential way to achieve higher voltage [19,20]. The nominal voltage is 19 V for Jetson TX2. However, we discover that the minicomputer can operate steadily at much lower voltages, and hence use 10.5 V.

In this way, we measure the total power consum‘ption of the CPU, the memory, the motherboard and PSU. For the evaluation of the total energy consumed during one benchmark run, we multiply the average power value during the run by the time of the first iteration τ_{iter}.

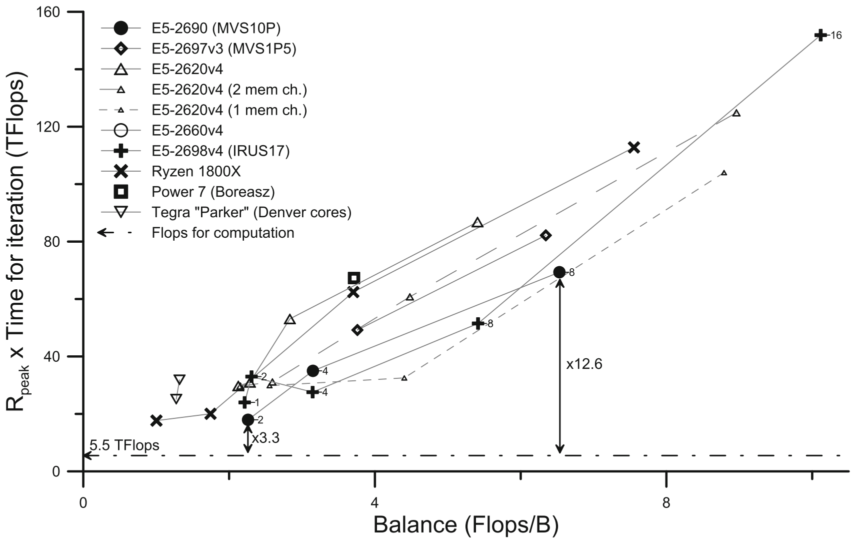


Fig. 1. The dependence of the time for the first iteration of the GaAs test on the number of cores per socket in the reduced parameters $R_{peak}\tau_{iter}$ and balance B (here R_{peak} is the total peak performance of all the core used, and the balance B corresponds to total bandwidth for a single/dual socket server.)

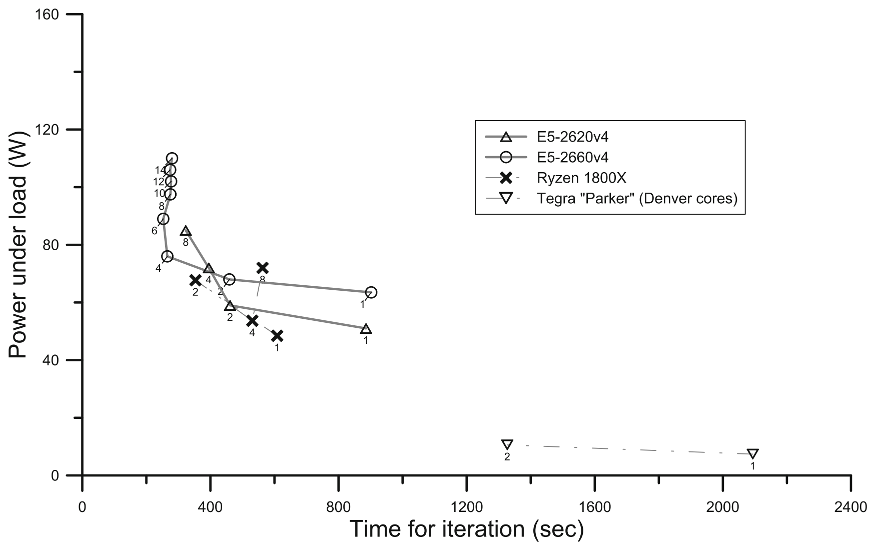
4 Results and Discussion

4.1 Computational Efficiency and the Balance Between R_{peak} and DRAM Bandwidth

The performance comparison of different CPUs resembles a comparison of “apples and oranges”. For comparison of CPUs with different frequencies and different peak numbers of Flops/cycle, it is better to use the reduced parameter of $R_{peak}\tau_{iter}$ [19, 22], where R_{peak} is the theoretical peak performance in Flops/sec. $R_{peak}\tau_{iter}$ gives the number floating-point operations that could be performed during the time τ_{iter} if the data are available in registers without any delay.

Another reduced parameter that characterizes the memory subsystem is the so-called balance B that is the ratio of R_{peak} to the CPU memory bandwidth (in this work, we measure the latter quantity using the STREAM benchmark).

We detect that for Broadwell systems considered (with 4 memory channels per socket) τ_{iter} saturates at 4 cores per socket and shows no significant decrease for higher core counts. In order to better understand the dependence on the number of memory channels, we perform tests with E5-2620v4 CPU with only 2 or 1 memory channels activated (with only 2 DIMMs or 1 DIMM installed into the motherboard). Figure 1 shows the results for different systems. In this way, we have eliminated the differences of CPUs considered in floating point performance and in the memory bandwidth.



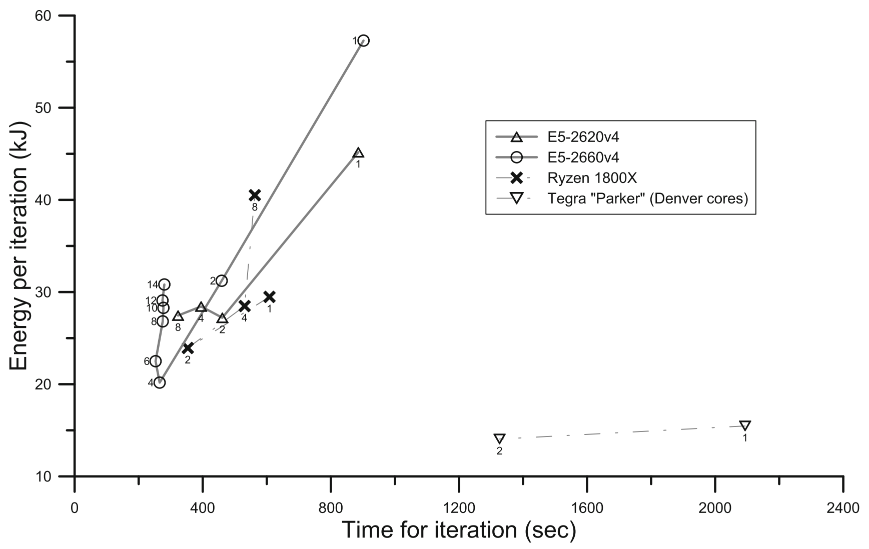


Fig. 2. The average power draw and energy consumption of the single socket systems under the VASP test model load. The number of active cores is shown near each data point.

One can see that in these reduced coordinates there is an evident common trend. The data point for the IBM Power 7 CPU is located at the same trend that suggests the low sensitivity of the results to the hardware and software differences between Intel, AMD, Nvidia and IBM systems considered.

The test model considered defines (by its choice) the total number of arithmetic operations (Flops) required for its solution N_{FP}. The increase of $R_{peak}\tau_{iter}$ (that is proportional to the number of CPU cycles) shows the increase of the overhead due to the limited memory bandwidth. More CPU cycles are required for the CPU cores involved in computations to get data from DRAM.

We have calculated the number of floating-point operations that corresponds to τ_{iter}. We used a system with Intel Core i7 640UM CPU. This CPU does not support AVX instructions and the performance counters work unambiguously. The resulting value of $N_{FP} = 5.5$ TFlops is shown at Fig. 1 as a dash-dotted horizontal line. The ratio of $R_{peak}\tau_{iter}/N_{FP}$ shows the overhead of the CPU cycles that are not deployed for computations because of the required data from DRAM are not available. As it is shown at Fig. 1, this overhead is about 3 for the cases with the lowest balance values considered. The overhead becomes about 10–20 if 8 cores per CPU are deployed for the VASP test run.

We should notice that the overall trend at Fig. 1 corresponds very well to the limiting case $R_{peak}\tau_{iter} \rightarrow N_{FP}$ when $B \rightarrow 0$.

4.2 Analysis of the Energy-to-Solution

For the single socket systems considered (see Table 1) the power consumption measurements are performed together with the VASP model test runs. The results are summarized in Fig. 2 that shows the average power and the total consumed energy as functions of τ_{iter}.

Comparing E5-2620v4 (with 8 cores in total) and E5-2660v4 (with 14 cores in total), we conclude that non-active cores do not contribute significantly to the power draw during VASP test runs.

AMD Ryzen shows a competitive level of power consumption. However, the increase of average power consumption after the transition from 1 to 2 cores for AMD Ryzen is more pronounced than for Intel Broadwell CPUs considered. The probable reason is the activation of both quad-core CPU-Complexes (CCX) of the Ryzen 1800X CPU.

In most cases, there is a minimum in energy consumption for a given CPU. This minimum is mainly connected with the reduction of τ_{iter}. Beyond this minimum, when more cores come into play, further acceleration is connected with essentially higher power draw, or there is no acceleration at all.

The most power-efficient and energy-efficient case among the x86_64 variants considered is the use of 4 cores of E5-2660v4. The reason for this advantage is the large L3 cache size E5-2660v4 in comparison with E5-2620v4. AMD Ryzen CPU shows a competitive level of energy efficiency and performance.

Benchmarks of the Jetson TX2 minicomputer using one and two Nvidia Denver cores show a much better energy efficiency. Extrapolating the results for 1 and 2 cores for higher core counts, one can say that the energy efficiency is about 2 times better. Here, we should notice, however, the differences between the motherboards and PSUs of Intel and AMD systems considered and the Jetson TX2 minicomputer. The latter was designed for an essentially lower power load and thus has a lower power overhead [17].

From the point of view of the silicon technology, there is no evident reason of the observed results on energy efficiency. Tegra "Parker" SoC is based on the 16 nm technology. The Ryzen architecture is based on 14 nm technology, as well as the Broadwell architecture (the oldest among considered).

5 Conclusions

In this work, we have considered several Intel CPUs (from Sandy Bridge, Haswell and Broadwell families), the novel AMD Ryzen CPU and Nvidia Tegra "Parker" SoC with novel 64-bit Denver cores. Moreover, we have used the data on IBM Power 7 for comparison. In all the cases, we have used the test VASP model of GaAs crystal as a benchmark tool. Power consumption measurements have been carried out.

Additionally to the variation of the CPU types, we have considered the variations in the number of active memory channels for E5-2620v4 CPU.

For comparison of different systems, we have used the reduced parameters: the time for iteration normalized by the floating point peak performance $R_{peak}\tau_{iter}$ and the balance B that is the ratio of the peak floating-point performance to the maximum sustained memory bandwidth. The benchmark results correlate with these reduced parameters quite well. This fact allows us to make several conclusions on optimal VASP performance.

For VASP, the optimal number of cores per memory channel is 1–2. Using more that 2 cores per channel provides no acceleration. Comparing different CPUs at the same level of performance, we conclude that CPUs with larger L3 cache size needs less power and consumes less energy.

The AMD Ryzen system demonstrates a competitive level of performance and energy efficiency in comparison with the Intel Broadwell systems. The benchmarks of the Jetson TX2 system with Nvidia Denver cores show a very promising level of energy efficiency that is about 2 times better than the results for Broadwell and Ryzen systems.

Acknowledgment. The authors are grateful to Dr. Maciej Cytowski and Dr. Jacek Peichota (ICM, University of Warsaw) for the data on the VASP benchmark [21].

The authors acknowledge Joint Supercomputer Centre of Russian Academy of Sciences (http://www.jscc.ru) and Shared Resource Center "Far Eastern Computing Resource" IACP FEB RAS (http://cc.dvo.ru) for the access to the supercomputers MVS10P, MVS1P5 and IRUS17.

The work was supported by the grant No. 14-50-00124 of the Russian Science Foundation. A part of the equipment used in this work was purchased with the financial support of MIPT and HSE.

References

1. Kresse, G., Hafner, J.: Ab initio molecular dynamics for liquid metals. Phys. Rev. B **47**, 558–561 (1993). http://link.aps.org/doi/10.1103/PhysRevB.47.558
2. Kresse, G., Hafner, J.: Ab initio molecular-dynamics simulation of the liquid-metal-amorphous-semiconductor transition in germanium. Phys. Rev. B **49**, 14251–14269 (1994). http://link.aps.org/doi/10.1103/PhysRevB.49.14251
3. Kresse, G., Furthmuller, J.: Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. Computat. Mater. Sci. **6**(1), 15–50 (1996). https://doi.org/10.1016/0927-0256(96)00008-0. http://www.sciencedirect.com/science/article/pii/0927025696000080
4. Kresse, G., Furthmüller, J.: Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. Phys. Rev. B **54**, 11169–11186 (1996). http://link.aps.org/doi/10.1103/PhysRevB.54.11169
5. Bethune, I.: Ab initio molecular dynamics. In: Introduction to Molecular Dynamics on ARCHER (2015). https://www.archer.ac.uk/training/course-material/2015/06/MolDy_Strath/AbInitioMD.pdf
6. Hutchinson, M.: VASP on GPUs. When and how. In: GPU Technology Theater, SC 2015 (2015). http://images.nvidia.com/events/sc15/pdfs/SC5107-vasp-gpus.pdf
7. Zhao, Z., Marsman, M.: Estimating the performance impact of the MCDRAM on KNL using dual-socket Ivy Bridge nodes on Cray XC30. In: Proceedings of Cray User Group – 2016 (2016). https://cug.org/proceedings/cug2016_proceedings/includes/files/pap111.pdf
8. Boggs, D., Brown, G., Tuck, N., Venkatraman, K.S.: Denver: Nvidia's first 64-bit arm processor. IEEE Micro **35**(2), 46–55 (2015). https://doi.org/10.1109/MM.2015.12
9. Kogge, P., Shalf, J.: Exascale computing trends: adjusting to the "new normal" for computer architecture. Comput. Sci. Eng. **15**(6), 16–26 (2013). https://doi.org/10.1109/MCSE.2013.95
10. Burtscher, M., Kim, B.D., Diamond, J., McCalpin, J., Koesterke, L., Browne, J.: Perfexpert: an easy-to-use performance diagnosis tool for HPC applications. In: Proceedings of 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC (2010). https://doi.org/10.1109/SC.2010.41
11. Rane, A., Browne, J.: Enhancing performance optimization of multicore/multichip nodes with data structure metrics. ACM Trans. Parallel Comput. **1**(1), 3:1–3:20 (2014). http://doi.acm.org/10.1145/2588788
12. Stanisic, L., Mello Schnorr, L.C., Degomme, A., Heinrich, F.C., Legrand, A., Videau, B.: Characterizing the performance of modern architectures through opaque benchmarks: pitfalls learned the hard way. In: IPDPS 2017–31st IEEE International Parallel & Distributed Processing Symposium (RepPar Workshop), Orlando, USA (2017). https://hal.inria.fr/hal-01470399
13. Hoefler, T., Belli, R.: Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, pp. 73:1–73:12. ACM, New York (2015). https://doi.org/10.1145/2807591.2807644
14. Scogland, T., Azose, J., Rohr, D., Rivoire, S., Bates, N., Hackenberg, D.: Node variability in large-scale power measurements: perspectives from the Green500, Top500 and EEHPCWG. In: Proceedings of International Conference for High

Performance Computing, Networking, Storage and Analysis, SC 2015, pp. 74:1–74:11. ACM, New York (2015). http://doi.acm.org/10.1145/2807591.2807653

15. Calore, E., Schifano, S.F., Tripiccione, R.: Energy-performance tradeoffs for HPC applications on low power processors. In: Hunold, S., et al. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 737–748. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_59
16. Rojek, K., Ilic, A., Wyrzykowski, R., Sousa, L.: Energy-aware mechanism for stencil-based MPDATA algorithm with constraints. Concurr. Comput.: Pract. Exp. e4016-n/a (2016). http://dx.doi.org/10.1002/cpe.4016.Cpe.4016
17. Luijten, R.P., Cossale, M., Clauberg, R., Doering, A.: Power measurements and cooling of the DOME 28nm 1.8GHz 24-thread ppc64 μServer compute node. In: 2015 International Conference on IC Design Technology (ICICDT), pp. 1–4 (2015). https://doi.org/10.1109/ICICDT.2015.7165919
18. Nikolskiy, V., Stegailov, V.: Floating-point performance of ARM cores and their efficiency in classical molecular dynamics. J. Phys.: Conf. Ser. **681**(1), 012,049 (2016). http://stacks.iop.org/1742-6596/681/i=1/a=012049
19. Nikolskiy, V.P., Stegailov, V.V., Vecher, V.S.: Efficiency of the Tegra K1 and X1 systems-on-chip for classical molecular dynamics. In: 2016 International Conference on High Performance Computing Simulation (HPCS), pp. 682–689 (2016). https://doi.org/10.1109/HPCSim.2016.7568401
20. Vecher, V., Nikolskii, V., Stegailov, V.: GPU-accelerated molecular dynamics: energy consumption and performance. In: Voevodin, V., Sobolev, S. (eds.) RuSCDays 2016. CCIS, vol. 687, pp. 78–90. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-55669-7_7
21. Cytowski, M.: Best Practice Guide – IBM Power 775. PRACE, November 2013. http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-IBM-Power-775.pdf
22. Stegailov, V.V., Orekhov, N.D., Smirnov, G.S.: HPC hardware efficiency for quantum and classical molecular dynamics. In: Malyshkin, V. (ed.) PaCT 2015. LNCS, vol. 9251, pp. 469–473. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21909-7_45

GPU Power Modeling of HPC Applications for the Simulation of Heterogeneous Clouds

Antonios T. Makaratzis[1], Malik M. Khan[2], Konstantinos M. Giannoutakis[1](✉), Anne C. Elster[2], and Dimitrios Tzovaras[1]

[1] Information Technologies Institute, Centre for Research and Technology Hellas, 57001 Thessaloniki, Greece
{antomaka,kgiannou,Dimitrios.Tzovaras}@iti.gr

[2] Norwegian University of Science and Technology, Trondheim, Norway
{malikmk,elster}@ntnu.no

Abstract. Hardware accelerators have been widely used in the scientific community, as the gain in the performance of HPC applications is significant. Hardware accelerators have been used in cloud computing as well, though existing cloud simulation frameworks do not support modeling and simulation of such hardware. Models for the estimation of the power consumption of accelerators have been proposed by many researchers, but they require large number of inputs and computations, making them unsuitable for hyper scale simulations. In previous work, a generic model for the estimation of the power consumption of accelerators has been proposed, that can be combined with generic CPU power models suitable for integration in hyper scale simulation environments. This paper extends this work by providing models for the energy consumption of GPUs and CPU-GPU pairs, that are experimentally validated with the use of different GPU hardware models and GPU intensive applications. The relative error between the actual and the estimated energy consumption is low, thus the proposed models provide accurate estimations and can be efficiently integrated into cloud simulation frameworks.

Keywords: Power consumption modeling
Energy consumption modeling · Accelerators
High Performance Computing · Graphics Processing Units
Central processing units · Heterogeneous clouds

1 Introduction

The power consumption of cloud resources has been investigated by many researchers during the last years, due to the rapid increase of the energy consumed by massive hardware infrastructures. For simulating the energy consumption of cloud data centres, various energy aware simulators have been developed such as CloudSim [4], GreenCloud [8], DCSIM [16], iCanCloud [12], SimGrid

[13] and DCworms [5,9]. Hardware accelerators, such as Graphics Processing Units (GPUs), have been widely used in High Performance Computing (HPC), and recently have been adopted by cloud providers. Currently, no cloud simulation framework consider heterogeneity in terms of hardware accelerators in the computing nodes.

Cloud simulators tend to model the hardware infrastructure with simple generalized models that require small number of inputs and computations, in order to keep the computational complexity low. This is essential considering the large number of hardware resources that have to be simulated, especially when clouds of thousands or millions of servers are simulated. Simple models for CPU servers have already been proposed, i.e. linear functions of CPU utilization and its power consumption, and have been widely used in cloud simulators such as CloudSim and DCSIM. Models proposed by individual research efforts for hardware accelerators such as GPUs, are not suitable for such simulation platforms, since they are hardware/application specific and require large number of inputs, [7,11,15,17]. General models for hardware accelerators with simple inputs are recommended in cloud simulations of large data centers.

A generic power consumption model for accelerators was proposed recently in [6], which targets HPC cloud environments, that can be combined with existing CPU power models of cloud simulation frameworks. This model targets hyper scale cloud simulation environments, as it requires small number of inputs and computations. This work extends this model by proposing energy models for GPUs and computing nodes of CPU-GPU pairs, while it provides experiments for the evaluation of the proposed models.

In Sect. 2, related work on GPU power modeling is discussed. The proposed GPU power models are presented in Sect. 3, while experiments for the evaluation of the models are given in Sect. 4. Finally, conclusions and suggestions for future research are presented in Sect. 5.

2 Related Work

Many research efforts have been devoted aiming at modeling of the power consumption of GPUs. Nagasaka et al. proposed a power model for GPUs that uses performance counters collected from the runtime profiler, [11]. The proposed power model uses linear regression that models the GPU power consumption by assuming linearity between the power consumption of the GPU and the performance counter values. Song et al. [15], proposed a model of power-performance efficiency on GPU architectures, that computes the total energy consumption of GPU-based clusters. The model uses measurements on performance counters, which are used for the training of a back-propagation Artificial Neural Network in order to ease portability to different hardware architectures. The use of performance counters for the modeling of the GPU power consumption introduces the following limitations: (i) the model can estimate the average power of a given kernel but not the instantaneous power consumption at arbitrary timings, (ii) the number of simultaneously monitored counters is limited to four, thus each kernel needs to be executed multiple times in order to collect all the counter values, (iii) each counter records the number of specific events only on a single SM instead of the whole GPU, which makes the model effective only when the load is balanced between all the SMs of the GPU, and (iv) some important events in CUDA, such as data reads from DRAM through texture hardware, are not monitored by the performance counters, thus the models cannot accurately estimate the GPU power consumption. Finally, these approaches requires a considerable number of inputs/measurements regarding the specific GPU hardware and the type of application.

A GPU power model was presented by Hong and Kim in [7], that can predict the power consumption of GPGPU workloads by using empirical power consumption values of the GPU components and instruction mixture information of the application (access rates of the application in each architectural component). More specifically, the maximum power consumption of each architectural element (floating point unit, register file, ALU, etc.) is measured, and by using the access rates of the application in each GPU element, the energy consumption is computed. Such information is hard to be obtained for different types of applications and hardware types of GPUs. Xie et al. [17], presented a similar power model based on GPU native instructions. More detailed, the energy requirements of each native GPU instruction is measured, thus the energy of an application is computed if the number of each GPU instruction of the application is given. Thus, the model requires energy measurements of each instruction on the architectural element of the GPU hardware model, and the number/types of the instructions that the application is executing on each element. The authors state that the error of the model does not exceed 15%.

Recently, Sirbu et al. [14], proposed a power model for predicting the power consumption of future jobs in a hybrid CPU-GPU-MIC system. In this model, the prediction problem is formulated as a regression task, where the given feature values are divided into sub-problems for each hardware type (CPU, GPU, MIC). In each individual problem, Support vector Regression is used, while the total power is obtained by summing the individual power consumptions. The model is trained by using data from the user's history, and can achieve high performance when enough history data are available. Barik et al. [2], presented an energy aware scheduler for optimizing the power usage on integrated CPU-GPU systems. This approach is based on power models that are computed in order to characterize the power consumption of hardware for different kinds of workload. These efforts contribute significantly on the energy efficiency of heterogeneous resources, though do not give generic power consumption models that can be integrated in cloud simulation frameworks.

Concluding, the available GPU power models of the literature use detailed power measurements on the architectural components of the GPUs, and require detailed information on the type and the number of instructions that are executed by the application. Such approaches are not suitable for integration in cloud simulation environments, due to the large number of required inputs for each type of GPU and each type of application. More generic with less computational complexity models are required in order to be able to scale up to millions of hardware cloud nodes. The model proposed in [6] is examined in this work, that only requires the maximum and the minimum power consumption values that the application consumes on the GPU, as well as the percentage of the execution time that the application utilize the GPU. This information can be easily obtained and the proposed model can be combined with existed CPU power models for computing the total amount of the power consumption of heterogeneous nodes.

3 Energy Modeling of HPC Heterogeneous Resources

Generic models for estimating the power consumption of CPU servers, have been widely used in cloud simulators, [10]. The CPU power models that have been proposed in CloudSim are the following, [3]: linear ($P_{cpu}(u) = P_{min} + (P_{max} - P_{min})u$), square ($P_{cpu}(u) = P_{min} + (P_{max} - P_{min})u^2$), cubic ($P_{cpu}(u) = P_{min} + (P_{max} - P_{min})u^3$) and square root ($P_{cpu}(u) = P_{min} + (P_{max} - P_{min})\sqrt{u}$), where $u \in [0, 1]$, is the CPU utilization and P_{min}, P_{max} are the CPU's power consumption in idle and max state respectively. An additional model has been proposed, that applies linear interpolation on real measured power consumption values, obtained from SPEC [1].

Therefore, the energy consumption of an application that is executed on a CPU server for the time period $[t_1, t_2]$, can be computed as follows:

$$E_{cpu} = \int_{t_1}^{t_2} P_{cpu}(u(t))dt. \tag{1}$$

By considering constant CPU utilization, i.e. the mean utilization of an application is available, the energy consumption of the application on the CPU server can be computed as follows:

$$E_{cpu} = (t_2 - t_1) \cdot P_{cpu}(u_{mean}). \tag{2}$$

For modeling the GPU power consumption, it can be assumed that HPC applications use the full potential of the compute capabilities of GPUs, when the application run on the GPU. Thus, a binary model can be utilized, [6]:

$$P_{gpu}(\rho) = (1 - \rho) \cdot P_{gpu_{min}} + \rho \cdot P_{gpu_{max}}, \tag{3}$$

where $P_{gpu_{min}}, P_{gpu_{max}}$ are the minimum and the maximum GPU power consumption values that the application can consume, while ρ is the percentage of the application that is parallelized on the GPU. When the full potential of the GPU is used, $\rho(t) = 1$ when the GPU is utilized and $\rho(t) = 0$ when the GPU is idle. The model can be extended further to support applications that do not use the full potential of the GPU by setting $\rho(t) < 1$ when the GPU is utilized. In that case, the Eq. (3) is extended to the following form:

$$P_{gpu}(\rho) = (1 - \rho') \cdot P_{gpu_{min}} + \rho \cdot P_{gpu_{max}}, \tag{4}$$

where $\rho' = 0$ when $\rho = 0$, and $\rho' = 1$ when $\rho > 0$. The energy consumption that is consumed on the GPU during the interval $[t_1, t_2]$, can be computed as follows:

$$E_{gpu} = \int_{t_1}^{t_2} P_{gpu}(\rho(t))dt, \tag{5}$$

where $\rho(t) = 1$ when the GPU is utilized and $\rho(t) = 0$ when the GPU is idle. The mean value of the parameter ρ can be computed as the ratio of the time that the GPU is utilized over the total execution time of the application. Using the mean value of ρ (denoted as ρ_{mean}), the GPU energy consumption of the application can be computed as follows:

$$E_{gpu} = (t_2 - t_1) \cdot P_{gpu}(\rho_{mean}). \tag{6}$$

The combined power consumption of a CPU-GPU heterogeneous node can be computed as follows, [6]:

$$P_{cpu-gpu}(u, \rho) = P_{cpu}(u) + P_{gpu}(\rho), \tag{7}$$

where $P_{cpu}(u)$ can be any CPU power consumption model, while $P_{gpu}(\rho)$ is the GPU power model of Eq. (3). The energy consumption of an application that is executed on a CPU-GPU heterogeneous node is then:

$$E_{cpu-gpu} = \int_{t_1}^{t_2} P_{cpu-gpu}(u(t), \rho(t))dt. \tag{8}$$

By using the mean values of u and ρ, the energy consumption can be derived as:

$$E_{cpu-gpu} = (t_2 - t_1) \cdot P_{cpu-gpu}(u_{mean}, \rho_{mean}). \tag{9}$$

The proposed GPU power model does not require any detailed information on the GPU's hardware or the instruction mixture information of the application. The only required parameters are the maximum and minimum power values that the application consume on the GPU, and the percentage of the application that is run on the GPU. It is noted that various phenomena, such as memory bottlenecks, can be tuned through the ρ parameter (i.e., the increment/decrement that need to be applied on the ρ parameter when memory bound applications are considered, can be computed through experimentation). Thus, any application characteristic can be modeled through the ρ parameter. In the next section the proposed theoretical model is evaluated, for predicting the total energy consumption of various GPU intensive applications for various GPU hardware models.

4 Experimentation

4.1 Evaluation of the GPU Power Model

The proposed GPU power model was tested for predicting the energy consumption of three GPU intensive applications[1] and on three different GPU hardware

[1] Applications retrieved from http://docs.nvidia.com/cuda/cuda-samples/index.html#simple.

models. The same instances of applications were executed on all GPUs, which where the following: a dense matrix multiplication (17920 × 17920), an FFT convolution (2D, 10240 × 10240) and an NBody simulation (120832 bodies, 10 iterations). The hardware models of the GPUs were the following: a "GeForce GTX 980", a "Tesla K20C" and a "Tesla P100".

The power and utilization measurements of the GPUs were collected using the *nvidia-smi* utility, which periodically provides the power consumption and the utilization of the selected GPU. In order to collect measurements for a longer time period, the applications were executed multiple times successively in each experiment. These power consumption measurements were used for computing the actual energy consumption of each experiment.

In Table 1 the estimation of the mean GPU power consumption of each experiment is depicted, using the mean value of ρ (ρ_{mean}) in Eq. (3). The $P_{gpu_{min}}$ and $P_{gpu_{max}}$ parameters (minimum and maximum GPU power consumption that each application consumes) were obtained from the recorded power values of each experiment. The mean value of the parameter ρ (ratio of the time that the GPU is utilized over the total execution time of the application) was computed from the GPU utilization measurements of each application as $\rho_{mean} = \frac{ExecutionTime(gpuUtil > 75\%)}{ExecutionTime(total)}$.

Table 1. Inputs of the power model of Eq. (3) and estimation of the mean power consumption of each experiment in Watts.

Application	GPU model	$P_{gpu_{min}}$	$P_{gpu_{max}}$	ρ_{mean}	P_{gpu}
NBodySim	GTX980	44.9900	194.9700	0.7662	159.9097
	K20C	48.6200	151.2500	0.8592	136.7951
	P100	31.4200	198.2500	0.9629	192.0607
MatrixMul	GTX980	18.9900	182.0900	0.8872	163.6975
	K20C	15.1300	176.8900	0.4794	92.6758
	P100	29.9500	250.0800	0.7755	171.4621
FFTConv	GTX980	54.0400	105.7400	0.0808	58.2178
	K20C	46.3300	125.1500	0.1140	55.3182
	P100	32.6200	182.1600	0.0522	32.2992

In Table 2, the actual GPU energy consumption for each experiment (in Wh) and the GPU energy consumption (in Wh) that was estimated with Eq. (6), are presented. The last column depicts the error of the estimation for each experiment. The error was computed with the following formula:

$$Error = \frac{|ActualEnergy - EstimatedEnergy|}{ActualEnergy}. \quad (10)$$

The GPU power consumption over time, for the three applications and the three GPU hardware implementations is depicted in Fig. 1, where the measured

Table 2. Errors in the estimation of the energy consumption.

Application	GPU model	Actual GPU energy	Estimated GPU energy	Error
NBodySim	GTX980	3.3662	3.4203	1.61%
	K20C	5.3372	5.3958	1.10%
	P100	28.3666	29.6093	4.38%
MatrixMul	GTX980	47.5562	48.7910	2.60%
	K20C	81.9927	82.4300	0.53%
	P100	147.4077	155.1644	5.47%
FFTConv	GTX980	1.6217	1.6010	1.28%
	K20C	1.6540	1.7517	5.91%
	P100	4.6581	4.5367	2.61%

values are presented together with the model estimation over time, using Eq. (3) for the instantaneous value of ρ.

4.2 Evaluation of the CPU-GPU Power Model

Additional experiments were performed for the estimation of the power consumption of a computing node, consisting of a pair of CPU and GPU. The experiments were conducted on the "Tesla P100", coupled with an "Intel® Xeon® CPU E5-2609 v3" processor running at 1.90 GHz. The power and utilization measurements of the computing node were collected with the command *ipmi-oem Dell get-instantaneous-power-consumption-data*, which periodically provides the instantaneous CPU utilization and power consumption of the computing node.

The total energy consumption (in Wh) for the three applications is given in Table 3 (using the inputs of Table 1). Additionally, the corresponding errors in the estimation of the energy (Eq. (9)) for using each of the generic CPU power models are given in Table 4.

It can be observed that the CPU-GPU power model achieves accurate estimations of the total energy consumption of the heterogeneous node, where the Square and the Cubic CPU power models proved less accurate than the Linear and the Square Root models. Matrix Multiplication also revealed increased errors compared to the other applications, especially when the Square and the Cubic power models were used. This error occurs due to the heavy data transmits that take place during the computations, which result in high frequency increments and decrements of the power consumption.

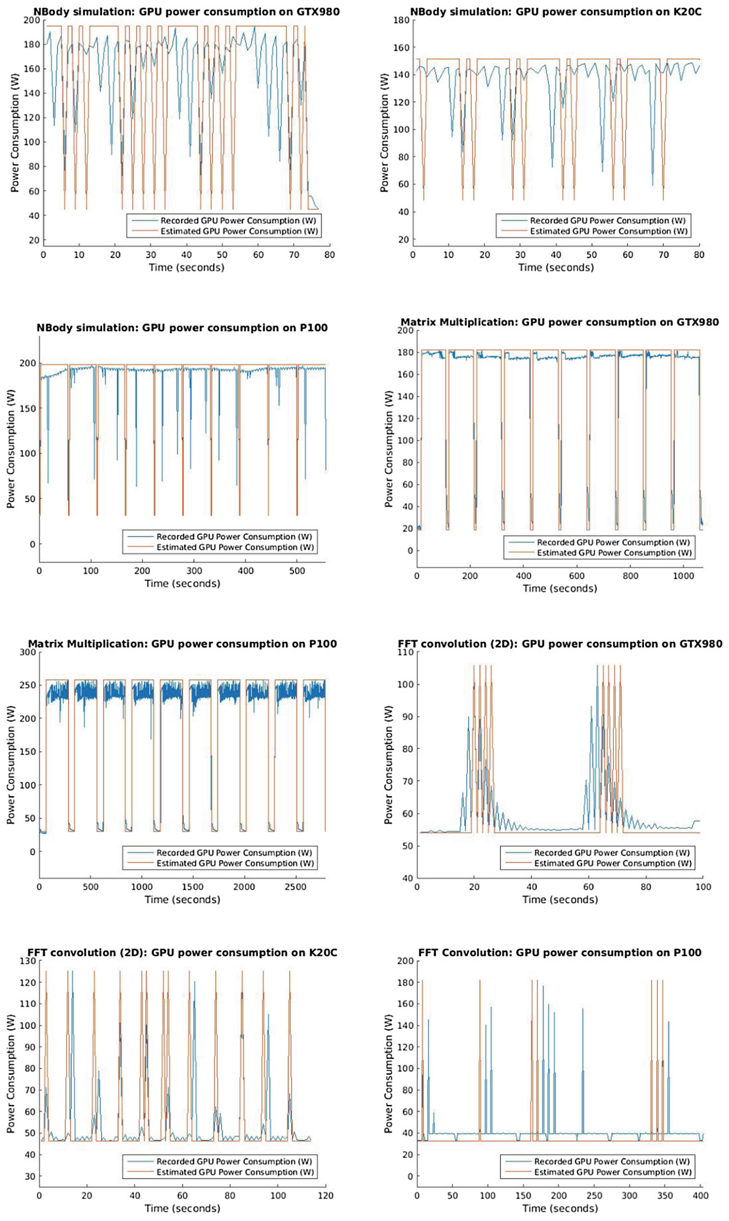


Fig. 1. GPU power consumption over time for the three applications. The blue line depicts the recorded power consumption, while the orange line depicts the GPU power model estimation (Eq. (3)). (Color figure online)

Table 3. Inputs of the CPU power model and total estimated and actual energy consumption for the three applications.

Application	$P_{cpu_{min}}$	$P_{cpu_{max}}$	u_{mean}	Linear	Square	Cubic	Sqrt	Actual
NBodySim	142.5800	257.7500	0.0828	53.0612	51.7123	51.6005	56.7007	54.1936
MatrixMul	134.0500	280.9200	0.0827	268.7223	260.0935	259.3802	292.0313	282.1008
FFTConv	137.3800	182.3800	0.0823	20.3695	19.9880	19.9566	21.4028	20.2350

Table 4. Errors in the estimation of the energy for each CPU power model.

Application	Linear	Square	Cubic	Sqrt
NBodySim	2.09%	4.58%	4.78%	4.63%
MatrixMul	4.74%	7.80%	8.05%	3.52%
FFTConv	0.66%	1.22%	1.38%	0.44%

5 Conclusions and Future Work

GPUs have been widely used as accelerators of HPC applications during the last years. Many models have been developed for the energy modeling of GPUs, though these models require a significant number of inputs and complex computations, which can hardly be integrated into cloud simulation platforms. In this paper, a generic power and energy model for GPUs was presented and evaluated. The proposed model can be easily integrated in cloud simulation platforms, as it requires substantially less number of inputs and number of computations, while it can be combined with generic CPU power models.

The experimentation on various GPU hardware models revealed that the proposed model can derive accurate estimations of the energy consumption of HPC applications. The model was also combined with generic CPU power models for the estimation of the total energy consumption of a computing node, where the errors of the estimation remained low (from 0.44% to 8.05%). Since the proposed model is parameterized (by tuning parameter ρ), it is expected to achieve accurate results also on different software/hardware configurations.

Future work will be focused on the improvement of the generic CPU power models and on the development of new and more accurate CPU-GPU power models. Additionally, evaluating the proposed model on other accelerator hardware types, such as Many Integrated Cores (MICs) and Field-Programmable Gate Arrays (FPGAs), will be investigated.

Acknowledgment. This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through CloudLightning project under Grant Agreement No. 643946.

References

1. Server power and performance characteristics (spec) (2008). http://www.spec.org/power_ssj2008/
2. Barik, R., Farooqui, N., Lewis, B.T., Hu, C., Shpeisman, T.: A black-box approach to energy-aware scheduling on integrated CPU-GPU systems. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization, pp. 70–81. ACM, New York (2016). https://doi.org/10.1145/2854038.2854052
3. Beloglazov, A., Buyya, R.: Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. Concurr. Comput.: Pract. Exper. **24**(13), 1397–1420 (2012). https://doi.org/10.1002/cpe.1867
4. 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. Softw. Pract. Exper. **41**(1), 23–50 (2011). https://doi.org/10.1002/spe.995
5. Fontoura Cupertino, L., Da Costa, G., Oleksiak, A., PiąTek, W., Pierson, J.M., Salom, J., Siso, L., Stolf, P., Sun, H., Zilio, T.: Energy-efficient, thermal-aware modeling and simulation of datacenters: the CoolEmAll approach and evaluation results. Ad Hoc Netw. J. **25**(B), 535–553 (2015). https://doi.org/10.1016/j.adhoc.2014.11.002
6. Giannoutakis, K.M., Makaratzis, A.T., Tzovaras, D., Filelis-Papadopoulos, C.K., Gravvanis, G.A.: On the power consumption modeling for the simulation of heterogeneous HPC clouds. In: Proceedings of the 1st International Workshop on Next Generation of Cloud Architectures, CloudNG 2017, pp. 1:1–1:6. ACM, New York (2017). https://doi.org/10.1145/3068126.3068127
7. Hong, S., Kim, H.: An integrated GPU power and performance model. SIGARCH Comput. Archit. News **38**(3), 280–289 (2010). https://doi.org/10.1145/1816038.1815998
8. Kliazovich, D., Bouvry, P., Khan, S.U.: Greencloud: a packet-level simulator of energy-aware cloud computing data centers. J. Supercomput. **62**(3), 1263–1283 (2012). https://doi.org/10.1007/s11227-010-0504-1
9. Kurowski, K., Oleksiak, A., Piatek, W., Piontek, T., Przybyszewski, A.W., Weglarz, J.: DCworms - a tool for simulation of energy efficiency in distributed computing infrastructures. Simul. Model. Practice Theory **39**, 135–151 (2013)
10. Makaratzis, A.T., Giannoutakis, K.M., Tzovaras, D.: Energy modeling in cloud simulation frameworks. Future Gener. Comput. Syst. (2017). https://doi.org/10.1016/j.future.2017.06.016
11. Nagasaka, H., Maruyama, N., Nukada, A., Endo, T., Matsuoka, S.: Statistical power modeling of GPU kernels using performance counters. In: International Conference on Green Computing, pp. 115–122, August 2010
12. Núñez, A., Vázquez-Poletti, J.L., Caminero, A.C., Castañé, G.G., Carretero, J., Llorente, I.M.: iCanCloud: a flexible and scalable cloud infrastructure simulator. J. Grid Comput. **10**(1), 185–209 (2012). https://doi.org/10.1007/s10723-012-9208-5
13. Pouilloux, L., Hirofuchi, T., Lebre, A.: SimGrid VM: virtual machine support for a simulation framework of distributed systems. IEEE Trans. Cloud Comput. (2015). https://hal.inria.fr/hal-01197274
14. Sîrbu, A., Babaoglu, O.: Power consumption modeling and prediction in a hybrid CPU-GPU-MIC supercomputer. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 117–130. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_9

15. Song, S., Su, C., Rountree, B., Cameron, K.W.: A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 673–686 (2013)
16. Tighe, M., Keller, G., Bauer, M., Lutfiyya, H.: DCSim: a data centre simulation tool for evaluating dynamic virtualized resource management. In: 2012 8th International Conference on Network and Service Management (CNSM) and 2012 Workshop on Systems Virtualization Management (SVM), pp. 385–392, October 2012
17. Xie, Q., Huang, T., Zou, Z., Xia, L., Zhu, Y., Jiang, J.: An accurate power model for GPU processors. In: 2012 7th International Conference on Computing and Convergence Technology (ICCCT), pp. 1141–1146, December 2012

Bi-cluster Parallel Computing in Bioinformatics – Performance and Eco-Efficiency

Paweł Foszner(✉) and Przemysław Skurowski

Institute of Informatics, Silesian University of Technology, Gliwice, Poland
{pawel.foszner,przemyslaw.skurowski}@polsl.pl
http://inf.polsl.pl

Abstract. The paper discusses the selected bi-clustering algorithms in terms of energy efficiency. We demonstrate the need for the power aware software development, elaborate bi-clustering methods and applications, and describe the experimental computational cluster with a custom built energy measurement instrumentation.

Keywords: Bioinformatics computation · Green computing
Bi-clustering

1 Introduction

Parallel processing plays pivotal role in the bioinformatics data processing. Because of tremendous amount of data and complex processing methods, the parallel systems makes feasible that computations. With the growing computing power, the more and more complex tasks can be accomplished, nevertheless there is notable cost of that capabilities in the growing energy demand for computations and cooling in the computation facilities. High performance computing (HPC) is of everyday usage in bioinformatics, it causes respective carbon footprint, that makes the HPC to be of an interest for the power saving.

The information and communication technologies (ICT) consume $\sim$5% of the world electrical energy [2] of which $\sim$20% is data center share; these numbers are growing [11]. Due to considerable share in energy consumption, power-aware ICT is an important aspect for the policy makers, it is one of the objectives in the European Commission research agenda [1]. Therefore, eco-efficiency evaluation of the algorithms and implementations seems to be important.

In the field of bioinformatics, we usually deal with data related to the list of genes or complex protein structures. These are, for example, gene expression matrices, genetic variation data, etc. Algorithms operating on these data seek to find patterns that carry relevant information. They may be expression profiles or patterns of behavior that can distinguish diseased tissue from healthy ones. Bi-clustering is a technique of data mining which has found itself particularly well in the analysis of gene expression matrices [5,21,26]. Such matrices are the

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 102–112, 2018.
https://doi.org/10.1007/978-3-319-78054-2_10

result of a series of microarray experiments. Microarrays are used to measure the expression levels of large numbers of genes simultaneously. The procedure is repeated for many tissues or under different conditions. From the combination of multiple expression vectors, we get a matrix that in one dimension consists of genes and in another dimension different tissues/patients/conditions.

Bi-clustering algorithms can be very useful for this type of data because they perform parallel clustering of both dimensions simultaneously. Whereby they seek for subgroups of genes highly correlated only to a subgroup of patients. This is an advantage over the classic approach (clustering) that loses some of the information by finding subgroups of genes of a similar profile from the perspective of all conditions or a subset of conditions correlated across the whole set of genes.

The dimensionality of biological data can make the computer short of breath. The number of dimensions ranges from several thousand to several dozens of thousands. Such an input data sizes do not even guarantee the finite time of calculations for some groups of algorithms. Due to the fact that bioinformatics is one of the most important fields of science, and in addition, every day comes with more and more data - the question arises whether we make the best use of our resources. In general, during research, the two most valuable resources are: (1) the scientist's time as a precious and non-renewable resource, and (2) the electricity that generates significant costs (financial and environmental). In our work, we will try to see usage of these resources by popular bi-clustering algorithms in typical application.

The articles is organized as follows: Sect. 2 brings overall background on the computation problem; Sect. 3 outlines the experiment – the computation methods, data, evaluation criteria, hardware, and software. Section 4 brings the experimental results and their interpretation. The work is summarized in Sect. 5.

2 Background

2.1 Green Computing in Bio-Informatics

Green computing or green IT is a whole set of manufacturing, organizational, business, and engineering efforts towards the eco-efficiency of computing [2]. During the computations, the energy is drawn, utilized, and then dissipated as a waste heat and it is proportional to the algorithm's computation time, intensity, and efficiency. Additionally, the power consumption in a computer system is related to the other factors, not directly related to the implemented algorithms – energy-efficient hardware design, system power management, software releases. The guidelines for evaluation of the program efficiency are provided in [2,23].

Most of AI techniques employed in bio-informatics are computationally intensive and operate on a massive amount of data – hundreds to thousands of objects (e.g. proteins, patients) and tens of thousands of features (e.g. genes). This makes the notable power consumption and the power-aware computing is known in bioinformatics for relatively long time [7]. Although, the motivation for the fast progress (not only) in research overcomes usually the power-efficiency of the computation. Nevertheless, proper selection of implementation or algorithm by the researcher or the HPC staff could result in both obtaining fine results in a reasonable time at reasonable cost.

Evaluation of energy efficiency in bio-informatics applications has no significant difference to the general computer science. The main discriminant of bioinformatics is in usage patterns of HPC by bio- and genomic researchers [10], who prefer easy available software tools over creating new ones. So it makes the software deployment decisions by the HPC staff even more important because these tools are highly re-used in numerous experiments.

2.2 Bi-clustering

Bi-clustering is a data mining technique that in two-dimensional matrix finds a subset of attributes from one dimensions that reveals similar behavior only on subset of attributes from second dimensions. In a very simple words bi-clustering is about finding sub-matrices in data matrix or finding a bi-cliques in bipartite graphs. Single bi-cluster consist of subset of rows and subset of columns very strongly correlated with each other. This correlation can take many forms. Rows and columns inside bi-clusters can be correlated by constant values, constant values with offset or scaling, etc. A very good survey of the bi-cluster data structures and algorithms available in literature can be found in work of Madeira, et al. [22]. Bi-clusters and its data structure is a latent information which must be extracted by the algorithm. Nevertheless, very often the choice of the appropriate method depends on the data and information that must be discovered. In order to solve this problem consensus method were developed [9,12]. The idea of these algorithms is to combine the results of different bi-clustering methods into one with better quality. And the quality of the bi-cluster itself is usually expressed by a function dependent on its data. For data with fixed values, good measures are functions that measure variance within a bi-cluster (MSR) [5]. More general measures are those based on the average correlation of each pair of rows and columns (ACV [34] and ASR [27]). An extensive survey of quality measures can be found in Orzechowski's paper [28].

In recent years interest in algorithms for bi-clustering of data has substantially grown due to many new areas of applications, e.g., text analysis [4], pattern recognition [17], signal analysis [14] bioinformatics [3].

2.3 Parallel Processing

The maximal speedup possible to achieve by means of parallel processing is described using Amdahl's law [15], which is given as follows:

$$S = T_{seq}/T_{par} = 1/\left((1-P) + P/N\right), \tag{1}$$

where: T_{seq}, T_{par} – sequential and parallel execution time, S – speedup, P – time fraction (as executed sequentially) of parallelizable part of code, $(1-P)$ – time fraction of non-parallelizable part, N – number of processing units (CPUs/cores).

The speedup according to the Amdahl's law is a theoretical value. It assumes perfect load balancing, neglects I/O and memory bandwidth, and it does not include the overhead of communication and synchronization. Just a selection of communication model – shared memory (SM) or message passing (MP) – can have notable influence even on the performance of a relatively simple data mining tasks [31]. Another issue for the computation that may affect efficiency is the computation locality [24]. For the high performance, the tasks should be well partitioned, without referencing to the data of other workers and independent on each other results. Alas, predictable load balancing is not possible in case of non-deterministic algorithms [8], which results and performance might depend on the initial data, and which is the case of many of AI methods. Starting from the initialization they iteratively minimize an objective function. Parallelism is employed using multi-start to avoid stuck in local minimum.

3 Experiment Plan

The purpose of this research is to evaluate the power efficiency of selected bi-clustering algorithms. As a final outcome, the energy in Wh (watt-hours) used by each of the implementations will be provided. The results are obtained for the small cluster, made of two off-the-shelf workstations.

The scenario includes data for which we know what structure to expect and would know what algorithm to choose. We will use data containing somatic mutation information - in which data we expect correlation based on constant values (or bi-clusters with very small variation to be exact) and we know how many bi-clusters to expect. The scenario will be divided into two parts: (1) 4 NMF algorithms (PLSA [16], LSE, K-L [21], nsK-L [29]) will be executed. Because the methods are based on searching for a local minimum - the step will be repeated 8 times, each with another initial condition. This will result in different results which, (2) using a consensus algorithm [9], will be combined into one that gives better quality. The main computational burden of the entire experiment rests on step (1). However, these are tasks that are very easy to divide and disperse between computational nodes. As a result, we expect many independent computing threads and little interaction between nodes. This scenario will be evaluated 10 times to obtain statistical values for every computation setup. We tested following configurations: (1) 1 node (4 cores), 1 worker thread; (2) 1 node (4 cores), 4 worker threads; (3) 1 node (4 cores), 8 worker threads; (4) 2 nodes ($2 * 4$ cores), 8 worker threads. If just one node works the other is turned-off.

The quality of results obtained in the bi-clustering process is evaluated with *average correlation value* (ACV $\in \langle 0, 1\rangle$), expressing inter-cluster consistency because. It is given as a maximal of average correlations R of all-versus-all n rows (objects) and m columns (attributes):

$$\mathrm{ACV}(A) = \max\left\{\frac{\sum_i \sum_j |R_{row_{ij}}| - n}{n^2 - n}, \frac{\sum_k \sum_l |R_{col_{kl}}| - m}{m^2 - m}\right\}. \tag{2}$$

3.1 Survey of Algorithms

In terms of computational complexity majority of formulations of bi-clustering problems belong to the NP-complete class, e.g., [19]. There are numerous algorithms for data bi-clustering, which use variety of different approaches, approximations and heuristics, and their running times scale differently with the data size. Published approaches can be classified into several groups, heuristic iterative searches along rows and columns of data matrices [32], iterative numerical optimization [35], using (bipartite) graph theory [33], algebraic operations on matrices including non-negative matrix factorizations [18], using statistical models of bi-clusters, likelihood maximization by using an appropriate formulation of the EM algorithm [20], using two-way clustering approach [13] or fuzzy bi-clustering techniques [25].

Non-negative Matrix Factorization. A very wide range of algorithms are algorithms based on data matrix decomposition. In such methods data matrix (A) is factorized into (usually) much smaller matrices. Such a distribution, because of the much smaller matrices is much easier to analyze, and the obtained matrices reveal previously hidden features. These algorithms are often called NMF algorithms. NMF stands for non-negative matrix factorization. Two efficient algorithms were introduced by Seung and Lee [21]. First minimize conventional least square error distance function and second generalized Kullback-Leibler divergence. Third and last from this group is algorithm that slightly modify the second approach. Author [29] introduce smoothing matrix for achieving a high degree of sparseness, and better interpretability of the results. Data matrix in this techniques is factorized into (usually) two smaller matrices:

$$A \approx WH \tag{3}$$

Finding the exact solution is computationally very difficult task. Instead, the existing solutions focus on finding local extrema of the function describing the fit of the model to the data. Following list represent most common algorithms based on non-negative matrix factorization:

– PLSA witch stands for Probabilistic Latent Semantic Analysis. Introduced by Hoffman [16], and based on maximizing log-likelihood function. For this purpose author use Expectation-Maximization (E-M) algorithm [6].
– Based on minimization of Least Square Error distance function

$$\|A - WH\|^2 = \sum_{ij}(A_{ij} - WH_{ij})^2 \tag{4}$$

– Based on minimization of Kullback-Leibler divergence

$$D(A\,||\,WH) = \sum_{ij}(A_{ij} log\frac{A_{ij}}{WH_{ij}} - A_{ij} + WH_{ij}) \tag{5}$$

– Based on minimization of non-smooth Kullback-Leibler divergence (where S is a smoothing matrix).

$$D(A\,||\,WSH) = \sum_{ij}(A_{ij} log \frac{A_{ij}}{WSH_{ij}} - A_{ij} + WSH_{ij}) \tag{6}$$

3.2 Data

The data was carefully prepared from a combination of 3 patient classes. The values represent the number of somatic mutations in a particular patient genome. We expect to find exactly 3 bi-clusters in the data and we assume that the structure of the found bi-clusters will be based on values of very low variance. This allows to significantly narrow down the spectrum of algorithms. In this case, we can focus more on finding a solution of very high quality.

Data set was taken from TCGA database. Have been retrieved all patients with one of three types of cancer - head & neck, prostate and thyroid. For each patient, we looked at theirs somatic mutations. The final data set was composed of 3 patient groups (according to cancer type). Such data matrix consist of gene – patient information where rows represent genes and columns represent patients. Data showing the relationship of these two dimensions is the number of somatic mutations of a given gene in a given patient genome. As the following numbers show - the patient classes in the selected set are very well balanced: (1) head & neck (510 patients), (2) prostate (505 patients), (3) thyroid (504 patients).

The total number of genes relevant for the above tumors is 43754. Mutation numbers for a single gene vary from 0 to 2130.

3.3 Hardware Setup

The testbed (see Fig. 1) for the considered algorithms comprises two parts – instrumentation and system under test (SUT). The instrumentation comprises a PC with controller/logger role and a custom built power measurement unit [30]. The controller records power consumption from the measurement device connected with USB/RS485. The measurement unit is sampling power consumption at 5 Hz and 1% relative error at the operation range.

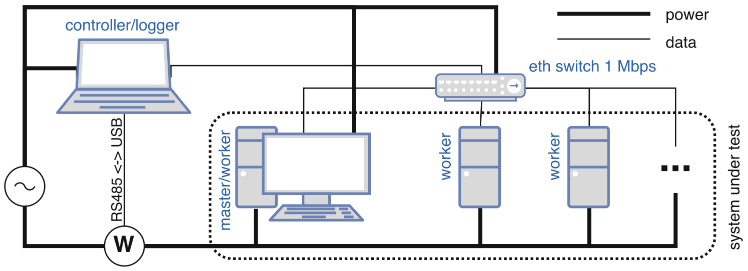


Fig. 1. Overview of a hardware setup, comprising instrumentation and SUT

The SUT is a cluster consisting of one master node (which is also a worker) and workers. For the testing purposes we set up minimalistic cluster of two off-the-shelf PCs – Lenovo ThinkStation S20 – equipped with: Xeon W3550 CPU with 4 cores at 3.07 GHz, and max. TDP 130 W; 4 GB RAM; NVidia Quadro FX580; 0.5 TB WD Caviar blue series HDD; 610 W power supply; GNU/Linux Debian 9 OS. Hosts' idle power consumption is between 46–48 W, they offer peak of 21.50 GFlops in the linpack test, being 155.81 MFlops/watt.

One of the PCs acts as a cluster master and hosts a number of worker threads as well. In order to obtain clearer results, master node display and network 1 Gb switch are considered to be out of SUT but they can be considered as a part of the cluster as well. The cluster operated in the room temperature around 20–22 °C with no air conditioning.

3.4 Software

For this work we used the AspectAnalyzer software [9], which was run on a test cluster (see Sect. 3.3). Figure 2 presents a diagram describing the fragment of the AspectAnalyzer environment that was used in this work. According to the experimental plan, only NMF algorithms and the author's own consensus algorithm were used. The software was written in C++ and mathematical calculations were made using the Armadillo library. Distributed computations was developed by authors as ad-hoc cluster and is based on direct TCP/IP communication.

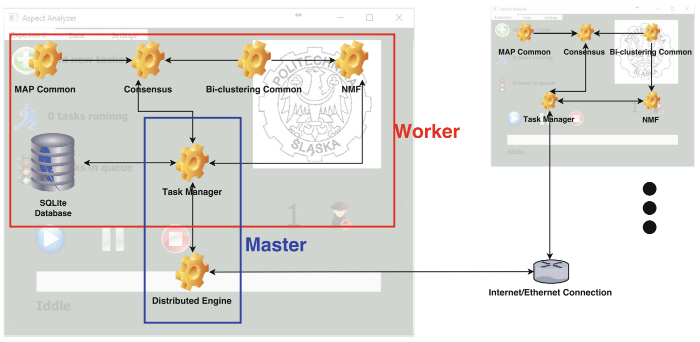


Fig. 2. Schematic diagram of a AspectAnalyzer fragment used in the experiments

4 Results and Discussion

The results are demonstrated for visual examination in Fig. 3 and they are aggregated into statistical descriptors in Tables 1, 2 and 3 representing, energy consumption, time and bi-clustering results quality estimation respectively.

The plots demonstrate a non-deterministic character of the computations. The clearly visible step in multi-core computations reveals passing form the computationally intensive error minimization stage to the consensus stage, which utilizes up to the declared number of threads but no more than a number of bi-clusters (3 in our case), so it is less intensive and causes lower power demand. Both the stages have varying durations as they are dependent on initial conditions. Another non-deterministic behavior observed in the energy profiles is valley between minimization and consensus stages. It appears due to need for synchronization need of all workers, it lasts until all realizations are completed.

For the PLSA and K-L we observe more consistent power usage than for the other two methods, it suggests these two methods are more consistent in convergence (less dependent of initialization) with just a few inconsistencies in timing and power drawn during the execution – it is reflected by small value of standard deviations in Tables 1 and 2.

A special attention should be given to the performance of the K-L method, which appears to be the most effective in terms of time and energy regardless of the hardware setup. Moreover, as it is shown in Table 3, K-L method outcomes are the best for our test data, although, it can be different for other data sets.

We observed no benefit of cluster computations in the tested scenario. Apparently, the test computations were a bit too small. The task migration cost overwhelmed the advantage of faster computations, which are observed as shorter intense power consumption during the minimization stage. Supposedly, the larger computational problem with more initializations would get gain. Another observation were 'jaggy' plots in third column, related to process contexts switching. Clearly, it had negligible impact on the overall energy consumption.

Table 1. Average energy consumption - mean and standard deviation [Wh]

	1 worker @ 4 cores		4 workers @ 4 cores		8 workers @ 4 cores		8 workers @ 8 cores	
	mean	**std dev**	**mean**	**std dev**	**mean**	**std dev**	**mean**	**std dev**
PLSA	47.7817	0.6833	30.2366	1.5892	29.3323	0.6391	44.6754	0.9258
LSE	25.3439	8.8501	16.2953	8.6965	14.5348	6.4935	27.9426	11.8277
K-L	20.2808	0.0760	10.8141	0.4462	9.4203	0.0361	15.2544	0.2253
nsK-L	56.0443	10.8252	25.2366	6.5378	25.4836	9.0289	55.8117	16.5715

Table 2. Average execution times [seconds]

	1 worker @ 4 cores		4 workers @ 4 cores		8 workers @ 4 cores		8 workers @ 8 cores	
	mean	**std dev**	**mean**	**std dev**	**mean**	**std dev**	**mean**	**std dev**
PLSA	1899.0727	21.6183	998.7361	78.4178	925.0467	22.9704	877.4406	23.4889
LSE	1004.1657	355.2545	546.6430	316.5878	476.4764	229.2346	571.1822	257.4949
K-L	799.4731	2.1240	298.6876	29.0432	238.9589	1.7275	261.0155	5.4058
nsK-L	2219.0811	428.1098	846.2043	218.4888	839.7787	307.4473	1142.7699	353.4086

Table 3. Stats of ACV values for ten experiment realizations

	PLSA		**LSE**		**K-L**		**nsK-L**	
	mean	**std dev**	**mean**	**std dev**	**mean**	**std dev**	**mean**	**std dev**
Cluster 1	0.9598	0.0263	0.9881	0.0108	0.9931	0.0046	0.9872	0.0058
Cluster 2	0.9697	0.0281	0.9881	0.0081	0.9930	0.0044	0.9904	0.0095
Cluster 3	0.9543	0.0229	0.9827	0.0062	0.9929	0.0047	0.9887	0.0070

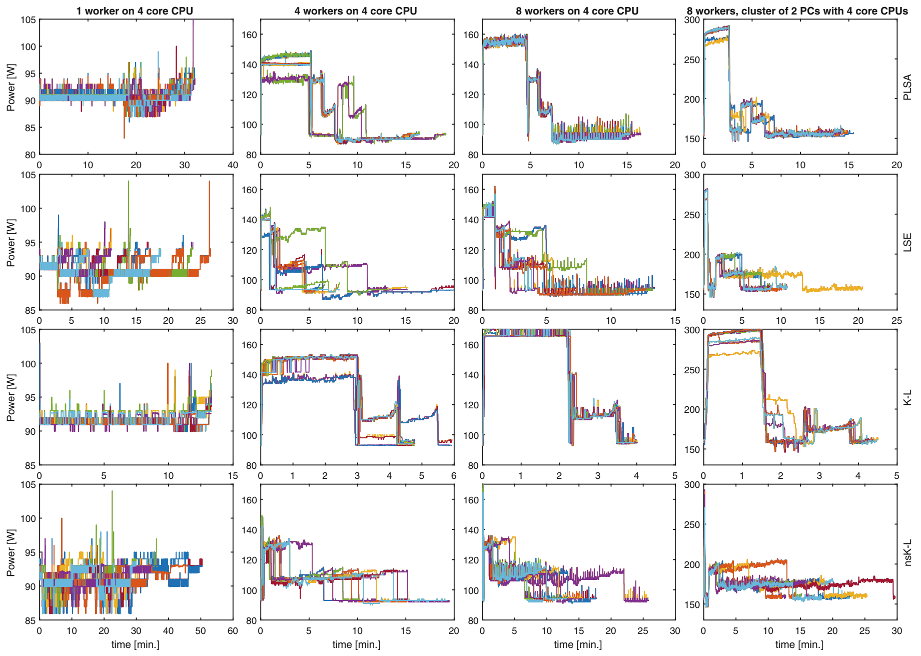


Fig. 3. Overview of power usage profiles during the test bi-clusterization task

5 Summary

In the paper we have studied a group of algorithms for a sophisticated and computationally demanding AI technique – bi-clustering. We have demonstrated that the parallel processing is a logical way for achieving better performance and energy-efficiency, but its efficiency scales in hardly predictable way. We also found out that, fortunately, there is no contradiction between the quality of results, time and energy efficiency for the considered case – K-L minimization offers the best performance according to all these criteria. As was shown in the Fig. 3 one simple guideline can be recommended for the users of NMF based bi-clustering methods - use K-L divergence measure to achieve results fast with small energetic cost and of decent quality.

Further studies in this area might include: other bi-clustering algorithms (and other AI techniques as well), larger computing facilities or different architectures, and alternative data sets.

Acknowledgment. This work is supported by Silesian Univ. of Technology grants: P. Foszner – 02/020/BKM_17/0115, P. Skurowski – 02/020/BK_17/0105.

References

1. ICT for Sustainable Growth. http://ec.europa.eu/information_society/activities/sustainable_growth/ict_sector/index_en.htm. Accessed 03 May 2017
2. Ardito, L., Morisio, M.: Green IT - available data and guidelines for reducing energy consumption in IT systems. Sust. Comput.: Inform. Syst. **4**(1), 24–32 (2014)
3. Ben-Dor, A., et al.: Discovering local structure in gene expression data: the order-preserving submatrix problem. J. Comput. Biol. **10**(3–4), 373–384 (2003)
4. de Castro, P.A.D., de França, F.O., Ferreira, H.M., Von Zuben, F.J.: Applying biclustering to text mining: an immune-inspired approach. In: de Castro, L.N., Von Zuben, F.J., Knidel, H. (eds.) ICARIS 2007. LNCS, vol. 4628, pp. 83–94. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73922-7_8
5. Cheng, Y., Church, G.M.: Biclustering of expression data. In: Ismb, vol. 8, pp. 93–103 (2000)
6. Dempster, A., Laird, N., Rubin, D.: Maximum likelihood from incomplete data via the EM algorithm. J. Roy. Stat. Soc. Ser. B 39 (1977)
7. Feng, W.: Green destiny + mpiBLAST = bioinfomagic. In: 10th International Conference on Parallel Computing (ParCo), September 2003
8. Floyd, R.W.: Nondeterministic algorithms. J. ACM **14**(4), 636–644 (1967)
9. Foszner, P., Polański, A.: Aspectanalyzer-distributed system for bi-clustering analysis. In: Gruca, A., Brachman, A., Kozielski, S., Czachórski, T. (eds.) Man-Machine Interactions 4. AISC, vol. 391, pp. 411–420. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-23437-3_35
10. Ganote, C.L., et al.: A voice for bioinformatics. In: Proceedings of the PEARC 2017, pp. 36:1–36:5 (2017)
11. Gelenbe, E., Caseau, Y.: The impact of information technology on energy consumption and carbon emissions. Ubiquity **2015**(June), 1:1–1:15 (2015)
12. Hanczar, B., Nadif, M.: Ensemble methods for biclustering tasks. Pattern Recogn. **45**(11), 3938–3949 (2012)
13. Hartigan, J.A.: Direct clustering of a data matrix. JASA **67**(337), 123–129 (1972)
14. Hibbs, M.A., et al.: Exploring the functional landscape of gene expression: directed search of large microarray compendia. Bioinformatics **23**(20), 2692–2699 (2007)
15. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**, 33–38 (2008)
16. Hofmann, T.: Unsupervised learning by probabilistic latent semantic analysis. Mach. Learn. J. 177–196 (2001)
17. Kerr, G., et al.: Techniques for clustering gene expression data. Comput. Biol. Med. **38**(3), 283–293 (2008)
18. Kluger, Y., et al.: Spectral biclustering of microarray data: coclustering genes and conditions. Genome Res. **13**(4), 703–716 (2003)
19. Kong, M., Partoens, B., Peeters, F.: Structural, dynamical and melting properties of two-dimensional clusters of complex plasmas. New J. Phys. **5**(1), 23 (2003)
20. Lazzeroni, L., Owen, A.: Plaid models for gene expression data. Statistica sinica 61–86 (2002)
21. Lee, D., Seung, S.: Algorithms for non-negative matrix factorization. In: Advances in Neural Information Processing Systems, pp. 556–562 (2000)
22. Madeira, S.C., Oliveira, A.L.: Biclustering algorithms for biological data analysis: a survey. IEEE/ACM Trans. Comput. Biol. Bioinform. 24–45 (2004)
23. Mair, J., et al.: Myths in power estimation with performance monitoring counters. Sust. Comput.: Inform. Syst. **4**(2), 83–93 (2014)

24. Markatos, E.P., LeBlanc, T.J.: Load balancing vs. locality management in shared-memory multiprocessors. Technical report, Rochester, NY, USA (1991)
25. Maulik, U., et al.: Multiobjective fuzzy biclustering in microarray data: method and a new performance measure. In: IEEE World Congress on Computational Intelligence Evolutionary Computation, CEC 2008, pp. 1536–1543. IEEE (2008)
26. Michalak, M., Lachor, M., Polański, A.: HiBi – the algorithm of biclustering the discrete data. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2014. LNCS (LNAI), vol. 8468, pp. 760–771. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07176-3_66
27. Myers, J.L., Well, A.D.: Research Design and Statistical Analysis (ed.) (2003)
28. Orzechowski, P.: Proximity measures and results validation in biclustering – a survey. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2013. LNCS (LNAI), vol. 7895, pp. 206–217. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38610-7_20
29. Pascual-Montano, A., et al.: Non-smooth non-negative matrix factorization. IEEE Trans. Pattern Anal. Mach. Intell. 403–415 (2006)
30. Rzepka, K., et al.: Design of portable power consumption measuring system for green computing needs. Studia Informatica (in press). arXiv:1512.08201 [cs]
31. Skurowski, P., Staniszewski, M.: Parallel distance matrix computation for matlab data mining. In: AIP Conference Proceedings, vol. 1738, no. 1, p. 070004 (2016)
32. Tanay, A., Sharan, R., Shamir, R.: Discovering statistically significant biclusters in gene expression data. Bioinformatics **18**(Suppl. 1), S136–S144 (2002)
33. Tanay, A., et al.: Revealing modularity and organization in the yeast molecular network by integrated analysis of highly heterogeneous genomewide data. PNAS **101**(9), 2981–2986 (2004)
34. Teng, L., Chan, L.: Discovering biclusters by iteratively sorting with weighted correlation coefficient in gene expression data. J. Sig. Proc. Syst. 1520–1527 (2010)
35. Yang, J., et al.: δ-clusters: capturing subspace correlation in a large data set. In: Proceedings of 18th International Conference Data Engineering, pp. 517–528. IEEE (2002)

Performance and Energy Analysis of Scientific Workloads Executing on LPSoCs

Anish Varghese(✉), Joshua Milthorpe, and Alistair P. Rendell

Research School of Computer Science, Australian National University, Canberra, Australia
{anish.varghese,josh.milthorpe,Alistair.Rendell}@anu.edu.au

Abstract. Low-power system-on-chip (LPSoC) processors provide an interesting alternative as building blocks for future HPC systems due to their high energy efficiency. However, understanding their performance-energy trade-offs and minimizing the energy-to-solution for an application running across the heterogeneous devices of an LPSoC remains a challenge. In this paper, we describe our methodology for developing an energy model which may be used to predict the energy usage of application code executing on an LPSoC system under different frequency settings. For this paper, we focus only on the CPU. Performance and energy measurements are presented for different types of workloads on the NVIDIA Tegra TK1 and Tegra TX1 systems at varying frequencies. From these results, we provide insights on how to develop a model to predict energy usage at different frequencies for general workloads.

Keywords: Energy usage model · LPSoC · Tegra SoC
Energy efficiency · DVFS

1 Introduction

Leading high performance computing (HPC) systems today consist of thousands of nodes containing heterogeneous elements such as CPUs, GPUS and other custom cores. The push to increase the scale of these systems presents considerable challenges in reliability, programmability and energy efficiency. The DARPA Exascale Technology study [10] has outlined power as the major bottleneck to achieving exascale computing. With the power requirements of future systems expected to increase by an order of magnitude using current technologies, meeting such high energy demands is not economically feasible.

Consequently, there has been an increased effort to investigate the suitability of low-power hardware, which are traditionally used in a mobile context, for HPC [15,16]. Low-power system-on-chip (LPSoC) processors, in particular, provide an interesting alternative as building blocks for future HPC systems. LPSoCs generally have heterogeneous compute units such as a multi-core CPU and on-chip accelerators such as Graphics Processing Units (GPU) or other custom cores on a single chip. Such heterogeneous SoCs have high floating point capability and consume less power per flop compared to conventional processors used in HPC platforms. There are, however, significant challenges for using such LPSoCs for HPC, including (i) developing application software capable of utilizing all the processing elements on the LPSoC, (ii) minimizing the energy-to-solution for an application running across the heterogeneous devices and (iii) understanding the performance-energy trade-offs. This work (currently in progress) focuses on how to model and consequently minimize the energy consumption of an application executing on an LPSoC.

The contributions of this paper are as follows - (i) We present our results from exploring the effect of Dynamic Voltage and Frequency Scaling (DVFS) on the performance and power consumption of an application running on an LPSoC system. (ii) We describe the methodology that we are developing (in progress) to extend our previous energy usage model [12], in order to predict the energy usage of application code executing under different frequency settings. The extended model may subsequently be used to determine the frequency settings to use in order to minimize total energy consumption. Our methodology involves applying frequency scaling to the processing elements and observing the variation in performance and energy consumption for various benchmark codes. Although our approach is applicable to the different components of an LPSoC (such as CPU, GPU, memory), in this work, we present results only for the CPU. We also measure the effect of scaling the number of active CPU cores. Results are presented from experiments on the NVIDIA Tegra K1 and X1 systems. Comparisons of energy consumption are presented for three benchmark codes - (i) GEMM kernel, (ii) 2D stencil computation and (iii) Block-Tridiagonal (BT) solver.

2 Related Work

The PEACH model [7] provides an analytical performance and energy model which captures the performance and energy impact of computation distribution and energy-saving scheduling to identify the optimal strategy for best performance or lowest energy consumption. Evaluation is done on heterogeneous systems with Intel Sandy Bridge processors and NVIDIA Tesla GPUs and does not consider ARM and SoC-based systems. Although this work is similar to ours, this model analyses how power varies with frequency for three particular applications in isolation and uses the results to make energy optimality predictions for the same applications. In contrast, our work attempts to study the power behaviour for different classes of workloads in order to model the energy consumption for a general application.

Du et al. [6] describe a family of models of power consumption for homogeneous multicore systems, which capture the effects of independent frequency scaling of different cores. Their results support a simple linear model which takes into account average core speed and speed variation between cores.

Other works [11,17] describe power as a cubic function of frequency and empirically derive power using non-linear regression models.

More complex models study the effects on power of micro-architectural features and use this information for developing detailed models [13]. Isci and Martonosi [8] express the power consumption of a Pentium 4 CPU as the sum of the power consumption of the processor's 22 major sub-units. Bertran et al. [1] propose a methodology to develop PMC-based models for multicore processors. They identify and classify micro-architectural components into three categories, namely, in-order engine, out-of-order engine, and memory. The authors develop a set of 97 micro-benchmarks to stress each of these components in isolation under different scenarios to detect those performance monitoring events (PMEs) that best reflect the activity level of each component, and use these PMEs to estimate the power consumption of the CPU cores. Other works employ similar techniques and methodologies to model the power of Intel and AMD based CPUs [2,3,18].

3 Platforms

We use two NVIDIA Tegra-based SoC platforms for our experiments as detailed in Table 1. The Jetson *TK1* development kit contains an NVIDIA Tegra K1 SoC and the Jetson *TX1* development kit contains an NVIDIA Tegra X1 SoC. Both the systems considered have NVIDIA GPUs which share memory with the host. For the ARM host in TK1, gcc 4.8.4 is used and for TX1, gcc 5.4 is used.

Table 1. Platforms

Platform	CPU	Max freq	RAM	GPU cores	Max freq	Linux kernel	CUDA
TK1	Cortex-A15	2.3 GHz	2 GB LPDDR3	192	852 MHz	3.10.40*armhf*	v6.5
TX1	Cortex-A57	1.7 GHz	4 GB LPDDR4	256	998 MHz	3.10.67*aarch64*	v8.0

4 Energy Model

We briefly describe our existing energy usage model [12] that describes the energy consumed by an application which is partitioned to execute between different devices on a chip. Consider two processing elements, a CPU and a GPU, denoted by c and g respectively. These processing elements execute an application at the computational rates R_c and R_g. The active power draw (in watts) of the CPU and GPU are denoted by P_c^a and P_g^a while their idle power draws are denoted by P_c^i and P_g^i respectively. The total computational cost of executing the application is labelled as N, with the fraction of the work given to the CPU denoted as w. The times spent by the CPU and GPU executing the application are denoted as T_{ac} and T_{ag} (s) respectively where $T_{ac} = \frac{Nw}{R_c}$ and $T_{ag} = \frac{N(1-w)}{R_g}$. The total time to solution is labelled as T_s (s) where $T_s = max[T_{ac}, T_{ag}]$. The total energy consumed by the system can then be modelled as the sum of the energy consumed by the CPU and that consumed by the GPU:

$$E(w) = \left[(P_c^a - P_c^i)\frac{Nw}{R_c} + P_c^i T_s\right] + \left[(P_g^a - P_g^i)\frac{N(1-w)}{R_g} + P_g^i T_s\right] \quad (1)$$

The results showed that this model is able to accurately predict the energy usage (error of less than 5% on TK1 and TX1) of a code which was partitioned to execute on both the CPU and GPU. However, the model is not able to predict the energy usage when the operating frequencies of the components is varied. It is also not able to predict the energy usage when number of active cores is scaled. In this work, we aim to address these shortfalls by characterizing the variation in energy usage when scaling frequency and number of active cores, which can then be substituted in Eq. 1.

5 Frequency Scaling

DVFS is a popular method for reducing energy consumption of processors. This is due to the possibility of getting the same results using much less energy and often without significant performance penalties. Although frequency scaling can be applied to different components such as CPU, GPU (or accelerator) and memory, for the purpose of this paper we focus only on the effects of CPU frequency scaling.

The power consumption of a processor consists of two parts: (1) dynamic part that is mainly related to CMOS circuit switching energy, and (2) static part that addresses the CMOS circuit leakage power. The dynamic portion of power can be modeled as a cubic function of frequency while the static portion can be modeled as a linear function of frequency where voltage is linearly proportional to frequency [9,17].

$$P = \lambda f^3 + \mu f \quad (2)$$

6 Microbenchmarks

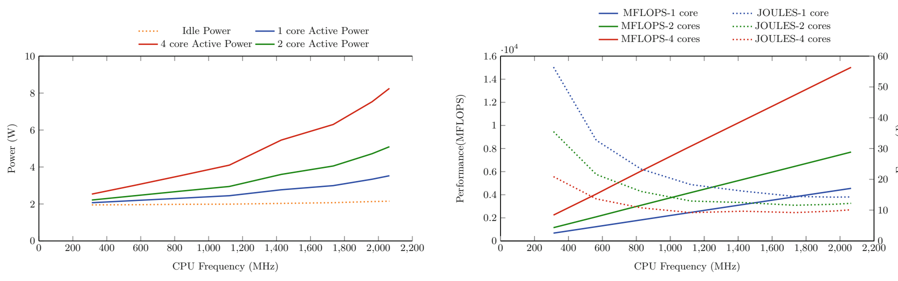
To characterize the power requirements of the CPU and memory components, we created two synthetic microbenchmarks: a “CPU-bound” code with a high proportion of floating-point and integer instructions that operate on data in (on-chip) cache, and a “memory-bound” code designed to be highly cache-inefficient, such that a high proportion of time and energy is devoted to off-chip memory access. We hypothesize that most real world applications will fall somewhere between these two extremes.

The CPU-bound code evaluates a polynomial function for every element of an array. The computation has a high flop intensity of 36 flops per memory access. The size of the vectors is chosen so that the array can be stored entirely in L1 cache. This is to ensure that there is no memory access during computation and the microbenchmark is thus bound by CPU performance.

The memory-bound code copies 32-bit values from one array to another. To ensure that copying of each element involves explicit memory access, after transferring one element in the array, the next element to be transferred is located at an offset such that the element will not be present in the cache, thereby necessitating an off-chip memory access.

6.1 CPU-Bound Workload

In the first experiment, we measured performance and power consumption for the CPU-bound workload while varying the CPU frequency, on 1, 2 and 4 cores. The mean of 20 samples are reported for all experiments with a margin of error, at a confidence level of 95%, of less than 0.5% for performance and 1% for energy and power. Power consumption was observed to increase non-linearly with increase in CPU frequency as shown in Fig. 1(a).



(a) Power (b) Performance and Energy

Fig. 1. CPU-bound workload on TK1 - varying CPU frequency

To identify the relation between power and frequency, a non-linear curve-fitting was done on the measured data. The result describes a cubic relation as identified by [11,17]. The relations are shown in Eq. 3 for TK1 and Eq. 4 for TX1.

$$P = [0.09, 0.18, 0.31] * f^3 + [0.36, 0.73, 1.66] * f + 1.95 \tag{3}$$

$$P = [0.025, 0.11, 0.25] * f^3 + [0.72, 1.04, 1.81] * f + 2.66 \tag{4}$$

where P is the power consumption in watts, and f is the CPU frequency in GHz. The equation consists of a cubic term in frequency, a linear term in frequency and a constant. The coefficients in brackets are for single core execution, 2-core execution and 4-core execution respectively. The constant term is fixed to the idle power of the system at lowest CPU frequency under no load, which is 1.95 W for the TK1 and 2.66 W for the TX1.

The measured performance and energy to solution are shown in Fig. 1(b). Performance increases linearly with CPU frequency, which is to be expected as the workload is executed entirely within the CPU. The overall energy to solution is lowest at the maximum CPU frequency as execution time is lowest in this case.

The behaviour for multicore execution was very similar, with the performance scaling linearly with the number of cores as expected. A speedup of around 3.1 is observed for 4 cores compared to 1 on the TK1. The overall energy to solution is lower when executing with a greater number of cores.

6.2 Memory-Bound Workload

A similar experiment was conducted for the memory bound workload. The observed increase in power consumption with increasing CPU frequency is shown in Fig. 2(a).

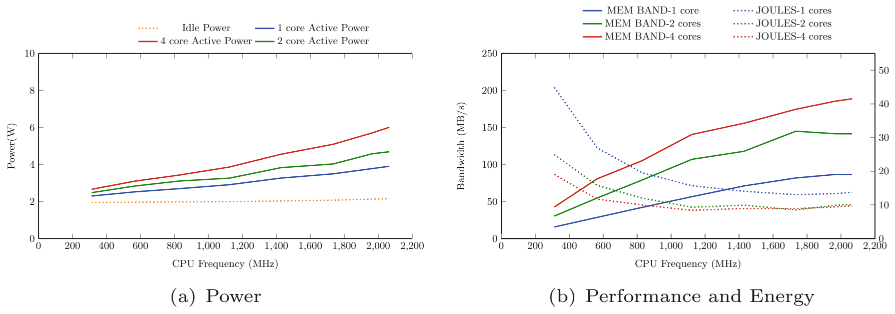


Fig. 2. Memory-bound workload on TK1 - varying CPU frequency

The relations obtained from curve-fitting are shown in Eq. 5 for TK1 and Eq. 6 for TX1. The coefficient for the cubic term is negligible, resulting in a linear relation between power and CPU frequency for the memory-bound workload. Similar behaviour is seen for multicore execution.

$$P = [0.89, 1.30, 1.74] * f + 1.95 \tag{5}$$

$$P = [0.90, 1.19, 1.72] * f + 2.66 \tag{6}$$

Memory bandwidth performance might not be expected to change much when varying CPU frequency [14]. However, in Fig. 2(b), memory bandwidth is seen to increase with CPU frequency. This might be explained by the CPU overhead of loop increment after each memory access and this becomes a bottleneck at very low CPU frequencies. It may also be explained by the cache performance at these very low frequencies since cache misses drive memory access.

As the CPU frequency reaches higher values, the memory bandwidth approaches an asymptote of around 90 MB/s for a single core and around 190 MB/s when using 4 cores on the TK1. Overall energy to solution is observed to decrease gradually with increase in CPU frequency and reaches a constant value at higher CPU frequencies. We also observed, by running the experiment at lower memory frequencies, that the overall energy to solution increased when memory frequency was reduced. This suggests it is more energy efficient to run workloads at higher memory frequencies.

For multicore execution, the performance and energy behaviour are seen to be similar. On the TK1 (at the highest memory frequency), the bandwidth using 4 cores was only 2.1 times higher than the bandwidth using a single core, due to memory contention between the cores. However, energy to solution is still lower with a greater number of cores, due to static power.

7 Predicting Power for a General Workload

Based on the observations of power consumption for the 2 microbenchmarks, we describe our approach to derive a model (work in progress) for determining the power consumed by any general workload at a given frequency, which can be then substituted in Eq. 1. From Eqs. 3 and 5, we observe how power consumption varies with frequency for a completely CPU-bound workload and for a completely memory-bound workload respectively. Naively one might expect the power consumed by any general application workload to lie within these two extremes. To test this hypothesis, we measured the power consumed by three benchmark codes – Matrix Multiplication, Stencil Computation and Block Tridiagonal Solver – at different CPU frequencies and compared them with the results from the microbenchmarks. The result for single core execution is shown in Fig. 3.

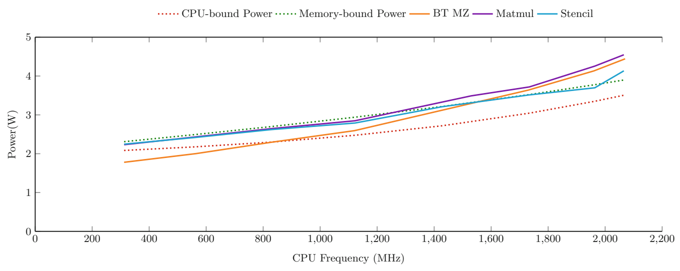


Fig. 3. Comparison of power consumed by benchmark codes - TK1

From Fig. 3 it is evident that the above hypothesis does not hold, as the application results fall outside the bounds provided by the CPU-bound and memory-bound workloads. An alternative would be to model application power consumption as a linear combination of Eqs. 3 and 5 (power behaviours for completely CPU-bound and completely memory bound workloads), as follows:

$$P = x * (a_1.f^3 + b_1.f + c) + y * (a_2.f^3 + b_2.f + c) \tag{7}$$

The first term in the equation signifies the contribution to the power by on-chip activity and the second term denotes the contribution by off-chip or memory activity. To test this hypothesis, we constructed a benchmark that executes both the CPU-bound and memory-bound workloads disjointly in segments spanning very short durations of time. We ran an experiment where we varied the time periods of each segment and measured the power consumption. We compared this to the power predicted by Eq. 7, by substituting the relative time proportion of each segment for x and y. We obtained the results for a few configurations of x and y. An average error of less than 5% is observed for all the predictions, validating this simple combination model for these workloads. The results for two configurations (single-core) are shown in Fig. 4.

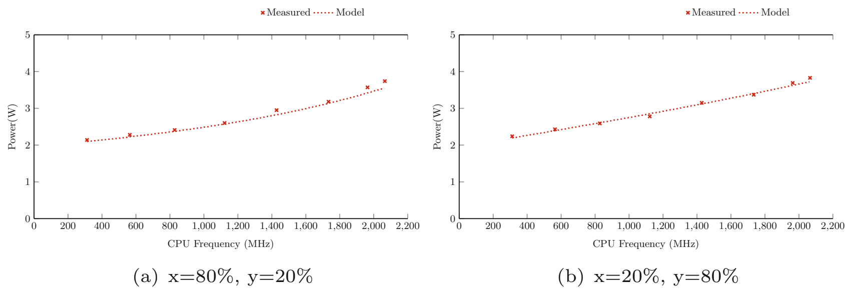


Fig. 4. Combination model results for two configurations - TK1

Our aim is to characterize a general workload by examining the relative mix of instructions executed by the workload in order to estimate the contributions of the on-chip components and off-chip components to the overall power. With this in mind, we also collected available hardware performance counters for the previous experiments with the aim of estimating x and y from the collected metrics. The operational intensity metric (in flops:byte) is widely used to describe compute or memory boundness [4,5]. For our experiments we measure hardware performance counter metrics such as *total instructions per cycle* and *last level cache misses per cycle* in order to estimate operational intensity since they are good indicators of levels of on-chip and off-chip activity. However detailed analysis of these metrics in order to characterize a general workload is currently in progress and not included in this work. Table 2 lists a few collected performance metrics for the microbenchmarks and the three benchmark codes.

Table 2. Collected hardware performance metrics on TK1

Metric	CPU-bound	Memory-bound	Stencil	BT solver	GEMM
inst_per_cyc	0.815	0.040	0.865	1.302	*
fp_ins_per_tot	0.676	0	0.356	0.264	*
mem_ins_per_tot	0.020	0.376	0.427	0.658	*
llc_miss_per_tot	0	0.347	0.005	0.001	0.011
llc_miss_per_mem	0	0.921	0.011	0.002	0.037

* GEMM counters could not be reliably measured.

8 Conclusions and Future Work

This paper presented our results from exploring the effect of DVFS on the performance and power usage of an application running on an LPSoC system. We presented our methodology to model the power consumed by an application under different frequencies. This involved executing specially designed workloads which stress on-chip components and off-chip components in isolation. From the results, it was observed that the power consumed by a CPU-bound workload varies as a cubic function of frequency while the power consumed by a memory-bound workload varies as a linear function of frequency on the experimental platforms. The power behaviour of the workloads under single core and multi-core execution was also described. It is observed that while power consumption reduces when reducing CPU frequency, the overall energy consumption is lowest at higher CPU frequencies for both the CPU-bound and memory-bound workloads. It was also observed that increasing the number of cores of execution also resulted in reduction in overall energy consumption for both types of workloads.

A comparison of the power measurements of three application benchmark codes was shown along with the measurements from the microbenchmark workloads. Based on this, we are in the process of developing a simple model which describes the power consumed by any general workload at varying frequencies as a combination of the power consumption behaviour of the on-chip components and that of the off-chip components. Since the contributions of these different components to the overall power depends on the mix of instructions involved for a particular workload, our approach is to collect hardware performance counters in order to estimate the contributions of the on-chip and off-chip activity to the overall power consumption. This is currently a work in progress. In our future work we aim to extend our model by applying the proposed methodology to the other components of an LPSoC, such as GPU (or other accelerators) and memory.

References

1. Bertran, R., Gonzelez, M., Martorell, X., Navarro, N., Ayguade, E.: A systematic methodology to generate decomposable and responsive power models for CMPs. IEEE Trans. Comput. **62**(7), 1289–1302 (2013)
2. Bircher, W.L., John, L.K.: Complete system power estimation using processor performance events. IEEE Trans. Comput. **27**(11), 563–577 (2015)
3. Chen, X., Xu, C., Dick, R.P., Mao, Z.M.: Performance and power modeling in a multi-programmed multi-core environment. In: ACM/IEEE Design Automation Conference (DAC) (2010)
4. Choi, J., Dukhan, M., Liu, X., Vuduc, R.: Algorithmic time, energy, and power on candidate HPC compute building blocks. In: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS) (2014)
5. Choi, J.W., Bedard, D., Fowler, R., Vuduc, R.: A roofline model of energy. In: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS) (2013)

6. Du, Z., Ge, R., Lee, V.W., Vuduc, R., Bader, D.A., He, L.: Modeling the power variability of core speed scaling on homogeneous multicore systems. Sci. Program. **2017**, 13 p. (2017). https://doi.org/10.1155/2017/8686971. Article ID 8686971
7. Ge, R., Feng, X., Burtscher, M., Zong, Z.: Performance and energy modeling for cooperative hybrid computing. In: Proceedings of IEEE International Conference on Networking, Architecture, and Storage (NAS) (2014)
8. Isci, C., Martonosi, M.: Runtime power monitoring in high-end processors: methodology and empirical data. In: Proceedings of Annual International Symposium on Microarchitecture (MICRO) (2003)
9. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: Proceedings of International Symposium on Low Power Electronics and Design (ISLPED), vol. 1, no. 1 (1998)
10. Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., et al.: Exascale computing study: technology challenges in achieving exascale systems. Technical report TR-2008-13, DARPA (2008)
11. Lang, J., Rünger, G.: An execution time and energy model for an energy-aware execution of a conjugate gradient method with CPU/GPU collaboration. J. Parallel Distrib. Comput. **74**(9), 2884–2897 (2014)
12. Mitra, G., Haigh, A., Varghese, A., Angove, L., Rendell, A.P.: Split wisely: When work partitioning is energy-optimal on heterogeneous hardware. In: Proceedings of International Conference on High Performance Computing and Communications (HPCC) (2016)
13. Möbius, C., Dargie, W., Schill, A.: Power consumption estimation models for processors, virtual machines, and servers. IEEE Trans. Parallel Distrib. Syst. **25**(6), 1600–1614 (2014)
14. Nikl, V., Hradecky, M., Keleceni, J., Jaros, J.: The investigation of the ARMv7 and intel Haswell architectures suitability for performance and energy-aware computing. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) ISC 2017. LNCS, vol. 10266, pp. 377–393. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58667-0_20
15. Rajovic, N., Carpenter, P.M., Gelado, I., Puzovic, N., Ramirez, A., Valero, M.: Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC? In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC). ACM (2013)
16. Rajovic, N., Rico, A., Puzovic, N., Adeniyi-Jones, C., Ramirez, A.: Tibidabo: Making the case for an ARM-based HPC system. Future Gener. Comput. Syst. **36**, 322–334 (2014)
17. Rizvandi, N.B., Zomaya, A.Y., Lee, Y.C., Boloori, A.J., Taheri, J.: Multiple frequency selection in DVFS-enabled processors to minimize energy consumption. In: Energy-Efficient Distributed Computing Systems, pp. 443–463 (2012)
18. Singh, K., Bhadauria, M., McKee, S.A.: Real time power estimation and thread scheduling via performance counters. ACM SIGARCH Comput. Archit. News **37**(2), 46 (2009)

Energy Efficient Dynamic Load Balancing over MultiGPU Heterogeneous Systems

Alberto Cabrera(✉), Alejandro Acosta, Francisco Almeida, and Vicente Blanco

HPC Group, Escuela Superior de Ingeniería y Tecnología,
Universidad de La Laguna, ULL, 38270 La Laguna, Tenerife, Spain
Alberto.Cabrera@ull.edu.es

Abstract. Current HPC technologies demand high amounts of power and energy to achieve good performances. In order to address the next milestone in peak power, powerful graphic processing units and many-core processors present in current HPC systems need to be programmed having energy efficiency in mind. As energy efficiency is a major issue in this area, the existing codes and libraries need to be adapted to improve the use of the available resources. Rewriting the code requires deep knowledge of programming and architectural details to achieve good efficiency, which is an ad-hoc solution for a concrete system. We present the *Ull_Multiobjective_Framework*, an interface that allows automatic dynamic balance of the workload for parallel iterative codes in heterogeneous environments. *UllMF* allows to include the overall energy consumption as a parameter during the balancing process. This tool hides all architectural measurement details, requires very low effort to the programmer and introduces a minimum overhead. The calibration library has been used to solve iterative problems over heterogeneous platforms. To validate it, we present an analysis of different Dynamic Programming problems over different hardware configurations of a MultiGPU heterogeneous system.

Keywords: Dynamic load balancing · Energy efficiency
Iterative algorithms

1 Introduction

There is a major concern in high performance computing regarding to the increasing energy consumption of the top end systems as current Petaflop systems draw a huge amount of energy at full performance. With the current growth, by 2020, exascale systems will start to appear, and the main challenge will be to have them limited to 20 MW to have a bearable infrastructure cost.

Highly heterogeneous environments are nowadays available as computational resources for institutions. However, applications usually have a performance penalty when they are not tuned explicitly for the heterogeneous platform and there is a strong dependence between parallel code and target architectures [4].

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 123–132, 2018.
https://doi.org/10.1007/978-3-319-78054-2_12

New architectures are still affected by the same well known problems that affect conventional parallel programming, including the load balancing problem while solving irregular problems, an aggravated problem when the systems used have different efficiencies in terms of both energy and time.

As new architectures are developed, good performances per watt can be achieved by sacrificing performance, allowing distinction between time and energy efficiency. The principles of load balancing, developed to improve execution time before the exascale objective, can now be applied to achieve good performances per watt are achieved by sacrificing performance. We investigate the load balancing problem appearing when parallel problems are executed in multiGPU heterogeneous systems and how the energy consumption is related to imbalance problems.

We designed a simple mechanism to balance the workload between different parallel processes dynamically, with few changes to the source code of an application, the *ULL_Multiobjective_Framework* (*UllMF*), that allows dynamic workload balancing inside a parallel program running on a heterogeneous system, adapting to the conditions of the system during the execution of the application. This software allows calibration for different objectives, time and energy, with the time module being `ULL_Calibrate_lib` [2] and a new module for energy calibration. This module uses the Energy Measurement Library (EML) [3] for energy measurement, that has proven to have a negligible overhead, and provides all the tools needed to perform the measurement for the calibration phase of the dynamic load balance operations. We prove that performing different approaches for balancing is feasible to improve energy efficiency.

The main contributions of this paper are:

- Extending the ideas and methodologies developed for the load balancing of heterogeneous clusters to energy consumption.
- A new library design, capable of being extended for calibrating based on a user objective by adding new modules.
- Dynamic load balancing to reduce energy consumption in a heterogeneous multiGPU environment, with an energy module developed for *UllMF*.

To validate our proposal, we have executed several Dynamic Programming algorithms where the portability of the single GPU code to a multiGPU system is compromised due to load imbalance inefficiencies. The chosen problems correspond to an iterative scheme that is representative of many other problems like Jacobi, GaussSeidel, Longest Common Subsequence, Matrix Parenthesization and to some classes of stencil codes. The efficiency level obtained, considering the minimum code intrusion, makes this methodology a useful tool in the context of the energy efficiency in heterogeneous platforms.

This paper is structured as follows: Sect. 2 covers the related work in the field of load balancing workload on heterogeneous systems. Section 3 describes the problem we address and our proposal. Section 4 illustrates our hardware configuration and the results obtained after multiple executions performed with our energy balancing methodology. Finally, Sect. 5 summarizes our conclusions and future research possibilities.

2 Related Work

Extensive work has been done in the field of load balancing in heterogeneous systems to improve time performance. The most direct approach to address the problem requires to develop code specifically for the architecture in order to maximize performance. This requires a deep understanding on parallel programming for the target systems and also a manual task allocation according to the capability of every processing unit. A different approach consist of developing the applications using skeleton based programming. This concept implies finding an skeleton that covers your implementation, so that you can develop the particularities of your exact problem. Once done, this skeletons that are optimized to obtain an optimal load balance for maximum time efficiency, could be modified to obtain the best configuration for energy efficiency. Frameworks like SkelCL [11] and DPSKEL [1,9] are examples for this paradigm. Our approach to balance the workload of iterative algorithms on heterogeneous systems is closer to the work presented in ADITHE [7]. This environment and `ULL_Calibrate_lib` [1] follow a similar strategy. Starting from an homogeneous distribution of the workload on the heterogeneous system, the speed of every node is estimated during the first iterations of the algorithm. According to the speed of every node, new workload distributions are carried out for the remaining iterations of the algorithm. A similar approach can be used to balance workloads in terms of energy consumption.

Energy-aware scheduling algorithms can be found in datacenters with virtual machines deployments. Examples of this can be found at HEROS [6], that proposes a load balancing algorithm for energy-efficient resource allocation in heterogeneous systems, or PVA [12], peer VMs aggregation to enable dynamic discovery of communication patterns and reschedule VMs based on the determined communication patterns using VM migration. These techniques minimize network traffic and deal with an energy efficient scheduling.

In manycore systems, ALEPH [10] is an algorithm that solves the bi-objective optimization problem for performance and energy (BOPPE) in multicore architectures by introducing the problem size as a decision variable. In iterative schemes, E-ADITHE [5] was introduced as a two step process for integrated CPU-GPUs system on chip (SoC), where a first step is made for selecting the most energy efficient resources and then the load balance technique is applied to maximize performance.

Other techniques use DVFS processor capabilities to implement energy-aware schedulers to balance energy in iterative algorithms [8].

In *UllMF* we start the execution of an iterative application with an homogeneous distribution of work. The difference with the other authors is that we redistribute workload depending on the current performance of the application rather than using a model of the application, without a preparation phase. Depending on the overall performance, the workload gets dynamically redistributed to improve the execution time or the energy consumption. With this approach, we exchange a slight performance in order to reduce the overhead of the calibration methodology.

3 Dynamic Load Balancing for Energy Efficiency

3.1 Dynamic Load Balancing

The problem we are trying to address is formulated as a variation of the classic technique of load balancing in iterative algorithms to improve performance in heterogeneous environments. As every system is different, performing an uniform workload distribution would result in a resource loss caused by the different computing capabilities of each computing element. The most powerful hardware would end its work, and to continue its next step, it would need to wait for the slower hardware to end. By redistributing the work appropriately, it is possible to minimize these waiting times in order to improve the overall performance of an iterative algorithm. The same principle can be applied to energy consumption.

However, when the problem to be solved presents an irregular nature, i.e., every iteration requires a different amount of work, the load balance has to be done not only at the beginning of the execution but multiple times along the execution. Furthermore, the input may cause variation, forcing to rebalance after some iterations have been performed. This is why a simple, but dynamic, method of load balance is required, even if it is less accurate than algorithms based on models, as it needs to be performed multiple times without sacrificing the possible gains from a more precise load balance.

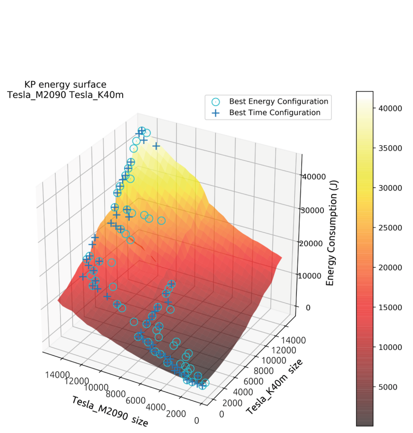
Four examples of irregular iterative algorithms have been chosen for the experimentation using a dynamic programming implementation based on programming skeletons. These algorithms are the Knapsack Problem (KP), the Resource Allocation Problem (RAP), the Cutting Stock Problem (CSP) and the Triangulation of Convex Polygons (TCP). Their implementations follow the recurrence formulations described in Table 1.

Table 1. Dynamic programming problems description.

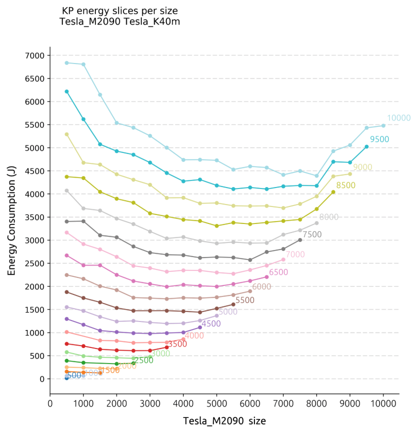
Problem name	Recurrence equation
KP	$f_{i,j} = \{f_{i-1,j}, f_{i-1,j-w} + p_i\}$
RAP	$f_{i,j} = p_{1,j}\ if\ i = 1\ and\ j > 0$ $f_{i,j} = \max_{0\le k<j}\{f_{i-1,j-k} + p_{i,k}\ if\ (i > 1)\ and\ (j > 0)$
CSP	$f_{i,j} = max \begin{cases} \max_{0\le k<object}\{profit_k\} \\ \max_{0\le z\le i/2}\{f_{z,j} + f_{i-z,j}\} \\ \max_{0\le y\le j/2}\{f_{i,y} + f_{i,j-y}\} \end{cases}$
TCP	$f_{i,j} = cost_i \cdot cost_{i+1} \cdot cost_{i+2}\ if\ i = (j-2)$ $f_{i,k} = \min_{i<j<k}\{f_{i,j} + T_{k,j} + (cost_i \cdot cost_j \cdot cost_k)$

3.2 Workload Distribution to Increase Energy Efficiency

Current technologies do not always operate at maximum energy efficiency when they work as fast as possible, minimizing execution time. Figure 1a illustrates



(a) 3D representation of the solution space for the *Knapsack Problem*



(b) 2D representation of the solution space for the *Knapsack Problem*, each curve represents a problem size

Fig. 1. Knapsack problem solution space study

this phenomenon through an exhaustive analysis of the space solutions for KP, a fine grained memory bounded problem. The 3D surface summarizes different workload distributions for multiple problem sizes using two different GPUs. X and Y axis, represent the workload given to each process, the former executed in a Tesla M2090, and the latter in the Tesla MK40c. The surface itself, in the Z axis, represents the energy consumed in total by each execution. Additionally, the best workload distribution for minimizing energy consumption is labelled with an O symbol and the best workload distribution for minimizing execution time with the $+$ symbol, for every problem size, which can be calculated by adding the X and Y values. These sets of optimal solutions are different enough to demonstrate the disparity between energy consumption and execution time.

The same data set is represented in the two dimensional chart labeled as Fig. 1b. Each curve with a given color represents a different problem size, labeled at the end of the curve itself. Energy consumption is depicted in the Y axis, the lower the better. The X axis is the workload given to the Tesla M2090 Process, i.e. to left part of the axis less workload is given to the M2090 GPU and to the right more workload is given to it. As the size of the problem increases, the workload curves start to display local minima and irregular shapes.

Observing the results obtained in the study, it is valid to conclude that an optimization for energy consumption can yield results that differ to an optimization performed to reduce the execution time of a parallel application. Considering the following terminology:

- p, number of parallel processes.
- w_i, normalized amount of work given to process i, with $0 \leq w_i \leq 1$ and $\sum_{i=1}^{p} w_i = 1$.

- $d(d_1, d_2, \cdots, d_p)$, workload displacement, with $-w_i \leq d_i \leq (1 - w_i)$ and $\sum_{i=1}^{p} d_i = 0$.
- E_i, energy consumed by process i.
- $\frac{E_i}{w_i}$, estimated energy consumed per unit of work by process i.

Our proposal is to reassign the workload between processors dynamically, on each step of the iterative algorithm by finding the best combination of workload that minimizes the whole energy consumed by all the processors. To achieve it, we generate a small set of different workload displacements $d(d_1, d_2, \cdots, d_p)$, adding or subtracting workload of process i depending on its performance. This set of possible data movements is used to simulate the estimated energy consumption with Eq. 1.

$$min f(d), f(d) = \sum_{i=1}^{p} \left[(w_i + d_i) \cdot \frac{E_i}{w_i} \right] \tag{1}$$

This equation is summarized as *the total cost of the current iteration if process i workload is $w_i - d_i$*. The amplitude of the movement, determined by $max(d_i)$, is decreased after every iteration, stopping when a local minima is reached, as the effort of finding the optimal solution using this methodology outweights the potential energy savings that can be achieved.

3.3 Ull Multiobjective Framework

We have developed *UllMF*, a calibration framework, to support the proposal in the previous section, as well to allow the reutilization of the code of traditional calibration techniques. *UllMF* is currently composed of two calibration modules, the energy calibration module (our proposal) and a time calibration module (based on *Ull_calibrate_lib*). Additionally, we have developed the usage of energy measurement libraries independently, and while we are using EML to measure energy, it is possible to modify and substitute the energy measurement module. The resulting structure is visualized in Fig. 2.

This implementation was done to offer dynamic load balancing capabilities to non expert programmers, focusing on the ease of use. The overhead introduced by our system is at most 2% of the total energy consumption in its current state, as shown in Fig. 3.

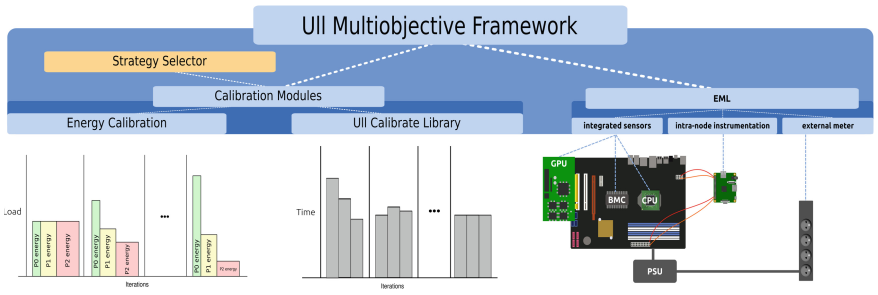
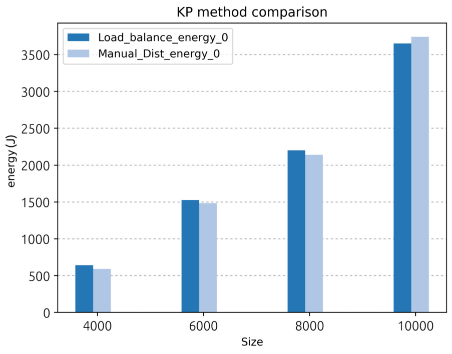


Fig. 2. *UllMF* component diagram. In orange, work in progress. (Color figure online)

Nodes	CPUs (Xeon)	GPU	Memory
Verode16	2x E5-2660	M2090	64 GB
Verode17	2x E5-2660	K20c	64 GB
Verode18	2x E5-2660	K40m	64 GB
Verode20	2x E5-2698 v3	M2090	128 GB

GPU Type	M2090	K20c	K40m
# Cores	512	2496	2880
RAM	6GB	5GB	12GB
Mem BW	177.6 GB/s	208 GB/s	288 GB/s
Power	225 W	225 W	235 W

(a) GPU Cluster



(b) Overhead

Fig. 3. GPU cluster and overhead caused by energy calibration

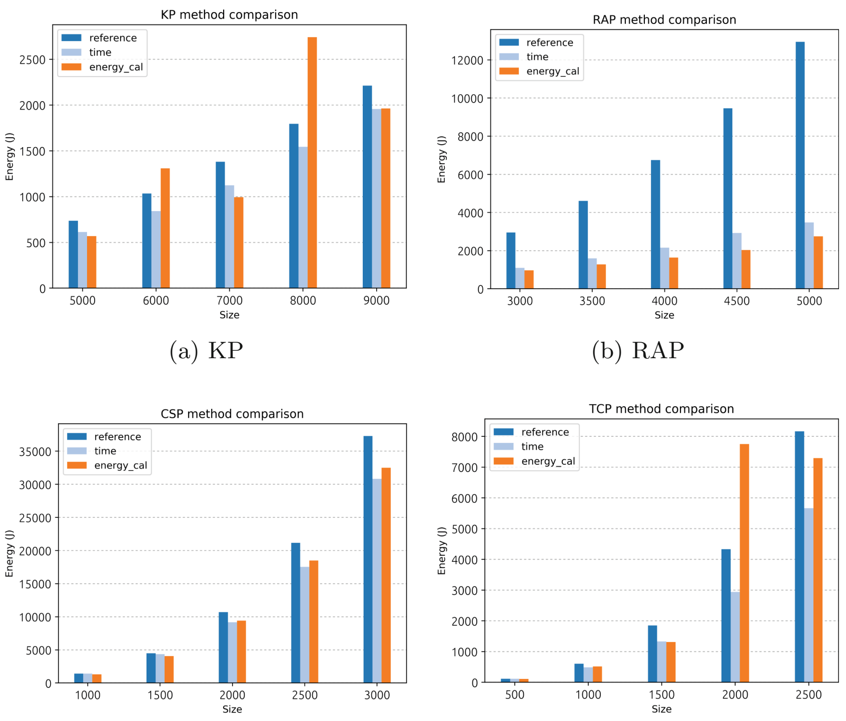
The framework provides an unified API to be used. Initially, it requires four API calls in the code to be used: *UllMF_setup*, *UllMF_calibrate_start*, *UllMF_calibrate_stop* and *UllMF_finalize*. To simplify usage, the framework provides a datatype, *calibration_t*, to encapsulate the workload distribution, which calibration technique to use, the measurement module to use, and a threshold to determine when to stop the dynamic calibration.

First, *UllMF_setup* should be called with the workload distribution, the calibration module, the measurement module and the stop threshold. Then, in the core of the iterative procedure, *UllMF_calibrate_start* and *UllMF_calibrate_stop* need to be called at every computation phase. Workload distributions are modified by the measurements obtained between these two points, and are redistributed to every process inside the *UllMF_calibrate_stop* procedure. After the work is done, *UllMF_finalize* is called to free the memory allocated for this purpose, and to stop the energy measurements.

4 Computational Results

We have used an heterogeneous cluster as tested architecture to evaluate our proposals. The cluster consists of 4 nodes with different CPU-GPU configurations. They have two types of CPUs and three types of GPUs that are detailed in Fig. 3. These nodes use Debian 9 with the kernel 4.9.0-2-amd64. The build and execution environments have GCC version 4.4.5, OpenMPI 1.6.3 and CUDA version 7.5. Every compilation was done with the $-O2$ optimization flag. Energy measurement was performed for the GPUs using the Nvidia Management Library (NVML) interface through EML. Every problem is executed for multiple sizes in different hardware combinations.

Figure 4 collects a subset of all the experimentation that properly represents the strengths and weaknesses of our methodology. Each subfigure represent a



(a) KP (b) RAP

(c) CSP (d) TCP

Fig. 4. Energy consumption of *UllMF* compared to an evenly distributed workload

different problem using multiple column sets, with different problem sizes represented in the X axis, and the energy consumed by the execution in the Y axis. Each column set is composed by three columns that visualize a comparison between three procedures: a uniform workload distribution, labeled as *reference*; a workload distribution performed using the *UllMF* version of *Ull_Calibrate_Lib*, labeled as *time*; and the newly developed procedure of workload distribution to minimize energy consumption, labeled as *energy_cal*.

The results for the KP are summarized in Fig. 4a. The *UllMF* framework is capable of balancing the work in order to minimize energy consumption, using one of its modules. The energy module, which we will refer as *energy_cal*, however, misbehaves in 18% of the cases, where the obtained workload distribution reaches the opposite effect. In this figure, we observe how for size 6000 and 8000, the energy consumption increases due to a poor load balance, since the solution gets trapped in a workload distribution that is worse than the reference, and is not able to recognize the situation. In the rest of the cases, it achieves results that improve between 2% and 9% the energy consumption of the *Ull_Calibrate_Lib* module. On average, including the results were the calibration makes the solution worse, energy consumption is reduced by 4%. When *energy_cal* works properly, the average energy consumption is reduced by 13%.

The RAP shares 90% of the code with the KP. The main difference is that the KP is a fine grain and memory bounded problem, while the RAP is compute bounded. Figure 4b displays how *energy_cal* obtains very good performances, in terms of energy savings, in this environment. This is expected as the RAP irregularity lies in being unbalanced when the work is distributed evenly. The interesting result here, apart from the noticeable improvement from the reference (up to 400%), is that *energy_cal* is better than the *Ull_Calibrate_Lib* module in every case studied, with an average improvement of 10.2%.

The CSP, as shown in Fig. 4c, is improved by both the *Ull_Calibrate_Lib* module and *energy_cal*, with a 2.1% worse performance by the latter, as opposed by the last problem. *Ull_Calibrate_Lib* improves the result of the reference problem by an average of 15.3% while *energy_cal* improves it by 13.2%.

Both modules behave in the TCP similarly to the CSP as shown in Fig. 4d. In this case, we observe that the *energy_cal* produces bad balancing when size is 2000, additionally in size 2500, in spite of improving the reference energy consumption, its performance is subpar compared to *Ull_Calibrate_Lib*. Despite the possibility of misbehaviour, the energy module of *UllMF* improves on average the energy consumption by 6%. When balanced properly, *energy_cal* provides an average reduction of energy consumption of 18.5%. However, *Ull_Calibrate_Lib* improves the overall energy consumption by 30.5% compared to these values.

These results, though they could still be improved, demonstrate that our approach is competitive enough to justify its simplicity. In spite of the evident misbehaviour for some cases, we observe problems where the calibration technique improve energy consumption greatly, with a little overhead of 3% shown in Fig. 3.

5 Conclusion

We have developed a library that allow us to perform dynamic load balancing for energy consumption in heterogeneous systems. We included our library in a calibration framework we have developed, *UllMF*. It has been proven to work using iterative problems and can be easily applied to a wider range of problems with little effort required by the programmer, adding a few lines to the available codes. It also has been proven to work properly with most presented cases, but still requires improvements in order to compete with a time based load balancing. In the future we will extend the library to allow users a multi-objective approach where multiple objectives are taken into account so that optimizing workload distributions can be feasible with a Strategy Selector that choose automatically the proper load balance calibration technique in order to increase the efficiency of the system.

Acknowledgment. This work was supported by the Spanish Ministry of Education and Science through the TIN2016-78919-R project, the Government of the

Canary Islands through the grant with reference TESIS2017010134, partially funded by FEDER funds; the Spanish network CAPAP-H4, and the European COST Actions NESUS and CHIPSET.

References

1. Acosta, A., Almeida, F.: Skeletal based programming for dynamic programming on MultiGPU systems. J. Supercomput. **65**(3), 1125–1136 (2013). https://doi.org/10.1007/s11227-013-0895-x
2. Acosta, A., Blanco, V., Almeida, F.: Dynamic load balancing on heterogeneous multi-GPU systems. Comput. Electr. Eng. **39**(8), 2591–2602 (2013). https://doi.org/10.1016/j.compeleceng.2013.08.004
3. Cabrera, A., Almeida, F., Arteaga, J., Blanco, V.: Measuring energy consumption using EML (energy measurement library). Comput. Sci. - Res. Dev. **30**(2), 135–143 (2014). https://doi.org/10.1007/s00450-014-0269-5
4. Dongarra, J., Bosilca, G., Chen, Z., Eijkhout, V., Fagg, G.E., Fuentes, E., Langou, J., Luszczek, P., Pjesivac-Grbovic, J., Seymour, K., You, H., Vadhiyar, S.S.: Self-adapting numerical software (SANS) effort. IBM J. Res. Dev. **50**(2/3), 223–238 (2006)
5. Garzón, E.M., Moreno, J.J., Martínez, J.A.: An approach to optimise the energy efficiency of iterative computation on integrated GPU-CPU systems. J. Supercomput. **73**(1), 114–125 (2017). https://doi.org/10.1007/s11227-016-1643-9
6. Guzek, M., Kliazovich, D., Bouvry, P.: HEROS: energy-efficient load balancing for heterogeneous data centers. In: Pu, C., Mohindra, A. (eds.) 8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, 27 June–2 July 2015, pp. 742–749. IEEE (2015). https://doi.org/10.1109/CLOUD.2015.103
7. Martínez, J., Garzón, E., Plaza, A., García, I.: Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE. J. Supercomput. 1–9 (2009). https://doi.org/10.1007/s11227-009-0350-1
8. Padoin, E.L., Castro, M.B., Pilla, L.L., Navaux, P.O.A., Méhaut, J.: Saving energy by exploiting residual imbalances on iterative applications. In: 21st International Conference on High Performance Computing, HiPC 2014, Goa, India, 17–20 December 2014, pp. 1–10. IEEE (2014). https://doi.org/10.1109/HiPC.2014.7116895
9. Peláez, I., Almeida, F., Suárez, F.: DPSKEL: a skeleton based tool for parallel dynamic programming. In: 7th International Conference Parallel Processing and Applied Mathematics, PPAM2007, Gdansk, Poland, pp. 1104–1113, September 2007. https://doi.org/10.1007/978-3-540-68111-3_117
10. Reddy, R., Lastovetsky, A.: Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy. IEEE Trans. Comput. **PP**(99), 1 (2017)
11. Steuwer, M., Gorlatch, S.: SkelCL: a high-level extension of OpenCL for multi-GPU systems. J. Supercomput. **69**(1), 25–33 (2014). https://doi.org/10.1007/s11227-014-1213-y
12. Takouna, I., Rojas-Cessa, R., Sachs, K., Meinel, C.: Communication-aware and energy-efficient scheduling for parallel applications in virtualized data centers. In: IEEE/ACM 6th International Conference on Utility and Cloud Computing, UCC 2013, Dresden, Germany, 9–12 December 2013, pp. 251–255. IEEE (2013). https://doi.org/10.1109/UCC.2013.50

Workshop on Scheduling for Parallel Computing (SPC 2017)

Scheduling Data Gathering with Maximum Lateness Objective

Joanna Berlińska(✉)

Faculty of Mathematics and Computer Science, Adam Mickiewicz University in Poznań, Umultowska 87, 61-614 Poznań, Poland
Joanna.Berlinska@amu.edu.pl

Abstract. In this paper, scheduling in a star data gathering network is studied. The worker nodes of the network produce datasets that have to be gathered by a single base station. The datasets may be released at different moments. Each dataset is assigned a due date by which it should arrive at the base station. The scheduling problem is to organize the communication in the network so that the maximum dataset lateness is minimized. As this problem is strongly NP-hard, we propose a heuristic algorithm for solving it. The performance of the algorithm is evaluated on the basis of computational experiments.

Keywords: Scheduling · Data gathering · Maximum lateness
Release times

1 Introduction

Parallel and distributed processing has become very popular in recent years. Complex computations may often be divided between many computers working in parallel. Distributed networks are also used for sensing and measuring the environment. In many cases the obtained data have to be collected together on a single machine for further analysis or storing. The efficiency of the data gathering process influences the overall performance of a distributed application. Therefore, scheduling data gathering becomes an important research area.

Data gathering scheduling problems were studied in [5–7,10,21]. In [10,21] data gathering wireless sensor networks were analyzed. The scheduling problem was to assign the amounts of measured data to the network nodes and organize the communications with the base station so that the total time of sensing and gathering data was minimized. An algorithm for maximizing the lifetime of a data gathering network with limited node memory was proposed in [5]. Scheduling in data gathering networks whose nodes were able to compress data at some monetary cost was studied in [6]. The goal was to transfer all data to the base station within given time at the minimum cost. In [7] scheduling algorithms for data gathering networks with variable communication speed were proposed.

In this work we analyze gathering the results of a set of applications run in parallel in a star network. When a worker node completes its job, the dataset it produced becomes ready to be sent to the base station. Each dataset has a due date by which it should arrive at the base station. The communications between the worker nodes and the base station have to be scheduled so that the maximum dataset lateness is minimized.

The rest of this paper is organized as follows. In Sect. 2 we present the network model and formulate the scheduling problem. Related work is outlined in Sect. 3. In Sect. 4 a heuristic scheduling algorithm is proposed. The results of computational experiments on its performance are described in Sect. 5. The last section comprises conclusions.

2 Problem Formulation

We analyze a data gathering network consisting of m worker nodes $P_1, \dots, P_m$ and a single base station, to which the computation results should be passed. Node P_j generates dataset D_j of size α_j at time r_j. The due date for receiving D_j at the base station is denoted by d_j. At most one worker can communicate with the base station at a time. Following the methodology of *divisible load theory* (see, e.g., [8,12]), we assume that a dataset can be divided into pieces of arbitrary (rational) sizes, sent separately. The communication capabilities of the network are described by two parameters: communication startup S and communication rate (inverse of speed) C. Thus, sending a message of size x from a worker node to the base station takes time $S+Cx$. Hence, if a dataset is sent in many separate messages, then each additional message increases the total dataset transfer time by S.

Let T_j denote the time when the whole dataset D_j arrives at the base station. The lateness of dataset D_j is $L_j = T_j - d_j$. Our goal is to schedule the communications in the network so that the maximum dataset lateness $L_{max} = \max_{j=1}^{m}\{L_j\}$ is minimized.

Since the network nodes communicate with the base station sequentially, our problem is equivalent to scheduling m preemptive jobs of size $p_j = C\alpha_j$ on a single machine, where setup time S is required when a job is started, whether initially or after preemption. Following the three-field notation [13], this problem can be denoted by $1|pmtn, r_j, s_j = S|L_{max}$.

3 Related Work

Single machine scheduling with setup times was studied in [9,14,16,19,20]. In [14] it was assumed that machine maintenance must be performed within certain intervals. If a job is not fully processed before the maintenance, then an additional setup is required when the job is resumed. The two analyzed problems were minimizing the total weighted job completion time and minimizing the maximum lateness. Both problems were shown to be NP-hard in the case of a long planning horizon, and dynamic programming algorithms were proposed for solving them. It was also shown that for a short planning horizon, minimizing the total weighted completion time is NP-hard, while the total completion time problem and the maximum lateness problem can be solved in polynomial time.

Article [19] considered a preemptive scheduling problem with job release dates, delivery times and setup times. A job-dependent setup time was incurred whenever a job was started or resumed. The objective was to minimize the maximum delivery time. It is known that such a scheduling problem is equivalent to that with due dates rather than delivery times, where the objective is to minimize the maximum lateness [15]. Thus, the results obtained for problem $1|pmtn, r_j, s_j|D_{max}$ in [19] can be also applied to problem $1|pmtn, r_j, s_j|L_{max}$. It was proved in [19] that problem $1|pmtn, r_j, s_j|D_{max}$ is strongly NP-hard even if each setup time is one unit. Hence, our data gathering scheduling problem is also strongly NP-hard. Moreover, [19] proposed a dynamic programming algorithm and a PTAS for solving the analyzed problem. However, both algorithms were based on an observation that job preemptions need to occur only at the release times of other jobs. Unfortunately, as we will show in Sect. 4, this observation is not true, and in consequence, the algorithms are not correct.

In [20] the analyzed problem was to minimize the total weighted completion time in the same model of processing as in [19]. This problem was also proved to be strongly NP-hard, and a greedy algorithm with the worst-case performance guarantee of 25/16 was proposed.

Online minimization of the total flow time subject to job release dates and setup times was analyzed in [16]. It was proved that the problem is strongly NP-hard even if the setup time is one unit, and a heuristic algorithm was proposed. Later on, it was proved in [9] that no online algorithm for this problem can be competitive.

Scheduling problems with job setup times were also studied for the parallel machines setting (see, e.g., [24]) and in the flow shop environment [3]. They emerge in the context of resource constrained project scheduling [1,2] and real time systems [22,23,25,26]. An extensive survey of scheduling with setup times can be found in [4].

To conclude this section, let us indicate two special cases of our problem that boil down to well-known scheduling problems without setup times. For $S = 0$ our scheduling problem is equivalent to $1|pmtn, r_j|L_{max}$, and hence can be solved in $O(m \log m)$ time [17]. Moreover, if $S \neq 0$ but $C = 0$, then the problem is equivalent to $1|p_j = p, r_j|L_{max}$, and hence can be solved in $O(m^2 \log m)$ time [11].

4 Heuristic Algorithm

As it follows from [19] that our scheduling problem is strongly NP-hard, in this section we will construct a heuristic algorithm for solving it. We will first show that considering schedules with preemptions taking place only at dataset release times (as suggested in [19]) is not enough to obtain optimum solutions for all instances.

Proposition 1. *Constructing an optimum schedule for the data gathering problem with maximum lateness objective may require preempting a communication at a moment when no dataset is released.*

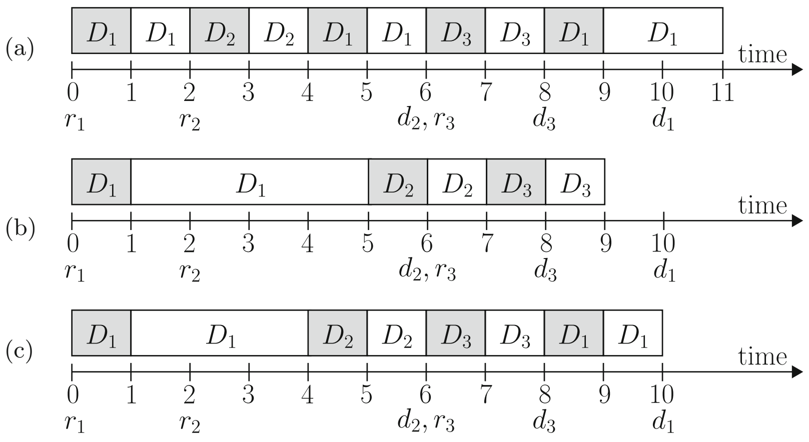

(a)
D_1 D_1 D_2 D_2 D_1 D_1 D_3 D_3 D_1 D_1
time
0 1 2 3 4 5 6 7 8 9 10 11
r_1 r_2 d_2, r_3 d_3 d_1
(b)
D_1 D_1 D_2 D_2 D_3 D_3
time
0 1 2 3 4 5 6 7 8 9 10
r_1 r_2 d_2, r_3 d_3 d_1
(c)
D_1 D_1 D_2 D_2 D_3 D_3 D_1 D_1
time
0 1 2 3 4 5 6 7 8 9 10
r_1 r_2 d_2, r_3 d_3 d_1

Fig. 1. Schedule structures for the proof of Proposition 1. Gray fields depict startup time, white fields depict data transfer. (a) D_1 preempted by D_2 at time $t = 2$, (b) D_1 not preempted by D_2, (c) D_1 preempted by D_2 at time $t = 4$.

Proof. Let $m = 3, S = C = 1$. Let the dataset parameters be $(\alpha_1, r_1, d_1) = (4, 0, 10)$, $(\alpha_2, r_2, d_2) = (1, 2, 6)$ and $(\alpha_3, r_3, d_3) = (1, 6, 8)$. If communications are preempted only at dataset release times, dataset D_1 can be preempted by D_2 either at time $t = 2$ or not at all. In both cases we obtain a schedule with $L_{max} = 1$ (cf. Fig. 1a and b). However, if dataset D_1 is preempted by D_2 at time $t = 4$, we obtain the optimum schedule with $L_{max} = 0$ (see Fig. 1c). □

It follows from Proposition 1 that the approach from [19] cannot be used to construct an exact algorithm for our data gathering problem, and that it does not always lead to obtaining optimum solutions for problem $1|pmtn, r_j, s_j|D_{max}$.

Let us now prove an intuitive observation that preempting dataset D_i by D_j can be beneficial only if $d_j < d_i$.

Proposition 2. *For any instance of the data gathering problem with maximum lateness objective there exists an optimum schedule such that for each pair of indices i, j, such that dataset D_i is preempted by D_j, we have $d_j < d_i$.*

Proof. Suppose that there exists an optimum schedule Σ in which dataset D_i is preempted by D_j and $d_j \geq d_i$. Let x_i, x_j denote the amounts of data from datasets D_i and D_j, correspondingly, transferred in the two messages forming the preemption. Let y_i be the size of the next message containing data from D_i. We will transform schedule Σ into a new schedule Σ' by increasing the amount of data from D_i sent just before the preemption by $\min\{y_i, x_j + \frac{S}{C}\}$. As the communication startup is non-preemptive and has to be performed directly before dataset transfer, we distinguish three cases: $y_i < x_j$ or $x_j \leq y_i \leq \frac{S}{C} + x_j$, or $\frac{S}{C} + x_j < y_i$. Figure 2 shows how to rearrange the communications in each case. The analyzed preemption is either deleted or delayed (if the message of size y_i was not the last part of D_i) in schedule Σ'. Let L_k and L'_k denote the

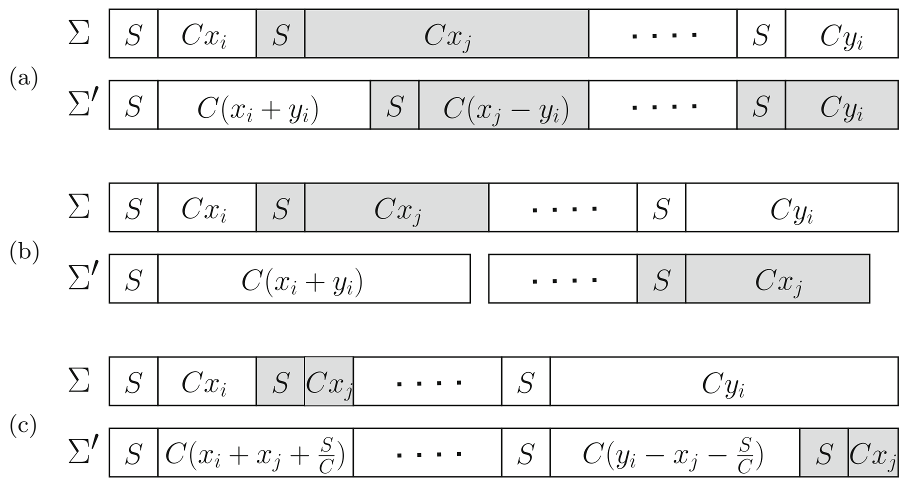

(a)
Σ
S | Cx_i | S | Cx_j | .... | S | Cy_i
Σ'
S | $C(x_i + y_i)$ | S | $C(x_j - y_i)$ | .... | S | Cy_i
(b)
Σ
S | Cx_i | S | Cx_j | .... | S | Cy_i
Σ'
S | $C(x_i + y_i)$ | .... | S | Cx_j
(c)
Σ
S | Cx_i | S | Cx_j | .... | S | Cy_i
Σ'
S | $C(x_i + x_j + \frac{S}{C})$ | .... | S | $C(y_i - x_j - \frac{S}{C})$ | S | Cx_j

Fig. 2. Schedule modifications for the proof of Proposition 2. Σ is the original schedule, Σ' is the modified schedule. White and gray fields depict sending datasets D_i and D_j, correspondingly. (a) Case 1: $y_i < x_j$, (b) Case 2: $x_j \leq y_i \leq \frac{S}{C} + x_j$, (c) Case 3: $\frac{S}{C} + x_j < y_i$.

lateness of dataset D_k in schedules Σ and Σ', respectively. We have $L'_k = L_k$ for $k \neq i, j$, and $L'_i \leq L_i$. Moreover, $L'_j \leq \max\{T_i - d_j, T_j - d_j\} \leq \max\{L_i, L_j\}$. Hence, Σ' is an optimum schedule. This procedure can be applied iteratively until all preemptions that do not have the required property are removed. □

On the one hand, preempting communications when new datasets with small due dates are released, may be necessary to construct a good schedule. On the other hand, each preemption extends the total communication time by S, which may damage the solution quality. Therefore, we propose algorithm **DS** which aims at finding some balance between the due date differences and the communication startup costs. As suggested by Proposition 2, the algorithm preempts transferring dataset D_i only when a dataset D_j with a smaller due date is released, and the difference between due dates d_i and d_j is large enough in comparison to startup time S. In more detail, algorithm DS consists in the following scheduling rules.

1. If the communication network is idle and some datasets are available (i.e. already released but not fully transferred to the base station), we start sending an available dataset with the smallest due date.
2. If dataset D_i is being transferred at time r_j, then it is preempted by D_j if $d_j < d_i - \gamma S$, where $\gamma \in [0, \infty]$ is a tunable algorithm parameter. If several datasets are released at time r_j and could preempt D_i according to this rule, we choose the one with the smallest due date.

Parameter γ controls the balance between the due dates and the communication startup times. When γ is small, more communication preemptions occur.

When γ is big, a dataset may be preempted only by a dataset with a much smaller due date. In particular, for $\gamma = 0$ a preemption is always made if $d_j < d_i$, i.e. algorithm DS follows the preemptive EDD rule [17]. When $\gamma = \infty$, no preemptions take place, and algorithm DS is equivalent to the non-preemptive EDD algorithm [18]. In the following text, algorithm DS with a given value of γ will be denoted by DS(γ).

Since algorithm DS preempts communications only at dataset release times, it introduces at most $m - 1$ preemptions, and the total number of messages is at most $2m - 1$. Thus, the algorithm can be implemented to run in $O(m \log m)$ time.

5 Computational Experiments

In this section we analyze the performance of algorithm DS for different values of parameter γ. Let us remind that this parameter controls the number of communication preemptions. The maximum number of preemptions introduced by algorithm DS is $m - 1$, and each preemption extends the total communication time by S. Therefore, we study the influence of parameters m and S on the quality of the obtained schedules. The algorithm was implemented in C++ in Microsoft Visual Studio.

The test instances were constructed as follows. The total size of data to be gathered was always $V = 1\text{E}9$. The number of datasets m belonged to the set $\{10, 20, \ldots, 100\}$. In order to select particular dataset sizes, numbers β_j were chosen from $U[1, 2]$ (the uniform distribution in range $[1, 2]$) for $j = 1, \ldots, m$. The size of dataset D_j was set to $\alpha_j = \beta_j V / (\sum_{j=1}^{m} \beta_j)$. Thus, the greatest dataset was at most twice bigger than the smallest one. This construction reflects gathering a diversified, but not very unbalanced collection of datasets. The communication rate was $C = 1\text{E-}8$, which corresponds with the bitrate of Gigabit Ethernet. The experiments were repeated for $S \in \{1\text{E}{-3},\ 2\text{E}{-3},\ 5\text{E}{-3},\ 1\text{E}{-2}\}$. We present here the results obtained for $S = 1\text{E}{-2}$ and $S = 2\text{E}{-3}$, which represent a big and a small startup time, correspondingly. For given m and S, the minimum possible communication time $T = mS + CV$ was computed. The release times r_j were selected randomly from $U[0, \delta_r T]$, and the due dates d_j from $U[0, \delta_d T]$. In order to construct more and less demanding sets of instances, we tested all combinations of δ_r and δ_d such that $\delta_r, \delta_d \in \{0.25, 0.5, 0.99\}$. Due to lack of space, we report here only on the results obtained for $\delta_r = \delta_d = 0.5$ and for $\delta_r = 0.25, \delta_d = 0.99$. Algorithm DS uses parameter γ to make the decision whether or not to preempt dataset D_i by D_j, only if $d_j < d_i$. Therefore, in order to emphasize the influence of γ, we sorted the selected release times and due dates so that $r_1 \le r_2 \le \cdots \le r_m$ and $d_1 \ge d_2 \ge \cdots \ge d_m$. Thus, a newly released dataset always has a smaller due date than all previous datasets.

In order to assess the quality of the obtained solutions, for each instance we computed a lower bound *LoBo* on the maximum dataset lateness by solving problem $1|pmtn, r_j|L_{max}$ with job execution times $p_j = S + C\alpha_j$, job release times r_j and due dates d_j, for $j = 1, \ldots, m$. It follows from the construction of

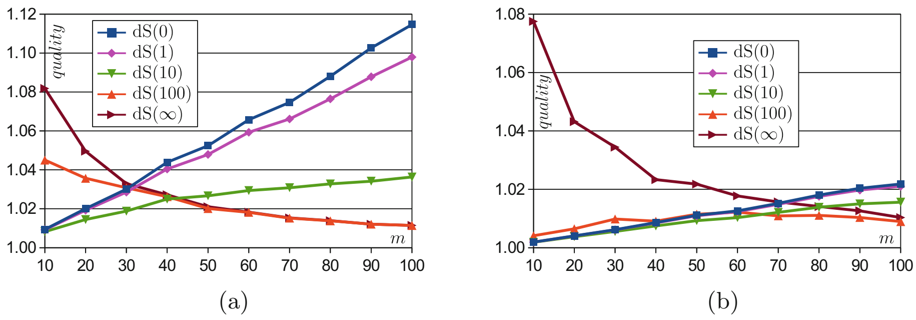
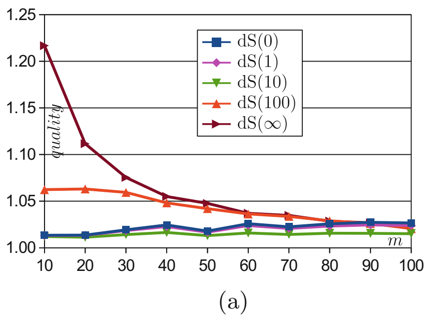


Fig. 3. Solution quality vs. m for $\delta_r = \delta_d = 0.5$. (a) $S = 1\text{E}{-2}$, (b) $S = 2\text{E}{-3}$.

our test instances that the last job in the above instance of $1|pmtn, r_j|L_{max}$ is completed after its due date, as $\sum_{j=1}^{m} p_j = mS + CV > \max_{j=1}^{m}\{d_j\}$. Hence, $LoBo > 0$, and the schedule quality can be measured by the ratio $L_{max}/LoBo$. As the maximum dataset lateness is to be minimized, a smaller number means better quality. The value of algorithm parameter γ was tuned experimentally. We chose to use $\gamma \in \{0, 1, 10, 100, \infty\}$. Each point in the following charts represents an average of 100 instances.

The results obtained for $\delta_r = \delta_d = 0.5$, $S = 1\text{E}{-2}$ and different values of m are shown in Fig. 3a. For small m, the best schedules are obtained for $\gamma = 10$. It seems that for $\gamma \leq 1$ the cost of preemptions is too big, whereas for $\gamma \geq 100$ the delays caused by waiting until previous communications finish are too high. When m gets bigger, the performance of the algorithm variants with $\gamma \leq 10$ gradually decreases. This is caused by the fact that the number of preemptions and hence, their total cost, increases with growing m. Contrarily, for $\gamma = 100$ and $\gamma = \infty$ the results get better with growing m. Indeed, when the fixed amount of data V is divided between more processors, dataset transfer times are shorter, and waiting for the end of some communication takes less time. Therefore, the delays in sending datasets with the smallest due dates are shorter than in the case of small m. All in all, in this experiment configuration, the best results are obtained for $\gamma = 10$ when $m \leq 40$ and for $\gamma = 100$ when $m \geq 50$.

Figure 3b presents the results obtained for small startup time, $S = 2\text{E}{-3}$. In most cases the influence of m on the performance of the algorithms is similar as in the previous chart. However, as for finite γ the value of γS is now smaller than for $S = 1\text{E}{-2}$, the differences between the algorithms with $\gamma < \infty$ become less significant. In particular, the performance of DS(100) is comparable to the variants with $\gamma \leq 10$ for small m. As the cost of additional startups is now lower, the performance of all algorithms with $\gamma < \infty$ improves in comparison to the case with $S = 1\text{E}{-2}$. The value of S has no influence on the behavior of DS(∞), which can be confirmed by comparing Fig. 3a and b. A slightly better quality reported for DS(∞) when $S = 2\text{E}{-3}$ is caused by the fact that when S is smaller, the lower bound $LoBo$ is closer to the actual optimum L_{max}. When the startup time is small, it is profitable to make a lot of preemptions for greater



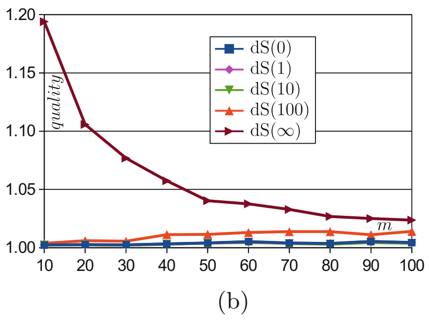


Fig. 4. Solution quality vs. m, for $\delta_r = 0.25, \delta_d = 0.99$. (a) $S = 1\mathrm{E}{-2}$, (b) $S = 2\mathrm{E}{-3}$.

values of m than in the case of a big startup time. Thus, algorithm DS(10) wins here for $m \leq 60$, and DS(100) for $m \geq 70$.

The results obtained for $\delta_r = 0.25, \delta_d = 0.99$ are depicted in Fig. 4. Decreasing δ_r means that the datasets are available earlier, and increasing δ_d that the due dates are bigger. Hence, these instances may be considered easier than those with $\delta_r = \delta_d = 0.5$. However, the results improve only for the algorithms that create many preemptions. The algorithms that introduce a small number of preemptions (DS(100) and DS(∞) for $S = 1\mathrm{E}{-2}$, and DS(∞) for $S = 2\mathrm{E}{-3}$) perform much worse than for $\delta_r = \delta_d = 0.5$. This can be explained in the following way. Since all datasets are released rather early, and the first datasets have quite big due dates, the algorithm should concentrate on transferring the last datasets (with the largest release times and the smallest due dates) as soon as possible. This can only be done if a sufficient number of previous datasets are preempted. Algorithm DS(∞), which does not allow preemptions, starts sending the last datasets too late, which damages its performance. Algorithm DS(10) wins for all combinations of m and S when $\delta_r = 0.25, \delta_d = 0.99$.

6 Conclusions

In this work we studied scheduling data gathering with maximum dataset lateness objective. As the problem is computationally hard, we proposed a heuristic algorithm DS(γ) running in $O(m \log m)$ time. The quality of schedules generated by this algorithm for different values of parameter γ was tested experimentally. The algorithm performance depends on the number of datasets m and the communication startup time S. If m is small, then the algorithm obtains better results for smaller γ. If both m and S are big, the algorithm variants with big γ perform better. The quality of the schedules is also influenced by the ranges of dataset release times r_j and due dates d_j. For all analyzed experiment settings, the best results were obtained either by algorithm DS(10) or DS(100), depending on the instance parameters. It is worth noting that for each tested setting, the average relative error of the solutions delivered by the algorithm with the best value of γ was below 3%.

Future research in this area may include a more detailed experimental analysis of the algorithm behavior, e.g. for different distributions of dataset sizes, release times and due dates. In particular, an important question is how to choose γ in general, in order to obtain good results for various types of problem instances. Theoretical performance guarantees of algorithm DS should also be investigated.

Acknowledgements. This research was partially supported by the National Science Centre, Poland, grant 2016/23/D/ST6/00410.

References

1. Afshar-Nadjafi, B.: Resource constrained project scheduling subject to due dates: preemption permitted with penalty. Adv. Oper. Res. **2014**, 10 p. (2014)
2. Afshar-Nadjafi, B., Majlesi, M.: Resource constrained project scheduling problem with setup times after preemptive processes. Comput. Chem. Eng. **69**, 16–25 (2014)
3. Allahverdi, A.: Minimizing mean flowtime in a two-machine flowshop with sequence-independent setup times. Comput. Oper. Res. **27**, 111–127 (2000)
4. Allahverdi, A., Ng, C.T., Cheng, T.C.E., Kovalyov, M.Y.: A survey of scheduling problems with setup times or costs. Eur. J. Oper. Res. **187**, 985–1032 (2008)
5. Berlińska, J.: Communication scheduling in data gathering networks with limited memory. Appl. Math. Comput. **235**, 530–537 (2014)
6. Berlińska, J.: Scheduling for data gathering networks with data compression. Eur. J. Oper. Res. **246**, 744–749 (2015)
7. Berlińska, J.: Scheduling data gathering with variable communication speed. In: Proceedings of the First International Workshop on Dynamic Scheduling Problems, pp. 29–32 (2016)
8. Bharadwaj, V., Ghose, D., Mani, V., Robertazzi, T.G.: Scheduling Divisible Loads in Parallel and Distributed Systems. IEEE Computer Society Press, Los Alamitos (1996)
9. Chan, H.-L., Lam, T.-W., Li, R.: Online flow time scheduling in the presence of preemption overhead. In: Gupta, A., Jansen, K., Rolim, J., Servedio, R. (eds.) APPROX/RANDOM 2012. LNCS, vol. 7408, pp. 85–97. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32512-0_8
10. Choi, K., Robertazzi, T.G.: Divisible load scheduling in wireless sensor networks with information utility. In: IEEE International Performance Computing and Communications Conference 2008, IPCCC 2008, pp. 9–17 (2008)
11. Condotta, A., Knust, S., Shakhlevich, N.V.: Parallel batch scheduling of equal-length jobs with release and due dates. J. Sched. **13**, 463–477 (2010)
12. Drozdowski, M.: Scheduling for Parallel Processing. Springer, London (2009). https://doi.org/10.1007/978-1-84882-310-5
13. Graham, R.L., Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Ann. Discret. Math. **5**, 287–326 (1979)
14. Graves, G.H., Lee, C.Y.: Scheduling maintenance and semiresumable jobs on a single machine. Nav. Res. Logist. **46**, 845–863 (1999)
15. Hall, L.A., Shmoys, D.B.: Jackson's rule for single-machine scheduling: making a good heuristic better. Math. Oper. Res. **17**, 22–35 (1992)

16. Heydari, M., Sadjadi, S.J., Mohammadi, E.: Minimizing total flow time subject to preemption penalties in online scheduling. Int. J. Adv. Manuf. Technol. **47**, 227–236 (2010)
17. Horn, W.A.: Some simple scheduling algorithms. Nav. Res. Logist. Q. **21**, 177–185 (1974)
18. Jackson, J.R.: Scheduling a production line to minimize maximum tardiness. Research Report 43, Management Sciences Research Project, UCLA (1955)
19. Liu, Z., Cheng, T.C.E.: Scheduling with job release dates, delivery times and preemption penalties. Inf. Process. Lett. **82**, 107–111 (2002)
20. Liu, Z., Cheng, T.C.E.: Minimizing total completion time subject to job release dates and preemption penalties. J. Sched. **7**, 313–327 (2004)
21. Moges, M., Robertazzi, T.G.: Wireless sensor networks: scheduling for measurement and data reporting. IEEE Trans. Aerosp. Electron. Syst. **42**, 327–340 (2006)
22. Nasri, M., Nelissen, G., Fohler, G.: A new approach for limited preemptive scheduling in systems with preemption overhead. In: 2016 28th Euromicro Conference on Real-Time Systems (ECRTS), pp. 25–35 (2016)
23. Phavorin, G., Richard, P.: Cache-related preemption delays and real-time scheduling: a survey for uniprocessor systems. Technical report, Laboratoire d'Informatique et d'Automatique pour les Systemes (2015)
24. Schuurman, P., Woeginger, G.J.: Preemptive scheduling with job-dependent setup times. In: Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms, pp. 759–767 (1999)
25. Thekkilakattil, A., Dobrin, R., Punnekkat, S.: The limited-preemptive feasibility of real-time tasks on uniprocessors. Real-Time Syst. **51**, 247–273 (2015)
26. Ward, B.C., Thekkilakattil, A., Anderson, J.H.: Optimizing preemption-overhead accounting in multiprocessor real-time systems. In: Proceedings of the 22nd International Conference on Real-Time Networks and Systems, pp. 235–246 (2014)

Fair Scheduling in Grid VOs with Anticipation Heuristic

Victor Toporkov[1](✉), Dmitry Yemelyanov[1], and Anna Toporkova[2]

[1] National Research University MPEI,
ul. Krasnokazarmennaya, 14, Moscow 111250, Russia
ToporkovVV@mpei.ru, YemelyanovDM@mpei.ru

[2] National Research University Higher School of Economics,
ul. Myasnitskaya, 20, Moscow 101000, Russia
atoporkova@hse.ru

Abstract. In this work, a job-flow scheduling approach for Grid virtual organizations (VOs) is proposed and studied. Users' and resource providers' preferences, VOs internal policies along with local private utilization impose specific requirements for scheduling according to different, usually contradictive, criteria. We study the problem of a fair job batch scheduling with a relatively limited resources supply. With increasing resources utilization level the available resources set and corresponding decision space are reduced. The main problem is a scarce set of job execution alternatives which eliminates scheduling optimization. In order to improve overall scheduling efficiency we propose a heuristic anticipation approach. It generates a reference, most likely infeasible, scheduling solution. A special replication procedure performs a feasible solution with a minimum distance to a reference alternative under given metrics.

Keywords: Scheduling · Grid · Resources · Utilization · Heuristic Job batch · Virtual organization · Anticipation · Replication

1 Introduction and Related Works

Virtual organization (VO) formation and performance largely depends on mutually beneficial collaboration between all the related stakeholders [1,2]. Application level scheduling [3] is based on the available resources utilization and, as a rule, does not imply any global resource sharing or allocation policy. Job flow scheduling [4] in user's VOs supposes uniform rules of resource sharing and consumption, in particular based on economic models [1–3,5]. The VO scheduling problems may be formulated as follows: to optimize users' criteria or utility function for selected jobs [5,6], to keep resource overall load balance [7,8], to have job run in strict order or maintain job priorities [9], to optimize overall scheduling performance by some custom criteria [10,11], etc. However, users', resource owners' and administrators' preferences may conflict with each other. Users are likely to be interested in the fastest possible running time for their jobs with least possible costs whereas VO preferences are usually directed to available resources load balancing or node owners' profit boosting. Thus, the most important aspect of rules suggested by VO is their fairness. A number of works understand fairness as it is defined in the theory of cooperative games, such as fair job flow distribution [8], fair quotas [12,13], fair user jobs prioritization [9], and non-monetary distribution [14].

The cyclic scheduling scheme (CSS) [15] implements a fair scheduling optimization which ensures stakeholders interests to some predefined extent. Scheduling of a job flow using CSS is performed in time cycles, by job batches. The actual scheduling procedure consists of two main steps. The first step involves a search for alternative scenarios of each job execution, or simply alternatives [16]. The launch of any job requires a co-allocation of a specified number of slots, as well as in the classic backfilling variation [17]. A single slot is a time span that can be assigned to run a part of a parallel job. The target is to scan a list of available slots and to select a window of parallel slots for the required resource reservation time. The user job requirements are arranged into a resource request containing the resource reservation time, characteristics of computational nodes (clock speed, RAM volume, disk space, operating system etc.), limitation on the selected window maximum cost. Window search algorithms were discussed in [16]. During the second step the dynamic programming methods [15] are used to choose an optimal alternatives' combination. One alternative is selected for each job with respect to the given VO and user criteria. The downside of centralized metascheduling approaches, including CSS [4,13,15], is that they lose their efficiency and optimization features in distributed environments with a limited resources supply. For example, the traditional backfilling [17] provides better scheduling outcome when compared to different optimization approaches in resource domain with a minimal performance configuration [2].

Main contribution of this paper is a heuristic CSS-based job-flow scheduling approach which retains efficiency even in distributed computing environments with limited resources. A special replication procedure is proposed and studied to ensure a feasible scheduling result. The rest of the paper is organized as follows. Section 2 presents a problem statement for cyclic fair scheduling. The proposed heuristic-based scheduling technique is presented in Sect. 3. Section 4 contains simulation experiment setup and results for the proposed scheduling approach. Finally, Sect. 5 summarizes the paper.

2 Problem Statement for Cyclic Fair Scheduling

Cyclic scheduling is based on a hierarchical job-flow management scheme [15]. Job-flow scheduling is performed in cycles by separate job batches on the basis of dynamically updated local schedules of computational nodes.

Let S_i be the family of appropriate sets of slots for executing job i, $i = 1, \dots, n$, in the batch, $s_j \in S_j$ be the set of slots that are appropriate by the resource request, the cost $c_i(s_j)$ and the execution time $t_i(s_j)$,

$j = 1, \dots, N, N = |\bigcup_{i=1}^{n} S_i|$. Denote by S the family of appropriate sets of slots and by $\overline{s} = (s_1, \dots, s_n)$, $\overline{s} \in S$, the sequence, which we call the combination of slots, for executing the batch of jobs. Let $f_i(s_j)$ be a function determining the efficiency of executing job i in the batch on the set of slots s_j under the admissible expenses specified by the function $g_i(s_j)$. For example, $f_i(s_j) = c_i(s_j)$ is the price of using the set s_j for the time $g_i(s_j) = t_i(s_j)$. The expenses are admissible if $g_i(s_j) \le g_i \le g^*$, where g_i is the level of the total expenses for the execution of a part of jobs from the batch (for example, jobs $i, i+1, \dots, n$ or $i, i-1, \dots, 1$) and g^* is the restriction for the entire set of jobs (in particular, the restriction on a total time t^* of slot occupation or a limitation on a VO budget b^*).

Formally, the statement of the problem of the optimal choice of a slot combination $\overline{s} = (s_1, \dots, s_n)$ is as follows:

$$\operatorname*{extr}_{\overline{s} \in S} f(\overline{s}) = \operatorname*{extr}_{s_j \in S_i} \sum_{i=1}^{n} f_i(s_j), g_i(s_j) \le g_i \le g^*, g^* = \sum_{i=1}^{n} g_i^0(s_j), \tag{1}$$

where $g_i^0(s_j)$ is the resource expense level function of executing the batch.

The recurrences for finding the extremum of the criterion in (1) for the set of slots $s_j \in S_i$, $i = \overline{1,n}, j \in \{1, \dots, N\}$ based on backward recursion are

$$\begin{aligned} &f_i(g_i) = \operatorname*{extr}_{s_j \in S_j} \{f_i(s_j) + f_{i+1}(g_i - g_i(s_j))\}, g_i(s_j) \le g_i \le g^*, i = \overline{1,n}, \\ &f_{n+1}(g_{n+1}) = 0, g_i = g_{i-1} - g_{i-1}(s_k), 1 < i \le n, g_1 = g^*, s_k \in S_{i-1}, \end{aligned} \tag{2}$$

where g_i are the total expenses (utilization time or cost) for using the slots for jobs $i, i+1, \dots, n$ of this batch. The optimal expenses are determined from the equation

$$g_i^*(s_j) = \operatorname*{arg\,extr}_{g_i(s_j) \le g_i} f_i(g_i), i = \overline{1,n}. \tag{3}$$

The optimal set of slots $s_i^* \in \{1, \dots, N\}$ in the scheme (2), (3) is given by the relation

$$s_i^*(s_j) = \operatorname*{arg\,extr}_{s_j \in S_i} f_i(g_i^*(s_j)), i = \overline{1,n}. \tag{4}$$

Here (4) represents the solution of the problem (1). An example of a resource expense level function in (1) is $t_i^0(s_j) = [\sum_{s_j} t_i(s_j)/l_i]$, where l_i is the number of admissible (alternative) sets of slots for the execution of job i, [.] is the ceiling of $t_i^0(s_j)$. Then the constraint on the total time of slot occupation in the current scheduling cycle can have the form

$$t^* = \sum_{i=1}^{n} t_i^0(s_j). \tag{5}$$

Let us consider several problems of practical importance. Suppose it is required to select a set of slots for executing a batch of n jobs so as to maximize the total cost of resource utilization $f_i(t_i) = \max_{s_j \ inS_i} \{c_i(s_j) + f_{i+1}(t_i - t_i(s_j))\}$, $i = 1, \dots, n$, $f_{n+1}(t_{n+1}) = 0$ (maximization of profit of resource owners under restrictions on the total time of slot utilization). The restriction on the total time of using slots by all the jobs is given by (5). Minimization of the total completion time of a batch of jobs under a restriction on the budget b^* of the virtual organization: $f_i(c_i) = \min_{s_j \in S_i}\{t_i(s_j)+f_{i+1}(c_i-c_i(s_j))\}$, $i = 1,\dots,n$, $f_{n+1}(c_{n+1}) = 0$. An example for a user scheduling criterion Z may be an overall job running time, an overall running cost, etc. This criterion describes user's preferences for that specific job execution and expresses a type of an additional optimization to perform when searching for alternatives. Alongside with time $T = \sum_{i=1}^{n} t_i(s_j)$ and cost $C = \sum_{i=1}^{n} c_i(s_j)$ properties each job execution alternative has a user utility (U) value: user evaluation against the scheduling criterion. We consider the following relative approach to represent a user utility U. A job alternative with the minimum (best) user-defined criterion value Z_{min} corresponds to the left interval boundary ($U = 0\%$) of all possible job scheduling outcomes. An alternative with the worst possible criterion value Z_{max} corresponds to the right interval boundary ($U = 100\%$). In the general case, for each alternative with value Z, U is set depending on its position in $[Z_{min}; Z_{max}]$ interval using the following formula: $U = \frac{Z-Z_{min}}{Z_{max}-Z_{min}} * 100$. Thus, each alternative gets its utility in relation to the *best* and the *worst* optimization criterion values user could expect according to the job's priority. A common VO optimization problem may be stated as either minimization or maximization of one of the properties, having other fixed or limited, or involve Pareto-optimal strategy search involving both kinds of properties [1,15,18].

The job batch scheduling performs consecutive allocation of multiple alternatives with nonintersecting slots for each job. Otherwise irresolvable collisions for resources may occur if different jobs will share the same time-slots. Sequential alternatives search and resources reservation procedures help to prevent such scenario. However in an extreme case when resources are limited or overutilized only at most one alternative execution could be reserved for each job. In this case the alternative-based scheduling will be no different from the First Fit resources allocation procedure [2]. In order to address this problem, we propose a heuristic job batch scheduling scheme discussed in the next section.

3 Anticipation-Based Heuristic Fair Scheduling

The proposed approach consists of three main steps. First, a set of all possible execution alternatives is found for each job not considering time slots intersections and without any resources reservations. The resulting alternatives found for each job reflect a full range of different job execution possibilities user may expect on the current scheduling interval. Second, CSS procedure (1–4) is performed to select the alternatives combination optimal according to VO policy. The resulting combination most likely corresponds to an infeasible scheduling solution as possible time slots intersection will cause collisions on the resources allocation stage. The main idea of this step is that obtained infeasible solution will provide some heuristic insights on how each job should be handled during the scheduling. Third, a feasible resources allocation is performed by replicating alternatives selected in step 2. The base for this replication step is an **A**lgorithm searching for **E**xtreme **P**erformance (AEP) described in details in [16]. In the current step AEP helps to find and reserve feasible execution alternatives most similar to those selected in the reference optimal solution (4). After these three steps are performed the resulting solution is both feasible and efficient as it reflects scheduling pattern obtained from the reference solution from step 2.

```
Data: slotList - a list of available slots ordered by start time; job - a job for
      which the search is performed; refAlternative – reference alternative
      used to find similar job execution window.
Result: closestWindow – execution window similar to refAlternative
minDistance = MAX_VALUE;
for each slot in slotList do
    windowSlotList.add(slot);
    windowStartTime = slot.startTime;
    for each wSlot in windowSlotList do
        minLength = wSlot.node.getWorkingTimeEstimate();
        if (wSlot.endTime - windowStartTime) < minLength then
            windowSlotList.remove(wSlot);
        end
    end
    if windowSlotList.size() ≥ job.nodesNeed then
        distance = calculateDistance(windowSlotList, refAlternative);
        if distance < minDistance then
            minDistance = distance;
            closestWindow = windowSlotList;
        end
    end
end
```

Algorithm 1. AEP modification to allocate a set of job execution alternatives

We use AEP modification to allocate a diverse set of execution alternatives for each job. Originally AEP scans through a whole list of available time slots and retrieves one alternative execution satisfying user resource request and optimal according to user custom criterion. During this scan, we save all intermediate AEP search results (2) to a dedicated list of possible alternatives. For the replication purpose a new Execution Similarity criterion was introduced which helps AEP to find a window with a minimum distance to a reference alternative. Generally, we define a distance between two different alternatives as a relative difference or *error* between their significant criteria values. For example if the reference alternative has C_{ref} total cost, and some candidate alternative cost is C_{can}, then the relative cost error E_C is calculated as $E_C = \frac{|C_{ref} - C_{can}|}{C_{ref}}$. If one need to consider several criteria the *distance* D between two alternatives may be calculated as a linear sum of criteria errors $D_l = E_C + E_T + ...E_U$ or as a geometric distance in a parameters space $D_g = \sqrt{E_C^2 + E_T^2 + ... + E_U^2}$.

AEP modification with Execution Similarity criterion is represented as Algorithm 1. In this algorithm an expanded window *windowSlotList* of size M moves through a whole list of all available slots *slotList* sorted by their start time in ascending order. At each step any combination of m slots inside *windowSlotList* (in the case, when $m \leq M$) can form a window that meets all the requirements to run the job.

The main difference from the original AEP is that instead of searching for a window with an extreme criterion value (1), we retrieve one of the windows according to (2) with the minimum distance D_l or D_g to the reference alternative.

For a feasible job batch resources allocation AEP consequentially allocates for each job a single window with the minimum distance to the reference alternative from an infeasible solution. Time slots allocated for i-th job are reserved and excluded from the slot list when AEP search algorithm is performed for the following jobs $i+1, i+2, \ldots$, etc. Thus this procedure prevents any conflicts for resources and provides scheduling which in some sense reflects a near-optimal solution.

4 Simulation Study

An experiment was prepared as follows using a custom distributed environment simulator [2,15,16]. For our purpose, it implements a heterogeneous resource domain model: nodes have different usage costs and performance levels. A space-shared resources allocation policy simulates a local queuing system (like in GridSim or CloudSim [19]) and, thus, each node can process only one task at any given simulation time. The execution cost of each task depends on its execution time which is proportional to the dedicated node's performance level. The execution of a single job requires parallel execution of all its tasks.

During each experiment new instances for the computing environment and the job flow are automatically generated. VO and computing environment generation properties used for the simulation are the following. The resource pool includes 80 heterogeneous computational nodes. Each nodes performance level is given as a uniformly distributed random value in the interval [2,10]. A base cost of a node is an exponential function of its performance value, so any two nodes of the same resource type and performance have the same base cost. Effective node cost during the scheduling interval is then calculated by adding a variable distributed normally as ± 0.6 of a base cost, simulating discounts or extra charges up to 60%. The scheduling interval length is 800 time quanta. The initial resource load with owner jobs is distributed hyper-geometrically resulting in 5% to 10% utilization on each node.

Job batch properties are as follows. Jobs number in a batch is 125. Nodes quantity required simultaneously for a job execution is a whole number distributed evenly on [2; 6]. Node reservation time is a whole number distributed evenly on [100; 500]. Job budget varies in the way that some of jobs can pay as much as 160% of base cost whereas some may require a discount. Every request contains a specification of a custom user criterion which is one of the following: minimization of job execution runtime or job execution cost.

The selected parameters for computing environment are generally suitable to execute the whole job-flow during the simulation cycle, but not much more The generated resources are heterogeneous in terms of performance and cost, which increases fair scheduling efficiency [2]. At the same time we make sure that resource domain does not contain nodes with performance differing by more than an order of magnitude. The source code for the simulator as well as for the current experiment setup is freely available at https://github.com/dmieter/mimapr.

The main feature of the current simulation is to study and compare scheduling results provided by CSS, Anticipation and Backfilling algorithms. We are especially interested in integral job-flow execution criteria (such as an average finish time), as well as compliance with user's individual criteria.

4.1 Replication Scheduling Accuracy

The first experiment is dedicated to a replication scheduling accuracy study. First, a general CSS was performed in each experiment for the following job-flow execution cost maximization problem $C \rightarrow max$, $\lim U_a = 10\%$. U_a stands for the average user utility for one job, i.e. $\lim U_a = 10\%$ means that at average resulting deviation from the best possible outcome for each user did not exceed 10%. Next, linear and geometric replication algorithms were executed to replicate CSS solution using linear D_l and geometric D_g distances. In the current experiment we used job execution cost error E_C and processor time usage error E_t to calculate distances D_l and D_g.

Table 1. CSS replication average scheduling results

Job execution characteristic	$C \rightarrow max$, $U_a \leq 0\%$	$C \rightarrow max$, $U_a \leq 10\%$	Linear replication	Geometric replication	$C \rightarrow max$, $U_a \leq 100\%$
Cost	1283	1349	1353	1353	1475
Processor time	191.6	191.2	190.6	190.5	202.3
Finish time	367.1	353.8	356.2	356.4	358.5
$U_a, \%$	0	9.9	17.6	17.8	65

In order to evaluate the resulting difference in scheduling outcomes, we additionally performed CSS algorithm for $C \rightarrow max$, $\lim U_a = 0\%$ (ensuring users' individual preferences only) and $C \rightarrow max$, $\lim U_a = 100\%$ (ensuring VO preference, i.e. maximizing overall cost without taking into account users' criteria) problems. These additional problems reflect extreme boundaries for scheduling results, which can be used to evaluate a relative replication error. Table 1 contains scheduling results for all these three problems and two replication algorithms. To address this discrepancy in more details Fig. 1 shows average linear and geometric replication distances for each job of the batch. There values are practically independent from an ordinal job number and do not exceed 0.05. For comparison average distances between the most and the least expensive alternative executions for the first batch job amounted as follows: 1.15 for linear and 0.88 for geometric metrics.

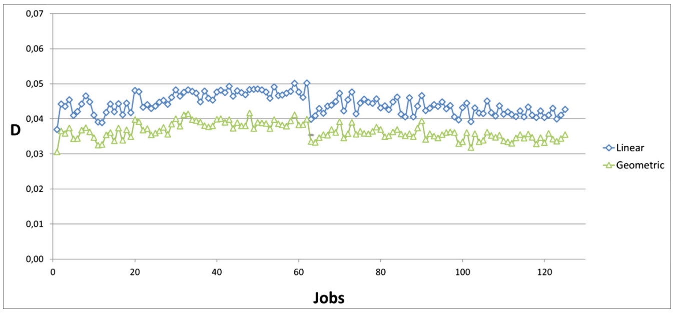


Fig. 1. Average replication error for user jobs

4.2 Anticipation Scheduling Simulation

The second experiment series consider anticipation scheduling efficiency. During each experiment a VO domain and a job batch were generated and the following scheduling schemes were simulated and studied. First, a general CSS solved the optimization problems $T \rightarrow min$, $\lim U$ with different limits $U_a \in \{0\%, 1\%, 4\%, 10\%, 16\%, 32\%, 100\%\}$. Second, a near-optimal but infeasible reference solution REF was obtained for the same problems. Third, a replication procedure CSSrep was performed based on CSS solution to demonstrate a replication process accuracy. For the heuristic anticipation scheduling ANT the same replication procedure was performed based on REF solution. We used a geometric distance as a replication criterion. Finally, two independent job batch scheduling procedures were performed to find scheduling solutions most suitable for VO users (USERopt) and VO administrators (VOopt).

An average number of alternatives found for a job in CSS was 2.6. This result shows that while for relatively small jobs usually a few alternative executions have been found, large jobs usually had at most one possible execution option. At the same time REF algorithm at average considered more than 100 alternative executions for each job. CSS failed to find any alternative executions for at least for one job of the batch in 209 experiments; ANT - in 155 experiments. These results show that simulation settings at the same time provided quite a diverse job batch and a limited set of resources not allowing executing all the jobs during every experiment. Figure 2 shows average job execution time (VO criterion) in $T \rightarrow min$, $\lim U$ optimization problem. Different limits U_a specify to what extent user preferences were taken into account. Two horizontal lines USERopt and VOopt represent practical T values when only user or VO administration criteria are optimized correspondingly, i.e. USERopt was obtained by using only user criteria to allocate resources for jobs without taking into account VO preferences. VOopt was obtained by using one VO optimization criterion ($T \rightarrow min$) for each job scheduling without taking into account user preferences.

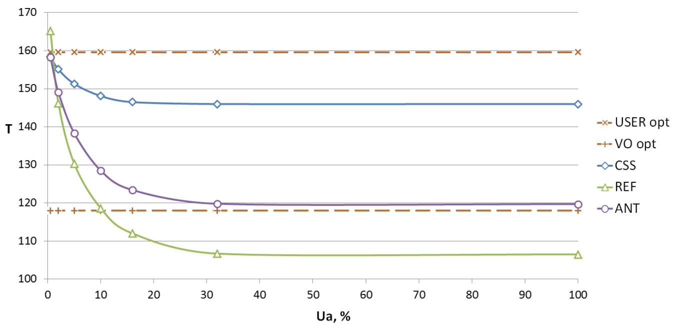


Fig. 2. Average job execution time in $T \rightarrow min$, lim U problem

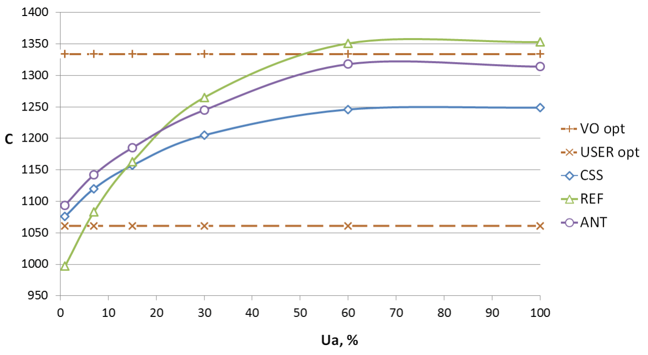


Fig. 3. Average job execution cost in $C \rightarrow max$, lim U problem

First thing that catches the eye in Fig. 2 is that REF for $U_a > 10\%$ provides job execution time value better (smaller) than those of VOopt. However such behavior is expected as REF generates an infeasible solution and may use time-slots from more suitable (according to VO preferences) resources several times for different jobs. Otherwise ANT provided better VO criterion value than CSS for all $U_a > 0\%$. The relative advantage reaches 20% when $U_a > 20\%$ is considered. ANT algorithm graph gradually changes from USERopt value at $U_a = 0\%$ to almost VOopt value at $U_a = 100\%$ just with changing average user utility limit. Thereby, ANT represents a general scheduling approach allowing balancing between VO or user criteria optimization.

A similar pattern can be observed in Fig. 3, where $C \rightarrow max$, $\lim U$ scheduling problem is presented. However, in this case ANT advantage over CSS amounts to 10% against VO criterion.

5 Conclusions and Future Work

In this paper, we study the problem of fair job batch scheduling with a relatively limited resources supply. The main problem arise is a scarce set of job execution alternatives which eliminates optimization efficiency.

We propose the heuristic scheme which generates an infeasible reference solution and then replicates it to allocate a feasible accessible solution. A special replication procedure is proposed which provides 2–5% error from the reference scheduling solution. The obtained results show that the new heuristic approach provides flexible and efficient solutions for different fair scheduling scenarios. Future work will be focused on replication algorithm study and its possible application to fulfill complex user preferences expressed in a resource request.

Acknowledgments. This work was partially supported by the Council on Grants of the President of the Russian Federation for State Support of Young Scientists and Leading Scientific Schools (grants YPhD-2297.2017.9 and SS-6577.2016.9), RFBR (grants 18-07-00456 and 18-07-00534), and by the Ministry on Education and Science of the Russian Federation (project no. 2.9606.2017/8.9).

References

1. Kurowski, K., Nabrzyski, J., Oleksiak, A., Weglarz, J.: Multicriteria aspects of grid resource management. In: Nabrzyski, J., Schopf, J.M., Weglarz, J. (eds.) Grid Resource Management. State of the Art and Future Trends, pp. 271–293. Kluwer Academic Publishers, Dordrecht (2003). https://doi.org/10.1007/978-1-4615-0509-9_18
2. Toporkov, V., Toporkova, A., Tselishchev, A., Yemelyanov, D., Potekhin, P.: Heuristic strategies for preference-based scheduling in virtual organizations of utility grids. J. Ambient Intell. Human. Comput. **6**(6), 733–740 (2015). https://doi.org/10.1007/s12652-015-0274-y
3. Buyya, R., Abramson, D., Giddy, J.: Economic models for resource management and scheduling in grid computing. Concurr. Comput. **14**(5), 1507–1542 (2002). https://doi.org/10.1002/cpe.690
4. Rodero, I., Villegas, D., Bobroff, N., Liu, Y., Fong, L., Sadjadi, S.M.: Enabling interoperability among grid meta-schedulers. J. Grid Comput. **11**(2), 311–336 (2013). https://doi.org/10.1007/s10723-013-9252-9
5. Ernemann, C., Hamscher, V., Yahyapour, R.: Economic scheduling in grid computing. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 128–152. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36180-4_8
6. Rzadca, K., Trystram, D., Wierzbicki, A.: Fair game-theoretic resource management in dedicated grids. In: IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2007), pp. 343–350. IEEE Computer Society, Rio De Janeiro (2007). https://doi.org/10.1109/ccgrid.2007.52

7. Vasile, M., Pop, F., Tutueanu, R., Cristea, V., Kolodziej, J.: Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. Future Gener. Comput. Syst. **51**, 61–71 (2015). https://doi.org/10.1016/j.future.2014.11.019
8. Penmatsa, S., Chronopoulos, A.T.: Cost minimization in utility computing systems. Concurr. Comput.: Pract. Exp. **16**(1), 287–307 (2014). https://doi.org/10.1002/cpe.2984
9. Mutz, A., Wolski, R., Brevik, J.: Eliciting honest value information in a batch-queue environment. In: 8th IEEE/ACM International Conference on Grid Computing, New York, USA, pp. 291–297 (2007). https://doi.org/10.1109/grid.2007.4354145
10. Blanco, H., Guirado, F., Lrida, J.L., Albornoz, V.M.: MIP model scheduling for multi-clusters. In: Caragiannis, I., et al. (eds.) Euro-Par 2012. LNCS, vol. 7640, pp. 196–206. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-36949-0_22
11. Takefusa, A., Nakada, H., Kudoh, T., Tanaka, Y.: An advance reservation-based co-allocation algorithm for distributed computers and network bandwidth on QoS-guaranteed grids. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2010. LNCS, vol. 6253, pp. 16–34. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16505-4_2
12. Carroll, T., Grosu, D.: Divisible load scheduling: an approach using coalitional games. In: Proceedings of the Sixth International Symposium on Parallel and Distributed Computing, ISPDC 2007, p. 36 (2007). https://doi.org/10.1109/ispdc.2007.16
13. Kim, K., Buyya, R.: Fair resource sharing in hierarchical virtual organizations for global grids. In: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, pp. 50–57. IEEE Computer Society, Austin (2007). https://doi.org/10.1109/grid.2007.4354115
14. Skowron, P., Rzadca, K.: Non-monetary fair scheduling cooperative game theory approach. In: Proceeding of SPAA 2013 Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 288–297. ACM, New York (2013). https://doi.org/10.1145/2486159.2486169
15. Toporkov, V., Yemelyanov, D., Bobchenkov, A., Tselishchev, A.: Scheduling in grid based on VO stakeholders preferences and criteria. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds.) Dependability Engineering and Complex Systems. AISC, vol. 470, pp. 505–515. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39639-2_44
16. Toporkov, V., Toporkova, A., Tselishchev, A., Yemelyanov, D.: Slot selection algorithms in distributed computing. J. Supercomput. **69**(1), 53–60 (2014). https://doi.org/10.1007/s11227-014-1210-1
17. Dimitriadou, S.K., Karatza, H.D.: Job scheduling in a distributed system using backfilling with inaccurate runtime computations. In: Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems, pp. 329–336 (2010). https://doi.org/10.1109/CISIS.2010.65
18. Farahabady, M.H., Lee, Y.C., Zomaya, A.Y.: Pareto-optimal cloud bursting. IEEE Trans. Parallel Distrib. Syst. **25**, 2670–2682 (2014). https://doi.org/10.1109/TPDS.2013.218
19. 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. Softw.: Pract. Exp. **41**(1), 23–50 (2011). https://doi.org/10.1002/spe.995

A Security-Driven Approach to Online Job Scheduling in IaaS Cloud Computing Systems

Jakub Gąsior[1](✉), Franciszek Seredyński[1], and Andrei Tchernykh[2]

[1] Department of Mathematics and Natural Sciences, Cardinal Stefan Wyszyński University, Warsaw, Poland
j.gasior@uksw.edu.pl

[2] CICESE Research Center, Ensenada, Baja California, Mexico

Abstract. The paper presents a general framework to study issues of multi-objective on-line scheduling in the Infrastructure as a Service model of Cloud Computing (CC) systems taking into account the aspects of the total work-flow execution cost while meeting the deadline and risk rate constraints. Our goal is providing fairness between concurrent job submissions by minimizing tardiness of individual applications and dynamically rescheduling them to the best suited resources. The system, via the scheduling algorithms, is responsible to guarantee the corresponding Quality of Service (QoS) and Service Level Agreement (SLA) for all accepted jobs.

Keywords: Cloud Computing · Service Level Agreement
Security-aware scheduling · Infrastructure as a Service

1 Introduction

The emerging CC paradigm represents an important architectural shift from the traditional distributed computing approaches [4,13]. From the users' point of view, CC moves the application software from their own devices to the Cloud side, which makes users able to plug-in anytime from anywhere and utilize large-scale storage and computing resources by dynamically expanding and contracting their demands with the natural flow of the involved business life-cycle. Specifically, end-users rent resources from Cloud infrastructure providers, according to a *pay-per-use* pricing model, to deploy specific service instances in the form of Virtual Machines (VMs) or runtime applications [13].

On the other hand, the providers are responsible for dispatching the execution of these instances on their physical resources. In doing this, they have to focus on how to automatically and dynamically distribute and schedule the involved tasks according to the current workloads experienced on their infrastructures. This elastic management aspect of CC platform allows applications to scale and grow without needing traditional upgrades.

The official commitment that prevails between a service provider and the customer defining aspects of the service such as quality, availability and responsibilities is specified in the SLA. SLAs can be extended to include provider and consumer responsibilities, bonuses and penalties, availability, conditions of services supporting, rules and exceptions, excess usage thresholds and charges, payment and penalty regulation, purchasing options, pricing policy, payment procedure, security and privacy issues, etc.

In addition to the optimal utilization of computing resources, security has become another critical concern for a wide range of applications on cloud computing systems [7–10]. Unfortunately, since distributed computing allows a vast number of users to execute a broad spectrum of unverified third-party applications, both applications and users can be sources of security threats to computing environments. However, many existing cloud computing environments have not employed any security measures to counter security threats [1].

Hence, there is an emergent need to exploit security services to protect security-critical applications from attacks during their operation in the cloud data center. However, adding the security services to applications inevitably incurs overhead in terms of computation time, which increase the makespan and operational cost of applications. In this study, we restrict ourselves to the SLA performance guarantees with the aim of minimizing the total workflow execution cost while meeting the deadline and risk rate constraints.

The remainder of this paper is organized as follows. In Sect. 2, we present the works related to the distributed scheduling and load balancing in distributed computing systems. In Sect. 3, we describe the problem definition, while Sect. 4 presents the proposed scheduling model. The experimental evaluation of the proposed approach is given in Sect. 5. We end the paper in Sect. 6 with some conclusions and indications for future work.

2 State of the Art

Distributed scheduling has been widely studied in the context of real-time systems, when jobs have deadline constraints. Among others, in [15] authors proposed a distributed algorithm to solve general constraint optimization problem with a guaranteed convergence using only localized, asynchronous communication between agents involved in this process.

We address in our work the problem of resource provisioning in environments with multiple CC nodes. Emerging deadline-driven applications require access to several resources and predictable *Quality of Service* (QoS) metrics. However, it is often difficult to provision resources to these applications because of the complexity of providing guarantees about the start or completion times of applications currently in execution or waiting in the resources' queues.

To complicate matters further, users commonly access resources from a Cloud via mediators such as brokers or gateways [3]. The design of gateways that provision resources to deadline-driven applications may be complex and prone to scheduling decisions that are far from optimal. Furthermore, a gateway representing a Cloud can have peering arrangements or contracts with other gateways through which they coordinate the resource provisioning. This complicates provisioning as a gateway needs not only to provision resources to its users, but also provision spare capacity to other gateways. Previous work has demonstrated how information about fragments in the scheduling queue of clusters, or free time slots, can be obtained from RMSs and provided to gateways to be provisioned to Grid and Cloud applications [11,19].

Due to the distributed nature of such systems, several concurrent jobs originating from different users are likely to compete for the resources. Traditionally, schedulers aim at minimizing the overall completion time of a job [2]. Closer to our problem, Viswanathan in [18] proposed a distributed scheduling strategy specifically designed to handle large volumes of computationally intensive and arbitrarily divisible workloads submitted for processing involving multiple sources and processing nodes. In [6] authors proposed a distributed scheduling solution ensuring a fair and efficient use of the available resources by providing a similar share of the platform to every application through stretch optimization.

Research on SLAs in CC has also addressed the usage of SLAs for resource management and admission control techniques, automatic negotiation protocols, economic aspects associated with the usage of SLAs for service provision. For example, in [23] authors explored the benefits of power optimization for distributed systems by turning off or on (activating/deactivating) servers so that only the minimum number of servers required to execute a given workload is kept active. A similar concept is used in [17] where authors analyzed the effects of virtual machine allocation on power consumption.

In another study [16] authors focused on multiobjective planning of cloud datacenters considering SLAs and power profiles. Their experimental analysis performed on realistic datacenters demonstrates that accurate schedules, accounting for different trade-offs between power, temperature and QoS, can be computed by combining a traditional NSGA-II multiobjective evolutionary algorithm with a backfilling technique to deal with sleeping/switched off computing resources.

3 Problem Formulation

In this section we formally define basic elements of the model and provide corresponding notation, its characteristics and the type of jobs to be scheduled. We follow the job, system and cost function model presented in [21,22]. We are interested in providing QoS guarantees defined in the SLA and optimizing both the provider income, while meeting the deadline and security constraints.

3.1 Cloud Datacenter Model

We assume that cloud data-center offers a set of VM instances $M_1, M_2, \ldots, M_m$ specified by several characteristics, including their processing capacity s_i in million floating point operations per second (MFLOPS), cost per hour c_i, memory and storage space. For the purpose of this paper we follow the specification of

Compute Optimized - Current Generation VM series provided by Amazon EC2 and shown Table 1. Prices are adjusted for Dedicated On-Demand Instances in EU region.

Table 1. Compute optimized dedicated on-demand VM instances in Amazon EC2.

	CPU (ECU)	Processing capacity (MFLOPS)	Cost per hour
c4.large	2 (8)	8 800	\$0.131
c4.xlarge	4 (16)	17 600	\$0.261
c4.2xlarge	8 (32)	35 200	\$0.524
c4.4xlarge	16 (62)	70 400	\$1.045
c4.8xlarge	36 (132)	140 800	\$1.902

VMs are charged based on its leasing time in unit of time (e.g., hour, minute, etc.), and partial time unit of use is rounded up to the next whole time unit. As an infinite amount of resources can be accessed in clouds, there is no limit on the number of VMs that can be executed in the workflow. Moreover, we assume that all VMs in the cloud datacenter have the same communication bandwidth.

3.2 Job, Security and Pricing Model

Individual users $(U_1, U_2, \ldots, U_n)$ submit to the system workflow application J_k^j for execution. Each application is the set of n tasks or jobs. Users are expected to pay appropriate fees to the Cloud provider dependent on the SLA requested.

Job (denoted as J_k^j) is jth job produced (and owned) by user U_k. J_k stands for the set of all jobs produced by user U_k, while $n_k = |J_k|$ is the number of such jobs. Each task has varied parameters defined as a tuple $<r_k^j, size_k^j, t_k^j, d_k^j>$, specifying its release dates $r_k^j \geq 0$; its size $1 \leq size_k^j \leq m_m$, that is referred to as its processor requirements or *degree of parallelism*; its workload t_k^j defined in MFLOPs and a deadline d_k^j.

In mapping jobs onto cloud resources, we have to tackle a number of security-related problems [14]. The first step is for a user to issue a Security Demand (SD) to all submitted jobs. When setting up the SD values, users should be concerned about issues related to job sensitivity, job execution environment, access control and data integration [20], etc. On the other hand, the Security Level (SL) of a machine can be attributed to the available intrusion detection mechanisms, firewalls and anti-virus capabilities, as well as prior job execution success rates. This defense capability is evaluating the risk existing in the allocation of a submitted job to a specific machine.

Thus, a job is expected to be successfully carried out when SD and SL satisfy a security-assurance condition ($SD \leq SL$) during the job mapping process. The SD is a real fraction in the range [0,1] with 0 representing the lowest and 1 the highest security requirement. The SL is in the same range with 0 for the most risky resource site and 1 for a risk-free or fully trusted site. Specifically, we define a *Job Failure Model* as a function of the difference between the job security demand and a resource trust (Eq. 1):

$$P_{i,j}^{Failure} = \begin{cases} 0, & SD_j \leq SL_i, \\ 1 - exp^{-(SD_j - SL_i)}, & SD_j > SL_i. \end{cases} \tag{1}$$

Meeting the security assurance condition ($SD_j \leq SL_i$) for a given job-machine pair guarantees successful execution of that particular job. Such a scheduling will be further called as a *Secure Job Allocation*. On the other hand, successful execution of the job assigned to machine without meeting this condition ($SD_j > SL_i$), will be dependent on the calculated probability and further referred to as a *Risky Job Allocation*.

3.3 Job Scheduling Problem Formulation

Once the job is released, the provider has to decide, before any other job arrives, whether the job is accepted or not. In the case of acceptance, later submitted jobs cannot cause job J_k^j to miss its deadline. If a deadline violation occurs, provider will be expected to refund $\alpha_{Penalty}^{Secure} = 90\%$ of costs to the customer in a case of the *Secure Job Allocation* policy and $\alpha_{Penalty}^{Risky} = 45\%$ of costs in a case of the *Risky Job Allocation* policy. Two related objectives are considered in this work:

- The minimization of the *SLA Violation Count*, defined as a weighted sum of user's U_k tardy jobs in the schedule S_k, that is $SLA_{Violation}^{Count} = \sum_{j=1}^{n_k} w_j * D_k^j$. The weight coefficient w_j is equal to 0.25 for jobs with *Risky Job Allocation* SLA class and 0.75 for jobs with *Secure Job Allocation* SLA class. D_k^j stands for a total number of jobs that fail to meet their deadline or due date measured calculated as:
$$D_k^j = \begin{cases} 1, & C_k^j > d_k^j, \\ 0, & otherwise. \end{cases} \tag{2}$$
- The maximization of the *Total Provider's Income*, $V = \sum_{j=1}^{n_k} \Bigg((1-\alpha_{Penalty}) * p_k^j * size_k^j * u_k^j(size_k^j) \Bigg)$. Due to the definition of the problem, we have to assure a benefit for the service provider. The first term is the sum of the processing times of all released jobs multiplied by the penalty factor, a number of used processors and a cost function dependent on a number of used processors (see Table 1).

4 Scheduling Approach

The machine for job allocation can be determined by taking into account different criteria. In this work we apply a modified version of the approach proposed in [12]. The first step is estimating an accurate availability summary which describes node's capacity to process new jobs submitted by clients. The availability of a node can be characterized by the size (duration) of a free *Time Slot* that can be allocated to the arriving jobs.

Whenever a resource becomes overburdened in comparison with its neighbors, its local *SLA Violation Count*, exceeds the local average by a specified *threshold* value. Resources in such a state are considered as *Overloaded*. They will send all incoming traffic to their neighbors, as well as any surplus workload that cannot be completed in a required time frame (i.e., before deadline).

Resource that are not overburdened with workload can be considered as *Underloaded*. Their estimated *SLA Violation Count* is lower than the local average and they are capable of accepting excessive workload from their *Overloaded* neighbors, as well as any incoming workload submitted by users. Resources in the *Balanced* state are characterized by the estimated *SLA Violation Count* close to the local average. They will run jobs, which exist in their local queue and will accept new jobs as well.

If the arrival of the job triggers the *Overloaded* transition rule (i.e., causes workload imbalance), the excessive jobs will be sent to one of the available neighbors. Selection of an appropriate destination can be seen as a bin packing problem with variable bin sizes and prices, where bins represent the physical nodes; items are the VMs that have to be allocated; bin sizes are the available CPU capacities of the nodes; and prices correspond to the power consumption by the nodes.

5 Experimental Analysis and Performance Evaluation

In this section, we present the experiments in order to evaluate the performance of the proposed Sandpile CA-based scheduling and load balancing algorithm. All experiments were conducted using the CloudSim framework [5] to simulate a cloud environment.

5.1 Workloads

We evaluate the performance of our strategies with a series of experiments using traces of real HPC jobs obtained from the Parallel Workloads Archive, and the Grid Workload Archive. The workloads include nine traces from: DAS2-University of Amsterdam, DAS2-Delft University of Technology, DAS2-Utrecht University, DAS2-Leiden University, KHT, DAS2-Vrije University Amsterdam, HPC2N, CTC, and LANL. These workloads are suitable for assessing the system because our IaaS model with multiple heterogeneous parallel machines is intended to execute jobs traditionally executed on Grids and parallel machines.

To adapt these workloads to the problem studied in our work artificial deadline constraints were added according to the following formula. Each job's J_k^j deadline d_k^j was set to be $d_k^{j'}$ [Time Units] from its release dates r_k^j. Parameter $d_k^{j'}$ was generated from a uniform distribution $\mathcal{U}\{p_k^{i,j'}, 50\}$ [Time Units]. Variable $p_k^{i,j'}$ denotes the job's J_k^j processing time at the slowest machine in the system.

Table 2. VM selection policies

Name	Description
CA-Stretch (CS)	Our proposed algorithm, the purpose of which is to minimize the total workflow execution cost while meeting the deadline and risk rate constraints
Maximum Reliability (MR)	This algorithm always meets the security assurance condition ($SD_j \leq SL_i$) minimizing the failure rate
Minimum Cost (MC)	This algorithm always disregards the security assurance condition ($SD_j \leq SL_i$) minimizing the associated execution cost
Minimum Cost with Deadline (MCD)	This algorithm maps each job to the least expensive VM instance, which can meet the associated deadline constraints

5.2 Experimental Scenarios

For all scenarios we consider eight infrastructure sizes with the number of machines from 1 to 128. It does not exactly match all machines on which the workloads are recorded, and in some cases may cause artifacts in the single run. To obtain valid statistical values, 50 repetitions per run are simulated. We employ four allocation heuristics described in Table 2.

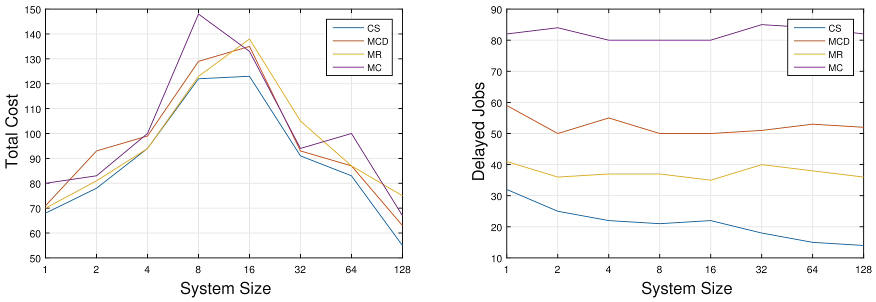
Additionally, we randomly assign Security Level (SL) factors to each machine uniformly selected from the range $\mathcal{U}\{0.3, \ldots, 1.0\}$ and Job Security Demand (SD) factors uniformly selected from the range $\mathcal{U}\{0.6, \ldots, 0.9\}$.

In the following simulation experiments, the efficiency of the analyzed job scheduling methods was measured in terms of:

- ***Total Cost:*** cost function dependent on both the requested level of SLA and a number of used machines;
- ***Delayed Jobs:*** stands for a total number of jobs that fail to meet their deadline or due date measured by: $D = \sum_{j=1}^{n_k} D_k^j$.

The simulation results are given in Fig. 1. It shows that for the given workloads we receive the best results by the CA-Stretch allocation strategy. In particular, the best return on infrastructure per machine can be seen when 8 to 16 machines are used. When we increase the number of machines, the income generated by each machine is decreased. Figure 1(b) shows the total number of delayed jobs during simulation. Once again, the *CA-Stretch* achieves the best performance of all analyzed lower level scheduling strategies.

It can be attributed to the employed reshuffling and rescheduling mechanisms in the proposed *CA-Stretch* scheme and its capability to equalize the workload in the local balancing domain. As a result, the number of concurrent requests submitted by the users has less negative impact on the scheduler's effectiveness. We can conclude that the QoS provided to the user can be optimized only by finding an optimal trade-off between various criteria inherent in the scheduling process. Single-minded approaches (such as *Maximum Reliability* or *Maximum Cost* policies) might seem like a simple solution to a complex problem, but as indicated by experimental simulation, such an outlook is not valid in practice. Their similar goals and optimization criteria result in allocating jobs to the same pool of machines which leads to the overall congestion and load imbalance, and in the effect, more frequent delays and higher execution costs.



(a) Total Cost (b) Delayed Jobs

Fig. 1. Performance results of conducted experiments with eight infrastructure sizes from 1 to 128 VMs: (a) Total Cost, (b) Delayed Jobs.

6 Conclusion and Future Work

In this paper we have proposed a novel parallel and distributed algorithm based on the *Sandpile* Cellular Automata (CA) model for dynamic load balancing and rescheduling of jobs in a distributed IaaS CC environment. We analyzed a variety of cloud configurations and workloads considering two objectives: provider income and SLA compliance. A user submits jobs to the service provider, which offers a guaranteed level of service. For a given service level the user is charged by a cost per unit of execution time. In return, the customer receives guarantees regarding the provided resources. These guarantees are maximum response time or deadlines used as QoS constraints.

As a future line of work we plan to extend our scheduling framework for tackling the problem of energy efficiency in CC systems. On the one hand, energy efficiency can be achieved by a smart consolidation of submitted workloads and optimizing system utilization. On the other hand, the proposed scheduler can be easily modified not only for spreading but also for consolidating workloads between VMs thus finding optimal trade-offs between system performance and energy consumption.

References

1. Ali, M., Khan, S.U., Vasilakos, A.V.: Security in cloud computing: opportunities and challenges. Inf. Sci. **305**, 357–383 (2015)
2. Benoit, A., Marchal, L., Pineau, J.-F., Robert, Y., Vivien, F.: Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. IEEE Trans. Comput. **59**(2), 202–217 (2010)
3. Buyya, R., Abramson, D., Giddy, Í.: Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid. In: Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region (HPC Asia 2000), pp. 283–289. IEEE Computer Society Press (2000)
4. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: vision, hype and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst. **25**(6), 599–616 (2009)
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. Softw. Pract. Exp. **41**(1), 23–50 (2011)
6. Celaya, J., Marchal, L.: A fair decentralized scheduler for bag-of-tasks applications on desktop grids. In: 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), pp. 538–541, May 2010
7. Chang, V.: The business intelligence as a service in the cloud. Future Gener. Comput. Syst. **37**, 512–534 (2014). Special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing. Special Section: Semantics, Intelligent Processing and Services for Big Data. Special Section: Advances in Data-Intensive Modelling and Simulation. Special Section: Hybrid Intelligence for Growing Internet and its Applications
8. Chang, V.: Towards a big data system disaster recovery in a private cloud. Ad Hoc Netw. **35**, 65–82 (2015). Special Issue on Big Data Inspired Data Sensing, Processing and Networking Technologies
9. Chang, V., Kuo, Y.-H., Ramachandran, M.: Cloud computing adoption framework: a security framework for business clouds. Future Gener. Comput. Syst. **57**, 24–41 (2016)
10. Chang, V., Walters, R.J., Wills, G.B.: Organisational sustainability modelling-an emerging service and analytics model for evaluating cloud computing adoption with two case studies. Int. J. Inf. Manag. **36**(1), 167–179 (2016)
11. de Assunção, M.D., Buyya, R.: Performance analysis of multiple site resource provisioning: effects of the precision of availability information. In: Sadayappan, P., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2008. LNCS, vol. 5374, pp. 157–168. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89894-8_17
12. Gąsior, J., Seredyński, F.: A Sandpile cellular automata-based scheduler and load balancer. J. Comput. Sci. **21**(Suppl. C), 460–468 (2017)
13. Jansen, W.A.: Cloud hooks: security and privacy issues in cloud computing. In: Proceedings of the 2011 44th Hawaii International Conference on System Sciences, HICSS 2011, pp. 1–10. IEEE Computer Society, Washington, DC (2011)
14. Kolodziej, J., Khan, S.U., Wang, L., Kisiel-Dorohinicki, M., Madani, S.A., Niewiadomska-Szynkiewicz, E., Zomaya, A.Y., Xu, C.-Z.: Security, energy, and performance-aware resource allocation mechanisms for computational grids. Future Gener. Comput. Syst. **31**, 77–92 (2014)

15. Modi, P.J., Shen, W.-M., Tambe, M., Yokoo, M.: Adopt: asynchronous distributed constraint optimization with quality guarantees. Artif. Intell. **161**(1–2), 149–180 (2005)
16. Nesmachnow, S., Perfumo, C., Goiri, I.: Controlling datacenter power consumption while maintaining temperature and QoS levels. In: 2014 IEEE 3rd International Conference on Cloud Networking (CloudNet), pp. 242–247, October 2014
17. Raycroft, P., Jansen, R., Jarus, M., Brenner, P.R.: Performance bounded energy efficient virtual machine allocation in the global cloud. Sustain. Comput.: Inform. Syst. **4**(1), 1–9 (2014)
18. Viswanathan, S., Veeravalli, B., Robertazzi, T.G.: Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems. IEEE Trans. Parallel Distrib. Syst. **18**(10), 1450–1461 (2007)
19. Singh, G., Kesselman, C., Deelman, E.: A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In: Proceedings of the 16th International Symposium on High Performance Distributed Computing, HPDC 2007, pp. 117–126. ACM, New York (2007)
20. Song, S., Hwang, K., Kwok, Y.-K.: Risk-resilient heuristics and genetic algorithms for security-assured grid job scheduling. IEEE Trans. Comput. **55**(6), 703–719 (2006)
21. Tchernykh, A., Lozano, L., Bouvry, P., Pecero, J.E., Schwiegelshohn, U., Nesmachnow, S.: Energy-aware on-line scheduling: ensuring quality of service for IaaS clouds. In: 2014 International Conference on High Performance Computing Simulation (HPCS), pp. 911–918, July 2014
22. Tchernykh, A., Lozano, L., Schwiegelshohn, U., Bouvry, P., Pecero, J.E., Nesmachnow, S., Drozdov, A.Y.: Online bi-objective scheduling for IaaS clouds ensuring quality of service. J. Grid Comput. **14**(1), 5–22 (2016)
23. Tchernykh, A., Pecero, J.E., Barrondo, A., Schaeffer, E.: Adaptive energy efficient scheduling in peer-to-peer desktop grids. Future Gener. Comput. Syst. **36**, 209–220 (2014). Special Section: Intelligent Big Data Processing. Special Section: Behavior Data Security Issues in Network Information Propagation. Special Section: Energy-Efficiency in Large Distributed Computing Architectures. Special Section: eScience Infrastructure and Applications

Dynamic Load Balancing Algorithm for Heterogeneous Clusters

Tiago Marques do Nascimento, Rodrigo Weber dos Santos, and Marcelo Lobosco(✉)

Graduate Program on Computational Modeling, Federal University of Juiz de Fora, Juiz de Fora, Brazil
tiago.nascimento@uab.ufjf.br, {rodrigo.weber,marcelo.lobosco}@ufjf.edu.br

Abstract. Half of the ten fastest supercomputers in the world use multiprocessors and accelerators. This hybrid environment, also present in personal computers and clusters, imposes new challenges to the programmer that wants to use all the processing power available on the hardware. OpenCL, OpenACC and other standards can help in the task of writing parallel code for heterogeneous platforms. However, some issues are not eliminated by such standards. Since multiprocessors and accelerators are different architectures and for this reason present distinct performance, data parallel applications have to find a data division that distributes the same amount of work to all devices, i.e., they have to finish their work in approximately the same time. This work proposes a dynamic load balancing algorithm that can be used in small-scale heterogeneous environments. A simulator of the Human Immune System (HIS) was used to evaluate the proposed algorithm. The results have shown that the dynamic load balancing algorithm was very effective in its purpose.

Keywords: Load balancing · Heterogeneous cluster · GPUs
Multiprocessors

1 Introduction

Heterogeneous clusters environments are becoming popular parallel platforms. These environments are composed by distinct processors and accelerators, such as GPUs. From a programmer perspective, it is not an easy task to write a parallel program to take advantage of all the computing resources, CPUs[1] and GPUs, present in such environment. This happens not only due to the fact that the computing resources have distinct computational power, but also because of the distinct types of parallelism they were designed to exploit. There are basically two types of parallelism in applications: Data-Level Parallelism (DLP) and Task-Level Parallelism (TLP). The first one arises due to the multiple data

The authors would like to thank UFJF and the Brazilian agencies FAPEMIG, CAPES, and CNPq.

[1] The term CPU in this work refers to multicore processors.

items that must be computed by an application, whereas the second one due to the multiple tasks that must be executed. CPUs were designed to deal with TLP and small amounts of DLP, whereas GPUs were designed to explore large amounts of DLP [6]. Programmers that want to explore DLP in all devices of a heterogeneous platform must take these differences into account, since they can impact the way the code is written and executed.

Some tools, such as those based on OpenCL (Open Computing Language) [15] and OpenACC (Open Accelerators) [1] standards, can help programmers to write code to execute on heterogeneous architectures. Some issues, however, remain open, such as the data division between GPUs and CPUs that balance the amount of work each one will execute. Since GPUs were designed to explore large amounts of DLP, they must receive more data than CPUs to compute, but how much more? Also, depending on the type of instruction that is executed (e.g., float point or integer instruction), the amount of data each computing resource must receive changes. A load balancing (LB) algorithm can help in this task. In this work we use the term LB in the sense of the data division that makes all devices in a heterogeneous cluster composed by CPUs and GPUs finish their computing in approximately the same time.

In previous works [16,17] we have proposed two distinct algorithms to deal with LB in Accelerated Processing Units (APUs) [3]. APUs merge, in a single silicon chip, the functionality of GPUs with the traditional multicore CPUs. In this paper two new contributions are presented. The first one is the extension done on the dynamic algorithm in order to execute it on a distinct architecture, a heterogeneous cluster. Some modifications in the original algorithm were introduced in order to deal with the new environment. The last one is the evaluation of the impact of the dynamic LB algorithm in performance, using for this purpose the HIS (Human Immune System) simulator.

The remaining of this work is organized as follows. Section 2 presents related works. Section 3 presents a brief overview of OpenCL. OpenCL and MPI were used in the implementation of HIS. In Sect. 4 we present the dynamic LB algorithm. Section 5 presents the performance evaluation. Finally, Sect. 6 presents our conclusions and plans for future works.

2 Related Work

A significant amount of research has been done on heterogeneous computing techniques [14]. Harmony [4] is a runtime supported programming and execution model that uses a data dependency graph to schedule and run independent kernels in parallel heterogeneous architectures. This approach is distinct from ours because we focus on data parallelism, while Harmony focus on task parallelism. Merge [11] is a library system that deals with map-reduce applications on heterogeneous system. Qilin [12] is an API that automatically partitions threads to one CPU and one GPU. SKMD [9] is a framework that transparently distributes the work of a single parallel kernel across CPUs and GPUs. SOCL [7] is an OpenCL implementation that allows users to dynamically dispatch kernels over devices.

StarPU [2] is a task programming library for hybrid architectures that provides support for heterogeneous scheduling. Our approach is distinct because we are not proposing a new library, API, framework or OpenCL implementation, nor we limit the number of CPUs or GPUs that can be used as Qilin does. Also, StarPU does not perform inter-node load-balancing as our approach does. Since the proposed dynamic scheduling approach is implemented in the application code, we do not have to pay the overheads imposed by the frameworks, runtime systems or APIs.

3 OpenCL

OpenCL (Open Computing Language) [15] is an open standard framework that was created by the industry (Khronos Group) in order to help the development of parallel applications in heterogeneous systems. An OpenCL platform includes a single host, which submits work to devices. OpenCL devices, such as CPUs, GPUs, and so on, are divided into compute units, which can further be divided into processing elements(PEs). An OpenCL application consists of two parts, the host program and one or more kernels. PEs execute the kernels, while the host program is executed by the host. The host sends commands to devices through a command-queue. There are three types of commands that can be issued: kernel execution, memory and synchronization commands. The commands issued to a specific queue can be executed in the same order they appear in the command-queue (in-order execution), or can be executed out-of-order. The programmer can use explicit synchronization mechanisms to enforce an order constrain to the execution of commands in a queue. An automatic LB scheme, based on the master-worker parallel pattern [13,15], can be implemented using command-queues, specially those that implements out-of-order execution. However, this parallel pattern is particularly suited for problems based on TLP [13]. In previous works [16,17] we proposed distinct solutions based on an in-order execution for problems based on DLP for an APU architecture.

4 Dynamic Load Balancing Scheme

Heterogeneous computers represent a big challenge to the development of applications that explore DLP. The use of all distinct PEs available to simultaneously operate in all data items is not easy due to the distinct hardware characteristics. In fact, heterogeneous computing on CPUs and GPUs using architectures like CUDA [8] has fixed the roles for each device: GPUs have been used to handle data parallel work while CPUs handle all the rest. The use of this fixed role has impacts on performance, since CPUs are idle while GPUs are handling the data parallel work. Actually CPUs could handle part of the work submitted to the GPU. In this scenario, OpenCL [15] represents an interesting alternative, since it is easy to program parallel codes that use all devices to operate in data items. The point is that the programmer is responsible for the data division between CPUs and GPUs. A good data division would give to each PE a distinct amount of data proportional to its relative performance. So if device A is 1.5 times faster than device B, it should receive 1.5 times more data to compute than device B.

In previous works we have presented two distinct LB algorithms [16,17] to be used with data parallel OpenCL codes running on an APU. The key idea behind the two algorithms is similar: data is split into two parts, one of which will be computed by the CPU, while the other one will be computed by the GPU. The amount of data that will be assigned to the CPU and GPU depends on their relative computing capabilities, which is measured in both LB algorithms during the execution of the application.

This work further extends our previous LB algorithms to be used in a different hardware platform: a heterogeneous cluster. Since an APU merges GPUs and CPUs cores on a single silicon chip, some modifications have to be done in the algorithm to deal with multiple GPUs and CPUs available in distinct nodes of a cluster. Also, the algorithm does not assume that all machines in a cluster have the same configuration, i.e., the same number and types of CPUs and GPUs.

The dynamic LB algorithm can be used in a wide variety of applications that explore DLP. Usually these applications have at least two aligned loops, in which the inner loop performs the same operations on distinct data items, as Algorithm 1 shows. Each step of the inner loop (or a collection of loops, if a multidimensional data structure is used) could be executed in any order, since no data dependency occurs between two distinct loop iterations. The number of steps the outer loop iterates is determined by the nature of the problem, but usually a dependency exists between two consecutive steps: a new step cannot proceed without the result of a previous one, since their results will be used during the computation of the new step. In many applications the outer loop is related to the progress of a simulation over time, and for this reason will be referred in this work as time-steps. The dynamic LB algorithm will decide the amount of data each PE will receive to compute in the inner loop.

During the computation of each data item, some applications require also access to its neighbors data, which can be located at distinct memory spaces due to data splitting between CPUs and GPUs. These data, called boundaries, must be updated between two consecutive iteration of the outer loop. This update requires the introduction of synchronization operations and the explicit copy of data. In the case of a heterogeneous cluster, this copy may occur inside a machine (e.g., copying data between two distinct GPUs or between the memory space of a CPU and a GPU, and vice-versa) or between machines, which imposes the use of communication primitives. Both data copy and synchronization operations are expensive, deteriorating performance, and for this reason should be avoided.

The dynamic LB algorithm is presented in Algorithm 2 and works as follows. For a single time-step, all GPUs and CPUs receive an equal amount of data to compute (data size divided by the total number of PEs) and the time required to compute them is recorded. This information is then used to compute the relative computing power of each PE and consequently determine the amount of data it will receive for the next time-steps. Equation 1 is used for this purpose.

```
for all time-steps do
    for each data item do
        call cpus/gpus devices to compute a piece of data;
    end
    send/receive boundaries;
    synchronize devices;
end
```

Algorithm 1. Data parallel algorithm

```
initialize MPI and OpenCL;
allocate memory in each device's memory space;
divide data equally among all devices;
start clock;
for a single time-step do
    call cpus/gpus to compute their data;
    synchronize;
end
finish clock;
compute $P_i^{(t)}$ and transfer data accordingly;
for all remaining time-steps do
    if time-step % LB interval == 0 then
        start clock;
        call cpus/gpus to compute their data;
        synchronize;
        finish clock;
        compute $P_i^{(t)}$;
        if $|P_i^{(t)} - P_i^{(t-1)}| > LB\ threshold$ then
            transfer data accordingly;
            synchronize devices;
        end
        else
            $P_i^{(t)} = P_i^{(t-1)}$ (keeps data distribution);
        end
    end
    else
        call cpus/gpus to compute interior points and transfer border points in
          parallel;
        synchronize;
        call cpus/gpus to compute border points;
        synchronize;
    end
end
```

Algorithm 2. The dynamic LB algorithm

$$P_i^{(t)} = \frac{P_i^{(t-1)} \times T_r^{(t-1)}}{T_i^{(t-1)} \times \sum_{k=1}^{n} \frac{P_k^{(t-1)} \times T_r^{(t-1)}}{T_k^{(t-1)}}}, \tag{1}$$

where $P_i^{(t)}$ is the percentage of data the PE i will receive to compute in the next time-step, $P_i^{(t-1)}$ is the percentage of data the PE i received in the previous time-step, $T_i^{(t-1)}$ is the time in which PE i executed the previous time-step, $T_r^{(t-1)}$ is the time in which an arbitrary reference PE r executed the previous time-step and k is the total number of PEs available in the heterogeneous cluster. In the first time-step ($t = 0$), the percentage of data each PE will receive to compute is divided equally among all PEs.

After the computation of the amount of data each PE will compute in the next time-step, memory should be reallocated and data copied from its last owner to the new one. However, in order to avoid memory reallocations, the dynamic LB algorithm allocates, at each PE, an additional amount of memory to avoid memory reallocations, and only data copies are required.

After the first time-step has finished, the LB algorithm will be executed from time to time to adjust the amount of data each PE will receive till the end of the computation. This occur because some applications exhibit an irregular behavior during computation, while other applications that seems to be regular parallel applications, such as the one that will be used in the performance evaluation, suffer from irregular execution time phases during their execution. This happens due to hardware optimizations done in the CPU, which would impact a static LB algorithm, i.e., an algorithm that keeps the percentage found in the first time-step until the end of computation [18].

The LB step is a time consuming task, specially due to data transfers between PEs located in distinct machines. If the change in the amount of data each PE must compute is minimal, the eventual performance gain is not compensated by the overhead of moving data. So a parameter, called *LB threshold*, was added to avoid this situation. If the difference between $P_i^{(t)}$ and $P_i^{(t-1)}$ is lower than this threshold, the PEs remain with their previous loads until another LB step is reached.

A final optimization is done in order to reduce the communication costs. Each PE divides its data into two subsets: borders and interior points. The border points are composed by the points that must be exchanged with the neighbors, whereas the interior points are not exchanged. The PE compute first the border points. While computing the interior points, the PE exchange borders with its neighbors, so computation and communication overlap.

5 Performance Evaluation

This section evaluates the performance of the LB algorithm presented in this work using for this purpose a simulator of the HIS [19,20]. This simulator was chosen because it is a representative of data parallel algorithm: the same set of operations must be executed on a large amount of data.

All tests were executed on a small cluster composed by 3 machines. The machines have two AMD 6272 processors (each machine has 32 cores), 32 GB of main memory, two Tesla M2075 GPUs, each one with 448 CUDA cores and 6 GB of global memory. Linux 2.6.32, OpenMPI version 1.6.2 and gcc version 4.4.7 were used to run and compile all codes. The machines are connected by a Gigabit Ethernet network. Although the AMD machines have a total of 96 cores, one Float-Point Unit (FPU) is shared by two cores, so only 48 FPUs are available in the machines.

5.1 Benchmark

A three dimensional simulator of the HIS [19,20] was used to evaluate the performance of the two load-balancing algorithms. The simulator implements a mathematical model that uses a set of eight Partial Differential Equations (PDEs) to describe how some cells and molecules involved in the innate immune response, such as neutrophils, macrophages, protein granules, pro- and anti-inflammatory cytokines, react to an antigen, which is represented by lipopolysaccharides. The diffusion of some cells and molecules are described by the mathematical model, as well as the process of chemotaxis. Chemotaxis is the movement of immune cells in response to chemical stimuli by pro-inflammatory cytokine. Neutrophils and macrophages move towards the gradient of pro-inflammatory cytokine concentration. A detailed discussion about the model can be found in [19,20].

The numerical method used in the computational implementation of the mathematical model was the Finite Difference Method [10], a method commonly used in the discretization of PDEs. The computation of the convective term (the chemotaxis term) is a complex part in the resolution of the PDEs. Our implementation is based on the finite difference method for the spatial discretization and the explicit Euler method for the time evolution. First-Order Upwind scheme [5] is used in the discretization of the chemotaxis term. More details about the numerical implementation, specially how the Laplace operator, that simulates the diffusion phenomenon, is implemented in 3D, can be found in a previous work [20]. This previous work used C and CUDA in the implementation, using only GPUs in the computation, while this work uses C and OpenCL, using all resources (CPUs and GPUs) available in the cluster.

There are two ways to divide the data mesh: division by planes and division by individual elements. The division by individual elements allows the algorithm to use of a fine-grain data partition in the LB. In a previous work [18], we have found that the division by individual elements performs better and, for this reason, this division will be used in this work. A mesh of size $50 \times 50 \times 3200$ was used in the experiments. The values used to set the initial conditions and parameters of HIS are the same used in our previous work [16]. A total of $10,000$ time-steps were executed. The LB interval is equal to 10% of the time-steps and the LB threshold is equal to 50 elements.

Three versions of the HIS were executed: a sequential one, a version that used the dynamic LB algorithm and one that did not use LB. In the version that did not use the LB, the mesh size was divided equally among all PEs that were used to execute the code. Each HIS version was executed at least 3 times, and all standard deviations of the execution time were below 1%.

5.2 Results

Table 1 presents the results. As one can observe, the sequential version of the code executes in more than 36 h. A typical simulation requires 1,000,000 time-steps, which represents more than 151 days of computation. The parallel version of the simulator that does not use the LB algorithm executes up to 435 times faster. But the dynamic LB algorithm improved the performance even more: using the same configuration, the application executed 916 times faster than the sequential one and 2.1 times faster than the version that does not use the LB algorithm.

Table 1. Experimental results for the parallel version of the code in a small cluster. Average execution time(s) and gains relative to the version without LB and the sequential one.

Platform	w/o LB	LB	Gain
32 CPUs + 2 GPUs	531.3	283.1	1.9
64 CPUs + 4 GPUs	308.3	182.1	1.7
96 CPUs + 6 GPUs	300.5	142.7	2.1
Sequential	130,694.33		916

Table 2 presents the HIS parallel execution time in a single machine, considering the use of each computational resource at a time, as well as using all of them simultaneously, with and without the use of the LB algorithm proposed in this work. The best result obtained with the use of a single computational

Table 2. Using all resources types available in a single machine × using one at a time. Times in seconds. Gains compared to the version that uses 2 GPUs to execute the code.

Platform	Execution time	Gain
32 CPUs	1,688	-
1 GPU	627	-
2 GPUs	317	-
32 CPUs + 2 GPUs (w/o LB)	531.3	0.6
32 CPUs + 2 GPUs (LB)	283.1	1.12

resource was 317 s, when 2 GPUs are used to execute the code. The simultaneous use of all resources does not guarantee a performance gain: if all CPUs and GPUs are included in the computation, the parallel execution time increases to 531 s. However, the same configuration can obtain a performance gain if the LB algorithm is used: the execution time reduces to 283 s.

6 Conclusions and Future Works

This paper presented the implementation of a dynamic LB algorithm in a heterogeneous cluster environment. Its key idea is to split data items of an application that explore DLP into multiple parts that will be computed simultaneously by CPUs and GPUs. The amount of data that will be assigned to CPUs and GPUs depends on their relative computing capabilities, which is measured and updated during all the execution of the application.

A performance evaluation of the dynamic LB algorithm was executed, using for this purpose the Human Immune System simulator. The results have shown that the algorithm was very effective in its purpose, resulting in gains up to 916-fold in execution time compared to the sequential one. Compared to the version that did not use the LB, the gains in performance were 2.1 times. We have also shown that performance gains could only be obtained using all resources in a single machine if the LB algorithm was used.

As future works, we plan: (a) to measure the overheads imposed by the algorithm, specially the time spent with communication due to a new data division; (b) to develop a static version of the algorithm, and compare it to the dynamic one; (c) to evaluate the proposed LB algorithm using other benchmarks; and (d) evaluate the impacts of the algorithm in the scalability of applications.

References

1. The OpenACC application programming interface - version 2.5. Technical report, OpenAcc.org (2015)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput.: Pract. Exp. **23**(2), 187–198 (2011)
3. Branover, A., Foley, D., Steinman, M.: AMD fusion APU: Llano. IEEE Micro **32**(2), 28–37 (2012)
4. Diamos, G.F., Yalamanchili, S.: Harmony: an execution model and runtime for heterogeneous many core systems. In: Proceedings of the 17th International Symposium on High Performance Distributed Computing, HPDC 2008, pp. 197–200. ACM, New York(2008)
5. Hafez, M.M., Chattot, J.J.: Innovative Methods for Numerical Solution of Partial Differential Equations. World Scientific Publishing Company, Singapore (2002)
6. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 5th edn. Morgan Kaufmann Publishers Inc., San Francisco (2011)

7. Henry, S., Denis, A., Barthou, D., Counilh, M.-C., Namyst, R.: Toward OpenCL automatic multi-device support. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 776–787. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09873-9_65
8. Kirk, D.B., Wen-Mei, W.H.: Programming Massively Parallel Processors: A Hands-on Approach, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco (2013)
9. Lee, J., Samadi, M., Park, Y., Mahlke, S.: Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT 2013, pp. 245–256. IEEE Press, Piscataway (2013)
10. LeVeque, R.: Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics). Society for Industrial and Applied Mathematics, Philadelphia (2007)
11. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: a programming model for heterogeneous multi-core systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 287–296. ACM, New York (2008)
12. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp. 45–55. ACM, New York (2009)
13. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional, Boston (2004)
14. Mittal, S., Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. ACM Comput. Surv. **47**(4), 69:1–69:35 (2015)
15. Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide, 1st edn. Addison-Wesley Professional, Boston (2011)
16. do Nascimento, T.M., de Oliveira, J.M., Xavier, M.P., Pigozzo, A.B., dos Santos, R.W., Lobosco, M.: On the use of multiple heterogeneous devices to speedup the execution of a computational model of the human immune system. Appl. Math. Comput. **267**, 304–313 (2015)
17. do Nascimento, T.M., dos Santos, R.W., Lobosco, M.: On a dynamic scheduling approach to execute OpenCL jobs on APUs. In: Osthoff, C., Navaux, P.O.A., Barrios Hernandez, C.J., Silva Dias, P.L. (eds.) CARLA 2015. CCIS, vol. 565, pp. 118–128. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26928-3_9
18. do Nascimento, T.M., dos Santos, R.W., Lobosco, M.: Performance evaluation of two load balancing algorithms on a hybrid parallel architecture. In: Malyshkin, V. (ed.) PaCT 2017. LNCS, vol. 10421, pp. 58–69. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62932-2_5
19. Pigozzo, A.B., Macedo, G.C., Santos, R.W., Lobosco, M.: On the computational modeling of the innate immune system. BMC Bioinform. **14**(6), S7 (2013)
20. Rocha, P.A.F., Xavier, M.P., Pigozzo, A.B., de M. Quintela, B., Macedo, G.C., dos Santos, R.W., Lobosco, M.: A three-dimensional computational model of the innate immune system. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012. LNCS, vol. 7333, pp. 691–706. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31125-3_52

Multi-Objective Extremal Optimization in Processor Load Balancing for Distributed Programs

Ivanoe De Falco[1], Eryk Laskowski[2](✉), Richard Olejnik[3], Umberto Scafuri[1], Ernesto Tarantino[1], and Marek Tudruj[2,4]

[1] Institute of High Performance Computing and Networking, CNR, Naples, Italy
{ivanoe.defalco,umberto.scafuri,ernesto.tarantino}@icar.cnr.it
[2] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
{laskowsk,tudruj}@ipipan.waw.pl
[3] Université Lille — CRISTAL, CNRS, Lille, France
richard.olejnik@univ-lille1.fr
[4] Polish-Japanese Academy of Information Technology, Warsaw, Poland

Abstract. The paper presents a multi-objective load balancing algorithm based on Extremal Optimization in execution of distributed programs. The Extremal Optimization aims in defining task migration as a means for improving balance in loading executive processors with program tasks. In the proposed multi-objective approach three objectives relevant in processor load balancing for distributed applications are jointly optimized. These objectives include: balance in computational load of distributed processors, total volume of inter-processor communication between tasks and task migration metrics. In the proposed Extremal Optimization algorithms a special approach called Guided Search is applied in selection of a new partial solution to be improved. It is supported by some knowledge of the problem in terms of computational and communication loads influenced by task migration. The proposed algorithms are assessed by simulation experiments with distributed execution of program macro data flow graphs.

Keywords: Extremal Optimization · Multi-objective optimization
Processor load balancing

1 Introduction

The focus of this paper is on using a multi-objective optimization approach based on Extremal Optimization (EO) [1] in the context of processor load balancing. EO is a nature inspired optimization approach based on improvements of a single solution which has small computational and memory complexity. These features justify using this approach in processor load balancing in distributed systems. Good reviews and classifications of classic load balancing methods are presented in [2,7,8]. Good reviews of load balancing methods based on evolutionary algorithms including EO are contained in [5,9]. Based on them, we can state that except for our works, the known solutions of load balancing methods have not yet shown interest in EO applied in load balancing algorithms.

In our previous research [3–5], we have proposed to use EO in iterative load balancing phases to determine periodic migration of tasks among processors leading to improvement of program task placement on processors. It is based on a special quality model which covers both computation and communication parameters of parallel application tasks and component features of the executive system. In these algorithms, a special EO-GS approach (EO with a Guided Search) is applied which assures that selection of a new partial solution to be improved is guided by some knowledge of the problem in terms of computational and communication loads. It replaces the fully random task and processor selection in solution improvement by stochastic selection which improves convergence of the algorithm. In papers [4,5] we have compared our load balancing algorithms based on EO-GS approach against algorithms based on greedy deterministic and genetic approaches. It has been shown the proposed algorithm based on the proposed by us EO-GS approach provides better applications parallel execution speedup than obtained with the use of the other mentioned approaches. Experience on EO-supported load-balancing gathered by us in our previous research was suggesting that a multi-criteria approach could be used to improve load balancing algorithms by search which covers a larger area of optimization space. The algorithms presented in the current paper concern the EO-GS based load balancing algorithms similar to the mentioned above but developed by using a multi-objective optimization approach instead of the single-objective one.

Large surveys on general methods of multi-objective optimization can be found in [13,14]. Extensive surveys on multi-objective optimization methods combined with evolutionary algorithms in general can be found in [15,16]. Multi-objective approach applied to EO has already been discussed in several papers [17–22]. They propose basic methods of multi-objective optimization and cover different technical aspects of this approach. However, they are oriented towards generalized optimization problems and do not cover algorithms applied to processor load balancing. In this respect our work has fully originality features.

In the current paper, we show that our multi-objective approach based on three parameters of program execution in cluster environments: computational load of processors, inter-processor communication intensity and the number of task migrations provides better results than that of the EO applied to processor load balancing presented in our previous papers. The three objectives are included into a generalized EO algorithm iterative structure in which the algorithm sequentially performs series of steps based on the dynamically changing three objectives (EO-GS global fitness functions) and on respective methods of solution element selection for improvement (EO local fitness function). The algorithm delivers the final compromise solution which minimizes the distance to the ideal point with respect to a given norm which is the Manhattan distance [6].

The proposed algorithms are validated by experimental results obtained by simulation.

The paper is organized as follows. Section 2 recalls EO algorithm principles. Section 3 presents the scheme of the load balancing based on the multi-objective EO and important details of the applied multi-objective approach. Section 4 describes the experimental assessment of the proposed algorithms including simulation results.

2 Extremal Optimization Algorithm Principles

Extremal Optimization is an attractive nature-inspired optimization method for NP–hard combinatorial and physical optimization problems. It was proposed by Boettcher and Percus [1], following the Bak–Sneppen approach of self–organized dynamic criticality [10].

A probabilistic version EO operates on a single solution S consisting of a given number of components s_i, each of which is a variable of the problem. At each algorithm iteration, a local fitness value ϕ_i is assigned to each of them. Then, for a minimization problem, the components are ranked in decreasing order of local fitness values. The worst component s_j is of rank 1, while the best one is of rank G, where G is the number of components. Then, a distribution probability over the ranks k is considered as follows: $p_k \sim k^{-\tau}$, $1 \leq k \leq G$ for a given value of the parameter τ. Finally, at each update, a rank k is selected according to p_k so that the component s_i of rank k randomly changes its state and the solution moves to a neighboring one, $S' \in \mathit{Neigh}(S)$, unconditionally. At the end of iteration, the global fitness $\Phi(S')$ is computed, and the new solution S' is saved if its global fitness value is better than that of the best solution found so far. The only parameters are the total number of iterations $\mathcal{N}_{\text{iter}}$ and the probabilistic selection parameter τ.

To foster the convergence speed of EO optimization, we have proposed a modified version of EO algorithm, called Extremal Optimization with Guided Search (EO-GS) [4,5]. In EO-GS, some knowledge of the problem properties is used during the next solution selection in consecutive EO iterations with the help of an additional local target function ω_s. The value of this function is evaluated for all neighbours $\mathit{Neigh}(S)$ of rank k. Then, the neighbour solutions are sorted and assigned GS-ranks g with the use of the function ω_s. The new state $S' \in \mathit{Neigh}(S)$ is selected in a stochastic way using the exponential distribution with the selection probability $p \sim \text{Exp}(g, \lambda) = \lambda e^{-\lambda g}$. Due to this, better neighbour solutions are more probable to be selected. The bias to better neighbours is controlled by the λ parameter.

3 Load Balancing Based on the Multi-Objective EO

3.1 Processor Load Balancing Scheme Based on Multi-Objective EO

The proposed Multi-Objective EO-GS-based load balancing algorithm is meant for a cluster of multi-core processors interconnected by a message passing

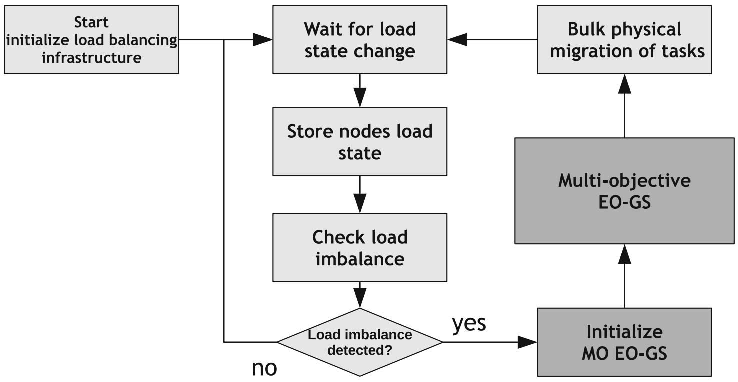


Fig. 1. The general scheme of load balancing based on Multi-Objective EO with guided search.

network. Load balancing actions are performed on-line to dynamically preserve the even distribution of application tasks on processors.

We assume that the load balancing algorithms dynamically control assignment of program tasks $t_k, k \in 1, \ldots, |T|$ to processors (computing nodes) $n, n \in 0, 1, \ldots, |N| - 1$, where T and N are the sets of all the tasks and the computing nodes, respectively. The goal is the minimal total program execution time, achieved by task migration between processors. The load balancing method is based on a series of steps in which detection and correction of processor load imbalance is done, Fig. 1. The imbalance detection relies on some run-time infrastructure which observes the state of processors in the executive computer system and the execution states of application programs. When load imbalance is discovered, processor load correction actions are launched. For them a multi-objective EO-GS algorithm is executed to identify the tasks which need migration and the processor nodes which will be migration targets. Following this, the required physical task migrations are performed with the return to the load imbalance detection.

To evaluate the load of the system two indicators are used. The first is the computing power of a node n: $\text{power}_{\text{CPU}}(n)$, which is the sum of potential computing powers of all the active cores on the node, available for application execution. The second is the percentage of the CPU power available for application threads on the node n: $\text{time}_{\text{CPU}}(n)$, periodically estimated on computing nodes.

System load imbalance I is a Boolean variable defined based on the difference of the CPU availability between the currently most heavily and the least heavily loaded computing nodes:

$$I = \max_{n=0,\ldots,|N|-1}(\text{time}_{\text{CPU}}(n)) - \min_{n=0,\ldots,|N|-1}(\text{time}_{\text{CPU}}(n)) \geq \alpha \qquad (1)$$

The load imbalance equal true requires a load correction. The value of α is determined experimentally (during experiments we set it between 25% and 75%).

Algorithm 1. Multi-objective EO with Guided Search (MOEO-GS)

```
initialize configuration $S$ at will
$S_{\text{best}} \leftarrow S$
$D_{\text{S}} \leftarrow \emptyset$ {the set of non-dominated solutions (Pareto-front)}
**while** total number of iterations $\mathcal{N}_{\text{iter}}$ not reached **do**
   $c \leftarrow$ a criterion for evaluation in the current iteration
   evaluate $\phi_{i,c}$ for each variable $s_i$ of the current solution $S$
   rank the variables $s_i$ based on their local fitness $\phi_{i,c}$
   choose the rank $k$ according to $k^{-\tau}$ so that the variable $s_j$ with $j = \pi(k)$ is selected
   evaluate $\omega_s$ for each neighbour $S_v \in Neigh(S, s_j)$, generated by $s_j$ change of the current solution $S$
   rank neighbours $S_v \in Neigh(S, s_j)$ based on the target function $\omega_s$
   choose $S' \in Neigh(S, s_j)$ according to the exponential distribution
   accept $S \leftarrow S'$ unconditionally
   **if** $S$ is non-dominated **then**
      include $S$ in $D_{\text{S}}$, remove dominated solutions from $D_{\text{S}}$
   **end if**
**end while**
select $S_{\text{best}}$ from $D_{\text{S}}$ using $\Phi(S)$
**return** $S_{\text{best}}$ and $\Phi(S_{\text{best}})$
```

An application is characterized by two programmer-supplied parameters, based on the volume of computations and communications tasks: $\text{com}(t_s, t_d)$ is a communication metrics between tasks t_s and t_d, $\text{wp}(t)$ is a load weight metrics introduced by a task t. $\text{com}(t_s, t_d)$ and $\text{wp}(t)$ metrics can provide exact values, e.g. for well-defined tasks sizes and inter-task communication in regular parallel applications, or only some predictions, e.g. when the execution time depends on the processed data. Even when the values are exact, we assume that there can some fluctuations of tasks execution or CPU power availability, so the dynamic load balancing is required.

A task mapping solution S is represented by a vector $\mu(S) = (\mu_1, \ldots, \mu_{|T|})$ of $|T|$ integers ranging in the interval $\{0, 1, \ldots, |N| - 1\}$. $\mu_i = j$ means that the solution S maps the i–th task t_i onto the computing node j.

In our solution we solve a processor load balancing problem in execution of distributed programs with the use of a multi-objective EO-GS algorithm (MOEO-GS), shown as Algorithm 1. The proposed MOEO-GS algorithm follows the general scheme of EO-GS approach described in the Sect. 2, with the exception that it uses a set of EO local and global fitness functions and maintains the Pareto front of non-dominated solutions.

During an iteration, the selection and solution improvement are performed using a single objective c (i.e. a single EO local and a respective single global fitness functions). It is selected in a probabilistic way from the MO objectives specified for our load balancing problem (the local and global fitness functions used in our MOEO-GS algorithm are defined in the Subsect. 3.2).

The Pareto front is analyzed at the end of the algorithm to deliver the S_{best} solution. The S_{best} is selected from D_{S} using some utility function, which in our implementation of the algorithm is the Manhattan distance metrics. The S_{best} is then used by the load balancing controller to decrease the load imbalance.

3.2 Global and Local Fitness Functions Applied in Multi-Objective EO

In our algorithm, we operate using three objective functions oriented on supporting the load balancing problem: total computational load imbalance in execution of application tasks on processors, total volume of communication between tasks placed on different computing nodes and task migration number which should be possibly small in fighting imbalance of processor loads. As MOEO-GS is a minimization algorithm, it looks for the solutions with lower values of the global fitness (or components with lower values of the local fitness, respectively).

The definitions of fitness functions use two auxiliary formulas:

$$\mathrm{nwp}(S,n) = \sum_{t\in T:\mu_t=n} \mathrm{wp}(t) \tag{2}$$

$$W_T = \sum_{t\in T} \mathrm{wp}(t) / \sum_{n=0,\ldots,|N|-1} \mathrm{power}_{\mathrm{CPU}}(n) \tag{3}$$

where $\mathrm{nwp}(S,n)$ is the sum of computational load of program tasks allocated to processor n in the solution S, and W_T is the average computational weight of program tasks attributed to one unit of computational power of processors.

A load imbalance normalization constant is equal to maximal numerical value of the imbalance (i.e. when all tasks are assigned to the slowest processor):

$$D_{\mathrm{norm}} = (|N|-2) * W_T + \sum_{t\in T} \mathrm{wp}(t) / \min_{n=0,\ldots,|N|-1} \mathrm{power}_{\mathrm{CPU}}(n) \tag{4}$$

The first objective concerns the reduction of the computational load imbalance among executive processors in the system during a given phase of distributed program execution i.e. defined by the current MOEO-GS solution S. The global fitness functions $\Phi(S)$ for **objective 1 (computational load imbalance)** is defined as follows:

$$\Phi_l^1(S) = \begin{cases} 1 & \text{exists at least one unloaded node} \\ \mathrm{deviation}(S)/D_{\mathrm{norm}} & \text{otherwise} \end{cases} \tag{5}$$

where:

$$\mathrm{deviation}(S) = \sum_{n=0,\ldots,|N|-1} |\,\mathrm{nwp}(S,n)/\,\mathrm{power}_{\mathrm{CPU}}(n) - W_T| \tag{6}$$

The function $\Phi_l^1(S)$ represents the numerical load imbalance metrics in the solution S. It is equal to 1 when in S there exists at least one unloaded (empty) computing node, otherwise it is equal to the normalized absolute load deviation of tasks from average load in S.

The local fitness function for MOEO-GS algorithm for the objective 1 is designed as follows:

$$\phi_l(t) = \gamma * \mathrm{load}(\mu_t) + (1-\gamma) * (1 - \mathrm{ldev}(t)) \tag{7}$$

where the function load(n) indicates how much the load of node n, which executes t, exceeds the average load of all nodes. It is normalized versus the heaviest load among all the nodes. The function ldev(t) is defined as the difference between the load metrics of the task t and the average task load on the node μ_t, normalized versus the highest such value for all tasks on the node [5].

The second objective for the MOEO-GS algorithm is the global EO-GS fitness function $\Phi(S)$ for **objective 2 (external communication)** defined as follows:

$$\Phi_c(S) = \sum_{s,d \in T : \mu_s \neq \mu_d} \mathrm{com}(s,d) / \sum_{s,d \in T} \mathrm{com}(s,d) \tag{8}$$

The function $\Phi_c(S) \in [0,1]$ represents the impact of the external (i.e. inter-node) communication between tasks on the quality of a given mapping S. It is a quotient of the sum of external communication volume and the total communication volume in a program. When all tasks are placed on the same node $\Phi_c(S) = 0$, when tasks are placed so that all communication is external $\Phi_c(S) = 1$.

The local fitness function for objective 2 is designed as follows:

$$\phi_c(t) = 1 - \mathrm{attr}(t) \tag{9}$$

where the attraction of the task t to its executive computing node attr(t) is defined as the amount of communication between task t and other tasks on the same node, normalized versus the maximal attraction inside the node [5].

The third objective for the MOEO-GS algorithm is concerned with task migrations induced by the current EO-GS solution S in terms of the computational load imbalance. The global EO-GS fitness function for **objective 3 (migration)** corresponds to the number of migrations:

$$\Phi^1_m(S) = \mathrm{migration}(S) \tag{10}$$

$$\mathrm{migration}(S) = |\{t \in T : \mu^S_t \neq \mu^{S*}_t\}| / |T| \tag{11}$$

where: μ^S_t is the current node of the task t in the solution S, and μ^{S*}_t is the node of the task t in the initial solution at the start of the algorithm. The function $\Phi^1_m(S) \in [0,1]$ is a migration number metrics. It is equal to 0 when there is no migration, when all tasks have to be migrated $\Phi^1_m(S) = 1$, otherwise $0 \leq \Phi^1_m(S) \leq 1$.

The local fitness function $\phi_m(t)$ for migration objective $\Phi^1_m(S)$ is designed as follows:

$$\phi_m(t) = \begin{cases} 1 \text{ when task } t \text{ is migrated} \\ 0 \text{ otherwise} \end{cases} \tag{12}$$

The $\phi_m(t)$ local fitness function forces the migration of already migrated tasks, thus increasing the probability that finally more tasks will occupy their initial computing nodes.

An **alternative version** of the global fitness functions $\Phi_m^1(S)$ for the migration objective includes the migration quality coefficient. The function improvement(n) indicates how much the current placement of tasks on a node n improves (i.e. decreases) the load imbalance of the application, comparing the initial task placement:

$$\text{improvement}(n) = |\frac{\text{nwp}(S,n)}{\text{power}_{\text{CPU}}(n)} - W_T| - |\frac{\text{nwp}(S^*,n)}{\text{power}_{\text{CPU}}(n)} - W_T| \tag{13}$$

where S is the currently considered solution and S^* is the initial task placement at the start of the algorithm.

$$\Phi_m^2(S) = \frac{\text{totalimpr}(S)+1}{2} * \text{migration}(S) \tag{14}$$

where:

$$\text{totalimpr}(S) = \frac{\sum_{n=0,\ldots,|N|-1} \text{improvement}(n)}{D_{\text{norm}}} \tag{15}$$

The function totalimpr$(S) \in [-1, 1]$ indicates whether there is the improvement (when totalimpr$(S) < 0$) or deterioration (when totalimpr$(S) > 0$) in the total load balance in the system comparing the initial placement of tasks of the application.

We have also designed another variant of the MOEO-GS global fitness function for objective 1 (computational load imbalance). Instead of the total load imbalance as in Eq. 5, the second variant of objective 1 function uses the totalimpr(S) metrics as a load balance indicator:

$$\Phi_l^2(S) = \frac{\text{totalimpr}(S)+1}{2} \tag{16}$$

In such a way the task placement which ensures the best total processors balance improvement is a preferred outcome of the algorithm.

To summarize, the following MOEO-GS variants were designed: **MO-1-GS** which uses $\Phi_l^1(S), \Phi_c(S), \Phi_m^1(S)$ global fitness functions, **MO-2-GS** with $\Phi_l^1(S), \Phi_c(S), \Phi_m^2(S)$ and **MO-3-GS** with $\Phi_l^2(S), \Phi_c(S), \Phi_m^1(S)$ global fitness functions, respectively. These variants use the respective local fitness functions, defined in Eqs. 7, 9 and 12, which match the used global fitness functions, respectively.

4 Experimental Assessment of the Proposed Algorithms

In this section we present an experimental assessment of the presented load balancing algorithms. The experiments have been conducted using simulated execution of application programs in a distributed system. The applied simulator was built following the DEVS discrete event system approach [11].

The simulated model of execution corresponds to parallelization based on message-passing, using the MPI library for communication. The general structure of a program resembled typical numerical programs or physical phenomena

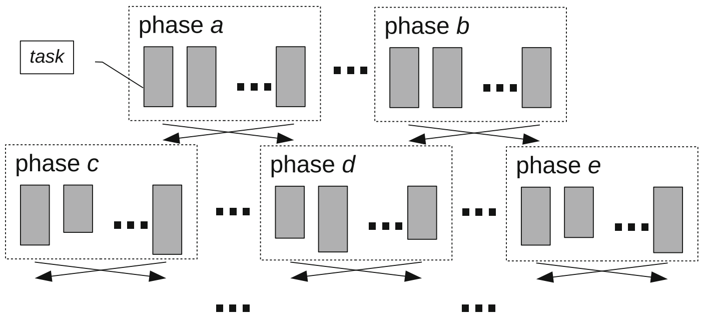


Fig. 2. The general structure of exemplary applications.

simulations. The exemplary programs were modeled as Temporal Flow Graphs, TFG, [12]. In the TFG model, an application program consists of a set of program modules called phases, composed of parallel tasks, Fig. 2. Tasks of the same phase can communicate. At the boundaries between phases there is a global exchange of data which enables the next phases for execution.

During experiments we used a set of 10 synthetic exemplary programs, which were randomly generated. The number of tasks in an application varied from 64 to 544. The communication/computation ratio C/E (the quotient of the communication time to the execution time in our experimental environment) for applications was in the range $[0.05, 0.20]$.

To assess performance of the presented multi-objective algorithms, we used two reference single objective load balancing algorithms based on sequential extremal optimization algorithms EO. One (denoted as SO-C) aims in balancing exclusively computational loads of processor nodes. It is based on a classical sequential EO without guided search. The second one (denoted as SO-WS-GS) is based on a single objective EO with the guided search and is using a global fitness function which is a weighted sum of the three aforementioned optimization criteria (see Subsect. 3.2) according to the following equation:

$$\Phi_{\mathrm{WS}}(S) = \Phi_c(S)\Delta_1 + \Phi_m^1(S)\Delta_2 + \Phi_l^1(S)[1-(\Delta_1+\Delta_2)] \qquad (17)$$

where Δ_1 and Δ_2 are weights from the range (0,1).

We have compared to these two algorithms three MOEO-GS-based algorithm variants (MO-1-GS, MO-2-GS, MO-3-GS, see Subsect. 3.2). A comparison of our reference algorithms SO-C, SO-WS-GS to other common methods of load balancing like genetic algorithm and deterministic local-search algorithm has been presented in [3,4].

Load-balanced execution of exemplary applications was studied in systems containing from 2 to 32 homogeneous processor nodes. The parameters used in EO, EO-GS, the load balancing control and the weighted sum of the global fitness function of the single objective EO-WS-GS were selected based on the results of the experimental research, presented in [3,4]. We applied such parameter values for which balanced performance between application speedup and migration count was obtained: $\alpha = 0.5$, $\tau = 1.5, \lambda = 0.14, \mathcal{N}_{\text{iter}} = 500$, $\beta = 0.5$, $\Delta_1 = 0.13$, $\Delta_2 = 0.17, \gamma = 0.75$. Our simulation environment allowed us to model the task migration cost.

The results are averages of 10 runs of each application. For each run 4 different methods of initial task placements (random, round-robin, METIS graph partitioning, packed /i.e. round-robin mapping of equal groups of tasks/) were used. Thus, the result for each parameter set is an average of 40 runs in total.

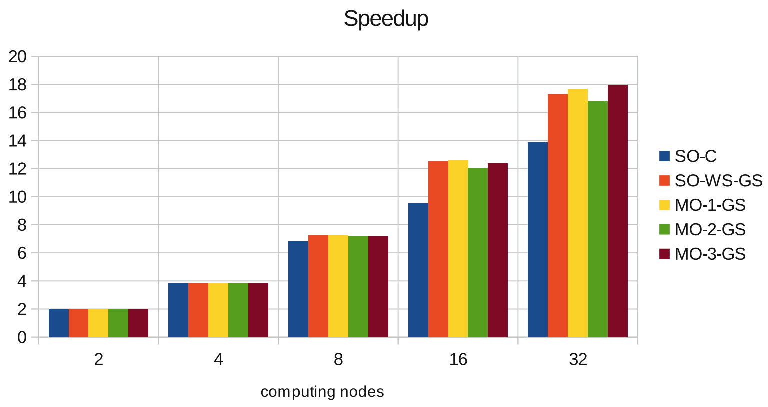


Fig. 3. Application parallel speedup for different numbers of computing nodes for all tested algorithms.

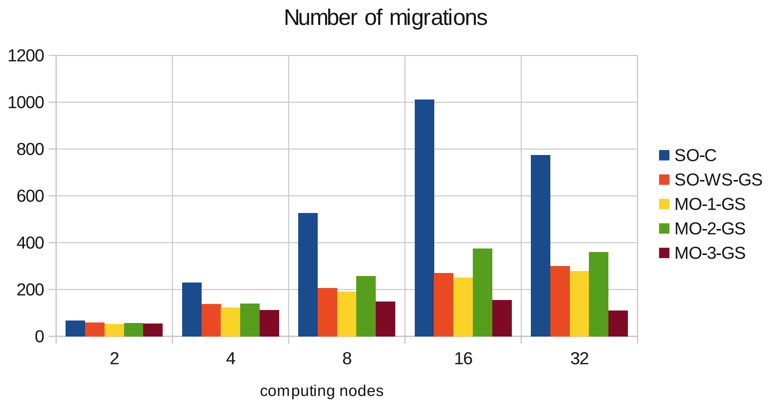


Fig. 4. The number of task migrations per single application execution on different numbers of computing nodes with different load balancing algorithms.

The application parallel speedup for both EO-based algorithms and the MO-EO algorithms as a function of the number of executive processors is shown

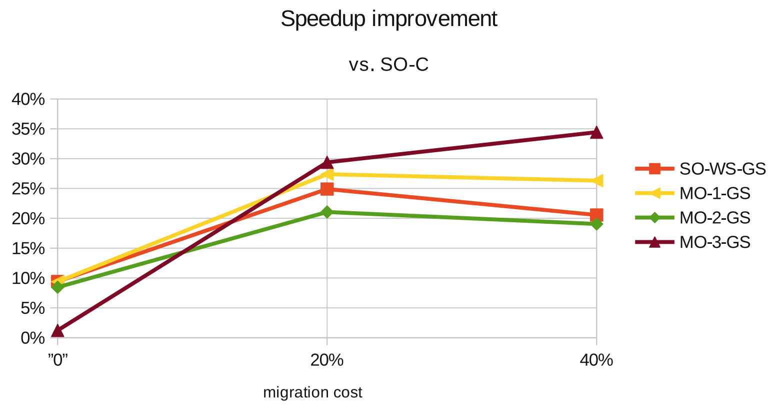


Fig. 5. Relative application speedup improvement for different load balancing algorithms versus SO-C for execution on 32 computing nodes with different migration costs.

in Fig. 3. We have assumed here the cost of migration of a single task to be equal to 20% of the task computational weight, as a medium migration cost between 0% and 40% considered in further experiments. The presented results are averaged over all tested exemplary applications. For these applications the parallel speedup obtained by multi-objective algorithms MO-1-GS and MO-3-GS is greater than that obtained with single-objective EO.

To better characterize our proposed MOEO-GS algorithms we collected also the migration number statistics, Fig. 4. It gives the average total number of task migrations in application execution resulting from applied different load balancing algorithms with the migration cost assumed as 20% of the task computational weight.

Since migration burden can be very different we decided to perform experiments also as a function of variable migration costs, Fig. 5, set as 0%, 20% and 40% of task computational weight.

Figures 3 and 4 have shown the superiority of two multi-objective algorithms over the single-objective ones, especially for a large number of computing nodes (32, 16) on which the application was executed. Here the multi-objective versions MO-1-GS and MO-3-GS are better than single objective versions SO-C and SO-WS-GS. Figure 5 has shown the superiority of the multi-objective approach (MO-1-GS and MO-3-GS) over the single objective approach (SO-WS-GS) for larger migration costs (20% and 40% of task computational weight).

The experimental results confirm advantages of the proposed load balancing multi-objective algorithms. For a realistic case with not null migration cost, all multi-objective algorithm variants are much better than the single-objective SO-C algorithm. The conclusions are that out from three MOEO-GS variants, MO-3-GS is the best suited for systems with the most demanding load balancing requirements (high migrations costs, big number of computing nodes), while MO-1-GS works very well for low or medium migrations costs and the medium number of computing nodes. The algorithm MO-2-GS requires further improvement before it can be advised to be used.

5 Conclusions

The paper has presented a multi-objective approach applied to Extremal Optimization used in processor load balancing in execution of distributed programs. Additional approach of the EO-GS has been embedded in the fundamental EO algorithm which improves the convergence of the entire algorithm. In the multi-objective EO approach, three objectives relevant in processor load balancing for distributed applications are simultaneously controlled: total computational load balance, total volume of external communication between tasks placed on different processors and parameters of task migration. Different global fitness function variants for computational load balancing and task migration minimalization were designed and verified. The proposed algorithms were assessed by simulation experiments on EO-controlled execution of macro data flow graphs of distributed programs. The experiments have shown that the multi-objective approach added to the EO algorithms for load balancing has improved the quality of obtained results. More detailed coverage of the internal properties of the proposed multi-objective algorithms including the analysis of the Pareto front graph itself will be presented in a larger journal paper which is currently under preparation.

References

1. Boettcher, S., Percus, A.G.: Extremal optimization: methods derived from co-evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999). Morgan Kaufmann, San Francisco, pp. 825–832 (1999)
2. Barker, K., Chrisochoides, N.: An evaluation of a framework for the dynamic load balancing of highly adaptive and irregular parallel applications. In: Proceedings of the ACM/IEEE Conference on Supercomputing, Phoenix. ACM Press (2003)
3. De Falco, I., Laskowski, E., Olejnik, R., Scafuri, U., Tarantino, E., Tudruj, M.: Load balancing in distributed applications based on extremal optimization. In: Esparcia-Alcázar, A.I. (ed.) EvoApplications 2013. LNCS, vol. 7835, pp. 52–61. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37192-9_6
4. De Falco, I., Laskowski, E., Olejnik, R., Scafuri, U., Tarantino, E., Tudruj, M.: Improving extremal optimization in load balancing by local search. In: Esparcia-Alcázar, A.I., Mora, A.M. (eds.) EvoApplications 2014. LNCS, vol. 8602, pp. 51–62. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45523-4_5
5. De Falco, I., Laskowski, E., Olejnik, R., Scafuri, U., Tarantino, E., Tudruj, M.: Extremal optimization applied to load balancing in execution of distributed programs. Appl. Soft Comput. **30**, 501–513 (2015)
6. Taxicab geometry. https://en.wikipedia.org/wiki/Taxicab_geometry. Accessed 20 Nov 2017
7. Xu, C., Lau, Francis C.M.: Load Balancing in Parallel Computers: Theory and Practice. Kluwer Academic Publishers, Dordrecht (1997)

8. Khan, R.Z., Ali, J.: Classification of task partitioning and load balancing strategies in distributed parallel computing systems. Int. J. Comput. Appl. **60**(17), 48–53 (2012)
9. Mishra, M., Agarwal, S., Mishra, P., Singh, S.: Comparative analysis of various evolutionary techniques of load balancing: a review. Int. J. Comput. Appl. **63**(15), 8–13 (2013)
10. Sneppen, K., et al.: Evolution as a self-organized critical phenomenon. Proc. Nat. Acad. Sci. **92**, 5209–5213 (1995)
11. Zeigler, B.: Hierarchical, modular discrete-event modelling in an object-oriented environment. Simulation **49**(5), 219–230 (1987)
12. Roig, C., Ripoll, A., Guirado, F.: A new task graph model for mapping message passing applications. IEEE Trans. Parallel Distrib. Syst. **18**(12), 1740–1753 (2007)
13. Collette, Y., Siarry, P.: Multi-objective Optimization: Principles and Case Studies. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-08883-8. p. 293
14. Ehrgott, M.: Multi-criteria Optimization. Springer, Heidelberg (2005). https://doi.org/10.1007/3-540-27659-9. p. 324
15. Coello Coello, C.A.: Evolutionary multi-objective optimization: a historical view of the field. IEEE Comput. Intell. Mag. **1**, 28–36 (2006)
16. Coello Coello, C.A., Lamont, G.B., Van Veldhuizen, D.A.: Evolutionary Algorithms for Solving Multi-Objective Problems. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-36797-2. p. 800
17. Chen, M.-R., Lu, Y.-Z.: A novel elitist multi-objective optimization algorithm: multi-objective extremal optimization. Shanghai Jiao Tong University
18. Ahmed, E., Elettreby, M.F.: On multi-objective evolution model. Int. J. Mod. Phys. C **15**(9), 1189–1195 (2004)
19. Gómez-Meneses, P., Randall, M., Lewis, A.: A hybrid multi-objective extremal optimisation approach for multi-objective combinatorial optimisation problems. Bond University, Griffith University, Australia (2010)
20. Galski, R.L., de Sousa, F.L., Ramos, F.M., Muraoka, I.: Spacecraft thermal design with the generalized extremal optimization algorithm. In: Proceedings of Inverse Problems, Design and Optimization Symposium, Rio de Janeiro, Brazil, 2004
21. Chen, M., Lu, Y., Yang, G.: Multi-objective extremal optimization with applications to engineering design. J. Zhejiang Univ. Sci. A **8**(12), 1905–1911 (2007)
22. De Falco, I., Della Cioppa, A., Maisto, D., Scafuri, U., Tarantino, E.: A multiobjective extremal optimization algorithm for efficient mapping in grids. In: Mehnen, J., Köppen, M., Saad, A., Tiwari, A. (eds.) Applications of Soft Computing. Advances in Intelligent and Soft Computing. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-89619-7_36

Workshop on Language-Based Parallel Programming Models (WLPP 2017)

Pardis: A Process Calculus for Parallel and Distributed Programming in Haskell

Christopher Blöcker(✉) and Ulrich Hoffmann

Department of Computer Science,
FH Wedel, University of Applied Sciences, Wedel, Germany
{chb,uh}@fh-wedel.de

Abstract. Parallel and distributed programming involve substantial amounts of boilerplate code for process management and data synchronisation. This leads to increased bug potential and often results in unintended non-deterministic program behaviour. Moreover, algorithmic details are mixed with technical details concerning parallelisation and distribution. Process calculi are formal models for parallel and distributed programming but often leave details open, causing a gap between formal model and implementation. We propose a fully deterministic process calculus for parallel and distributed programming and implement it as a domain specific language in Haskell to address these problems. We eliminate boilerplate code by abstracting from the exact notion of parallelisation and encapsulating it in the implementation of our process combinators. Furthermore, we achieve correctness guarantees regarding process composition at compile time through Haskell's type system. Our result can be used as a high-level tool to implement parallel and distributed programs.

Keywords: Process calculus · Parallel programming
Distributed programming · Domain specific language · Haskell

1 Introduction

Since the start of the multi-core revolution, parallel programming has gained more and more importance. Nowadays, multi-core hardware is found not only in servers and desktop computers, but also in low-end laptops and smartphones. In order to leverage the available processing power of modern hardware, software must be implemented accordingly, i.e., in a parallel fashion.

Parallel programs should create the exact same results as their sequential counterparts and must be distinguished from *concurrent* programs. While *parallelism* is used to execute and obtain a program's result faster by running parallel parts of the program on different processing units, *concurrency* is more of a software engineering technique to better structure multiple tasks a program has

C. Blöcker—This work was done while the author was working on his Master's degree at FH Wedel.

to carry out. *Concurrency* may involve hundreds of threads that are executed on one or more processing units in an interleaved way and can, but does not necessarily use *parallelism*. Whereas determinism is generally a requirement for parallel programs, there are concurrent algorithms that rely on non-determinism to find their solution.

Contemporary ways to express *parallelism* and *concurrency* are based on the same programming constructs which allow non-determinism. This includes processes and threads which usually have a mutable internal state and can depend on other processes or threads, making process management and data synchronisation necessary. Process management and data synchronisation, however, result in large amounts of repetitive boilerplate code and increased software complexity and give rise to new classes of bugs, such as deadlocks, livelocks or unintended non-deterministic program behaviour.

On the theory side, process calculi are used to model parallel and concurrent systems and to reason about them in a mathematical way. Typically, processes are modelled as stateful sequential programs and communication between them is expressed explicitly. Semantics of parallel process execution is defined non-deterministically so that, when carried over into a programming model, a choice of how to represent this non-determinism has to be made, resulting in a gap between specification and implementation. In order to achieve deterministic programs, software developers have to take special care, but this is far from trivial.

In this paper, we propose a deterministic process calculus for parallel and distributed programming to address these problems. We define a set of process combinators to compose simpler processes into more complex ones and eliminate the need for manual process management and data synchronisation, thus reducing error-prone boilerplate code and removing potential sources for bugs. We show that the same abstraction can be used to express both parallel and distributed programs. Our Haskell implementation is in one-to-one correspondence with our formal model with no gap between specification and implementation and gives developers a high-level tool to implement parallel and distributed programs.

2 Related Work

Various process calculi have been proposed over the past decades, including Hoare's *Communicating Sequential Processes* (CSP) [8] and Milner's *Calculus of Communicating Systems* (CCS) [15]. CSP and CCS (and others) can be derived from a more general algebraic core [9] and share a set of properties, including mutable process-internal state, explicit communication between processes, and non-deterministic process compositions [8,9,15]. Non-deterministic behaviour is relevant for certain concurrent algorithms and gives rise to interesting research questions, but is undesired in many real-world applications where reproducible behaviour is an essential requirement [2,4].

Various models for parallel programming with different focus have been introduced: while some take a low level approach and require manual management of, e. g., threads and data locks, others take higher level approaches with different degrees of automation regarding process management and communication [1]. Although low level approaches have been considered prone to errors and bugs for over a decade [1,6,11], they are still widely used. High level approaches are promoted by the scientific community as a way to make parallel programming easier and better manageable [1,4].

Haskell as a purely functional, statically typed and highly extensible language with isolated side-effect is well suited for parallel programming [1,4]. Different embedded domain specific languages for parallel programming have reached a sufficiently sophisticated level for production use [3,5,12–14,21]. The *parallel* package provides the `Eval` monad for pure, parallel evaluation using so-called *strategies* [13,21]. The `Par` monad from the *monad-par* package provides a programming model based on dataflow parallelism [14]. *repa* implements parallelised operations on arrays, and *accelerate* can be used to exploit massive parallelism on GPUs [3]. Concurrent programming in *Concurrent Haskell* is supported by lightweight threads using `forkIO` with shared variables (`MVars`) [16] or Software Transactional Memory (`STM`) [7] for synchronisation. *Cloud Haskell*, which is heavily influenced by Erlang, is designed for distributed programming and uses message passing for process communication [5].

3 The Calculus

In this section, we develop our process calculus and define its syntax, static typing rules, and semantics. For background on the used methodology we point to [18,20].

But what do we mean by *process*? A process is (a piece of) a computer program. It can be executed by supplying it with an input of appropriate type and it then produces an output. Processes can either be *basic* or *composed*. *Basic* processes are structurally minimal, i. e., not created by composition. *Composed* processes are composed of other (basic or composed) processes.

3.1 Syntax

Our process calculus comprises four syntactic rules. Let P, Q, R be (basic or composed) processes, then syntactically correct (composed) processes can be formed using the following rules:

1. $(P \triangleright Q)$ (Sequence)
2. $(R \propto P)$ (Repetition)
3. $(R \rightarrow P \vee Q)$ (Choice)
4. $(R \leftarrow P \,|\, Q)$ (Parallel)

The set of syntactically correct processes is the smallest set that includes all basic processes and composed processes constructed with the above rules.

3.2 Static Typing

For each syntactic rule, there is a corresponding static typing rule. Process composition is valid if and only if it follows the static typing rules, all other compositions are invalid. Without loss of generality, we consider unary processes in the following, i. e., processes that take exactly one argument, (un-)currying where necessary. Other arities can be simulated by using the unit type or product types.

Let $\mathcal{P}$ be the set of processes, let $P, Q, R \in \mathcal{P}$ be processes, let a, b, c, d be type variables and let *Bool* be the boolean type. Let $P\colon a \rightarrow b$ denote that process P is of type $a \rightarrow b$, i. e., P takes an input value of type a and produces and output value of type b. Let $\frac{x_1 \ \cdots \ x_n}{z}$ denote that if $x_1 \wedge \cdots \wedge x_n$, then z. The typing rules for process composition are:

$$\frac{P\colon a \rightarrow c \quad Q\colon c \rightarrow b}{(P \triangleright Q) : a \rightarrow b} \tag{1}$$

$$\frac{R\colon a \rightarrow \mathit{Bool} \quad P\colon a \rightarrow a}{(R \propto P) : a \rightarrow a} \tag{2}$$

$$\frac{R\colon a \rightarrow \mathit{Bool} \quad P\colon a \rightarrow b \quad Q\colon a \rightarrow b}{(R \rightarrow P \vee Q) : a \rightarrow b} \tag{3}$$

$$\frac{R\colon (c, d) \rightarrow b \quad P\colon a \rightarrow c \quad Q\colon a \rightarrow d}{(R \leftarrow P \,|\, Q) : a \rightarrow b} \tag{4}$$

3.3 Semantics

We define the semantics of processes in a denotational fashion and introduce the polymorphic higher order function $\mathit{sem}\colon \mathcal{P} \rightarrow (a \rightarrow b)$ for this purpose. *sem* can be applied to a process $P \in \mathcal{P}$ to obtain the function calculated by P, we write $\mathit{sem}\,\langle P \rangle$ for this. Note that we use lifted types to represent partial functions and processes where the undefined value is denoted $\perp$[1].

The semantics of a basic process is straightforward: let $P \in \mathcal{P}$ be a basic process and f_P be the function calculated by P, then the semantics of P is

$$\mathit{sem}\,\langle P \rangle = f_P \tag{5}$$

The semantics of a composed process depends on the semantics of its *subprocesses* and the process combinator. Let $P, Q, R \in \mathcal{P}$ be processes with appropriate types for the respective composition in the following. Then the semantics of composed processes are:

$$\mathit{sem}\,\langle (P \triangleright Q) \rangle = x \mapsto \mathit{sem}\,\langle Q \rangle\,(\mathit{sem}\,\langle P \rangle\,(x)) = \mathit{sem}\,\langle Q \rangle \circ \mathit{sem}\,\langle P \rangle \tag{6}$$

$$\mathit{sem}\,\langle (R \propto P) \rangle = x \mapsto \begin{cases} \mathit{sem}\,\langle (P \triangleright (R \propto P)) \rangle\,(x) & \mathit{if}\ \mathit{sem}\,\langle R \rangle\,(x) = \mathit{true} \\ x & \mathit{if}\ \mathit{sem}\,\langle R \rangle\,(x) = \mathit{false} \\ \perp & \mathit{otherwise}. \end{cases} \tag{7}$$

[1] To be precise, there is an undefined value $\perp_a$ for every type a, e. g., we have $\mathit{Bool} = \{\perp_{\mathit{Bool}}, \mathit{false}, \mathit{true}\}$. We omit the type index since it shall be clear from context.

$$sem\,\langle (R \rightarrow P \vee Q) \rangle = x \mapsto \begin{cases} sem\,\langle P \rangle\,(x) & if\; sem\,\langle R \rangle\,(x) = true \\ sem\,\langle Q \rangle\,(x) & if\; sem\,\langle R \rangle\,(x) = false \\ \bot & otherwise. \end{cases} \qquad (8)$$

$$sem\,\langle (R \leftarrow P \mid Q) \rangle = x \mapsto sem\,\langle R \rangle\,(sem\,\langle P \rangle\,(x)\,, sem\,\langle Q \rangle\,(x)) \qquad (9)$$

The existence of a solution satisfying Eq. (7) is not immediately obvious. However, this is a well-studied problem in domain theory and a solution can be characterised by a least fixed point [20].

4 Implementation

In principle, we could implement our model in virtually any programming language, including imperative languages such as C or Java. However, there are some important restriction: firstly, in order to guarantee type-safe process composition, we require a statically typed language. Secondly, in impure languages where threads operate on shared memory, it is harder, and in some cases not possible, to achieve determinism. For example, in Deterministic Parallel Java [2], fallbacks to runtime checks are necessary and runtime exceptions may be thrown when threads perform concurrent data accesses.

For our implementation, we choose Haskell. Haskell's purity, static type system, and effect system enable us to implement our model in one-to-one correspondence with the definitions given in Sect. 3.

```haskell
data Proc p a b where
  Lifted     :: (Pardis m, p ~ Basic m) => p a b              -> Proc p a b
  Sequence   ::                    Proc p a c -> Proc p c b -> Proc p a b
  Repetition :: Proc p a Bool    -> Proc p a a              -> Proc p a a
  Choice     :: Proc p a Bool    -> Proc p a b -> Proc p a b -> Proc p a b
  Parallel   :: Proc p (c, d) b -> Proc p a c -> Proc p a d -> Proc p a b
```

Listing 1: Implementation of the process data type `Proc`.

We implement the data type of processes `Proc` as a generalised algebraic data type using the language extension `GADTs` [17], c.f. listing 1. There is one data constructor for each syntactic rule from Sect. 3.1 with type signatures corresponding to Eqs. (1)–(4). We introduce another type constructor to lift basic processes into the calculus, namely `Lifted`. `Proc` has three type parameters: a basic process type `p`, an input type `a`, and an output type `b`. In order for processes to be composable, they must share the same basic process type `p`.

```haskell
type P m a b = Proc (Basic m) a b  ---- type alias for convenience

class Monad m => Pardis m where
  data Basic m :: * -> * -> *      ---- data type of basic processes
  data Env   m :: *                ---- environment for process execution

  runBasic      :: Env m -> Basic m a b  ->                       a -> m b
  runSequence   :: Env m ->                P m a c -> P m c b -> a -> m b
  runRepetition :: Env m -> P m a Bool   -> P m a a ->            a -> m a
  runChoice     :: Env m -> P m a Bool   -> P m a b -> P m a b -> a -> m b
  runParallel   :: Env m -> P m (c, d) b -> P m a c -> P m a d -> a -> m b

  runSequence env p q x = runProcess env p x >>= runProcess env q

  runRepetition env r p x = do
    b <- runProcess env r x
    if b then runProcess env (Sequence p (Repetition r p)) x
         else return x

  runChoice env r p q x = do
    b <- runProcess env r x
    runProcess env (if b then p else q) x
```

Listing 2: Implementation of type class `Pardis`.

We use the language extension `TypeFamilies` [19] to define the type class `Pardis` for process execution based on a monad `m`, c.f. listing 2. Basic processes and evaluation environments, i. e., `Basic m` and `Env m`, are associated with `m`. `Basic m` is of kind `* -> * -> *`, i. e., it is a type constructor that takes two type parameters, corresponding to input and output type of a process, and returns a type. `Env m` is of kind `*`, i. e., it is a type, and is used to provide information that is necessary for process execution in `m`. The type alias `P` is introduced for brevity. `Pardis` defines functions for process execution for each data constructor of `Proc`. We specifically do not implement a single function for processes execution *within* `Pardis` since execution of composed processes using `Sequence`, `Choice`, and `Repetition` is independent of the concrete monad `m`. We give default implementations according to Eqs. (6)–(8) for these cases.

```haskell
runProcess :: Pardis m => Env m -> Proc (Basic m) a b -> a -> m b
runProcess env (Lifted       p  ) = runBasic      env   p
runProcess env (Sequence     p q) = runSequence   env   p q
runProcess env (Repetition r p  ) = runRepetition env r p
runProcess env (Choice     r p q) = runChoice     env r p q
runProcess env (Parallel   r p q) = runParallel   env r p q
```

Listing 3: `runProcess` executes processes by pattern matching on the constructor.

The function `runProcess` executes processes by pattern matching on their constructor and dispatching to the corresponding function in `Pardis`, c.f. listing 3.

4.1 Parallel Implementation

For process execution in parallel, we use *Concurrent Haskell* [16] and the `IO` monad, c.f. listing 4. Basic processes in this case hold an `IO` action of type `a -> IO b` and a value of type `Dict (NFData b)`, i. e., a proof that `b` has an instance of `NFData`[2]. Evaluation environments are empty as there is no additional information needed for parallel processes execution in `IO`.

```haskell
instance Pardis IO where
  data Basic IO a b = Basic (a -> IO b) (Dict (NFData b))
  data Env   IO     = Env

  runBasic _ (Basic f _) x = f x

  runParallel env r p q x = case (getDict p, getDict q) of
    (Dict, Dict) -> do
      mvar <- newEmptyMVar
      forkIO $ runProcess env p x >>= putMVar mvar . force
      qx   <- force <$> runProcess env q x
      px   <- takeMVar mvar
      runProcess env r (px, qx)
```

Listing 4: `Pardis` instance for parallel processes execution using the `IO` monad.

We execute basic processes by applying their intrinsic action to their input as defined in Eq. (5), and processes involving the `Parallel` combinator in a (lightweight) concurrent thread with `forkIO`. The implementation is according to Eq. (9) and we enforce full evaluation of sub-results in parallel with `force`.

We make process composition deterministic by introducing implicit synchronisation points in the interpretation of the `Parallel` process combinator. For that, we use Haskell's `MVars` which represent a data container that can either be empty or hold exactly one element. Reading a value from an `MVar` succeeds if the `MVar` contains a value and blocks until a value becomes available otherwise.

Note that, in general, we would place constraints such as `NFData b` in the context of functions and data constructors where they are required. However, this is not an option here since adding constraints to `Proc` or `Pardis` would be too strict and enforce them on *all* possible instances.

[2] `NFData` stands for *normal form data* and values of data types with an instance of `NFData` can be fully evaluated. Haskell's evaluation strategy is *lazy evaluation*, i. e., values are only evaluated if they are needed. However, through `NFData` we can enforce full evaluation *in parallel* to benefit from parallelisation.

```haskell
getDict :: Proc (Basic Eval) a b -> Dict (NFData b)
getDict (Lifted     p  ) = case p of Basic _ dict -> dict
getDict (Sequence   _ q) = getDict q
getDict (Repetition _ p) = getDict p
getDict (Choice   _ p _) = getDict p
getDict (Parallel r _ _) = getDict r
```

Listing 5: `getDict` helper function to retrieve dictionaries from processes.

In order to use `force`, Haskell's type system requires instances of `NFData` for the respective types in scope. We retrieve these instances from the basic processes at the bottom layer of process composition with `getDict` and bring them into scope by pattern matching, cf. listings 4 and 5.

4.2 Distributed Implementation

For distributed process execution, we use the `Process` monad from *Cloud Haskell* [5], c.f. listing 6. Basic processes in this case hold a closure generator[3], i. e., a function that takes a value and returns a closure, and a value of type `Static (SerializableDict b)`, i. e., a compile time constant that describes how to serialise values of type `b`. Evaluation environments contain an action `getNode`[4] that, when executed, returns a `NodeId`, i. e., the address of a (local or remote) node that can be used to execute closures.

```haskell
instance Pardis Process where
  data Basic Process a b = Serializable b
    => Basic (a -> Closure (Process b)) (Static (SerializableDict b))
  data Env   Process     = Env { getNode :: Process NodeId }

  runBasic env (Basic f dict) x = getNode env >>= \n -> call dict n (f x)

  runParallel env r p q x = do
    mvar <- liftIO newEmptyMVar
    spawnLocal $ runProcess env p x >>= liftIO . putMVar mvar
    qx   <- runProcess env q x
    px   <- liftIO $ takeMVar mvar
    runProcess env r (px, qx)
```

Listing 6: `Pardis` instance for distributed processes execution using Cloud Haskell.

We execute basic processes by generating a closure which we then run on a (local or remote) node using `call`. Parallel processes are executed by spawning a (lightweight) local helper process[5]. One of the two parallel sub-processes is then

[3] Roughly speaking, a closure is a data structure that contains an executable computation together with inputs for that computation.
[4] We assume there is a way to obtain a node but omit node management for brevity.
[5] Note that there is a difference between Pardis processes and *Cloud Haskell* processes.

executed in the helper process while the other one is executed in the current process. As before, we use an MVar to introduce an implicit synchronisation point. Once both sub-processes have terminated, we combine their results. In this case, explicit full evaluation of the results is not necessary as this is done implicitly before serialisation.

5 Application Example: Web Crawler

Here, we present a rudimentary web crawler for indexing web pages with the purpose to illustrate how Pardis can be used to implement parallel and distributed programs. The parallel and distributed implementations are almost identical but differ in technicalities regarding the creation of basic processes. We display the relevant pieces of the parallel implementation here, while the full source code (including the distributed version) can be found on github[6].

```haskell
data Index = Index { done :: URLs
                   , todo :: URLs
                   , index :: Map String URLs
                   }
  deriving (Generic, NFData)

merge   :: Index -> Index -> Index  ---- for merging indices

mkBasic :: NFData b => (a -> IO b) -> Proc (Basic IO) a b
liftP   :: NFData b => (a ->    b) -> Proc (Basic IO) a b
```

Listing 7: The Index data type and helper functions.

The crawler's aim is to create an Index. An Index holds URLs that have been crawled, URLs that should be crawled, as well as the crawling results, i. e., a Map that associates words with URLs.

page retrieves the document from a given URL and creates a single-page index from its content, c.f. listing 8. In order to keep the code simple, we ignore errors and exceptions that may arise when retrieving web pages for now.

The helper functions mkBasic and liftP create basic processes and lift pure functions into processes, respectively, c.f. listing 7.

When crawling with crawl, we check with continue whether there is work left to do, c.f. listing 8. We repeat indexing all URLs currently in todo with crawlAll until todo becomes empty. crawlAll builds up a process structure in the shape of a binary tree for crawling all URLs in todo. The leaf nodes of the tree contain crawlOne processes and the inner nodes contain crawlMany processes that use merge to combine the single-page indices created at the leaf nodes. Note that the web crawler does not involve any code for process management but is solely expressed in terms of our process combinators.

[6] https://github.com/chrisbloecker/pardis.

```haskell
page :: URL -> IO Index
page url = do
  html <- parseTags . BS.unpack <$> simpleHttp url
  let (links, words) = (getLinks html, getWords html)
  return $ Index [url] links (fromList [(w, [url]) | w <- words])

crawl :: Proc (Basic IO) Index Index
crawl = Repetition (liftP continue) (liftP todo `Sequence` crawlAll)
  where
    continue  = not . null . todo
    bisect l  = let n = length l `div` 2 in (take n l, drop n l)
    crawlAll  = Choice (liftP $ length `is` 1) crawlOne crawlMany
    crawlOne  = liftP head   `Sequence` mkBasic page
    crawlMany = liftP bisect `Sequence` Parallel (liftP $ uncurry merge)
                                           (liftP fst `Sequence` crawlAll)
                                           (liftP snd `Sequence` crawlAll)
```

Listing 8: A rudimentary web crawler that uses `Pardis` for parallelisation.

6 Conclusion

Our main goal was to eliminate sources of boilerplate code in parallel and distributed programming and thereby reducing bug potential. For that, we designed a process calculus with process combinators for sequence, repetition, choice, and parallel composition and gave their semantics in a denotational fashion. Our abstractions are similar to those of arrows [10], a general interface to computation. However, we make specialisations in order to allow for parallelism and distribution in the interpretation of computations. Process management and synchronisation points between processes are implicit and incorporated into the semantics of our process combinators. In contrast to other calculi, the semantics of our calculus are deterministic and therefore well suited to express parallel and distributed programs that behave the same as their sequential counterparts.

We implemented our process calculus as a domain specific language in Haskell and used Haskell's type system to ensure that only well-typed processes can be expressed. We presented two implementations of our calculus, i. e., one for parallel and one for distributed programming. The implementations are in one-to-one correspondence with our formal model which lays the foundation for automated reasoning about and optimisation of processes, which we believe is worth investigating further.

We demonstrated the utility of our approach by implementing a web crawler that does not involve code for process management or communication but uses our calculus to model parallelism. Other potential use cases include, e. g., implementations of data processing pipelines, linear algebra algorithms, or algoritms for discrete optimisation.

Typical steps to transform sequential programs into parallel ones are to find a suitable partition into parallel pieces and to implement these pieces.

The implementation step is where error-prone boilerplate code for process management is used and bugs are easily introduced. However, our approach replaces the implementation step by modelling parallel processes using the parallel combinator which removes this source of bugs. Our calculus completely isolates the exact notion of parallelisation inside its implementation which in effect can be interchanged without modifying programs that use it. As such, `Pardis` can be seen as a high-level tool to describe parallel and distributed programs.

Acknowledgements. We would like to thank Jan-Philip Loos and Uwe Schmidt for inspiring discussions and helpful feedback.

References

1. Belikov, E., Deligiannis, P., Totoo, P., Aljabri, M., Loidl, H.W.: A survey of high-level parallel programming models. Technical report. HW-MACS-TR-0103, Department of Computer Science, Heriot-Watt University (December 2013)
2. Bocchino Jr., R.L., Adve, V.S., Adve, S.V., Snir, M.: Parallel programming must be deterministic by default. In: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, HotPar 2009, p. 4. USENIX Association, Berkeley (2009). http://dl.acm.org/citation.cfm?id=1855591.1855595
3. Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating haskell array codes with multicore gpus. In: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, pp. 3–14. ACM, New York (2011). http://doi.acm.org/10.1145/1926354.1926358
4. Coutts, D., Löh, A.: Deterministic parallel programming with haskell. Comput. Sci. Eng. **14**(6), 36–43 (2012)
5. Epstein, J., Black, A.P., Peyton-Jones, S.: Towards haskell in the cloud. In: Proceedings of the 4th ACM Symposium on Haskell, Haskell 2011, pp. 118–129. ACM, New York (2011). http://doi.acm.org/10.1145/2034675.2034690
6. Gorlatch, S.: Send-receive considered harmful: myths and realities of message passing. ACM Trans. Program. Lang. Syst. **26**(1), 47–56 (2004). http://doi.acm.org/10.1145/963778.963780
7. 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, PPoPP 2005, pp. 48–60. ACM, New York (2005). http://doi.acm.org/10.1145/1065944.1065952
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall Inc., Upper Saddle River (1985)
9. Hoare, T., van Staden, S.: The laws of programming unify process calculi. In: Gibbons, J., Nogueira, P. (eds.) MPC 2012. LNCS, vol. 7342, pp. 7–22. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31113-0_2
10. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1–3), 67–111 (2000). http://dx.doi.org/10.1016/S0167-6423(99)00023-4
11. Lee, E.A.: The problem with threads. Computer **39**(5), 33–42 (2006). http://dx.doi.org/10.1109/MC.2006.180
12. Marlow, S.: Parallel and concurrent programming in haskell. In: Zsók, V., Horváth, Z., Plasmeijer, R. (eds.) CEFP 2011. LNCS, vol. 7241, pp. 339–401. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32096-5_7

13. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.: Seq no more: better strategies for parallel haskell. In: Proceedings of the Third ACM Haskell Symposium on Haskell, Haskell 2010, pp. 91–102. ACM, New York (2010). http://doi.acm.org/10.1145/1863523.1863535
14. Marlow, S., Newton, R., Peyton Jones, S.: A monad for deterministic parallelism. In: Proceedings of the 4th ACM Symposium on Haskell, Haskell 2011, pp. 71–82. ACM, New York (2011). http://doi.acm.org/10.1145/2034675.2034685
15. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York Inc., Secaucus (1982). https://doi.org/10.1007/3-540-10235-3
16. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent haskell. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1996, pp. 295–308. ACM, New York (1996). http://doi.acm.org/10.1145/237721.237794
17. Peyton Jones, S., Washburn, G., Weirich, S.: Wobbly types: type inference for generalised algebraic data types. Technical report, MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104–6389, July 2004
18. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. William C. Brown Publishers, Dubuque (1986)
19. Schrijvers, T., Peyton Jones, S., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. SIGPLAN Not. **43**(9), 51–62 (2008)
20. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge (1977)
21. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + strategy = parallelism. J. Funct. Program. **8**(1), 23–60 (1998). http://dx.doi.org/10.1017/S0956796897002967

Towards High-Performance Python

Ami Marowka(✉)

Parallel Research Lab, Jerusalem, Israel
amimar2@yahoo.com

Abstract. Python became the preferred language for teaching in academia, and it is one of the most popular programming languages for scientific computing. This wide popularity occurs despite the weak performance of the language. This weakness is the motivation that drives the efforts devoted by the Python community to improve the performance of the language. In this article, we are following these efforts while we focus on one specific promised solution that aims to provide high-performance for Python applications.

Keywords: Python · Numba · Just-in-Time compilation

1 Introduction

Python is widely used for teaching programming in academia today [1]; it has been the first choice of coding language among software programmers in the last four years [2]; it is ranked fifth in the TIOBE Index [3], a measure of popularity of programming languages; and its popularity is also ranked third in the IEEE Spectrum ranking for 2016 [4].

Python is also widely used for scientific computing. Scientific applications make use of a very rich readymade collection of scientific libraries such as SciPy [5], NumPy [6], and Matplotlib [7]. However, the performance of Python is considered slow compared to compiled languages such as C, C++, and FORTRAN, especially for heavy computations. The main reasons for its slowness lie in being an interpreted programming language and its limitations on concurrency [8]. Therefore, the Python community has developed various solutions in order to improve the performance and speed of Python.

The range of solutions and techniques offered in order to overcome the slowness of Python adopt different approaches and various optimization techniques. The following are some examples. NumPy uses specialized data structures (densely packed arrays of homogeneous type) that use efficient in-memory representation. Furthermore, NumPy uses efficient, specialized implementations of operations in C and dynamic optimization; SciPy usually implements time-intensive loops in C or FORTRAN and uses sophisticated wrappers to wrap up existing optimized scientific algorithms. PyPy [9] is a Python implementation written in RPython (a subset of pure Python). RPython code is statically compiled and optimized to generate the PyPy interpreter in C while automatically adding in features as garbage collection and a Just-In-Time compiler. PyCUDA [10] and PyOpenCL [11] are Python wrappers, of the software libraries CUDA and OpenCL, respectively, which exploit the massive parallelism computing power provided by GPGPU platforms.

In this article, we describe a select number of solutions that are widely used by the community and focus on one specific solution: Numba [12]. Numba offers a comprehensive, user-friendly solution for portable high performance computing.

This article is organized as follows. Section 2 describes a few popular solutions that enhance Python's performance. Section 3 introduces the Numba compiler. Section 4 analyses the performance of a Numba implementation of Matrix Multiplication algorithm. Finally, Sect. 5 concludes the article.

2 Python Accelerators

In this section, we review a few notable solutions that are widely used in the scientific computing domain, which is characterized by high performance applications.

NumPy (Numerical Python) is an open source extension module for scientific computing in Python. NumPy handles the inefficiency problem of Python by providing fast, precompiled functions for mathematical and numerical routines that operate efficiently on multi-dimensional arrays of homogeneous data types called *ndarray* objects. *Ndarray* is internally based on C arrays, and therefore, it can easily interface NumPy with existing C code. NumPy arrays are a data structure of elements of the same data type that allows it to efficiently pack the data and store it in a way that NumPy can perform arithmetic and mathematical operations at high speed. These routines include mathematical and logical routines, sorting, shape manipulation, selecting, discrete FFT, basic linear algebra (based on BLAS and LAPACK), basic statistical operations, and random simulation. *Vectorization* and *broadcasting* are the two most powerful features of the package. Specifically, *vectorization* enables arithmetic operations to be performed on an array without writing any for loops. By implementing in C element-wise operations on the array rather than applying element-by-element computation, NumPy accelerates computations dramatically. For example, evaluation of the function f below over 1e5 numbers stored in an array x while using a for-loop appears as follows:

Listing 2.1. NumPy Example.

```
In [24]: def f(x):
    ...:  ...: return x**3 - 4*x**2 + 5*x
In [25]: x = np.arange(1e5)
In [26]: y = np.arange(1e5)

In [27]: %timeit y = [f(i) for i in x]
10 loops, best of 3: 77.9 ms per loop
```

Applying the function f on the NumPy array x using vectorized loop appears as follows:

```
In [28]: %timeit y = f(x)
100 loops, best of 3: 213 micro seconds per loop
```

As can be observed, a speedup of x365 is obtained.

SciPy (Scientific Python) is an open source package for science and engineering for Python. It extends the functionality of NumPy with a substantial collection of efficient numerical routines for minimization, regression, signal and image processing, Fourier-transformation, and applied mathematical techniques. Comparing the performance of LU factorization of square random matrices using SciPy against pure Python shows incomprehensible speed-up of x4142 [13].

PyPy is a Just-in-Time compiler and interpreter for Python. It aims to provide a faster, efficient, and compatible alternative implementation of Python language. It achieves these features by using a different interpreter language (RPython) instead the original CPython interpreter. Moreover, PyPy has a Just-in-Time compiler that is capable to compile Python code on-the-fly into a machine code. Applications written with PyPy demand less memory footprint and have the capability to use micro-threads for massive concurrency. PyPy trunk (with JIT) benchmark yielding a geometric average speed-up of x7.6 compared to pure Python [14].

Cython [15] is a superset compiled language of Python aimed to achieve comparable performance to C programming language. By enabling declarations of static typing to functions, variables, and classes, Cython allows C code to be generated once and then compiles with C/C++ compilers to produce efficient C code. In average, Cython exhibits speedup of x30 compared to pure Python.

Numexpr [16] is a module that aims to accelerate evaluation of a numerical expression operation on NumPy arrays while using less memory footprint compared to pure Python. Moreover, Numexpr uses an efficient multithreading mechanism for speedup computations by taking advantage of multi-core architectures. These capabilities are reached by avoiding memory allocation of temporary results that improve cache utilization, using an integrated virtual machine that parses expressions into its own op-codes, and splitting array operands into chunks that fit in cache memory. In average, Numexpr outperforms NumPy by x30 for four threads.

Parallelism is often one of the solutions offered by almost any programming language today for increasing performance. The Python community offers many solutions based on parallelism. For example, the multiprocessing module [17], which is part of the standard library of the language, implements process-based parallel programming for shared memory systems.

In the next section, we describe in more detail how to use Numba compiler, in order to increase the performance of applications that are written in Python language.

Listing 3.1. An example for annotating a function by using @jit compiler directive.

```
from numba import jit
@jit (nopython=True | nogil=True | cache=True)
def mult(x, y):
    return x * y
```

3 Numba in a Nutshell

Numba is a NumPy-aware Python programming model and a Just-in-Time compiler based on source-code annotations. Numba uses the LLVM compiler for generating optimized machine code similar in performance to C. It was designed in mind for array-oriented and numerical code that supports CPUs, CUDA GPUs, and HSA APUs. Numba is in active development (the current version is 0.30.1). Numba is part of Anaconda Accelerate [18], which is available under a free license for academic users. It runs on top of Anaconda Python [19], which is a completely free package and environment manager for large-scale data processing and scientific computing. It includes hundreds of open source packages to include the popular packages of NumPy, SciPy, Matplotlib, IPython, and Spider IDE.

Numba Just-in-Time Compilation

The example in Listing 3.1 demonstrates how annotation of a function *mult* that adds two scalars is performed. The compiler directive *@jit*, which is attached to the function header, will cause the function *mult* to be compiled on-the-fly for generating optimized code.

There are three compiler options that can be passed to the called function. The first one is *nopython=True* that instructs the compiler to run in a *nopython* mode that generates faster code while preventing Numba to fall back to the slower mode (*object mode*) when an error occurs. The *nogil=True* option instructs the compiler to release Python's *global interpreter lock* (GIL) that allows the compiled code to run in parallel while using the code's threads to take advantage of multi-core machines. The *cache=True* option instructs the compiler to write the compiled function into a file-based cache for avoiding the need to compile the code each time the Python program is invoked.

Numba Vectorization

Numba vectorization can be applied in one of two forms: *vectorize* or *guvectorize. Vectorize* uses scalar arguments, which are passed by Python functions, as NumPy universal functions that work on one element of input array at a time (but not in a loop fashion), while *guvectorize* work on a chunk of elements of input arrays. Listing 3.2 and Listing 3.3 show how to apply *@vectorize* and *@guvectorize* decorators respectively to the function *mult* that computes 2-vectors multiplication.

The first argument of the annotation is the type signatures. The second argument in Listing 3.3 is the declaration, in symbolic form, of input and output *lay-*

Listing 3.2. An example of using @vectorize for multiplication two vectors.

```
from numba import vectorize, float64
@vectorize([float64(float64, float64)])
def mult(X, Y):
    return X * Y
```

Listing 3.3. An example of using @guvectorize for multiplication of two vectors.

```
@guvectorize([(int64[:], int64[:], int64[:])],
    '(n),(n)->(n)', target='cpu')
def mult(X, Y, RES):
    for i in range(X.shape[0]):
        RES[i] = X[i] * Y[i]
```

outs. In our example, the expression $(n), (n)->(n)$ tells that the function f takes two n-element one-dimension arrays and returns a n-element one-dimension array. The third argument, *target*, instructs the compiler to generate code for one of three target architectures: *cpu, parallel or cuda*. The *cpu* target means a single-threaded CPU, *parallel* means multi-core CPU and *cuda* means CUDA GPU.

Numba for Accelerators

Numba supports CUDA GPU programming and HSA APU programming. For example, Numba's NVIDIA CUDA GPU support is able to compile a restricted subset of Python code into CUDA kernels and device functions and then following the CUDA execution framework. NumPy arrays have direct access from those kernels, while their transformation between the host and the driver is done automatically. Furthermore, managing the memory hierarchy of threads can be performed by Numba's CUDA specific functions that are similar to those supported by CUDA C language. Listing 3.4 demonstrates how to use the *@cuda.jit* compiler directive in order to compile the function *mult* into a CUDA kernel. This function multiplies two vectors on the device. The memory hierarchy of threads is determined in lines 10–11 before calling to the function *mult* by initializing the variable *threadsperblock* and computing the variable *blockspergrid*. Then, these variables are passed as parameters when calling the function *mult* in line 13.

4 Test Case: Matrix-Matrix Multiplication

In this section, we examine and analyze the performance of various implementations of Matrix-Matrix Multiplication algorithm. We compare the performance of the pure Python version against those of Numba, NumPy, Numba CUDA API, Numba scientific library, pure CUDA, and pure C code. Table 1 summarizes the times and the computed speedups of our measurements. The matrices sizes used for all the benchmarks are 320×320. The times in Table 1 are the average running time of each implementation over one-hundred runs. Our benchmarking environment includes Python 2.7; Anaconda version 4.2.13; Numba

Listing 3.4. An example of using @cuda.jit for multiplication of two vectors.

```
import numpy as np
from numba import cuda

@cuda.jit
def mult(x, y, res):
    pos = cuda.grid(1)
    if pos < x.size:
        res[pos] = x[pos] + y[pos]

threadsperblock = 32
blockspergrid = (x.size + (threadsperblock - 1))
    //threadsperblock

mult[blockspergrid, threadsperblock](x,y,res)
```

0.30.1; Spyder 3.0; NumPy 1.11.1; SciPy 0.18.1; Cython 0.25.2; Numexpr 2.6.2; PyPy2.7 v5.6.0; Visual Studio C++ 2015; OpenMP 3.0; CUDA 8.0; GPU processor NVIDIA GeForce GTX 550 Ti with compute capability 2.1 (It is a relative old processor but was sufficient for our purpose); and Intel Core-i7 3.4 GHz multicore processor. *All the running times of the GPU implementations do not include the time required to transmit data from the host to the driver and vice versa.* The rationale for not taking into account the time overhead of the communication is due to the fact that heterogeneous processors that will be lunched in the foreseeable future will include physical unified shared memory for the host and the driver. Therefore, the communication time between them will be negligible.

Table 1 is devised into two groups of implementations: serial and parallel implementations. All the computed speedups are relative to the running time of the pure Python implementation. The implementations in Table 1 are sorted in ascending order of the speedup obtained from our benchmarks.

Listing 4.1 presents the pure Python implementation based on for-loop of the Matrix Multiplication algorithm.

Let's start with the serial implementations. Annotating the function *matmul* with the annotation *@jit (nopython=True)* instructs the compiler to compile the function *matmul* to the fastest code possible. The time measurement in Table 1 shows an impressive speedup of x534. This result reflects the difference in performance that can be obtained from an interpreted code compared to a compiled code.

Generating *matmul* function as a NumPy generalized universal function is completed by annotating the function *matmul* by the following annotation:

```
@guvectorize(['void(float64[:,:], float64[:,:],
float64[:,:])'], '(m,n),(n,p)->(m,p)',
nopython=True, target='cpu')
```

Table 1. Times and speedups of various Matrix Multiplication implementations.

Program	Time (in seconds)	Speedup
Serial implementations		
Pure Python	18.6031195509	x1
Numba @jit	0.0347767734007	x534
Numba @guvectorize Target = 'cpu'	0.0333741619215	x558
C on windows	0.02513	x740
Pure NumPy	0.000834	x22,302
Parallel implementations		
Numba @guvectorize Target = 'parallel'	0.0668405298234	x278
Python threading 1 thread Nogil = False 4 thread Nogil = False 4 thread Nogil = True	 0.03643389......x1 0.03849981......x0.94 0.01314321......x2.73	 x1415
C-OpenMP, 4 cores	0.00860	x2162
Pure CUDA	0.000830	x22,409
Numba @guvectorize Target = 'cuda'	0.000377642710865	x49,258
Numba CUDA API	0.000131226685019	x141,740
cuBLAS GEMM	0.0000753513070663	x246,848

The first list is the type signatures of the input and the output arguments of the function. The second argument is a symbolic declaration of the function *matmul* that takes as input (m,n)-element and (n,p)-element two-dimension arrays and returns a (m,p)-element two-dimension array. The third argument, *target*, instructs the compiler to generate code for a single-threaded CPU. As observed in Table 1, when the target is 'cpu,' the speedup achieved is x558 compared to the result of pure Python which is 4.4% better than the performance achieved by automatic compilation using *@jit* decorator.

Implementing the matrix-matrix multiplication algorithm using pure C language with Microsoft Visual Studio C++ 2015 increases the speedup up to x740.

Listing 4.1. A pure Python Matrix Multiplication algorithm.

```
def matmul(A, B, C ):
    m, n = A.shape
    n, p = B.shape
    for i in range(m):
        for j in range(p):
            C[i, j] = 0
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]
```

This result is usually considered, by mistake, to the upper limit of the speedup attainable. However, comparing the running time of the pure Python algorithm against pure NumPy: $C = np.dot(A, B)$, yield an imaginary speedup of x22302! In other words, the *upper limit* term in applied computer science must be taken with a grain of salt.

Now, let's examine the results of the parallel implementations. Generating parallel *matmul* function as a NumPy generalized universal function is done by annotating the function *matmul* similar to the single-core case, but this time with the compiler option *target='parallel'*

As observed in Table 1, the achieved speedup is x278. However, this result shows no gain in performance compared to the running time on a single-core. There are two reasons for this disappointing outcome. First, *@guvectorize* actually parallelizes the "loop" dimension of the matrices which has only one entry in our case. Second, the overhead incurred by attempting to parallelize the algorithm is high relative to the gain from parallelizing the matrices. Another example involves parallelism overhead cost which will be further discussed a few lines ahead when the results of explicit parallelism of the algorithm will be introduced.

Numba compiler supports explicit parallelism using Python *threading* package [20]. However, the threading package is unable to parallelize code using its multithreading mechanism. The reason for this limitation is that the Python interpreter, CPython, is not *thread-safe*. In order to enforce it to be thread-safe, the *threading* module makes use of a global lock called the *Global Interpreter Lock* (GIL). This means that only one thread can execute a bytecode instruction at the same time; Python routinely switches between threads after a quantum of time. In other words, all threads in Python run on the same core, so no performance is gained by using multiple threads. Nevertheless, Numba allows the bypass of this problem by using the option *nogil=True*. The measurements that appear in Table 1 confirm our claims. When using the option *nogil=False*, we get speedup of x0.94 when running with four threads. However, when we set the option to *nogil=True*, then for four threads, we get speedup of x2.7 relative to the running time of one thread. The speedup relative to pure Python is x1415. The parallelism overhead inhibits us from gaining better speedup. However, by increasing the input size of the matrices to 2000 by 2000, we increase the work of each thread and therefore increase the computation time relative to the overhead time. The speedup, as expected, increases to x3.38 (times are not shown in Table 1).

The next entry in Table 1 is the implementation using C-OpenMP with four threads. In this case, the speedup soared to x2162.

All other implementations in Table 1 are exploiting the massive parallelism of the GPU processors. The first in this group of implementations is the Pure CUDA implementation that was borrowed from the CUDA 8.0 samples that were inspired from [21]. It has been written for clarity of exposition to illustrate various CUDA programming principles, not with the goal of providing the optimal performance of generic kernel for matrix multiplication. This implementation

Listing 4.2. Matrix Multiplication using CuBLAS GEMM routine.

```
import numpy as np
import time
import accelerate.cuda.blas as cublas

blas = cublas.Blas()

n = 320
A = np.random.random((n, n)).astype(np.float64)
B = np.random.random((n, n)).astype(np.float64)
C = np.zeros_like(A, order='F')

blas.gemm('T', 'T', n, n, n, 1.0, A, B, 1.0, C}

assert(np.allclose(np.dot(A, B), C))
```

has achieved a result that is a quantum leap compared with the previous ones with a remarkable speedup of up to x22169.

Now, we return to the generation of a parallel matmul function as a NumPy generalized universal function, but this time with the compiler option *target='cuda'*.

The recorded result shows more than double improvement in performance compared to the previous results, while the speedup reached to a level of x49,258. It is worth mentioning again that the timing measurements of the implementations running on the GPU are net time which does not include the communication costs of downloading and uploading the data from the host to the driver and vice versa.

The implementation of matrix-matrix multiplication using Numba CUDA GPU programming API introduces a new level of performance while attaining an astonishing speedup of x141740 that looks like an unbreakable achievement. However, the last implementation leaves no doubt that when it comes to optimizing the performance of parallel applications, the sky is the limit.

Anaconda Accelerate provides access to optimized numerical libraries for high performance on Intel CPUs and NVidia GPUs. One such binding is cuBLAS that provides an interface that accepts NumPy arrays and Numba's CUDA device arrays. The binding automatically transfers NumPy array arguments to the device as required. Listing 4.2 presents an example that uses cuBLAS GEMM routine to perform matrix-matrix multiplication. GEMM transposes the input matrices so that they can be in C order. Note that the output matrix is still in FORTRAN array. The string arguments in GEMM tell it to apply transformation on the input matrices. GEMM routine archives speedup of x246848 compared to pure Python code.

5 Conclusion

Today, the presence of Python in many academic courses and as emerging software development tool for scientific applications is not in doubt. This presence has accelerated the development community's motivation and the users of the language to find effective solutions to improve its performance. In this article, we reviewed the main proposed solutions that are adopted by the Python developers' community while we focused on Numba, a much-promised solution that is in its advanced stages of development. This solution also preserves the performance improvement of code while it is ported to different target architectures.

References

1. Guo, P.: Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities. 7 July 2014. http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext
2. Most Popular Coding Languages of 2016. 2 February 2016. http://blog.codeeval.com/
3. TIOBE Index. http://www.tiobe.com/tiobe-index/
4. IEEE Spectrum ranking. http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages
5. Scipy. http://www.scipy.org/
6. Numpy. http://www.numpy.org/
7. Matplotlib. http://matplotlib.org/
8. Marowka, A.: On parallel software engineering education using Python. Educ. Inf. Technol. **23**, 357–372 (2017). Springer
9. PyPy. http://pypy.org/
10. PyCUDA. https://mathema.tician.de/software/pycuda/
11. PyOpenCL. https://mathema.tician.de/software/pyopencl/
12. Numba. http://numba.pydata.org/
13. A Speed Comparison of C, Julia, Python, Numba and Cython on LU Factorization. https://www.ibm.com/developerworks/community/blogs/jfp/entry_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization?lang=en
14. PyPy Speed Center. http://speed.pypy.org/
15. Cython. http://cython.org/
16. Numexpr. https://github.com/pydata/numexpr
17. Python Multiprocessing module. https://docs.python.org/2/library/multiprocessing.html
18. Anaconda Accelerate. https://docs.continuum.io/accelerate/
19. Anaconda Python. https://www.continuum.io/downloads
20. Python threading module. https://docs.python.org/3.3/library/threading.html
21. Volkov, V., Demmel, J.: Benchmarking GPUs to tune dense linear algebra. In: Proceedings of ACM/IEEE Conference on Supercomputing (SC 2008), pp. 31:1–31:11. IEEE Press, Piscataway (2008)

Actor Model of a New Functional Language - Anemone

Paweł Batko[1] and Marcin Kuta[2](✉)

[1] VirtusLab, Smoleńsk 21/15, 31-108 Krakow, Poland
pbatko@virtuslab.com
[2] Department of Computer Science, AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Krakow, Poland
mkuta@agh.edu.pl

Abstract. This paper describes actor system of a new functional language called *Anemone* and compares it with actor systems of Scala and Erlang. Implementation details of the actor system are described. Performance evaluation is provided on sequential and concurrent programs.

Keywords: Compiler construction · Concurrent programming
Actors · Threads · Message passing

1 Introduction

Creating a new language is a complex task, requiring a lot of time and human resources. Creating a functional language is even more difficult, as it contains many high-level concepts, e.g., higher-order functions or closures. This work presents design and implementation of *Anemone* – fully useful, secure, functional language with user-friendly syntax, which can be enriched with new components and optimized in future.

Anemone [4] supports concurrent programming model based on actors communicating via messages. Polymorphic type system of *Anemone* supports error detection at compile time and encourages code reuse. Complete type inference disposes a programmer of defining explicit type signatures. *Anemone* functions are first class citizens [1] and memory is automatically managed with the garbage collector module. Mechanism of external functions gives access to functions and libraries written in C. *Anemone* compiler is based on the LLVM infrastructure, which generates high quality code for many computer architectures [6]. Created language draws inspiration from many mechanisms present in various programming languages, including Scala, Erlang, Haskell and ML. Actor system integrated directly into language, safe programming style encouraged by immutable variables, coherent type system with subtyping, pattern matching mechanism, and compiler retargetability make Anemone attractive compared to other languages.

The aims of this work were the following:

1. Implementation of a functional language compiler, analysis of such languages and ways of their implementation.
2. Convenient model of parallel processing based on actors mechanism and message passing.
3. Performance comparison of the implemented model with existing mechanisms in other languages.

2 Related Work

Threads are a low-level abstraction of concurrent programming, but using them is difficult because they share memory. Secure, concurrent access to shared memory requires synchronization with a critical section along with structures like semaphores, mutexes or critical sections. A programmer has to pay attention to avoid race conditions, starvation, dead locks or live locks.

Actor model [2,5] offers several advantages over thread model. Actors intuitively model real world, and writing reliable applications using actors is simple. Asynchronous communication between actors leads to better utilization of CPU time, as idle cycles are not wasted while waiting for an answer. Separation between actors makes programs easier to understand. The whole actor system can be understood by analysing each actor behaviour in isolation. Actors are well suited for problems requiring high performance, responsiveness or multiscaling [7], as they can delegate tasks to workers. Actors are also lightweight and thousands of instances can be created, which is impossible for system threads. With actor model, synchronization mechanisms present in thread model are no longer needed. As actors do not share a global state, critical sections are neither needed. Taking into account above advantages, actor model has been adopted as concurrency model in *Anemone*.

2.1 Actors in Erlang and Scala

Erlang implements actors at a language level and calls them processes. Erlang processes are lightweight, as they are implemented at the level of Erlang virtual machine and do not involve threads or processes of a operating system. Each actor is assigned its own dynamic memory, and garbage collection is performed for each actor independently. Sent messages are copied between heaps, which prevents them sharing by two or more actors. Actor scheduling is a duty of the Erlang virtual machine, and no synchronization mechanism, e.g., semaphores, is needed. Erlang actor system ensures scalability (one actor needs only 300 bytes of memory) and high reliability of written programs [3].

Figure 1(a) presents two actors, *ping* and *pong*, defined as functions in Erlang. It is worth noting code conciseness, built-in message sending operator, a block for receiving messages, and message identification with pattern matching.

Scala provides actors implementation in the Akka library [8]. Message passing is done in shared memory, if actors run within one Java virtual machine. If actors perform on different virtual machines, messages are serialized before sending them. Garbage collection applies to all the actors from a given JVM instance. Actors are scheduled by the Akka library. Akka does not impose immutable messages and avoiding global state although strongly recommends them.

```
ping(N, Pong) ->
    Pong ! {self(), ping}.

pong() ->
    receive
        {From, ping} ->
            From ! pong,
            pong();
    end.
```

(a) Erlang implementation

```
class Ping extends Actor {
  val state: Int = 1
  def receive = {
    case Ping =>
        pong ! PingMsg
  }
}

class Pong extends Actor {
  def receive = {
    case PingMsg =>
        sender ! PongMsg
  }
}
```

(b) Scala implementation

Fig. 1. Two actors *ping* and *pong* implemented in Erlang and Scala

Erlang actors are implemented as *ping* and *pong* functions, while Scala actors are represented with instances of *Ping* and *Pong* classes. In both cases, messages are sent with `!` operator and received within `receive` block.

Figure 1(b) presents actor definitions with Akka. In contrast to Erlang, Akka actors are defined as classes. Pattern matching, message sending operator, and a block for receiving messages are exploited similarly as in Erlang. Actor state is represented as a field of an actor instance. Continuity of actor work is ensured by Akka, so an actor does not have to call itself, as it is in Erlang.

3 Actor Model of Anemone

Anemone does not provide a programmer low-level model of threads, which, although very expressive, does not fit needs of a high-level, secure functional language.

Actor model is more restrictive than thread model, as actors communicate only through message passing. It makes manual synchronization redundant and allows to write programs more easily, due to imposed style of communication and message identification. Continuity of actor work is provided in *Anemone* by internal implementation, which, similarly to Akka, calls actor function for each new message. Similarly to Erlang, *Anemone* models actors as functions, which can accept an initial state and pass it through. Message passing model in *Anemone* is similar to Akka, as they both use shared memory to implement it.

Actor model of *Anemone* characterizes with:

– asynchronous communication
– possibility of creating many actors in one thread
– possibility of dynamic creation of new actors
– complete integration with the garbage collector
– modeling actors as functions, similarly to Erlang
– actor control similar to Akka
– message handling through pattern matching.

Implementation of many actors within one system thread was especially challenging, as it required their proper scheduling. Construction of the garbage collector which correctly and effectively cooperates with multithreaded actors architecture was equally challenging.

Figure 2 presents a complete program implementing two actors which communicate with each other via messages. Function `createActorSystem` creates an actor system on the basis of two system threads, and next, actors *ping* and *pong* are created. Function `sendFromOutside` starts activities of actors, i.e., sending messages. Actors exploit pattern matching to identify received messages, function `sendMsg` to send messages, and maintain the state identifying the address of the second actor.

4 Implementation of Actor System

Anemone models concurrent computing with the actor module, which provides abstraction for convenient work on multicore architectures. Figure 3 presents detailed architecture of the actor module. Actor module consists of the following submodules:

– `Mailbox module` defines how received messages are stored and is responsible for actors adding and removing.
– `Actor management module` defines architecture of a single actor.
– `Threads module` is responsible for management of OS threads.
– `Dispatcher module` is responsible for running actors on system threads.

In *Anemone* using actors is possible due to the actor module. The actor library is an interface to this module from *Anemone*. Figure 4 presents functions to handle actors. The actors library provides the following functions to manipulate actors:

– `createActorSystem` creates an actor system running on a given number of system threads.
– `createActor` creates a new actor in the actor system. Two parameters define actor behaviour and an initial state of actor. Function returns the id of a created actor.

```
fun ping(state, msg) {
    var otherActorId = state in {
        match msg {
            | s :: String => {
                printStr(s)
                sendMsg(otherActorId, "fromPing")
                nap(1)
                state
            }
            | otherActorId :: ActorId => {
                sendMsg(otherActorId, "fromPing - first")
                otherActorId
}   }   }   }

fun pong(state, msg) {
    var otherActorId = state in {
        match msg {
            | s :: String => {
                printStr(s)
                sendMsg(otherActorId, "fromPong")
                nap(1)
                state
}   }   }   }

fun main_fun() {
    createActorSystem(2)
    var pingActorRef = createActor(ping, 0),
    pongActorRef = createActor(pong, pingActorRef) in {
        sendFromOutside(pingActorRef, pongActorRef)
}   }
```

Fig. 2. Creation and starting actor system

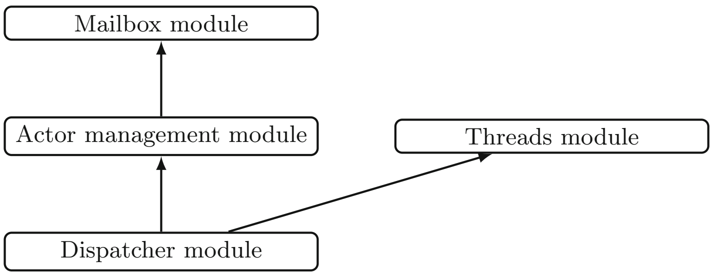


Fig. 3. Main submodules of module of actors and message passing

- `killActor` returns an object, which sent to an actor or passed as its new state terminates its action.
- `sendMsg` sends a message, specified in the second parameter, to an actor defined by the first parameter. Function can be called only by an actor.
- `become` can be called only by an actor. The function takes as its argument a function determining actor new behaviour.

```
type:: (double) -> unit
fun createActorSystem(n)

type:: (('s, 'm) -> 's, 's) -> ActorId
fun createActor(f, a)

type:: (ActorId, 'a) -> unit
fun sendMsg(to, msg)

type:: (('s, 'm) -> 's) -> unit
fun become(f)
```

Fig. 4. Functions of actor library of *Anemone*

4.1 Creating Actor System

Creating actor system is equivalent to creating a number of threads and initialization of their data structures. The actor system of *Anemone* distinguishes two kinds of threads: system threads and actor threads. Each actor thread contains a system thread and a number of actors.

```
struct __athread_t {
        int64_t thread_id;
        int64_t actors_array_capacity;
        int64_t actors_array_occupied;
        int64_t current_actor_idx;
        actor_t** actors_array;
};
typedef struct __athread_t athread_t;
```

Fig. 5. Structure of actor module describing an actor thread of *Anemone*

Function `createActorSystem` creates a new actor system with a given number of threads. In particular, it creates a table of `athread_t` structures, describing actor threads (Fig. 5). Each `athread_t` structure corresponds to one system thread and many actors (`actors_array`).

System threads are created and managed with the help of the POSIX *pthreads* library. Function `athread_main_fun` creates system threads and defines their behaviour. Function `athread_main_fun` works as follows:

- initializes structures describing an actor thread.
- waits for the first actor in this thread.
- takes the next actor assigned to this actor thread.
- starts handling at most `max_msg_handled_per_actor` messages in the mailbox of the actor. Variable `max_msg_handled_per_actor` is tunable.
- removes an actor from the system in the following cases:
  - an actor received a message being a result of `killActor` call
  - an actor returned a new state equal to the return value of `killActor` call.
- serves next actor, if it is present.
- terminates, if no actors are present in this actor thread.

4.2 Implementation of an Actor and Message Passing

Actors and message passing are implemented in the actor module in C. Figure 6 presents the structure of the actor module used by the actor management module. It consists from the following fields:

- `user_cls` - a closure describing actor behaviour,
- `state` - actor state, it is passed as a parameter to a closure describing actor behaviour,
- `actor_id` - actor id in the actor management module,
- `mailbox` - a pointer to the mailbox of an actor.

```
struct __actor_t {
        closure_t *user_cls;
        void *state;
        uint64_t actor_id;
        mailbox_t *mailbox;
};
typedef struct __actor_t actor_t;
```

Fig. 6. Structure of actor module describing a single actor of *Anemone*

An actor mailbox stores messages sent to a given actor. Each message has a sender, a receiver, and the content. Messages are processed under the *FIFO* regime. Sending a message does not incur copying the whole message but only a lightweight copying of a pointer to the message, as message passing is done in shared memory.

Function `pong` takes as its arguments a current actor state and a new message. It sends a message of type *string* to an actor identified by *pingActorID*. The function returns a new state (in this case unchanged) as a result of its call.

At any time, an actor thread can run at most one actor. Figure 7 presents a simple actor definition. Function `sendMsg`, responsible for sending a message to

```
fun pong(state, msg) {
    printStr("Pong: " ^ msg)
    sendMsg(pingActorId, "PONG" ^ msg)
    state
}
```

Fig. 7. An example of a function defining actor behaviour

```
void* threads_getspecific() {
    return pthread_getspecific(THREAD_KEY);
}

void threads_setspecific(void* data) {
    pthread_setspecific(THREAD_KEY, data);
}
```

Fig. 8. Getting and setting local state of a given actor thread

another actor, does not specify explicitly the sender of a message. The sender (currently running actor) is identified due to the local context of the actor thread, implemented by the actor module.

Function `threads_getspecific` gets local state of a actor thread, while `threads_setspecific` sets it. Key `THREAD_KEY` defines local data of a given actor thread. Both functions are implemented in *pthreads*.

Figure 8 presents functions of the actor management module, responsible for associating actor threads with contextual data. Function `threads_setspecific` is used in the main loop of `athread_main_fun` to set thread context to current actor. When function responsible for sending messages is called, it can access local context of a thread to get the id of the actor. The advantage of such a solution is a support for many actors in one system thread through changing value of thread context and conciseness of function `sendMesg`.

In addition to local context setters and getters, implementation of the actor module uses synchronization primitives – mutexes and conditional variables of *pthreads*. Correct and effective realization of the actor system on the basis of above mechanisms is a duty of the runtime system of *Anemone*. *Anemone* users profit from multicore architectures without using difficult and insecure low-level synchronization mechanisms.

4.3 Scheduling Many Actors in One System Thread

The actor module can create a huge number of actors, significantly exceeding the number of running threads of the operating system, as an *Anemone* actor is defined by a lightweight data structure (several hundred bytes). An actor thread associates a system thread with many actors. Actor scheduling in *Anemone* is a duty of the actor module and is based on the number of received messages. Each running actor may receive no more than `max_msg_handled_per_actor` messages. If an actor receives all the messages from its mailbox (but not exceeding the `max_msg_handled_per_actor` threshold), the next actor belonging to the same actor thread will be scheduled. Above scheduling algorithm promotes implementation of actors as quickly performing functions. Time-consuming tasks should be delegated by an actor to its child workers. The drawback of this solution is that some system threads will be blocked by these heavy computations.

4.4 Pattern Matching

With pattern matching supported by *Anemone*, defining actors and their behaviour is easier.

```
fun anActor(state, msg) {
    match msg {
        | Bar(b)   => { ... }
        | f :: Foo => { ... }
    }
}
```

Fig. 9. Usage of pattern matching in definition of *Anemone* actor

Figure 9 presents an actor defined with pattern matching. Message `msg`, received by an actor, is matched to the first pattern, `Bar(b)`, where `Bar` denotes a data type defined in *Anemone*, and `b` denotes a field of this data type. If the first match fails, pattern `f :: Foo` will be checked. The second pattern will be matched if `msg` is of type `Foo`. Identifier `f` introduces variable `f`, which refers to matched object, `msg`, and has type `Foo`.

5 Experiments

To assess quality of *Anemone* implementation and its runtime system, performance tests have been conducted and execution time of programs written in *Anemone*, Scala and Java have been compared. For each language, the arithmetic mean of ten measurements with *time* command was reported. The experiments were performed under dual-core Intel Core i3-2310M CPU with 2.10 GHz clock and Linux Ubuntu 13.10. Heap size for Scala and Java was set to default values. For *Anemone* heap size was set to 100 KiB and the threshold triggering collection to 0.8.

The first experiment, which assessed quality of generated code and efficiency of the runtime system, compared time performance of a sequential program (computation of 20th element of the Fibonacci sequence) written in Java, Scala and *Anemone*. Results in Table 1 show that implementation in Scala was two times faster, while implementation in Java three times faster. It can be partly attributed to memory organization in *Anemone*. Language performance is significantly influenced by efficiency of the memory allocation module. While *Anemone* is a new language, memory allocation algorithm in JVM used by Scala and Java has been fine-tuned for many years.

Table 1. Execution time of the program computing 20th element of the Fibonacci sequence

Language	Execution time [s]	Ratio
Java (1.7.0_60)	0.45	3.09
Scala (2.10.4)	0.75	1.87
Anemone	1.41	1.00

The second experiment assessed efficiency of the *Anemone* actor system. Implementations in *Anemone* and Akka of two actors communicating with each other were compared. Actors sent in total 10000 messages.

Table 2. Execution time of the program creating simple actor system

Language	Execution time [s]	Ratio
Scala (2.10.4) + *Akka* (2.2-M3)	1.86	6.12
Anemone	11.41	1.00

Implementation in Akka turned out to be six times faster than implementation in *Anemone* (Table 2). Efficiency of the memory allocation module of *Anemone* could have significant impact on results.

6 Conclusions

Model and implementation of actor system of a new functional language, *Anemone*, have been described. *Anemone* was developed within a limited period of time. Proposed language was few times slower than Scala or Java, which is a very good result, taking into account that latter languages have been already developed and optimized for a long period of time by large teams of experts.

Acknowledgements. This research was financed by AGH University of Science and Technology Statutory Fund.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs, 2nd edn. MIT Press/McGraw-Hill, Cambridge (1996)
2. Agha, G.A.: ACTORS - A Model of Concurrent Computation in Distributed Systems. MIT Press series in artificial intelligence. MIT Press, Cambridge (1990)
3. Armstrong, J.: Erlang. Commun. ACM **53**(9), 68–75 (2010)
4. Batko, P.: A compiler for functional language with support for message passing (in Polish). Master's thesis, Department of Computer Science, AGH University of Science and Technology, Krakow, Poland (2014)
5. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI 1973, pp. 235–245 (1973)
6. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004), pp. 75–88 (2004)
7. Rycerz, K., Bubak, M.: Using Akka actors for managing iterations in multiscale applications. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9573, pp. 332–341. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32149-3_32
8. Wyatt, D.: Akka Concurrency. Artima Incorporation, Sunnyvale (2013)

Almost Optimal Column-wise Prefix-sum Computation on the GPU

Hiroki Tokura, Toru Fujita, Koji Nakano(✉), and Yasuaki Ito

Department of Information Engineering, Hiroshima University, Kagamiyama 1-4-1, Higashihiroshima 739-8527, Japan
nakano@hiroshima-u.ac.jp

Abstract. The row-wise and column-wise prefix-sum computation of a matrix has many applications in the area of image processing such as computation of the summed area table and the Euclidean distance map. It is known that the prefix-sums of a 1-dimensional array can be computed efficiently on the GPU. Hence, the row-wise prefix-sums of a matrix can also be computed efficiently on the GPU by executing this prefix-sum algorithm for every row in parallel. However, the same approach does not work well for computing the column-wise prefix-sums, because inefficient stride memory access to the global memory is performed. The main contribution of this paper is to present an almost optimal column-wise prefix-sum algorithm on the GPU. Since all elements in an input matrix must be read and the resulting prefix-sums must be written, computation of the column-wise prefix-sums cannot be faster than simple matrix duplication in the global memory of the GPU. Quite surprisingly, experimental results using NVIDIA TITAN X show that our column-wise prefix-sum algorithm runs only 2–6% slower than matrix duplication. Thus, our column-wise prefix-sum algorithm is almost optimal.

Keywords: Prefix computation · Parallel algorithms · GPU · CUDA

1 Introduction

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [2,5,14]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [13], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [5], since they have thousands of processor cores and very high memory bandwidth.

When we develop programs running on GPUs, we can use the CUDA programming model [13]. Usually, a CUDA program executed on the host computer invokes *CUDA kernels* one or more times. A CUDA kernel executes one or more *CUDA blocks* with at most 1024 threads running on streaming multiprocessors of the GPU. CUDA blocks in a CUDA kernel are identical in the sense that they have the same number of threads executing the same program. CUDA blocks are dispatched to a streaming multiprocessor in turn. Threads in a CUDA block is partitioned into groups of 32 threads each called *warp*. All threads in a warp are dispatched to cores in a streaming multiprocessor at the same time and execute the same instruction.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [13]. The shared memory is an extremely fast on-chip memory with lower capacity such as 16–96 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity such as 1.5–12 Gbytes, but its access latency is quite large. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *bank conflicts* of the shared memory access and *coalescing* of the global memory access [5,12]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access different addresses in the same memory bank at the same time, they are processed sequentially in turn. Hence, to maximize the memory access performance, threads should access distinct memory banks to avoid bank conflicts. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

Let $a_0, a_1, \ldots, a_{n-1}$ be n numbers. The prefix-sums $\hat{a}$ of a are n numbers such that $\hat{a}_i = a_0 + a_1 + \cdots + a_i$ for all i $(0 \le i \le n-1)$. Suppose that each variable $A[i]$ stores a_i. After executing $A[i] \leftarrow A[i] + A[i-1]$ for all i $(1 \le i \le n-1)$ in turn, each $A[i]$ stores the prefix-sum $\hat{a}_i$. The computation of the prefix-sums of a 1-dimensional array is one of the most important computation for many algorithms. For example, list ranking problem which determines the position of each item in a linked list can be solved by computing the prefix-sums. It is also used for computing the positions of keys in radix sort. In [1], several fundamental algorithms for computing the prefix-sums on the GPU have been shown. Also, Merrill *et al.* [6,7] has presented a more sophisticated GPU implementation for the prefix-sums using decoupled look-back technique. As far as we know this algorithm is the most efficient GPU implementation for computing the prefix-sums of a 1-dimensional array. For later reference, we call this algorithm *CUB-prefix* in this paper.

Suppose that a matrix a with $n \times n$ elements $a_{i,j}$ $(0 \le i, j \le n-1)$ is given. As usual, we assume that each $a_{i,j}$ is an element in the i-th row and j-th column. The row-wise prefix-sums correspond to a matrix r of the same size such that $r_{i,j} = a_{i,0} + a_{i,1} + \cdots + a_{i,j}$ for all i and j $(0 \le i, j \le n-1)$. Similarly, the column-wise prefix-sums correspond to a matrix c such that $c_{i,j} = a_{0,j} + a_{1,j} + \cdots + a_{i,j}$ for all i and j $(0 \le i, j \le n-1)$. Figure 1 illustrates the row-wise and the column-wise prefix-sums of a 4×4 matrix. They have many applications in the

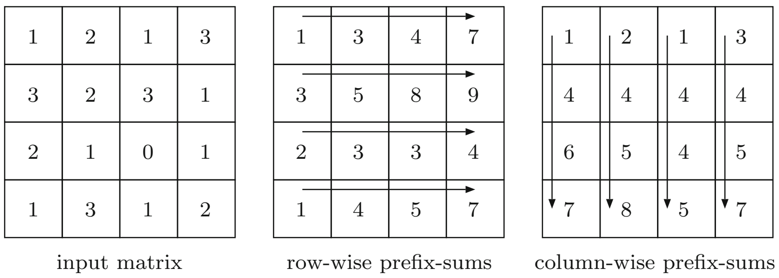


Fig. 1. Row-wise and column-wise prefix-sums of a 4×4 matrix

area of image processing. For example, the summed area table [3,4,11] can be obtained by computing the row-wise prefix-sums and the column-wise prefix-sums. Also, in the computation of Euclidean distance map of a binary image, the column-wise prefix-minima is computed [5].

The main contribution of this paper is to present *the Look-back Column-wise Prefix-sum (LCP) algorithm*, which computes the column-wise prefix-sums of a matrix very efficiently on the GPU. It partitions the matrix into small tiles and the column-wise sums and prefix-sums of every tile are computed using one CUDA block for each tile in parallel. The LCP algorithm involves several GPU computing techniques including *the warp prefix scan* [1], *the diagonal arrangement of a matrix* [10], and *the decoupled look-back* [7] to minimize memory access and synchronization overhead. The LCP algorithm does not perform stride access to the global memory, shared memory access with bank conflicts, or separated kernel calls for global synchronization, which involve large overhead. Clearly, no GPU implementation of column-wise prefix-sum computation of an $n \times n$ matrix can be faster than matrix duplication, in which n^2 elements are read and written. Thus, we can say that a column-wise prefix-sum algorithm is optimal if the computing time is equal to matrix duplication. Quite surprisingly, the experimental results on NVIDIA TITAN X GPU show that our LCP algorithm runs only 2–6% slower than matrix duplication. Thus, our LCP algorithm is almost optimal.

2 Preliminary

This section shows several fundamental techniques on the GPU necessary to understand our LCP algorithm and naive algorithms for computing the column-wise prefix-sums.

Let A be an $n \times n$ 2-dimensional array storing a matrix of $n \times n$ numbers. We assume that each $A[i][j]$ storing $a_{i,j}$ is arranged in offset $i \cdot n + j$ of the memory space for A. Suppose that A stores the values of an $n \times n$ matrix a. The row-wise prefix-sums of a can be computed using n threads as follows:

[Naive row-wise prefix-sum algorithm]

```
for $i \leftarrow 0$ to $n-1$ do in parallel
  for $j \leftarrow 1$ to $n-1$ do
    thread $i$ performs $A[i][j] \leftarrow A[i][j] + A[i][j-1]$;
```

Clearly, n threads access $A[0][j], A[1][j], \ldots, A[n-1][j]$ for each j. Since these n elements are not coalesced, a lot of clock cycles are necessary to read/write these elements. Similarly, the column-wise prefix-sums can be computed by executing $A[j][i] \leftarrow A[j][i]+A[j-1][i]$. Since n threads access $A[j][0], A[j][1], \ldots, A[j][n-1]$ for each j, the memory access is coalesced and this naive algorithm may run efficiently on the GPU. However, only n threads are used and it is not possible to fully utilize high memory bandwidth of the GPU.

Clearly, the row-wise prefix-sums can be computed efficiently by executing CUB-prefix [6,7] for every row in parallel. Since CUB-prefix can compute the prefix-sums of a 1-dimensional array efficiently, this CUB-based row-wise prefix-sum computation also works very efficiently. However, the same parallel computation does not work well for the column-wise prefix-sums, because each CUDA block computes the prefix-sums of stride elements and memory access to the 2-dimensional array in the global memory is not coalesced. Thus, the naive algorithm and the CUB-based algorithm should be used for the column-wise and the row-wise prefix-sums computations. Further, the CUB-based algorithm run faster than the naive algorithm because it uses more threads. Hence, we will develop more efficient algorithm for the column-wise prefix-sums.

Let $w = 32$ denote the number of threads in a warp of the GPU. Suppose that each thread in a warp has a register a storing a number and we write $A[i]$ ($0 \leq i \leq w-1$) to denote register A of thread i. We assume that a 1-dimensional array a of size n are stored in register A's such that each $A[i]$ stores a_i. The prefix-sums of a can be computed in $\log_2 w$ steps as follows:

```
[Warp prefix scan]
for $k \leftarrow 0$ to $\log_2 w - 1$ do
   for $i \leftarrow 0$ to $w-1$ do in parallel
      thread $i$ performs $A[i] \leftarrow A[i] + A[i-2^k]$ if $i \geq 2^k$;
```

Figure 2 illustrates how the warp prefix scan computes the prefix-sums. The reader should refer [1,8,9] for the details. In the warp prefix scan, each thread i ($0 \leq i \leq w-1$) must read register $A[i-2^k]$ of thread $i-2^k$. This register read can be done very efficiently by warp shuffle function __shfl_up(A,2^k), which directly reads the value of register A of thread $i-2^k$. Since no memory access to the shared memory or the global memory is performed, the warp prefix scan runs very efficiently on the GPU.

Suppose that we have a $w \times w$ 2-dimensional array A stored in the shared memory with w memory banks. Each $A[i][j]$ is in offset $wi+j$ of S, which is arranged in bank $(wi+j) \bmod w = j$. Thus, the row-wise memory access to $A[i][0], A[i][1], \ldots, A[i][w-1]$ has no bank conflict while the column-wise memory access to $A[0][j], A[1][j], \ldots, A[w-1][j]$ is destined for the same bank j. By the diagonal arrangement which maps each $A[i][j]$ to offset $wi+((i+j) \bmod w)$, both the row-wise memory access and the column-wise memory access have

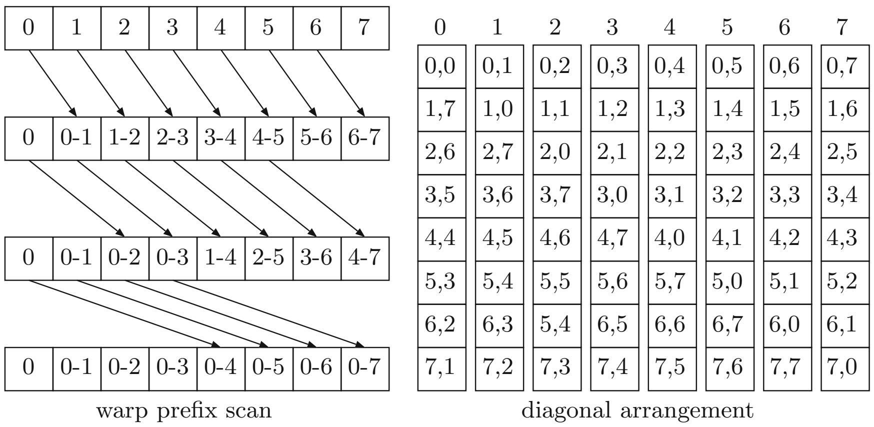


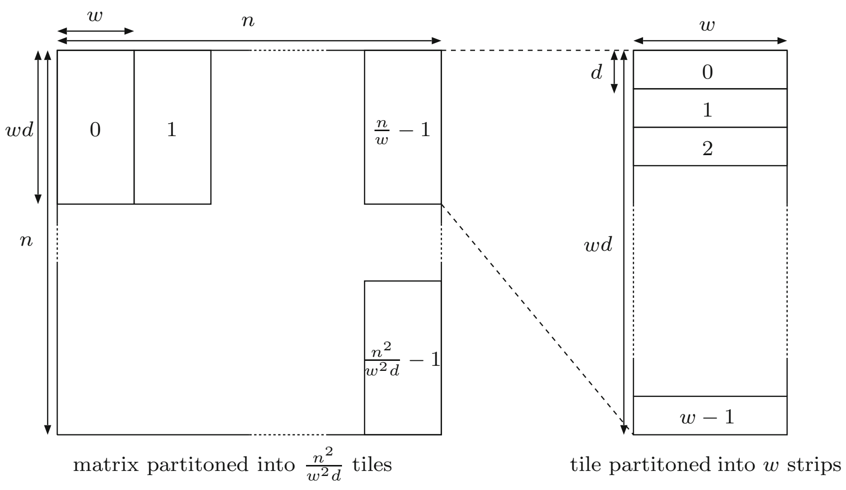
Fig. 2. Illustrating warp prefix scan and diagonal arrangement for $w = 8$

no bank conflict. Figure 2 illustrates the diagonal arrangement for $w = 8$. Since w elements $A[i][0], A[i][1], \ldots, A[i][w-1]$ are arranged in banks $i \bmod w$, $(i+1) \bmod w$, ..., $(i+w-1) \bmod w$, respectively, the row-wise memory access is conflict-free. Similarly, the column-wise memory access has no bank conflict. Thus, the column-wise (or the row-wise) prefix-sums can be computed very efficiently by executing warp prefix scan for each column (or row) in parallel. For later reference, we call the column-wise prefix-sum computation by this technique *column-wise warp scan*.

As we have mentioned, the CUB-based row-wise prefix-sum computation runs very efficiently, while the CUB-based column-wise prefix-sum performs stride memory access with large overhead. To avoid stride memory access, we can transpose the input matrix in advance. More specifically, the column-wise prefix-sums can be computed, by matrix transposition, the row-wise prefix-sum computation, and matrix-transposition. Since matrix transposition can be done very efficiently by coalesced memory access to the global memory [10], this 3-step algorithm may run more efficiently on the GPU.

3 The Look-back Column-wise Prefix-sums (LCP) Algorithm on the GPU

This section shows our LCP algorithm that computes the column-wise prefix-sums on the GPU. Again, let $w = 32$ denote the number of threads. We use CUDA blocks with $w^2 = 1024$ threads each and let $t_{i,j}$ $(0 \le i, j \le w-1)$ denote thread j in warp i, *i.e.* thread $iw + j$ in a CUDA block. Suppose that an $n \times n$ matrix a in the global memory is partitioned into $\frac{n}{wd} \times \frac{n}{w}$ tiles of size $wd \times w$ each as illustrated in Fig. 3, where $d \ge 1$ is an integer parameter. Let $T(i, j)$ $(0 \le i \le \frac{n}{wd} - 1$ and $0 \le j \le \frac{n}{w} - 1)$ denote a tile. We assume that serial numbers from 0 to $\frac{n^2}{w^2 d} - 1$ are assigned to tiles in row major order, that is, each



w
n
wd
n
0
1
$\frac{n}{w}-1$
$\frac{n^2}{w^2d}-1$
matrix partitoned into $\frac{n^2}{w^2d}$ tiles
w
d
0
1
2
wd
$w-1$
tile partitoned into w strips

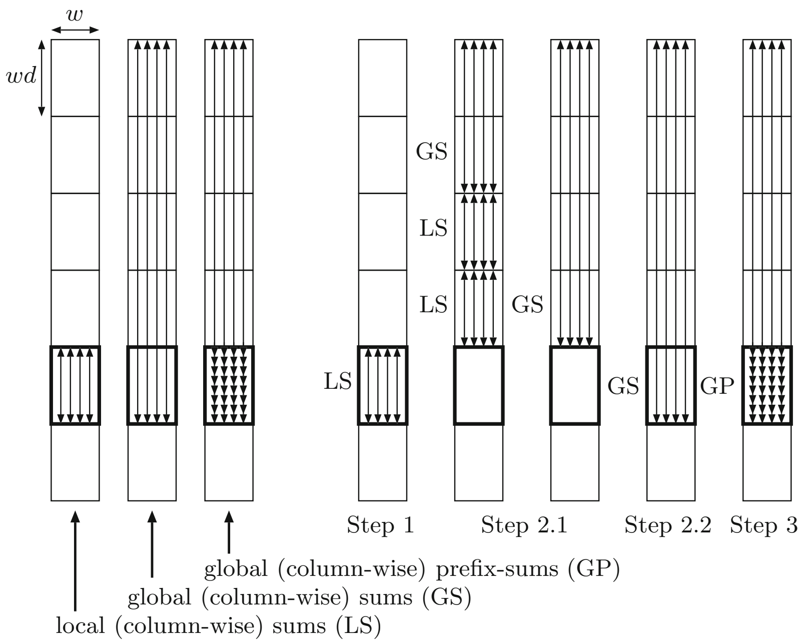
Fig. 3. Serial numbers assigned to tiles

$T(i,j)$ is assigned a serial number $i\frac{n}{w}+j$. We also call $T(i,j)$ *tile* k for $k = i\frac{n}{w}+j$ and the computation performed for tile k *task* k. Each tile is further partitioned into w *strips* 0, 1, ..., $w-1$ of size $d \times w$ each as illustrated in Fig. 3.

Each tile k ($0 \le k \le \frac{n^2}{w^2d} - 1$) is assigned a CUDA block, which performs task k in three steps. Let $a[i][j]$ ($\alpha \le i \le \alpha + wd - 1$ and $\beta \le j \le \beta + w - 1$) be elements of tile k. In Step 1 of task k, *the local column-wise sums (LS)* of tile k, $a[\alpha][j]+a[\alpha+1][j]+\cdots+a[\alpha+wd-1][j]$ for all j ($\beta \le j \le \beta+w-1$), are computed and written in the global memory. Step 2 computes *the global column-wise sums (GS)*, $a[0][j] + a[1][j] + \cdots + a[\alpha + wd - 1][j]$ for all j ($\beta \le j \le \beta + w - 1$) and writes them in the global memory. Finally, *the global column-wise prefix-sums (GP)*, $a[0][j] + a[1][j] + \cdots + a[i][j]$ for all i and j ($\alpha \le i \le \alpha + wd - 1$ and $\beta \le j \le \beta + w - 1$) are computed and written in the global memory in Step 3. Thus, when Step 3 of all tasks is completed, all column-wise prefix-sums of a are stored in the global memory. Figure 4 illustrates the LS, the GS, and the GP of a tile.

For later reference, we define the state of a tile in the LCP. Initially, all tiles are in null state. A tile changes to State LS when values of the LS are written in the global memory in Step 1. After that, it changes to State GS when values of the GS are written in the global memory in Step 2. The LCP algorithm uses a 2-dimensional array of size $\frac{n}{wd} \times \frac{n}{w}$ in the global memory to store the states of all $\frac{n}{wd} \times \frac{n}{w}$ tiles. A CUDA block assigned to a tile updates the corresponding element of this 2-dimensional array when the tile changes the state.

A CUDA block is assigned to one of the tiles in increasing order of serial number in turn. For this purpose, a global counter c initialized by zero in the global memory is used. A CUDA kernel for the LCP algorithm invokes $\min(\frac{n^2}{w^2d}, m)$ CUDA blocks, where m is the maximum number of CUDA blocks that can be



w
wd
GS
LS
LS
GS
LS
GS
GP
Step 1
Step 2.1
Step 2.2
Step 3
global (column-wise) prefix-sums (GP)
global (column-wise) sums (GS)
local (column-wise) sums (LS)

Fig. 4. The LS, the GS, and the GP of a tile and the computation of three steps

dispatched in the GPU at the same time. For example, NVIDIA TITAN X has 28 streaming multiprocessors with 2048 resident threads each and the LCP uses CUDA blocks with 1024 threads each, we have $m = \frac{28 \cdot 2048}{1024} = 56$. The first thread 0 of every CUDA block performs atomicAdd(&c,1), which exclusively increments c and returns the value of c before addition. Thus, atomicAdd(&c,1) returns 0, 1, ... in turn and no two threads receive the same return value. A CUDA block with the first thread receiving return value k performs task k for tile k if $k < \frac{n^2}{w^2 d}$. It terminates if $k \geq \frac{n^2}{w^2 d}$. After task k is completed, it executes $k \leftarrow$atomicAdd(&c,1) again and performs task k provided that return value k satisfies $k < \frac{n^2}{w^2 d}$. Otherwise, it terminates. The same procedure is repeated as long as return value k satisfies $k < \frac{n^2}{w^2 d}$.

We first show how w^2 threads in a CUDA block perform task 0 for tile 0. Tasks 1, 2, ..., $\frac{n}{w} - 1$ can be done in the same way. Let $a[i][j]$ $(0 \leq i \leq wd - 1$ and $0 \leq j \leq w - 1)$ be elements in tile 0. A CUDA block assigned to tile 0 uses a $w \times w$ 2-dimensional array E with the diagonal arrangement. Thus, the row-wise/column-wise memory access to E is conflict-free. The details of the algorithm are spelled out as follows:

Step 1.1. Each thread $t_{i,j}$ $(0 \leq i, j \leq w-1)$ reads d elements $a[id][j]$, $a[id+1][j]$, ..., $a[id+d-1][j]$ one by one and store them in d registers.

Step 1.2. Each thread $t_{i,j}$ computes the sum $a[id][j] + a[id+1][j] + \cdots + a[id+d-1][j]$ of the d registers and write it in $E[i][j]$ in the shared memory.

Step 1.3. Execute the column-wise warp prefix scan for E. Clearly, each $E[i][j]$ stores the value of $a[0][j] + a[1][j] + \cdots + a[id + d - 1][j]$ for all i and j.

Step 1.4 and 2. The LS of tile 0, the values stored in $E[w-1][0], E[w-1][1]$, …, $E[w-1][w-1]$ are written in the global memory. Since tile 0 is in the topmost row, they are also the GS of tile 0.

Step 3. Each thread $t_{i,j}$ $(0 \leq i, j \leq w-1)$ computes the prefix-sums of $E[i-1][j]+a[id][j], a[id+1][j], a[id+2][j], \ldots, a[id+d-1][j]$ in an obvious way, and writes them in the global memory. For simplicity, we assume $E[-1][j] = 0$ for all j. Since $E[i-1][j] = a[0][j] + a[1][j] + \cdots + a[id-1][j]$, these prefix-sums thus obtained are the GP of tile 0.

Clearly, all memory access operations to the global memory are coalesced, and those to the shared memory are conflict-free.

Next, we will show how tasks $\frac{n}{w}$ and larger are performed. For simplicity, we show the algorithm for task $l\frac{n}{w}$ for tile $T(l, 0)$ such that $1 \leq l \leq \frac{n}{wd} - 1$. The other tasks can be done in the same way. Step 1, which computes the LS, can be done in the same way as task 0 for tile 0. Steps 2 and 3 of task $l\frac{n}{w}$ are spelled out as follows:

Step 2.1. The GS of tile $T(l-1, 0)$, $a[0][j] + a[1][j] + \cdots + a[lwd-1][j]$ for all j $(0 \leq j \leq w-1)$, are computed and stored in registers. We will show how these values are computed later. Let $g[j] = a[0][j]+a[1][j]+\cdots+a[lwd-1][j]$ be the GS of tile $T(l-1, 0)$ thus obtained.

Step 2.2. Each thread $t_{w-1,j}$ $(0 \leq j \leq w-1)$ computes $g[j]+E[w-1][j]$, which is equal to the GS of tile $T(l, 0)$, $a[0][j] + a[1][j] + \cdots + a[(l+1)wd-1][j]$, and writes it in the global memory.

Step 3. Each thread $t_{i,j}$ $(0 \leq i, j \leq w-1)$ computes the prefix-sums of $g[j]+E[i-1][j]+a[lwd+id][j], a[lwd+id+1][j], a[lwd+id+2][j], \ldots, a[lwd+id+d-1][j]$, which are equal to the GP of tile $T(l, 0)$, and writes them in the global memory.

The reader should refer to Fig. 4 illustrating computation performed in three steps. Step 2.1 is implemented by looking back above tiles. If $T(l-1, 0)$ is in State GS, then the GS of $T(l-1, 0)$ can be obtained simply by reading the global memory. If it is not in State GS then tiles are looked back upwards until it finds a tile with State GS. Since CUDA blocks run asynchronously, tile 0 may not be in State GS when it is looked back. If this is the case, it repeatedly reads the state of tile 0 until it is changed to State GS. Let tile $T(l', 0)$ $(l' \leq l-1)$ be the first tile in State GS. The GS of tile $T(l-1, 0)$ can be computed by summing the GS of $T(l', 0)$, the LS of $T(l'+1, 0)$, the LS of $T(l'+2, 0)$, …, and the LS of $T(l-1, 0)$. To read the LS of these tiles, the state of each tile stored in the global memory is repeatedly read until it changes to State LS.

4 Experimental Results

We have used NVIDIA TITAN X GPU, which has 28 streaming multiprocessors with 128 processor cores each to evaluate GPU implementations of column-wise prefix-sum computation.

Table 1. The running time (in milliseconds) of the column-wise prefix-sums computation for a $n \times n$ matrix and the ratio of the running time over that of matrix duplication

	n	1K	2K	4K	8K	16K	32K
Duplicate	Time	0.0274	0.0974	0.379	1.50	6.00	22.3
Naive	Time	0.277	0.557	1.26	3.12	8.48	41.0
	Ratio	10.1	5.72	3.33	2.07	1.41	1.84
Column-wise CUB	Time	0.138	0.376	1.38	5.25	20.5	83.3
	Ratio	5.04	3.86	3.64	3.49	3.41	3.74
Transposed CUB	Transpose time	0.101	0.246	0.877	3.90	15.8	61.9
	Row-wise time	0.0527	0.125	0.433	1.62	6.32	25.6
	Total time	0.154	0.370	1.31	5.52	22.1	87.5
	Ratio	5.60	3.80	3.46	3.67	3.68	3.93
LCP	Time	0.0281	0.101	0.392	1.58	6.33	23.5
	Ratio	1.02	1.04	1.04	1.05	1.05	1.06

Table 1 shows the running time in milliseconds for an $n \times n$ matrix with 4-byte single precision floating point numbers from n =1K (1024) to 32K (32768). In "duplicate" the input matrix is duplicated using cudaMemcpy, which reads all n^2 elements of the matrix and writes them in the other space of the global memory. Clearly, no column-wise prefix-sum algorithm cannot be faster than "duplicate", we can say that the running time of "duplicate" is the lower bound of that of any column-wise prefix-sum computation. In "naive", the naive column-wise prefix-sum algorithm executed using $\frac{n}{32}$ CUDA blocks with 32 threads each. In the table, "ratio" is the running time ratio over "duplicate", that is, the running time of "naive" divided by that of "duplicate." Thus, the ratio indicates the computation overhead, that is, the algorithm has ϵ overhead if it is $1 + \epsilon$. Since "naive" uses only n threads, the ratio is much larger than 1. In particular, the overhead is more than 900% when n = 1K, because it uses only 1024 threads on the GPU with 3584 cores.

In "column-wise CUB", CUB-prefix is executed for every column in parallel. Since memory access to the global memory is not coalesced, the overhead is more than 200% for all n. In "transposed CUB", matrix transpose, row-wise CUB, and matrix transpose are executed to compute the column-wise prefix-sums. In the table, "transpose time" is the time for performing matrix transpose twice and "row-wise time" is that for executing CUB-prefix for every row in parallel. We can see that row-wise CUB is much faster than column-wise CUB, because memory access to the global memory by row-wise CUB is coalesced. However, due to large overhead of matrix transpose, the running time is almost the same as "column-wise CUB."

In our LCP, we fix $d = 8$, because the running time of LCP for $d = 8$ is faster than $d = 1, 2, 4, 16$, and 32 for all n. Thus, the size of each tile is fixed to 256×32. Since each of 28 streaming multiprocessor has 2048 resident threads, we should invoke $\min(\frac{n^2}{8192}, 56)$ CUDA blocks with 1024 threads each. Since $n \geq 1024$, only 56 CUDA blocks are invoked. From the table, we can see that the LCP is much faster than "naive", "column-wise CUB", and "transposed CUB." In particular, the ratio is very close to 1 and the overhead is only 2–6%. Thus, we can say that computation performed by our LCP is almost hidden by necessary coalesced memory access to the global memory.

5 Conclusion

In this paper, we have shown an almost optimal parallel algorithm for computing the column-wise prefix-sums of a matrix on the GPU. The computation overhead over necessary global memory access is only only 2–6%. Hence, our implementation is almost optimal.

References

1. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. In: GPU Gems 3. Addison-Wesley (2007). Chapter 39
2. Hwu, W.W.: GPU Computing Gems Emerald Edition. Morgan Kaufmann, Burlington (2011)
3. Kasagi, A., Nakano, K., Ito, Y.: Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations. In: Proceedings of International Conference on Parallel Processing (ICPP), pp. 251–260, September 2014
4. Lauritzen, A.: Summed-area variance shadow maps. In: GPU Gems 3. Addison-Wesley (2007). Chapter 8
5. Man, D., Uda, K., Ueyama, H., Ito, Y., Nakano, K.: Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs. Int. J. Netw. Comput. **1**(2), 260–276 (2011)
6. Merrill, D.: CUB: a library of warp-wide, block-wide, and device-wide GPU parallel primitives (2017). https://nvlabs.github.io/cub/
7. Merrill, D., Garland, M.: Single-pass parallel prefix scan with decoupled look-back. Technical report NVR-2016-002, NVIDIA, March 2016
8. Nakano, K.: An optimal parallel prefix-sums algorithm on the memory machine models for GPUs. In: Xiang, Y., Stojmenovic, I., Apduhan, B.O., Wang, G., Nakano, K., Zomaya, A. (eds.) ICA3PP 2012. LNCS, vol. 7439, pp. 99–113. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33078-0_8
9. Nakano, K.: Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models. IEICE Trans. Inf. Syst. **E96–D**(12), 2626–2634 (2013)
10. Nakano, K.: Simple memory machine models for GPUs. Int. J. Parallel Emerg. Distrib. Syst. **29**(1), 17–37 (2014)
11. Nehab, D., Maximo, A., Lima, R.S., Hoppe, H.: GPU-efficient recursive filtering and summed-area tables. ACM Trans. Graph. **30**(6), 176 (2011)
12. NVIDIA Corporation: NVIDIA CUDA C best practice guide version 3.1 (2010)
13. NVIDIA Corporation: NVIDIA CUDA C programming guide version 8.0, March 2017
14. Takeuchi, Y., Takafuji, D., Ito, Y., Nakano, K.: ASCII art generation using the local exhaustive search on the GPU. In: Proceedings of International Symposium on Computing and Networking, pp. 194–200, December 2013

A Combination of Intra- and Inter-place Work Stealing for the APGAS Library

Jonas Posner(✉) and Claudia Fohry

Research Group Programming Languages/Methodologies,
University of Kassel, Kassel, Germany
{jonas.posner,fohry}@uni-kassel.de

Abstract. Since today's clusters consist of nodes with multicore processors, modern parallel applications should be able to deal with shared and distributed memory simultaneously. In this paper, we present a novel hybrid work stealing scheme for the APGAS library for Java, which is a branch of the X10 project. Our scheme extends the library's runtime system, which traditionally performs intra-node work stealing with the Java Fork/Join framework. We add an inter-node work stealing scheme that is inspired by lifeline-based global load balancing. The extended functionality can be accessed from the APGAS library with new constructs. Most important, locality-flexible tasks can be submitted with `asyncAny`, and are then automatically scheduled over both nodes and cores. In experiments with up to 144 workers on up to 12 nodes, our system achieved near linear speedups for three benchmarks.

Keywords: Task pool · Work stealing · APGAS · Java

1 Introduction

Almost all present high performance computing systems deploy multicore processors and a high speed network. Efficient use of these architectures requires a combination between the traditional approaches of shared and distributed memory parallel programming.

The Partitioned Global Address Space (PGAS) model is a popular base for such hybrid programming. In this model, a partition of the global address space, together with some computational resources, is denoted as *place*. Each place can access every memory partition, but accesses to the local partition are faster than accesses to remote partitions. Beside the original PGAS model, there is an asynchronous variant, in which sequential tasks can be spawned at runtime on user-defined places.

This asynchronous PGAS variant is, for instance, implemented in the language X10 [5] and the related APGAS library for Java [15]. In both, tasks can be created on the current place with the `async` construct, and on a specified remote place with a combination of the constructs `async` and `at`.

Many applications involve locality-flexible tasks that can run equally well on any resource of the overall system. For these tasks, programmers want to achieve load balancing without the need to worry about their placement. HabaneroUPC++, another implementation of the asynchronous PGAS variant, provides an `asyncAny` construct for spawning locality-flexible tasks [7]. These tasks are automatically distributed over all places and place-internal computing resources, called *workers*. Implementation strategies are described in [7]. Unfortunately, that publication does not report speedup values, and we were not able to obtain speedups with the code provided by the authors [14].

As of yet, `asyncAny` is unique to HabaneroUPC++, whereas other asynchronous PGAS implementations only support intra-place work stealing in their runtime systems. X10's standard library additionally supports inter-place work stealing for task-based irregular applications with the separate Global Load Balancing (GLB) framework [18]. In previous work, we developed a multithreaded GLB variant for APGAS [12]. Still, both GLB implementations have a limitation of one worker per place. Thus, exploiting a multicore system requires to start a separate place on each core, which results in unnecessary communication and increased memory load.

This paper introduces a novel hybrid work stealing scheme for the APGAS library. We selected this library, since it is based on Java, one of the most popular programming languages in general, which also gains more and more attention as an interesting language in high-performance computing [1].

Our hybrid work stealing scheme is inspired by the `asyncAny` construct of HabaneroUPC++, but the implementation is different. Our scheme combines the intra-place work stealing of Java's `ForkJoinPool` with GLB's lifeline-based inter-place work stealing. We customized the `ForkJoinPool` to enable polling out multiple tasks at once, which can then be sent to a remote thief. In contrast, HabaneroUPC++ utilizes the scalable locality-aware adaptive work stealing scheduler for intra-place work stealing [4]. In its inter-place scheme, a good victim is selected with the help of network Remote Direct Memory Access (RDMA), and an unlimited number of remote victims is contacted.

Our implementation extends the APGAS source code [6]. New constructs are `asyncAny`, `asyncAnyFinish` and a few others that supports storing and reducing task results. The `asyncAnyFinish` construct resembles X10's `finish`. It defines a scope, in which `asyncAny`-tasks may be spawned recursively and synchronize at the end. Overall, this paper makes the following contributions:

- It introduces a hybrid work stealing technique that combines GLB's lifeline scheme with Java's `ForkJoinPool`.
- It describes the implementation of this technique in the APGAS library for Java.
- It presents experimental results for three benchmarks (Unbalanced Tree Search, Betweenness Centrality and NQueens) that show near linear speedups over sequential execution with up to 144 workers on up to 12 places.

The paper is structured as follows. Section 2 provides background information about APGAS and the GLB lifeline scheme. After that, Sect. 3 explains programming with the novel constructs. The design and implementation of the hybrid work stealing technique are explained in Sect. 4. Experiments are described and discussed in Sect. 5. The paper finishes with related work and conclusions in Sects. 6 and 7, respectively.

2 Background

2.1 APGAS Library

The APGAS library is written in Java 8. It brings the parallel programming concepts of X10 to Java by using lambdas [15]. Each place is represented by a Java Virtual Machine (JVM).

APGAS programmers encapsulate computations in lightweight *asynchronous tasks*, and specify the mapping of each task to a place. The APGAS runtime executes the different tasks with multiple workers. Inside each place, it deploys a task pool that is an instance of the Java class `ForkJoinPool` [9]. Thus, workers correspond to Java threads. Users can set the number of workers via `APGAS_THREADS`, default is the number of CPU cores.

The `async` construct submits a task to the local `ForkJoinPool`. It is executed right away, or when a worker becomes available. The call always returns immediately. The `at` construct sends a task to a specified remote place, where it is inserted into the local `ForkJoinPool`. The `at` construct blocks until the transferred task has been executed. The `asyncAt` construct performs the same actions as `at`, but in an asynchronous manner such that it returns immediately. The `immediateAsyncAt` construct immediately starts a new Java thread on the remote place, which executes a specified task. Such a thread runs concurrently to the `ForkJoinPool`.

When task spawning is enclosed by a `finish` block, the block's execution ends only when all submitted `async` and `asyncAt`-tasks, including recursively spawned ones, have been executed.

2.2 Lifeline-Based Global Load Balancing

GLB deploys the so-called lifeline scheme. Each worker runs on a separate place, where it maintains its own task pool. It takes tasks from the pool for processing, and inserts any newly generated tasks there. The following task model is used:

- Tasks are free of side effects.
- Each task generates a result and possibly new tasks.
- Each worker accumulates task results in a partial result.
- The final result is computed from partial results by a reduction operation.
- All results have the same type. The reduction operator matches this type, and is commutative and associative.

If a worker runs out of tasks, it tries to steal tasks from another worker. First, the *thief* contacts up to w random *victims* and, if not successful, afterwards up to z *lifeline buddies*. If a victim has no tasks, it responds with a reject message, otherwise it sends tasks, called *loot*. When all $w + z$ steal attempts have returned unsuccessfully, the thief goes *inactive*. An inactive thief is restarted when a lifeline buddy sends loot in reaction to an earlier recorded steal request. When all workers have become inactive, the final result is computed.

3 Programming with AsyncAny-Tasks

We provide the following constructs for APGAS:

– `asyncAny`: Submits a locality-flexible task, which is initially placed in the local pool.
– `asyncAnyFinish`: Returns when all submitted tasks from a block have been processed.
– `staticInit`: Creates a copy of static data (e.g. constants), which are passed as parameter on each place.
– `staticAsyncAny`: Resembles `asyncAny`, but allows to specify an initial placement and refers to a list of tasks.
– `mergeResult`: Merges a task result into a partial result.
– `reduceAsyncAny`: Computes the final result by reduction and returns it.

Listing 1.1 shows an example. In line 1, an `asyncAnyFinish` block is spawned to detect when all `asyncAny`-tasks that are submitted in line 3 of the loop have been finished.

```
1  asyncAnyFinish(() -> {
2    for (int i = 0; i < n; i++) {
3      asyncAny(() -> {
4        UserResult r = new UserTask().compute();
5        mergeResult(r);
6      });
7    }
8  }, 1000);
9  reduceAsyncAny().display();
```

Listing 1.1. Submitting `asyncAny`-tasks within an `asyncAnyFinish` block

The class `UserTask` must be provided by the user for each type of task. Its `compute()` method contains the application code for the task. The computation returns a result of type `UserResult` (line 4). This class has to extend the abstract APGAS class `AsyncAnyResult<T>`, where `T` is the result type. Therefore, the user result class must implement the methods `display()`, `getResult()` and a few others.

Internally, the APGAS runtime deploys a `result` array of type `AsyncAnyResult<T>` on each place. Its length corresponds to the number of workers. The call of `mergeResult` (line 5) saves the task result by accumulating it in the worker's array entry.

The final result of all tasks is computed by the call to `reduceAsyncAny` (line 9). This call must be placed behind an `asyncAnyFinish`-block and refers to the result computed in this block. The `asyncAnyFinish` construct provides an optional second parameter which switches on a periodic computation and display of a preliminary result. In line 8, the parameter is set to `1000`, such that the preliminary result is displayed every 1000 ms.

4 Design and Implementation

We extended the open source code of APGAS and will commit our changes to the official APGAS repository [6].

As mentioned in Sect. 2, APGAS deploys the Java class `ForkJoinPool` for the place-internal pools. A call of `asyncAny` inserts a task into the local instance of this pool, from where it can be stolen away later. We combined the intra-place work stealing of the internal pool with inter-place work stealing, which is performed by a dedicated management worker.

4.1 Management Worker

A call to `asyncAnyFinish` starts one management worker on each place. This worker is not scheduled by the `ForkJoinPool`, but runs in a separate Java thread concurrently to the pool. The management worker realizes a modified variant of the lifeline scheme. Pseudocode of its main loop is shown in Listing 1.2. The operations in the loop are performed once per second, except when the worker is *inactive* (line 10). In GLB, in contrast, a worker performs these operations after processing n tasks.

If the local pool contains less unprocessed tasks than there are available workers, work stealing starts (line 4). This is a difference to GLB, where the stealing only starts when a worker has no tasks left. Stealing is performed with `immediateAsyncAt`. Unlike the lifeline scheme, our work stealing scheme involves direct accesses to remote task pools, in which a thief tries to pull half of the unprocessed tasks from the victim's pool itself. We modified the `ForkJoinPool` to enable pulling multiple tasks at once. This polling can be performed concurrently to the running computation, because the task pool deploys internal synchronization. If the victim is out of work, the thief aborts the stealing request. Otherwise, the loot is sent with `staticAsyncAny`.

If $w + z$ stealing attempts were unsuccessful, the management worker sends a notification to place 0 and goes inactive (line 6–9).

The management worker is reactivated when an `asyncAny`-task is submitted on its place. This can happen through a user call to `asyncAny` or when loot arrives via `staticAsyncAny`.

```
 1  while (tasks available) {
 2    send loot to recorded lifeline thieves;
 3    if (not enough local tasks left) {
 4      try to steal from up to w+z victims;
 5    }
 6    if (all local tasks have been executed &&
          all potential victims were contacted) {
 7      notify place 0;
 8      go inactive;
 9    }
10    sleep one second;
11  }
```

Listing 1.2. Main loop of management worker

4.2 AsyncAnyFinish

The existing `finish` implementation observes every single task, which induces a high overhead when the number of tasks is large. Therefore, we implemented a new `asyncAnyFinish` construct, which observes loot only. This construct internally starts a new thread that executes the corresponding block, and afterwards cyclically checks whether (1) the internal pool contains unprocessed tasks, and (2) there are unprocessed tasks in remote pools.

The first condition is checked with standard methods from the `ForkJoinPool` class. To check the second condition, each place holds an `int` array `stealCounts` with one entry per place. Entries are initialized with 0. Before a victim sends loot, it increments its local `stealCounts[thief]`. When a thief receives loot, it decrements its local `stealCounts[thief]`. Just before a management worker goes inactive, it sends its `stealCounts` array to place 0 (line 7 in Listing 1.2). On place 0, the received arrays are added to the local `stealCounts` array. If all entries are 0, the second condition from above must be met, and all management workers have become inactive. Thus, the `asyncAnyFinish` thread on place 0 terminates all management workers and itself.

5 Experiments

Experiments were run on a cluster with 12 homogeneous nodes. Each node has 256 GB of main memory and two six-core Intel Xeon E5-2643-v4E5 CPUs [16]. First, we measured the intra-place speedup by varying the number of workers from 1 to 12. Then, we measured the inter-place speedup by varying the number of places from 1 to 12. Here, each place was mapped to a separate node and deployed 12 workers internally. Java was used in version 1.8.0_121.

As benchmarks we adapted Unbalanced Tree Search (UTS) [8], NQueens and Betweenness Centrality (BC) [2]. UTS and BC were taken from X10's standard library [5], and NQueens was taken from the HabaneroUPC++ repository [14]. We ported the three benchmarks to Java, thereby using APGAS and our novel constructs.

UTS counts the number of nodes in a highly irregular tree, which is dynamically generated from node descriptors. NQueens calculates the number of placements of N queens on an $N \times N$ chessboard, so that no two queens threaten each other. BC calculates a centrality score for each node of a given graph. UTS and NQueens start with a single task, which is submitted with `asyncAny`, while the other tasks are spawned dynamically. BC initializes all tasks statically at the beginning and spreads them evenly over all places with `staticAsyncAny`. The result is a single `long` value for UTS and NQueens, and a `long` array with one entry per graph node for BC.

UTS was started with geometric tree shape, branching factor $b = 4$, random seed $s = 19$, and tree depth $d = 17$. Like in GLB [5], the initial task processes up to 511 tree nodes, and afterwards submits a new `asyncAny`-task for half of the remaining tree nodes. Any subsequent task does the same. NQueens was started with $N = 17$, but `asyncAny`-tasks are only spawned up to a tree depth of 6, as in the HabaneroUPC++ implementation [14]. BC was started with random seed $s = 2$, number of graph nodes $N = 2^{15}$ for intra-place experiments and with $N = 2^{17}$ for inter-place experiments. Moreover, we observed the best performance with a chunk size of 32.

Figures 1(intra-place) and 2(inter-place) depict the measured speedups for the three benchmarks. As we can see, all benchmarks achieve near linear speedup. NQueens even has a slightly superlinear speedup with 12 workers, which is probably caused by its exploratory decomposition [3]. NQueens deviates up to 2.40%

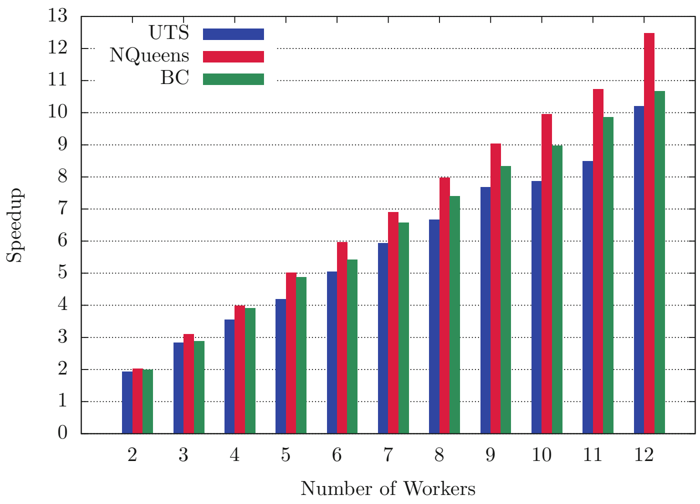


Fig. 1. Intra-place speedup over sequential execution time

from the linear speedup, UTS deviates up to 22.78% (both with 11 workers), and BC deviates up to 11.12% (with 12 workers).

Figure 2 shows that all benchmarks perform well with the new hybrid work stealing scheme. They get close to near linear speedup, although the deviation is a bit higher with 14.78% for BC and 26.54% for NQueens, both on 12 places. UTS, on the other hand, has a smaller deviation of 17.62% on 12 places. The overall slightly lower performance of inter-place work stealing is probably caused by communication costs. Comparing the hybrid variants with 144 workers to the sequential base variants, UTS achieves a speedup of 100 and NQueens of 109.

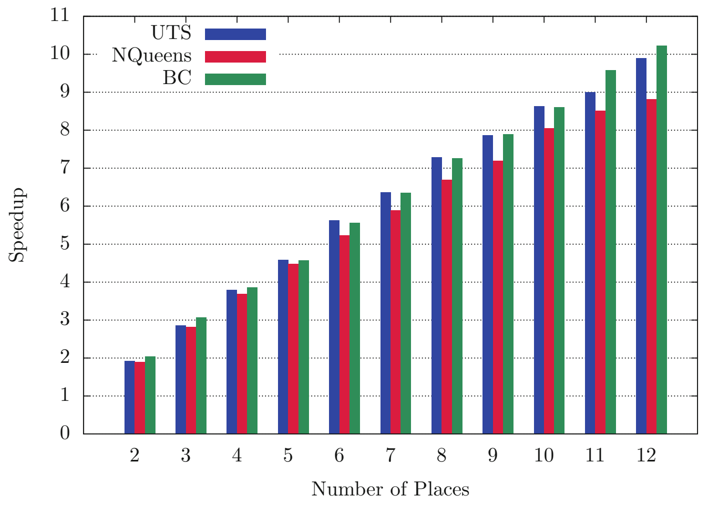


Fig. 2. Inter-place speedup over one place execution time with 12 workers

Furthermore, we ran experiments with 11 workers per place instead of 12, so that one CPU core was reserved for the management worker. The loss in speedup was about proportional to the reduction in the number of workers, from which we conclude that the management workers needs almost no computational resources. Consequently, it does not pay off to reserve a core for it.

6 Related Work

As mentioned before, the idea of hybrid work stealing with `asyncAny` tasks was adopted from HabaneroUPC++ [7], but our implementation is fundamentally different. The HabaneroUPC++ scheme does not limit the number of remote victims and selects them with the help of network Remote Direct Memory Access (RDMA). A performance comparison between variants with and without RDMA in [7] reveals an enhancement by up to 7% with RDMA. In contrast, we have adopted the lifeline scheme from GLB [18], and thus steal from up to $w + z$

remote victims, which are selected without RDMA. Like HabaneroUPC++, we utilize a dedicated management worker for the inter-place work stealing. However, HabaneroUPC++ runs it on a dedicated CPU core that does not participate in the actual computation, whereas we use as many computation workers as cores. Finally, HabaneroUPC++ binds to C++, and APGAS to Java.

An earlier outcome of the Habanero project is Habanero-C MPI (HCMPI). It integrates the place-internal task parallelism of Habanero-C [13] with MPI's message passing model between the places.

Yamashita and Kamada [17] present multistage execution and multithreading for GLB in X10. Each worker maintains an own queue, and each place holds two shared queues for intra- and inter-place work stealing, respectively. However, the overall scheme is quite complicated, and the implementation has problems with network message scheduling.

Paudel *et al.* investigate hybrid task placement in X10 with work stealing and work dealing, respectively [10,11]. Both papers deploy a dedicated thread for inter-place communication. Programmers use annotations to distinguish tasks into location-sensitive and location-flexible ones. Both task types are handled together in their work stealing scheme, which uses multiple deques per place. Work dealing achieves speedups from 2% to 16% [10], and work stealing achieves speedups from 12% to 31% [11], each in comparison to X10's default scheduler.

7 Conclusions

In this paper, we have presented a combined intra- and inter-place work stealing technique for the APGAS library. APGAS programmers can submit locality-flexible tasks via `asyncAny`, and the APGAS runtime automatically schedules them over all places and their computational resources. We have described the usage and implementation of the extended APGAS library. Moreover, we ported three benchmarks to this library, and observed near linear speedups with up to 144 workers on up to 12 places.

Future work should measure performance on a larger scale. Furthermore, we plan to combine the submission of locality-flexible and standard APGAS tasks inside the same `finish`, and support nested `asyncAnyFinish`s.

Acknowledgments. This work is supported by the Deutsche Forschungsgemeinschaft, under grant FO 1035/5-1.

References

1. Diaz, J., Munoz-Caro, C., Nino, A.: A survey of parallel programming models and tools in the multi and many-core era. IEEE Trans. Parallel Distrib. Syst. **23**(8), 1369–1386 (2012)
2. Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry **40**(1), 35 (1977)
3. Grama, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. Addison-Wesley, Boston (2003)

4. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP 2010. ACM Press (2010)
5. IBM: Core implementation of X10 programming language including compiler, runtime, class libraries, sample programs and test suite (2017). https://github.com/x10-lang/x10
6. IBM: The APGAS library for fault-tolerant distributed programming in Java 8 (2017). https://github.com/x10-lang/apgas
7. Kumar, V., Murthy, K., Sarkar, V., Zheng, Y.: Optimized distributed work-stealing. In: Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms, IA3 2016, pp. 74–77. IEEE Press (2016)
8. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.-W.: UTS: an unbalanced tree search benchmark. In: Almási, G., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72521-3_18
9. Oracle: Class ForkJoinPool (2017). http://download.java.net/java/jdk9/docs/api/java/util/concurrent/ForkJoinPool.html
10. Paudel, J., Tardieu, O., Amaral, J.N.: Hybrid parallel task placement in X10. In: Proceedings of the ACM SIGPLAN X10 Workshop. ACM Press (2013)
11. Paudel, J., Tardieu, O., Amaral, J.N.: On the merits of distributed work-stealing on selective locality-aware tasks. In: Proceedings of the International Conference on Parallel Processing (ICPP). IEEE (2013)
12. Posner, J., Fohry, C.: Cooperation vs. coordination for lifeline-based global load balancing in APGAS. In: Proceedings of the ACM SIGPLAN X10 Workshop. ACM Press (2016)
13. Rice University: Habanero C language (2017). https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C
14. Rice University: HabaneroUPC++: a Compiler-free PGAS Library (2017). https://github.com/habanero-rice/habanero-upc
15. Tardieu, O.: The APGAS library: resilient parallel and distributed programming in Java 8. In: Proceedings of the ACM SIGPLAN X10 Workshop. ACM Press (2015)
16. University of Kassel: Scientific data processing (2017). https://www.uni-kassel.de/its-handbuch/en/daten-dienste/wissenschaftliche-datenverarbeitung.html
17. Yamashita, K., Kamada, T.: Introducing a multithread and multistage mechanism for the global load balancing library of X10. J. Inf. Process. **24**(2), 416–424 (2016)
18. Zhang, W., Tardieu, O., Grove, D., Herta, B., Kamada, T., Saraswat, V., Takeuchi, M.: GLB lifeline-based global load balancing library in X10. In: Proceedings of the ACM Workshop on Parallel Programming for Analytics Applications (PPAA). ACM Press (2014)

Benchmarking Molecular Dynamics with OpenCL on Many-Core Architectures

Rene Halver[1], Wilhelm Homberg[1], and Godehard Sutmann[1,2](✉)

[1] Jülich Supercomputing Centre (JSC), Institute for Advanced Simulation (IAS), Forschungszentrum Jülich (JSC), 52425 Jülich, Germany
g.sutmann@fz-juelich.de
[2] ICAMS, Ruhr-University Bochum, 44801 Bochum, Germany

Abstract. Molecular Dynamics (MD) is a widely used tool for simulations of particle systems with pair-wise interactions. Since large scale MD simulations are very demanding in computation time, parallelisation is an important factor. As in the current HPC environment different heterogeneous computing architectures are emerging, a benchmark tool for a representative number of these architectures is desirable. OpenCL as a platform-overarching standard provides the capabilities for such a benchmark. This paper describes the implementation of an OpenCL MD benchmark code and discusses the results achieved on different types of computing hardware.

Keywords: Molecular Dynamics · OpenCL
Shared memory parallelisation · Many-core architectures

1 Introduction

Molecular Dynamics (MD) is widely used in various scientific domains, e.g. materials science or biophysics, where the evolution of specific systems can be described by point-like or extended particles, obeying the classical equations of motion [5,8]. Parametrised potentials describe pair-wise interactions between particles that may have either short-ranged (e.g. Lennard Jones interactions [5]) or long-ranged (e.g. electrostatics [7]) influences. The essential difference between long- and short-range interactions is the number of interaction partners, which strongly determines the performance of the method and also determines different types of, e.g., parallelisation schemes. While long range interactions require essentially all atoms in the system as interaction partners, short range interactions can be restricted to a narrow range, defined by a spherical region of radius R_c (the cutoff radius), in which interactions decrease to a sufficiently small value which can be tolerated as error.

In the present paper we will focus on short-range interactions, for which also neighbour list techniques will be considered which allow for a linear computational complexity with increasing number of particles in the system (cmp. Fig. 1).

For a shared-memory parallelisation the construction of these neighbour lists must avoid race conditions which would occur, when multiple threads try to update the list of a single cell at the same time, which can be expected for the case of a thread-parallel implementation where particles are divided between threads and are sorted into the cell structure simultaneously. To avoid possible race-conditions there exist basically three options: (i) explicit synchronisation; (ii) list copies; or (iii) atomic memory access implemented via compare-and-swap (CAS) operations.

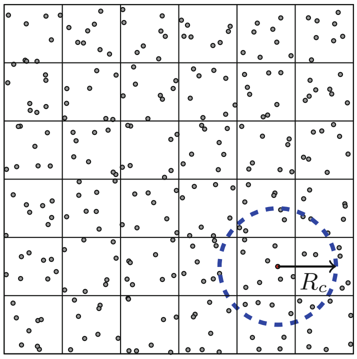


Fig. 1. Schematic of a particle system in 2D with overlayed cell structure for sorting. The circle with radius R_c illustrates the interaction range of a tagged particle.

In order to implement a function portable benchmark that can be run on a variety of different architectures, it is required to use a programming language that supports a large variety of architectures. OpenCL is one possible choice as it supports code execution on CPUs, GPUs, FPGAs as well as Intels Xeon Phi architecture. [1] Although other languages or language extensions, such as Intel TBB or OpenACC exist, which provide the possibility to run code on a set of different architectures, for this work OpenCL was chosen, because it supports all of the platforms available at the Jülich Supercomputing Centre (JSC). The set of features of OpenCL that can be used for a platform-overarching benchmark is limited by the lowest commonly supported OpenCL standard as well as by the set of extensions commonly available on these platforms. While the standard level defines the syntax and general features that can be used, some functionality is kept in extensions, e.g. the use of double-precision calculations or CAS functionality. To execute the same program version on all considered architectures we had to comply with OpenCL 1.2, which was found as common standard level on all machines.

2 Benchmark

In the present article we focus on short range interacting particle systems, where the range of influence is defined by the cutoff radius R_C. When introducing the concept of linked-cell lists, the computational complexity is $\mathcal{O}(NM)$, where N is the number of particles in the system and M the maximum number of particles in a cell [8,10]. As benchmark system we consider particles in a cuboid 3-dimensional box of lengths L_α $(\alpha = x, y, z)$ with periodic boundary conditions, interacting via the repelling part of the Lennard-Jones potential [5], which is a typical representative for a short range potential

$$U(r \leq r^*) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} + \frac{1}{4} \right] \tag{1}$$

with cut-off radius $r^* = 2^{1/6}\sigma$ and $U(r > r^*) = 0$ from where forces $\mathbf{F} = -\nabla U(r)$ onto particles are computed; r is the distance between two particles, ε the depth of the potential well and σ the characteristic size of a particle. To propagate particles continuously in space the classical equations of motion are integrated via the standard Verlet algorithm [5].

In the present article we consider a constant average number of particles per cell $\langle M\rangle$, which leads to a total number of particles in the system $N = n_x n_y n_z \langle M\rangle$, with n_x, n_y, n_z being the number of cells in each cartesian direction. The relation of cell size l_c to the cutoff radius R_c is simply given by $R_c = l_c = 2^{1/6}\sigma$ or $\sigma = l_c/2^{1/6}$ and therefore the boxsize is $L_\alpha = n_\alpha R_c$.

In the next section different techniques are described, which improve the performance of the simulation before presenting the results of the benchmarks in the final section.

3 Implementation of the Algorithm and Data Structures

3.1 Algorithmic Implementation Details

While a multi-node parallelization of MD simulations, e.g. based on a domain decomposition [9], is standard, the shared-memory parallelization of neighbor cell construction within each of the decomposed domains is not trivial. Since the most efficient way to sort particles into spatial cells is to distribute particles onto different threads. This might lead to possible concurrent memory accesses, if more than one thread attempts to sort particles into the same cell. As the memory access is not synchronized by default, race conditions can occur due to simultaneous write operations into single memory locations, leading to erroneous accounting of particles and list constructions.

Three different approaches will be described to avoid such race conditions: (i) list copying, (ii) explicit synchronization and (iii) atomic memory access. Each of these approaches has its specific advantages and disadvantages.

Of the three different approaches the synchronization-based approach is the one that requires the fewest changes to the sequential implementation (ref. Algorithm 1). The array containing the cell entries needs to be initialized with a terminating value indicating the end of the list, the array containing the particle lists needs to be initialized similarly.

Algorithm 1. Sequential implementation of neighbor cell sorting

```
entry ← EOA; list ← EOA // initialize to end-of-array (EOA)
for all particles pidx ∈ $\{i\}_N$ do
    cidx ← f (x[pidx], y[pidx], z[pidx]) // calculate cell index from particle coordinates
    list[pidx] ← entry[cidx] // update list element to current cell entry
    entry[cidx] ← pidx // update cell entry;
end
```

As is seen in Algorithm 1 a problem might occur if two threads try to simultaneously update the cell information. If these threads execute the first step, i.e. updating `list[pidx]`, before any of them completes the second step, i.e. updating `entry[cidx]`, it follows that their particles are pointing in `list` to the same former entry particle, `entry[cidx]`, ignorant of each other. Then, after updating `entry[cidx]` in the second step, `entry[cidx]` will contain only one of those particle indices, the linked list is broken and excludes the other particle from being accessible through the list.

In order to combine the two statements into one 'atomic' statement, one can either use critical sections or locks. Both of these techniques introduce overhead cost due to the implicit synchronization. Of the two the latter shows a much better scaling behavior than the first, since the creation of a critical section will serialize that section [6]. For this benchmark all of these synchronization-based approaches are not feasible, since the asynchronous execution of work groups excludes global synchronization. Therefore race-conditions happening between two different work groups cannot be avoided by these techniques.

In contrast to the synchronization-based approaches the copy-based approach utilizes thread-local partial copies of the final result in order to avoid race-conditions. Each thread independently works on its local copy and sorts all of its particles into this copy. After each thread has finished the local copies are merged. While this variant can be very efficient for a small number of threads, it becomes less effective once the number of threads reaches a threshold value, which depends on memory size and bandwidth [6]. This is due to the increased number of copies, which have to be filled simultaneously, leading to random access to (main) memory. For massively parallel systems, this approach is not feasible since the memory size requirements grow linearly with the number of threads, particularly on GPUs or Intel MICs, where hundreds and thousands of threads work concurrently. As a consequence, this approach has been discarded for the present benchmark.

Listing 1. Creation of a neighbor list using CAS operations in OpenCL

```
int old, cmp;
// arrays of particle positions (x,y,z), NC = no. of cells per dim
int cidx = (int)(x[pidx]/l_c) +
           (int)(y[pidx]/l_c) * NC +
           (int)(z[pidx]/l_c) * NC * NC;
//application of CAS operation to update list/entry
do
{
  // store old entry particle index
  old = entry[cidx];
  // update next particle in list for particle pidx
  list[pidx] = old;
  // try to update entry particle of target cell cidx
  cmp = atomic_cmpxchg(entry+idx,old,gid);
}
// if update failed, repeat the process
while(old != cmp);
```

Therefore, an approach is required which ensures the correctness of a parallel list construction without exacerbating the memory demand. Atomic memory access is a possible solution utilizing the compare and swap (CAS) operation. This hardware operation compares the value stored at a memory address to a test value before updating the memory address. Listing 1 shows as an example for the parallel list construction implemented with a CAS operation in OpenCL: First, the cell index `cidx` is calculated for the local particle `pidx`. Within a loop the first entry of the particle list of this cell `entry[cidx]` is stored in a temporary test value `old`. Then, the particle list at index `pidx` is updated to the value of `old`. This operation can be performed safely, since no other work item processes particle `pidx`. Next, the CAS operation is used to attempt an update of `entry[cidx]`. Now two different outcomes might occur: (i) `entry[cidx]` was changed in the mean time by another work item. In this case the value of `old` is not equal to the current value of `entry[cidx]`, the update is omitted and the loop is repeated. (ii) the value of `entry[cidx]` was not changed and its value is identical to that of `old`. In this case `entry[cidx]` is updated. The CAS operation returns the current value of `entry[cidx]`, for (i) a value different from `old`, for (ii) the same value as `old`. In order to check the success of the update, the return value of the CAS operation is stored to `cmp` and compared with `old` at the end of the loop.

Compared to an update of a memory location by an assignment, hardware supported CAS operations slightly increase the runtime due to the performed compare operation. The essential advantage of the CAS operation is that it can be applied to work items of different work groups in an OpenCL implementation. Therefore the implementation using the CAS operation was the method of choice for this benchmark.

3.2 Organization and Distribution of Data Structures

An important aspect concerns data locality, i.e. both the important difference between sorted and unsorted particle data and their access as well as the layout of the data structures containing this data. To this end, two different memory layouts were implemented for the arrays used to store the particle data. The first is an *array of structures* (AoS), where each structure contains the data of a single particle, the second is a *structure of arrays* (SoA), where a single structure contains a collection of arrays, each representing a parameter of a particle. In a SoA the data of a single particle has the same index in each of the arrays. For both implementations (AoS and SoA) a sorting algorithm was implemented, that sorts the particle data array with regard to the neighbor list, i.e. particles in the same neighbor cell are grouped together in the particle data array to have higher data locality. In the result section differences between these four possible implementations will be shown.

Table 1. List of implemented OpenCL kernels: distribution of (p) particles or (c) cells onto work items.

Kernel function	Distrib.
Creation of particles	p
Initial. of cells	c
Initial. of particle lists	p
Creation of neighbor lists	p
Counting of cell contents	c
Prefix sums calc. (sorting)	c
Resort of particle array	c
Calculation of interactions	c
Integration of particles	p

In order to compare a representative set of multi- and many-core architectures, while avoiding a rewrite of the code for each individual type, OpenCL was selected as a portable programming language. Basic parts of the MD algorithm were implemented into different OpenCL kernels (see Table 1), in order to benchmark the functional units of the program independently. The table shows the different ways of distributing data to the work items and related kernel functions which either act on a single particle or on a complete cell. Some of the kernel functions require data of all particles in a cell to compute properties resulting from the whole environment of a particle, e.g. the calculation of total forces on a particle. Others can update the individual particle state information independently from other particles, e.g. the propagation of position or velocities.

If not mentioned otherwise, the number of work items (wi) within a work group (wg) is kept constant, so that the number of required work groups for a given system is calculated by $n_{wg} = n/n_{wi}$, where $n = N$ (number of particles) or $n = N_c$ (number of cells), depending on the kernel (Table 1).

4 Architectures

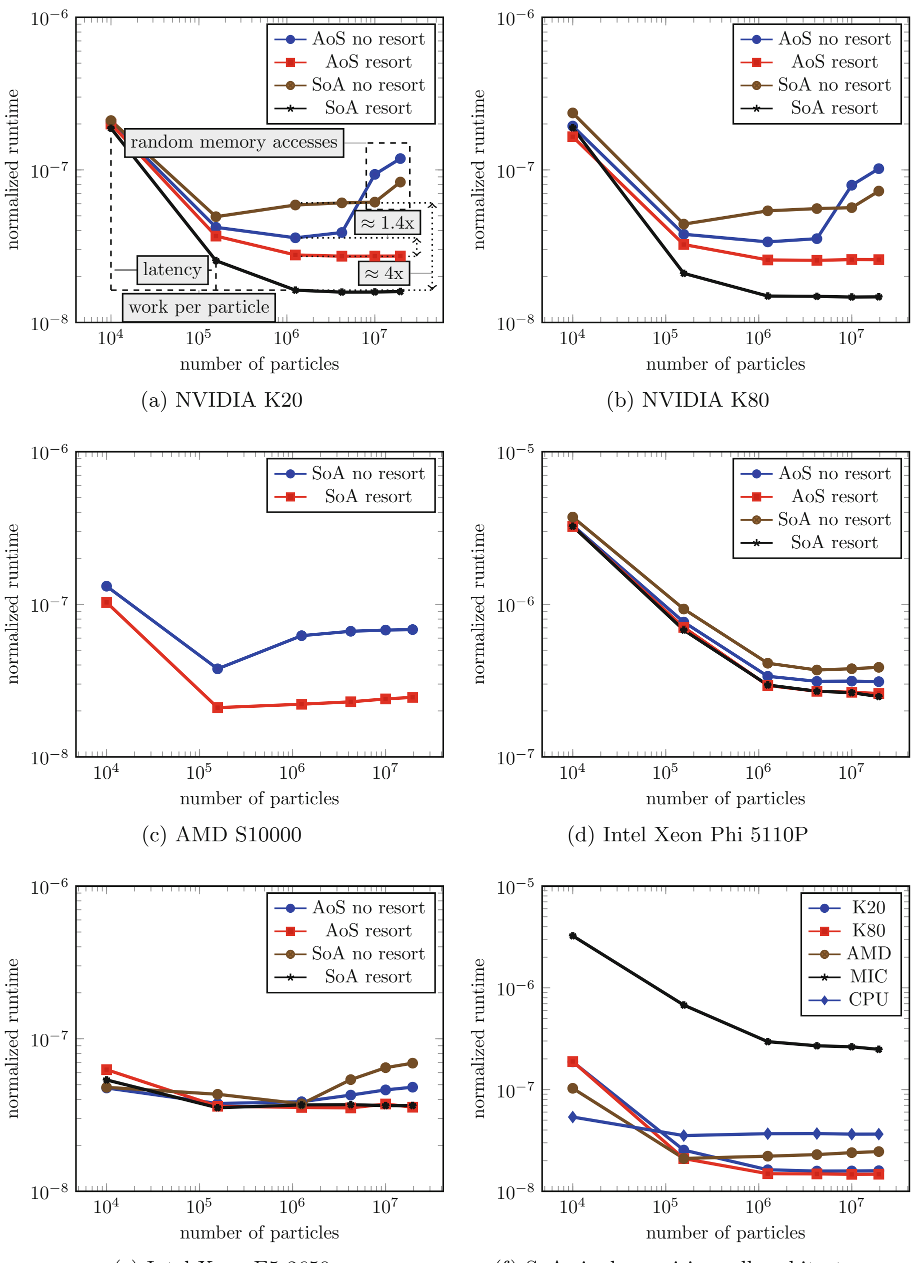
The benchmarks were conducted on three different machines: JURECA [2], JUROPA3 [3] (two of the supercomputer systems at JSC) as well as on a testing system for GPUs. Table 2 shows the specifications for the different architectures used for benchmarking. JUROPA3 has different partitions, that employ the same CPUs (Intel E5 2650), but contain different accelerators. For the scope of this paper the benchmarks were run on the GPU and the Intel Xeon Phi partitions. Due to the availability of the Intel OpenCL driver on the Intel Xeon Phi partition, the CPU comparison was conducted on JUROPA3 instead of running them on the faster E5-2680 Haswell processors on JURECA, where no OpenCL support is available for the CPU. Since the AMD GPU has less memory available than its NVIDIA counterparts, larger benchmarks could not be performed on the card. For all the tests the code was compiled with the GNU compiler version 4.9.3., with operating system CentOS 7.

Table 2. Specifications of the systems used for the different architectures: (sp) single precision (dp) double precision

Architecture	Relevant components	Peak performance
GPU (K20)	NVIDIA K20X	3.95 TF (sp), 1.31 TF (dp)
GPU (K80)	NVIDIA K80	5.6 TF (sp), 1.87 TF (dp)
GPU (S10000)	AMD S10000	5.91 TF (sp), 1.48 TF (dp)
Xeon Phi (5110P)	Intel Xeon Phi 5110P	1.011 TF (sp)
CPU (SandyBridge)	Intel Xeon E5 2650	128 GF

5 Results

To compare the performance on the different architectures, several benchmark runs were conducted. The most basic one was the comparison of the normalised runtime of the interaction kernel on each architecture (see Fig. 2a–f).



(a) NVIDIA K20

(b) NVIDIA K80

(c) AMD S10000

(d) Intel Xeon Phi 5110P

(e) Intel Xeon E5-2650

(f) SoA single precision, all architectures

Fig. 2. Runtime comparison on all architectures, using single-precision calculations. Note the shift in scale in (d) and (f) in order to show the full range.

Here, normalisation is defined as the measured runtime divided by the number of particles and the number of timesteps. No data transfer times were included, since the data is kept resident in the device memory for the complete benchmark and no additional data transfer is required. For the case of resorting the time required to resort the data is included into the presented times, i.e. the runtime is the sum of the time spent in the interaction kernel and the time spent to resort the data. All results were obtained with single precision calculations. For the AMD GPU (Fig. 2c) the AoS variant of the benchmark failed to execute for unknown reasons and therefore only the SoA results are presented. When comparing the benchmark results, it can be seen that the different architectures show a specific behavior; since the K20X (Fig. 2a) and the K80 GPUs (Fig. 2b) are different versions of the same production line, their results look fairly similar. With the exception of the Xeon E5-2650 CPU (Fig. 2e) all benchmarks suggest an architecture-dependent minimum problem size that must be reached before a stable performance is achieved. I.e. for small system sizes the specific runtime is exceedingly large compared with large system sizes. Exemplary this is shown in Fig. 2a and is due to the latency of high frequency accesses to small chunks of memory. Since transfer times are excluded in the results, an additional contribution to this behaviour is expected to result from a device-dependent overhead induced by the scheduling of the work-groups on the device. On the CPU this behaviour can only be observed to a smaller extent than on the other devices. For nearly all architectures, except the Xeon Phi (Fig. 2d) it can be observed that in the case of non-sorted data the runtime for larger problem sizes increases again. This can be understood by the fact that non-sorted data are scattered over memory and will have an unfavourable access pattern compared to the case of sorted data, where all particles belonging to a single cell are stored consecutively in memory. A possible explanation, why this behaviour cannot be observed on the Xeon Phi is the much lower overall performance of the Xeon Phi which hides the data access time behind a large computation time. For all other architectures the compute time is already so small that data access time is a crucial measure for the overall performance.

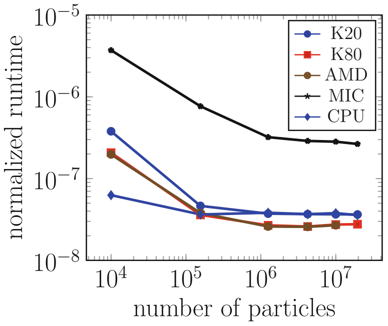


Fig. 3. All architecture comparison (double precision)

The choice of the data structure has a strong impact on the GPU performance and to a smaller extent also on the Xeon and Xeon Phi architectures. Since for the calculation of the forces within the interaction kernel only parts of the particle data is required (position and forces), it is very inefficient to store the whole set of data (velocities, masses, indices) within a single structure. In this case all the particle data within the complete structure would be loaded into the cache, filling it with nonessential data, i.e. leading to unnecessary data transfer and inefficient cache usage. However, if data is stored in individual arrays, data of consecutive particles is loaded into cache, and can be reused more efficiently. The difference between GPU and CPU performance comes into play when considering the size of data loaded into the registers. CPUs have a smaller capacity of data size loaded into registers, i.e. the load-operations need to be performed with a higher frequency than on a GPU. Therefore, the ratio between performance and load-operations is less favourable on a CPU and data layout patterns have less impact on the overall performance. On the other hand a GPU can operate most efficiently on large streams of data which can be consecutively processed. If the overall number of data loading operations is increased due to nonessential items in the data structures, performance degradation becomes more severe. This might explain the differences in the performance between GPU architectures (NVIDIA GPUs in Fig. 2a, b) and CPUs (Fig. 2e).

One peculiar detail that was observed on the AMD S10000 is the low single-precision performance compared to the NVIDIA cards. From the specification of the peak performance characteristics (cmp. Table 2), the AMD card should perform on the same level as the K80. However, the memory bandwidth of the S10000 is lower when compared to the NVIDIA GPUs leading to a reduced overall performance of the single-precision benchmark. For the case of double precision calculations, the frequency of load operations gets lower and therefore the effect of memory bandwidth limitations gets less pronounced (cmp. Fig. 3). Therefore, measuring double-precision performance on the AMD GPU shows comparable results to the NVIDIA K80, as could be expected.

Overall the benchmark shows a better performance for GPUs in comparison to the Xeon E5-2650 when test systems have a sufficient size, i.e. beyond showing memory latencies (Fig. 2f). Only for double-precision calculations it is observed that the K20X is only as fast as the CPU (cmp. Fig. 3). We only note here that the performance on the Xeon Phi is lacking behind all other architectures, since it is roughly a magnitude slower than other machines (Fig. 2f). A main reason for this behaviour is the missing vectorisation optimisation of the code, which was out of focus for this paper. Furthermore, the OpenCL drivers might lack best optimisation for the Intel Xeon Phi KNC, since the OpenCL support for KNC is deprecated [1].

6 Conclusion

One of the main goals of the present investigation was to investigate the performance characteristics of a function portable cell based MD program on various architectures. OpenCL has been selected as a programming language which allows for interoperability on different types of architectures, e.g. CPU and GPU based systems. Without any changes of the code it was possible to execute the code on several multi- and many-core systems available at JSC. As a downside of that approach the benchmark was not optimised for either of the architectures and therefore did not present optimal performance achievable. This especially accounts for the Xeon Phi system, where vectorisation is an essential issue to outperform a simple porting of a given program in comparison to running (even not optimised) on other architectures. On the other side this approach offers the possibility to have a comparison between basic features on the different architectures, e.g. memory bandwidth and core speed. To account for different data access patterns, we included the comparison of AoS and SoA, which is an important issue for the GPU architectures. It could be shown, that the SoA layout even improves the performance on the CPU for the non-sorted case and does not degrade performance in the sorted case. In this respect it can be concluded that further optimisation for GPU architectures will lead most probable to a significant performance gain compared to CPUs. Since the OpenCL support for Xeon Phi is already deprecated, it remains to be seen, if OpenCL will remain to be a valid option supporting newer Intel Xeon Phi architectures, e.g., KNL. Nevertheless, the choice of OpenCL provides the option to design MD simulation packages that can run on a variety of different architectures, without the need to provide specialised kernels or programs for each individual machine. In the present article we focused the work on function portability. At least as important as this is the request for performance portability, which we excluded from the present investigation, but which is of high importance in view of the developments towards more complex and heterogeneous architectures. International consortia are considering this aspect and it has to be awaited whether this leads to simplified porting and improved accessibility of future architectures (for a collection of contributions, see e.g. [4]).

References

1. Intel OpenCL SDK. https://software.intel.com/en-us/articles/opencl-drivers
2. JURECA. http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/JURECA_node.html
3. JUROPA3. http://www.fz-juelich.de/ias/jsc/EN/Research/HPCTechnology/ClusterComputing/JUROPA-3/JUROPA-3_node.html
4. Performance Portability WS DOE. https://asc.llnl.gov/DOE-COE-Mtg-2016/
5. Frenkel, D., Smit, B.: Understanding Molecular Simulation: From Algorithms to Applications. Academic Press, San Diego (2002)
6. Halver, R., Sutmann, G.: Multi-threaded construction of neighbour lists for particle systems in OpenMP. In: 11th International Conference on Parallel Processing and Applied Mathematics, Krakow, Poland, 6–9 Sep 2015 (2015). http://juser.fz-juelich.de/record/279249
7. Hockney, R.W., Eastwood, J.W.: Computer Simulation Using Particles. McGraw-Hill, New York (1981)
8. Rapaport, D.: The Art of Molecular Dynamics Simulation. Cambridge University Press, Cambridge (2001)
9. Sutmann, G.: Classical molecular dynamics. In: Grotendorst, J., Marx, D., Muramatsu, A. (eds.) Quantum Simulations of Many-Body Systems: From Theory to Algorithms, vol. 10, pp. 211–254. John von Neumann Institute for Computing, Jülich (2002)
10. Sutmann, G., Stegailov, V.: Optimization of neighbor list techniques in liquid matter simulations. J. Mol. Liq. **125**, 197–203 (2006)

Efficient Language-Based Parallelization of Computational Problems Using Cilk Plus

Przemysław Stpiczyński(✉)

Institute of Mathematics, Maria Curie–Skłodowska University,
Pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin, Poland
przem@hektor.umcs.lublin.pl

Abstract. The aim of this paper is to evaluate Cilk Plus as a language-based tool for simple and efficient parallelization of recursively defined computational problems and other problems that need both task and data parallelization techniques. We show that existing source codes can be easily transformed to programs that can utilize multiple cores and additionally offload some computations to coprocessors like Intel Xeon Phi. We also advise how to improve simplicity and performance of data parallel algorithms by tuning data structures to utilize vector extensions of modern processors. Numerical experiments show that in most cases our Cilk Plus versions of *Adaptive Simpson's Integration* and *Belman-Ford Algorithm* for solving single-source shortest-path problems achieve better performance than corresponding OpenMP programs.

Keywords: Cilk Plus · Multicore · Xeon Phi · Vectorization
Offload · Recursive algorithms · Shortest-path problems

1 Introduction

Recently, multicore and manycore computer architectures have become very attractive for achieving high performance execution of scientific applications at relatively low costs [8,17,20]. Modern CPUs and accelerators achieve performance that was recently reached by supercomputers. Unfortunately, the process of adapting existing software to such new architectures can be difficult if we expect to achieve reasonable performance without putting much effort into software development. For example, the use of OpenCL [10] leads to a substantial increase of software complexity. However, sometimes the use of high-level language-based programming interfaces devoted to parallel programming can get satisfactory results with rather little effort [19].

Software development process for modern Intel multicore CPUs and manycore coprocessors like Xeon Phi [8,17] requires special optimization techniques to obtain codes that would utilize the power of underlying hardware. Usually it is not sufficient to parallelize applications because in case of such computer architectures efficient vectorization is crucial for achieving satisfactory performance

[8,20]. Unfortunately, very often compiler-based automatic vectorization is not possible because of some non-obvious data dependencies inside loops [1,21]. On the other hand, people expect parallel programming to be easy and they prefer to concentrate on algorithms and use simple and powerful programming constructs that can utilize underlying hardware.

Cilk Plus introduces new extensions to C/C++ programming languages to express task and data parallelism using high-level constructs [8,9,12,18]. Although Cilk Plus has more usability than OpenMP [6], it is not very popular (several interesting applications can be found in [2,3,13,15]).

In this paper we show that Cilk Plus can be very easily applied to parallelize recursively defined adaptive Simpson's integration rule [11] and such implementation can be easily transformed to utilize coprocessors like Intel Xeon Phi. We also advise how to simplify move from OpenMP to Cilk Plus and improve the performance of such algorithms by tuning data structures to utilize hardware (i.e. vector units) of modern multicore and manycore processors. As an example we consider our Cilk Plus implementation of Belman-Ford algorithm for solving the single-source shortest-path problem [7] which achieves better performance than the corresponding simple OpenMP version of the algorithm. These two computational problems have been chosen to demonstrate the most important features of Cilk Plus that can be easily added to sequential C/C++ programs.

2 Short Overview of Cilk Plus

Cilk Plus offers several powerful extensions to C/C++ that allow to express both task and data parallelism [8,17]. The most important constructs are useful to specify and handle possible parallel execution of tasks:

cilk_for followed by the body of a `for` loop tells that iterations of the loop can be executed in parallel. Runtime applies the *divide-and-conquer* approach to schedule tasks among active workers to ensure balanced workload of available cores.

cilk_spawn permits a given function to be executed asynchronously with the rest of the calling function.

cilk_sync tells that all tasks spawned in a function must complete before execution continues.

Another important feature of Cilk Plus is the array notation which introduces vectorized operations on arrays. Expression `A[start:len:stride]` represents an array section of length `len` starting from `A[start]` with the given `stride`. Omitted `stride` means 1. The operator `[:]` can be used on both static and dynamic arrays. There are also several built-in functions to perform basic computations among elements in an array such as sum, min, max etc. It should be noticed that the array notation can also be used for array indices. For example, `A[x[0:len]]` denotes elements of the array `A` given by indices from `x[0:len]`.

Intel Cilk Plus also supports *Shared Virtual Memory* which allows to share data between the CPU and the coprocessor what is promising especially for complex data structures [8,17]. Such shared variables are declared using `_Cilk_shared` keyword. It also allows to declare functions that should be available for CPU and coprocessors. Computations can be offloaded to coprocessors for asynchronous execution using `_Cilk_spawn _Cilk_offload` construct. In such a case all necessary data are moved to the coprocessor. Memory synchronization between the CPU and the coprocessor takes place when an offloaded function is called by CPU or an offloaded function returns (i.e. when `cilk_sync` is used). The description of other features of Cilk Plus (like reducers) can be found in [17].

3 Two Examples of Computational Problems

Now we will present two exemplary problems which can be easily parallelized and optimized using Cilk Plus. All implementations have been tested on a server with two Intel Xeon E5-2670 v3 (totally 24 cores with hyperthreading, 2.3 GHz), 128 GB RAM, with Intel Xeon Phi Coprocessor 7120P (61 cores with multithreading, 1.238 GHz, 16 GB RAM), running under CentOS 6.5 with Intel Parallel Studio ver. 2017, C/C++ compiler supporting Cilk Plus. Experiments on Xeon Phi have been carried out using its native and offload modes.

3.1 Adaptive Simpson's Integration Rule

Let us consider the following recursive method for numerical integration called *Adaptive Simpson's Rule* [11]. We want to find the approximation of

$$I(f) = \int_a^b f(x)dx \tag{1}$$

with a user-specified tolerance ϵ. Let $S(a,b) = \frac{h}{6}(f(a) + 4f(c) + f(b))$, where $h = b - a$ and c is a midpoint of the interval $[a,b]$. The method uses Simpson's rule to the halves of the interval in recursive manner until the following stopping criterion is reached [14]:

$$\frac{1}{15}|S(a,c) + S(c,b) - S(a,b)| < \epsilon. \tag{2}$$

Figure 1 shows our parallel version of the straightforward recursive implementation of the method [4]. Note that we have only included keywords `_Cilk_spawn` and `_Cilk_sync`. The first one specifies that `cilkAdaptiveSimpsonsAux()` can execute in parallel with the remainder of the calling kernel. `_Cilk_sync` tells that all spawned calls in the current call of the kernel must complete before execution continues. For comparative purposes we have also implemented the method using OpenMP tasks [16], where the keywords `_Cilk_spawn` and `_Cilk_sync` are simply replaced with `task` and `taskwait` constructs.

Another Cilk implementation of the method assumes that some computations can be offloaded to a coprocessor (i.e. Xeon Phi, if available). The auxiliary kernel `cilkAdaptiveSimpsonsAux()` should be declared with the keyword

```
double cilkAdaptiveSimpsonsAux(double (*f)(double),double a,double b,
                double eps,double S,double fa,double fb,double fc,int depth)
{
  double c = (a + b)/2, h = b - a;
  double d = (a + c)/2, e = (c + b)/2;
  double fd = f(d), fe = f(e);
  double Sleft = (h/12)*(fa + 4*fd + fc);
  double Sright = (h/12)*(fc + 4*fe + fb);
  double S2 = Sleft + Sright;
  if (depth <= 0 || fabs(S2 - S) <= 15*eps)
    return S2 + (S2 - S)/15;
  double din1 =
    _Cilk_spawn adaptiveSimpsonsAux(f,a,c,eps/2,Sleft,fa,fc,fd,depth-1);
  double din2 = adaptiveSimpsonsAux(f,c,b,eps/2,Sright,fc,fb,fe,depth-1);
  _Cilk_sync;
  return  din1+din2;
}
double cilkAdaptiveSimpsons(double (*f)(double),double a,double b,double eps,
                                                                  int depth)
{ double c = (a + b)/2, h = b - a;
  double fa = f(a), fb = f(b), fc = f(c);
  double S = (h/6)*(fa + 4*fc + fb);
  return adaptiveSimpsonsAux(f,a,b,eps,S,fa,fb,fc,depth);
}
```

Fig. 1. Parallel version of Adaptive Simpson's method

`_Cilk_shared`, what makes the function available for CPU and coprocessors. In the main kernel `cilkAdaptiveSimpsonsOff()`, the integration over the first half of the interval $[a, b]$ can offloaded to Xeon Phi using `_Cilk_spawn _Cilk_offload` construct, while the rest is to be done by CPU.

Table 1 shows the execution time of our three parallel implementations applied for finding the approximation of $\int_{-4.4}^{4.4} \exp(x^2)dx$ with $\epsilon = 1.0e - 7$ and $depth = 40$ (namely OpenMP with tasks, Cilk, and Cilk with offload). We can observe that `cilkAdaptiveSimpsons()` outperforms `ompAdaptiveSimpsons()` significantly (about four times faster for CPU and three times for Xeon Phi). It should be noticed that the execution time (seconds) of the sequential version of the method is 62.8 for CPU and 638.04 for Xeon Phi. Thus, the speedup achieved by our Cilk implementation is 14.35 (CPU) and 70.66 (Xeon Phi), respectively.

Our non-offloaded Cilk version scales very well when the number of Cilk workers increases up to 24 for CPU and 60 for Xeon Phi, respectively, i.e. to the number of physical cores. The further increase in the number of workers results in smaller and rather marginal gains. For `cilkAdaptiveSimpsonsOff()`, we can observe that the shortest execution time is achieved for twelve workers. Then the execution time of `cilkAdaptiveSimpsonsAux()` on CPU and Xeon Phi working on the halves of the interval is approximately the same.

3.2 Bellman-Ford Algorithm for the Single-Source Shortest-Path Problem

Let $G = (V, E)$ be a directed graph with n vertices labeled from 0 to $n - 1$ and m arcs $\langle u, v\rangle \in E$, where $u, v \in V$. Each arc has its weight $w(u, v) \in \mathbf{R}$

Table 1. Execution time (s) of `ompAdaptiveSimpsons()`, `cilkAdaptiveSimpsons()` and `cilkAdaptiveSimpsonsOff()` for $\int_{-4.4}^{4.4} \exp(x^2)dx$

2x E5-2670 and Xeon Phi 7120P (coprocessor)						
number of threads/workers (CPU)	2	4	6	12	24	48
`ompAdaptiveSimpsons()`	202.71	101.43	68.03	34.36	17.43	15.45
`cilkAdaptiveSimpsons()`	61.99	31.06	20.64	10.57	5.39	4.32
`cilkAdaptiveSimpsonsOff()`	34.28	17.06	11.33	5.67	5.78	5.95
Xeon Phi 7120P (native mode)						
number of threads/workers	2	30	60	120	180	240
`ompAdaptiveSimpsons()`	1355.67	92.60	45.57	31.13	29.22	28.52
`cilkAdaptiveSimpsons()`	478.11	32.44	16.71	10.51	9.33	9.03

and we assume $w(u, v) = \infty$ when $\langle u, v\rangle \notin E$. For each path $\langle v_0, v_1, \ldots, v_p\rangle$ we define its length as $\sum_{i=1}^{p} w(v_{i-1}, v_i)$. We also assume that G does not contain negative cycles. Let $d(s, t)$ denotes the length of the shortest path from s to t or $d(s, t) = \infty$ if there are no paths from s to t.

Algorithm 1 is the well-known *Belman-Ford* method for finding shortest lengths of paths from a given source $s \in V$ to all other vertices [7].

Algorithm 1. Bellman-Ford Algorithm

```
   Data: $G = (V, E)$, $|V| = n$, $s \in V$, $w(u, v)$ for all $u, v \in V$
   Result: $D[v] = d(s, v)$ for all $v \in V$
1  for $v \in V$ do
2      $D[v] \leftarrow w(s, v)$
3  end
4  $D[s] \leftarrow 0$
5  for $k = 1, \ldots, n-2$ do
6      for $v \in V \setminus \{s\}$ do
7          for $u \in V$ such that $\langle u, v\rangle \in E$ do
8              $D[v] \leftarrow \min(D[v], D[u] + w(u, v))$
9          end
10     end
11 end
```

The most common basic implementations of the algorithm assume that a graph is represented as an array that describes its vertices. Each vertex is described by an array containing information about incoming arcs. Each arc is represented by the initial vertex and arc's weight. It is also necessary to store the length of arrays describing vertices. In order to parallelize such a basic implementation using OpenMP (see Fig. 2, left), we should notice that the entire algorithm should be within the `parallel` construct. Then the loops 7–13 and 18–26 can be parallelized using `for` construct with clause `schedule(dynamic,ChS)`. Thus, iterations are divided into pieces having a size specified by chunk size `ChS` and such pieces are dynamically dispatched to threads. The assignment in line 4 needs to be a single task (i.e. defined by `single`). Moreover, we need two copies of the array `D` for storing current and previous updates within each iteration of the loop 20–25. It should be noticed that this loop is automatically vectorized by the compiler. For the sake of simplicity, we also assume that the vertex labeled as 0 is the source.

```
 1  void
 2  ompBF1(DGraph & g,float *d1,float *d2)
 3  {
 4   float dist;
 5   #pragma omp parallel
 6   {#pragma omp for schedule(dynamic,ChS)
 7    for(int i=1;i<g.n;i++)
 8     {dist=INFTY;
 9      if((g.node[i].degIn>0)&&
10                 (g.node[i].in[0].v==0))
11         dist=g.node[i].in[0].weight;
12      d1[i]=dist;
13     }
14    #pragma omp single
15    { d1[0]=0; }
16    for(int k=1;k<g.n-1;k++)
17     {#pragma omp for schedule(dynamic,ChS)
18      for(int i=1;i<g.n;i++)
19       {d2[i]=d1[i];
20        for(int j=0;j<g.node[i].degIn;j++)
21        {
22         int u=g.node[i].in[j].v;
23         dist=g.node[i].in[j].weight;
24         d2[i]=std::min(d2[i],d1[u]+dist);
25        }
26       }
27      float *temp=d1; d1=d2; d2=temp;
28     }
29   }
30  }
```

```
 1  void
 2  cilkBF(DGraph & g,float *d1,float *d2)
 3  {
 4   float dist;
 5
 6
 7   cilk_for(int i=1;i<g.n;i++)
 8   {dist=INFTY;
 9     if((g.node[i].degIn>0)&&
10            (g.node[i].inv[0]==0)
11         dist=g.node[i].inw[0];
12     d1[i]=dist;
13   }
14
15   d1[0]=0;
16   for(int k=1;k<g.n-1;k++)
17    {
18     cilk_for(int i=1;i<g.n;i++)
19      {
20       int deg=g.node[i].degIn;
21       dist=__sec_reduce_min
22           (d1[g.node[i].inv[0:deg]]+
23                 g.node[i].inw[0:deg]);
24       d2[i]=std::min(d1[i],dist);
25
26      }
27     float *temp=d1; d1=d2; d2=temp;
28    }
29
30  }
```

Fig. 2. Belman-Ford algorithm implemented using OpenMP and Cilk Plus

In our Cilk Plus implementation (see Fig. 2, right), the loops 7–13 and 18–26 are parallelized using `cilk_for` construct. We also assume that each vertex of a given graph is represented by two arrays of the same size. The first one (i.e. `inv`) sorted in increasing order contains labels of initial vertices of incoming arcs. The next one (i.e. `inw`) stores weights of corresponding arcs. Then (lines 21–24) we can simply vectorize the body of the loop using built-in function `__sec_reduce_min()` to find minimum among elements in the array given by the sum of the array `inw` and necessary elements from the array `d1` given by indices from `inv`. This is a very fine example of using the array notation.

Table 2 shows the results of experiments performed for the considered implementation of Belman-Ford algorithm, namely **basic**, **ompBF1**, **ompBF2** and **cilkBF**. Note that **ompBF2** is another implementation that uses OpenMP and

Table 2. Execution time (in seconds) of three implementations of Algorithm 1

	2x E5-2670				Xeon Phi 7120P			
max deg	basic	ompBF1	ompBF2	cilkBF	basic	ompBF1	ompBF2	cilkBF
	The number of nodes $n = 4000$							
10	0.20	0.16	0.16	0.19	4.73	0.48	0.43	0.99
20	0.30	0.21	0.16	0.20	6.24	0.84	0.87	1.03
50	0.57	0.25	0.19	0.25	10.04	1.76	1.63	1.38
100	1.05	0.25	0.26	0.35	15.29	2.34	2.05	1.50
200	2.07	0.38	0.31	0.32	27.37	2.26	2.08	1.94
500	5.13	0.67	0.35	0.42	54.97	1.75	1.93	1.70
1000	10.34	0.71	0.92	0.59	100.37	2.60	2.87	2.40
2000	22.22	1.51	1.49	1.07	200.16	5.68	6.22	4.16
	The number of nodes $n = 10000$							
10	1.70	0.75	0.83	0.52	31.89	1.02	1.10	3.07
20	2.20	0.80	0.74	0.55	39.66	1.16	1.23	3.26
50	3.74	0.82	0.79	0.71	61.50	1.48	1.54	3.60
100	6.72	0.98	0.96	0.78	97.62	2.31	2.41	4.84
200	13.09	1.22	1.35	1.08	168.09	3.72	3.60	4.98
500	32.99	2.53	2.38	1.61	369.67	7.68	8.90	9.34
1000	71.88	3.90	4.87	4.01	684.38	14.49	16.01	10.85
2000	156.82	12.65	12.68	11.63	1331.26	27.82	30.62	18.25

the same data layout as **cilkBF**. All results have been obtained for graphs generated randomly for a given number of vertices and maximum degree (i.e. the maximum number of incoming arcs). We can observe that the parallel implementations are much faster than the basic (i.e. non-parallelized) implementation of Algorithm 1. Usually **ompBF2** is faster than **ompBF1**. **cilkBF** outperforms **ompBF1** and **ompBF2** for larger and wider graphs. However, in case of our OpenMP implementations, Table 2 shows the best results chosen from several runs for various values of `ChS`. Thus, one can say that our OpenMP versions have been manually tuned. In case of **cilkBF**, the runtime system has been responsible for load balancing.

We can observe that for sufficiently large graphs all parallel implementations utilize multiple cores achieving reasonable speedup (see Fig. 3). Moreover, **cilkBF** outperforms **ompBF** significantly, especially on Xeon Phi. This is the effect of the efficient and explicit vectorization of the loop 7–9 in Algorithm 1. For this architecture it is also important to vectorize sufficiently long loops. Indeed, the speedup grows when the maximum degree (i.e. the length of the arrays `inv` and `inw`) grows.

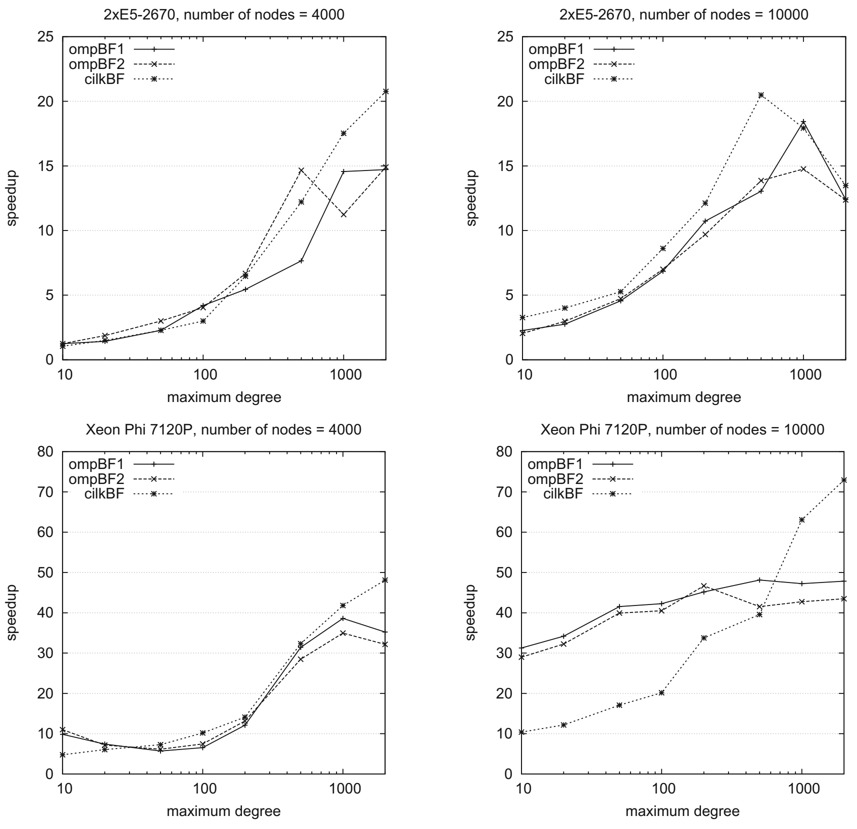


Fig. 3. Speedup of OpenMP and Cilk Plus implementations versus non-parallelized basic version of Belman-Ford algorithm

It should be noticed that we have also tested another version of **cilkBF** that uses `_Cilk_spawn _Cilk_offload` construct and where all data structures have been shared between CPU and coprocessors. Unfortunately, the need for synchronization of *Shared Virtual Memory* at the end of each iteration (i.e. the loop 16–28) leads to a very large increase in processing time and our implementation with offloading is over 10× slower than **cilkBF**. However, *Shared Virtual Memory* is perfect for exchanging irregular data with limited size, when explicit synchronization is not used frequently. Both sides (CPU and coprocessor) should operate on memory allocated locally. Local data can be persisted using more sophisticated techniques (the use of Cilk Plus together with `#pragma offload`).

4 Conclusions and Future Work

We have shown that Cilk Plus can be very easily applied to parallelize recursively defined problems like *Adaptive Simpson's Integration Rule* and such implementation can be easily modified to utilize coprocessors like Intel Xeon Phi. It is also easy to move from OpenMP to Cilk Plus and improve the performance of such algorithms by tuning data structures to utilize hardware (i.e. vector units) of modern multicore and manycore processors. For sufficiently large graphs, our Cilk implementation of Belman-Ford algorithm for solving the single-source shortest-path problem achieves really better performance than corresponding OpenMP versions of the algorithm. Thus, Cilk Plus is a good choice for people who want to concentrate on algorithms and prefer to use simple high-level programming constructs to express parallelism. Of course, it is clear that the use of OpenMP together with more advanced programming tools allows to fine-tune programs for a particular architecture [17,20]. However, this involves a much greater effort.

In the future, we plan to implement some other important computational problems using Cilk Plus. It would also be interesting and important to find problems that can benefit from using *Shared Virtual Memory*.

Acknowledgements. This work was partially supported by the National Centre for Research and Development under MICLAB Project POIG.02.03.00-24-093/13. The use of computer resources installed at Institute of Mathematics, Maria Curie-Skłodowska University in Lublin is kindly acknowledged.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann, Burlington (2001)
2. Asai, R., Vladimirov, A.: Intel Cilk Plus for complex parallel algorithms: "enormous fast Fourier transforms" (EFFT) library. Parallel Comput. **48**, 125–142 (2015). https://doi.org/10.1016/j.parco.2015.05.004
3. Basseda, R., Chowdhury, R.A.: A parallel bottom-up resolution algorithm using Cilk. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, 4–6 November 2013, pp. 95–100. IEEE Computer Society (2013). https://doi.org/10.1109/ICTAI.2013.24
4. Cameron, M.: Adaptive integration (2010). http://www2.math.umd.edu/~mariakc/teaching/adaptive.pdf
5. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers, San Francisco (2001)
6. Coblenz, M.J., Seacord, R., Myers, B.A., Sunshine, J., Aldrich, J.: A course-based usability analysis of Cilk Plus and OpenMP. In: 2015 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2015, Atlanta, GA, USA, 18–22 October 2015, pp. 245–249. IEEE (2015). https://doi.org/10.1109/VLHCC.2015.7357223
7. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. MIT Press, Cambridge (1994)

8. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High-Performance Programming. Morgan Kaufman, Waltham (2013)
9. Khaldi, D., Jouvelot, P., Ancourt, C., Irigoin, F.: Task parallelism and data distribution: an overview of explicit parallel programming languages. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 174–189. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37658-0_12
10. Kowalik, J.S., Puzniakowski, T.: Using OpenCL - Programming Massively Parallel Computers, Advances in Parallel Computing, vol. 21. IOS Press, Amsterdam (2012). http://ebooks.iospress.nl/volume/using-opencl
11. Kuncir, G.F.: Algorithm 103: Simpson's rule integrator. Commun. ACM **5**(6), 347 (1962). https://doi.org/10.1145/367766.368179
12. Leiserson, C.E.: Cilk. In: Padua, D.A. (ed.) Encyclopedia of Parallel Computing, pp. 273–288. Springer, Boston (2011). https://doi.org/10.1007/978-0-387-09766-4_2339
13. Lewin-Berlin, S.: Exploiting multicore systems with Cilk. In: Proceedings of the 4th International Workshop on Parallel Symbolic Computation, PASCO 2010, 21–23 July 2010, Grenoble, France, pp. 18–19. ACM (2010). https://doi.org/10.1145/1837210.1837214
14. Lyness, J.N.: Notes on the adaptive Simpson quadrature routine. J. ACM **16**(3), 483–495 (1969). https://doi.org/10.1145/321526.321537
15. Musaev, M., Khujayarov, I., Buriboev, A.: Accelerate the solution of problems of digital signal processing technology based INTEL CILK PLUS. Asian J. Comput. Inf. Syst. **3**, 48–51 (2015). https://doi.org/10.24203/ajcis.v3i2.2507
16. van der Pas, R., Stotzer, E., Terboven, C.: Using OpenMP - The Next Step. Affinity, Accelerators, Tasking, and SIMD. MIT Press, Cambridge (2017)
17. Rahman, R.: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers. Apress, Berkely (2013)
18. Robison, A.D.: Composable parallel patterns with Intel Cilk Plus. Comput. Sci. Eng. **15**(2), 66–71 (2013). https://doi.org/10.1109/MCSE.2013.21
19. Stpiczyński, P.: Semiautomatic acceleration of sparse matrix-vector product using OpenACC. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015 Part II. LNCS, vol. 9574, pp. 143–152. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_14
20. Supalov, A., Semin, A., Klemm, M., Dahnken, C.: Optimizing HPC Applications with Intel Cluster Tools. Apress, Berkely (2014)
21. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley, Boston (1996)

A Taxonomy of Task-Based Technologies for High-Performance Computing

Peter Thoman[1](✉), Khalid Hasanov[2], Kiril Dichev[3], Roman Iakymchuk[4], Xavier Aguilar[4], Philipp Gschwandtner[1], Pierre Lemarinier[2], Stefano Markidis[4], Herbert Jordan[1], Erwin Laure[4], Kostas Katrinis[2], Dimitrios S. Nikolopoulos[3], and Thomas Fahringer[1]

[1] University of Innsbruck, Innsbruck, Austria
{petert,philipp,herbert,tf}@dps.uibk.ac.at
[2] IBM Ireland, Dublin, Ireland
{khasanov,pierrele,katrinisk}@ie.ibm.com
[3] Queen's University of Belfast, Belfast, UK
{K.Dichev,D.Nikolopoulos}@qub.ac.uk
[4] KTH Royal Institute of Technology, Stockholm, Sweden
{riakymch,xaguilar,markidis,erwinl}@kth.se

Abstract. Task-based programming models for shared memory – such as Cilk Plus and OpenMP 3 – are well established and documented. However, with the increase in heterogeneous, many-core and parallel systems, a number of research-driven projects have developed more diversified task-based support, employing various programming and runtime features. Unfortunately, despite the fact that dozens of different task-based systems exist today and are actively used for parallel and high-performance computing, no comprehensive overview or classification of task-based technologies for HPC exists.

In this paper, we provide an initial task-focused taxonomy for HPC technologies, which covers both programming interfaces and runtime mechanisms. We demonstrate the usefulness of our taxonomy by classifying state-of-the-art task-based environments in use today.

Keywords: Task-based parallelism · Taxonomy · API
Runtime system · Scheduler · Monitoring framework · Fault tolerance

1 Introduction

A large number of task-based programming environments have been developed over the past decades, and the task-based parallelism paradigm has proven widely applicable for consumer applications. Conversely, in high-performance computing (HPC) loop-based and message-passing paradigms are still dominant. In this work, we specifically aim to categorize task-based parallelism technologies which are in use in HPC.

For the purpose of this work, we define a *task* as follows

A **task** is a sequence of instructions within a program that can be processed concurrently with other tasks in the same program. The interleaved execution of tasks may be constrained by control- and data-flow dependencies.

Many programming languages support task-based parallelism directly without external dependencies. Examples include the C++11 thread support library, Java via its Concurrency API, or Microsoft TPL for .NET. Except for C++, we do not study these languages in detail in this paper, since they are not common in the HPC domain.

The Cilk language[1] [19] allows task-focused parallel programming, and is an early example of efficient task scheduling via work stealing. OpenMP [4], which we consider a language extension, integrates tasks into its programming interface since version 3.0. Industry-standard and well-supported parallel libraries based on task parallelism have emerged, such as Intel Cilk Plus [24] or Intel TBB [25]. Task-based environments for heterogeneous hardware have also naturally developed with the emergence of accelerator and GPU computing; StarPU [8] is an example of such an environment.

In addition, task-based parallelism is increasingly employed on distributed memory systems, which constitute the most important target for HPC. In this context, tasks are often combined with a global address space (GAS) programming model, and scheduled across multiple processes, which together form the *distributed execution* of a single task-parallel program. While some examples of global address space environments with task-based parallelism are specifically designed languages such as Chapel [6] and X10 [18], it is also possible to implement these concepts as a library. For instance, HPX [10] is an asynchronous GAS runtime, and Charm++ [23] uses a global object space.

This already very diverse landscape is made even more complex by the recent appearance of task-based runtimes using novel concepts, such as the data-centric programming language Legion [1]. Many of these task-based programming environments are maintained by a dedicated community of developers, and are often research-oriented. As such, there might be relatively little accessible documentation of their features and inner workings.

Crucially, at this point, *there is no up to date and comprehensive taxonomy and classification of existing common task-based environments*. This makes it very difficult for researchers or developers with an interest in task-based HPC software development to get a concise picture of the alternatives to the omnipresent MPI programming model. In this work, we attempt to address this issue by providing a taxonomy and classification of both state-of-the-art task-based programming environments and more established alternatives. We consider a task-based environment as consisting of two major components: a *programming interface* (API) and a *runtime system*; the former is the interface that a given environment provides to the programmer, while the latter

[1] Note that we use the term "language" for Cilk and Cilk Plus, even though they build on C/C++. The reasoning is that a Cilk Plus compiler is strictly required for compilation (unlike e.g. OpenMP).

encompasses the underlying implementation mechanisms. We present a set of API characteristics allowing meaningful classification in Sect. 2. For discussing the more involved topic of runtime mechanisms, we further structure our analysis into the overarching topics of scheduling, performance monitoring, and fault handling (see Sect. 3). Finally, based on the taxonomy introduced, we classify and categorize existing APIs and runtimes in Sect. 4.

2 Task-Parallel Programming Interfaces (APIs)

The Application Programming Interface (API) of a given task-parallel programming environment defines the way an application developer describes parallelism, dependencies, and in many cases other more specific information such as the work mapping structure or data distribution options. As such, finding a way to concisely characterize APIs from a developer's perspective is crucial in providing an overview of task-parallel technologies.

In this work, we define a set of characterizing features for such APIs which encompasses all relevant aspects while remaining as compact as possible. A subset of these features was adapted from previous work by Kasim et al. [11]. To these existing characteristics we added additional information of general importance – such as technological readiness levels – as well as features which relate to new capabilities particularly relevant for modern HPC like support for heterogeneity and resilience management. We will now define each of these characteristics and their available options for categorization. *Explicit* (**e**) support refers to features which require extra effort or implementation by the developer, while *implicit* (**i**) support means that the toolchain manages the feature automatically.

Technology Readiness. The technology readiness of the given API and its implementations according to the European Commission definition.[2]

Distributed Memory. Whether targeting distributed memory systems is supported. Options are *no* support, *explicit* support, or *implicit* support. *explicit* refers to, for example, message passing between address spaces, while automatic data migration would be an example of *implicit* support.

Heterogeneity. Indicates whether tasks can be executed on accelerators (e.g. GPUs). *Explicit* support indicates that the application developer has to actively provision tasks to run on accelerators, using a distinct API.

Worker Management. Whether the worker threads and/or processes need to be started and maintained by the user (*explicit*) or are provided automatically by the environment (*implicit*).

Task Partitioning. This feature indicates whether each task is atomic – can, thus, only be scheduled as a single unit – or can be subdivided/split.

Work Mapping. Describes the way tasks are mapped to the existing hardware resources. Possibilities include *explicit* work mapping, *implicit* work mapping (e.g. stealing), or *pattern-based* work mapping.

[2] https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf. Accessed: 2017-05-03.

Synchronization. Whether tasks are synchronized in an *implicit* fashion, e.g. by regions or the function scope, or *explicitly* by the application developer.

Resilience Management. Describes whether the API has support for task resilience management, e.g. fine-grained checkpointing and restart.

Communication Model. Either *shared memory* (smem), *message passing* (msg), or *Partitioned Global Address Space* (pgas).

Result Handling. Whether the tasking model features *explicit* handling of the results of task computations – for example, return types accessed as *futures*.

Graph Structure. The type of task graph dependency structure supported by the given API: a *tree* structure, an *acyclic graph* (dag) or an *arbitrary graph*.

Task Cancellation. Whether the tasking model supports cancellation of tasks: *no* cancellation support; cancellation is supported either *cooperatively* (only at task scheduling points) or *preemptively*.

Implementation Type. How the API is implemented and addressed from a program. A tasking API can be provided either as a *library*, a *language extension*, e.g. pragmas, or an entire *language* with task integration.

3 Many-Task Runtime Systems

Many-task runtime systems serve as the basis for implementing these APIs, and are considered a promising tool in addressing key issues associated with Exascale computing. In this section we provide a taxonomy of many-task runtime systems, which is summarized and illustrated in Fig. 1.

A crucial difference among various many-task runtime systems is their **target architecture**. The evolution of many-task runtime systems started from *homogeneous shared-memory* computers with multiple cores and continued towards runtimes for *heterogeneous* shared-memory and *distributed-memory* systems. Support for distributed-memory systems varies significantly across different systems: in case of *implicit data distribution*, data distribution is handled by the runtime, without putting any burden on the application developer; on the other hand, in *explicit data distribution*, distribution across the nodes is explicitly specified by the programmer.

Modern HPC systems require efficiency not only in execution times, but also in power and/or energy consumption. Thus, whether the runtime provides **scheduling objectives** other than the total execution time is another important distinction. At the same time, there is not a single standard **scheduling methodology** that is being used by all many-task runtime systems. Some of them provide automatic scheduling within a single shared-memory machine while the application developer needs to handle distributed-memory execution explicitly, while others provide uniform scheduling policies across different nodes.

Many-task runtimes may require **performance introspection** and **monitoring** to facilitate the implementation of different scheduling policies. While traditionally it was not part of runtimes, requirements for on-the-fly performance information have surfaced. Thus, most task-based runtimes already provide and make use of introspection capabilities. **Fault tolerance** is another key factor that is important in many-task runtime systems in the context of Exascale requirements. As detailed in Sect. 3.3, a runtime may have no resilience capabilities, or it may target task faults or even process faults.

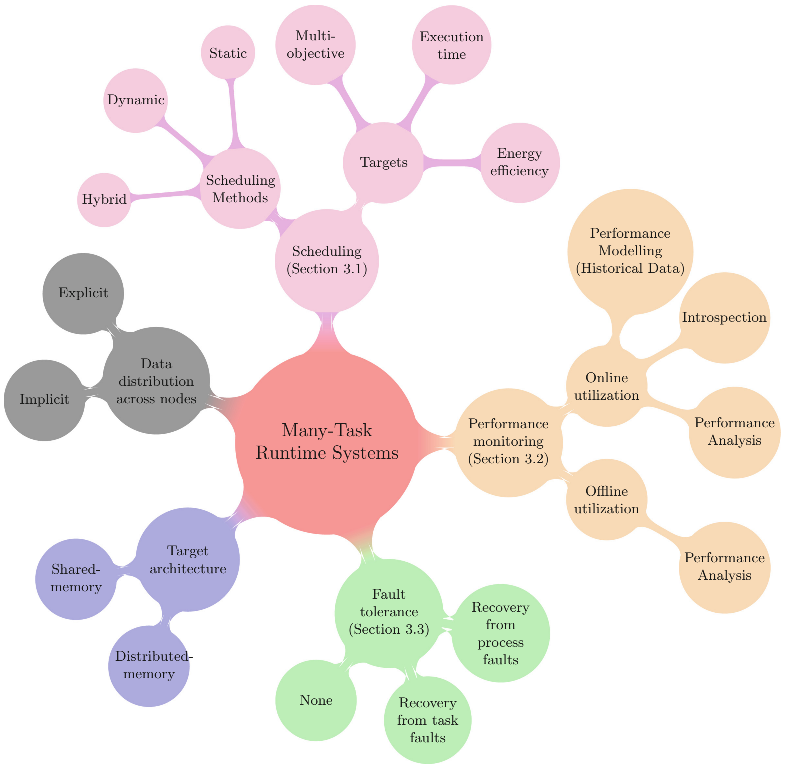


Fig. 1. Taxonomy of many-task runtime systems.

3.1 Scheduling in Many-Task Runtime Systems

Task Scheduling Targets. Depending on the capabilities of the underlying many-task runtime system, its scheduling domain is usually limited to a single shared-memory homogeneous compute node, a heterogeneous compute node with accelerators, homogeneous distributed-memory systems of interconnected compute nodes, or in a most generic form to heterogeneous distributed-memory systems. By supporting different types of heterogeneous architectures, the runtime can facilitate source code portability and support transparent interaction between different types of computation units for application developers.

Traditionally, *execution time* has been the main objective to minimize for different scheduling policies. However, the increasing scale of HPC systems makes it necessary to take the energy and power budgeting of the target system into account as well. Therefore, some many-task runtime systems have already started providing *energy-aware* [14] scheduling policies[3]. In addition, recent research projects, such as AllScale[4] focus on *multi-objective* scheduling policies trying to find optimal trade-offs among conflicting optimization objectives like execution time, energy consumption and/or resource utilization.

Task Scheduling Methods. Extensive research has been conducted in task scheduling methodologies. We do not try to list all different techniques for task scheduling, but rather highlight methods used in state-of-the-art many-task runtime systems. The task scheduling problem can be addressed either in a *static* or *dynamic* manner. In the former case, depending on the decision function it is assumed that either one or more of the following inputs are known in advance: the execution times of each task, inter-dependencies between tasks, task precedence, resource usage of each task, the location of the input data, task communications, and synchronization points. This is by no means an exhaustive list but it gives an indication of the multiple possible a priori inputs for static scheduling. Using all this information the scheduling can be performed offline during compilation time. On the other hand, dynamic scheduling is mainly used in the case where there is not enough information in advance or obtaining such information is not trivial. Additionally, *hybrid* policies which integrate static and dynamic information are possible.

Most static scheduling algorithms used in many-task runtime systems are based on the *list scheduling* methods. Here, it is assumed that the scheduling list of tasks is statically built before any task starts executing and the sequence of the tasks in the list is not modified. The list scheduling approach can easily be adapted and used for dynamic scheduling by re-computing and re-sequencing the list of tasks. As a matter of fact, heuristic policies based on list scheduling and performance models are employed in some many-task runtime systems [8].

Work-stealing. [2] can be considered as the most widely used dynamic scheduling method in task-based runtime systems. The main idea in work-stealing is to distribute tasks between per-processor work queues, where each processor operates on its local queue. The processors can steal tasks from other queues to perform *load-balancing*. There are two main approaches in implementing work-stealing, namely, *child-stealing* and *parent-stealing*. In parent-stealing, which is also called *work-first* policy, a worker executes a spawned task and leaves the continuation to be stolen by another worker. Child-stealing, which is also called *help-first* policy, does the opposite, namely, the worker executes the continuation and leaves the spawned task to be stolen by the other workers. Another approach to dynamic scheduling for many-task runtime systems is the **work-sharing** strategy. Unlike work-stealing, it schedules each task onto a processor when it is spawned and it is usually implemented by using a *centralized task pool*. In work-sharing, whenever a worker spawns a new task, the scheduler migrates it to a new worker to improve load balancing. As such, migration of tasks happens more often in work-sharing than that of in work-stealing.

[3] http://starpu.gforge.inria.fr/doc/html/Scheduling.html#Energy-basedScheduling.
[4] The AllScale EC-funded FET-HPC project: allscale.eu.

Few of the existing many-task runtime systems provide *energy efficient* scheduling policies. In the primitive case it is assumed that the application can provide an energy consumption model which can be used by a scheduling policy as part of its objective function. In more advanced cases, the runtime provides offline or online profiling data, such as, instructions per cycle (IPC) and last level cache misses (LLCM). This data is used to build a look-up table that maps each frequency setting with the triple of IPC, LLCM, and the number of active cores. Then, a scheduling decision based on this information [14].

3.2 Performance Monitoring

The high concurrency and dynamic behavior of upcoming Exascale systems poses a demand for performance observation and runtime introspection. This performance information is very valuable to guide HPC runtimes in their execution and resource adaption, thereby maximizing application performance and resource utilization.

When targeting performance observation, performance monitoring software is either generating data to be used *online* [1,7,13,15,16] or *offline* [1,5,8,15]. In other words, whether the collected data is going to be used while the application still runs or after its execution. Furthermore, this taxonomy can be extended with respect to who is consuming data – either the end user (*performance analysis*) or the runtime itself (*introspection* and *historical data*). Real-time performance data (*introspection* and performance models from *historical data*) will play an important role in Exascale for runtime adaptation and optimal task-scheduling.

3.3 Task, Process, and System Faults

For this topic, we extend a recent taxonomy [22] from the HPC domain to include the concept of task faults. We retain detectability of faults as the main criterion, but distinguish three levels of the system: distributed execution, process, and task (see Fig. 2). Each of these levels may experience a fault, and each of them has a different scope.

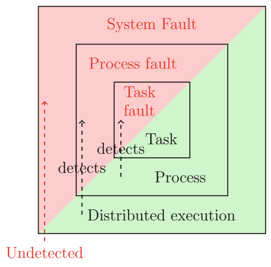


Fig. 2. A taxonomy of faults based on the detection capabilities: task faults, process faults, and system faults.

Task Faults: Tasks have the smallest scope of the three; still, a failure of a task may affect the result of a process, and subsequently of a distributed run. A typical example are undetected errors in memory. The process which runs a task is generally capable of detecting task faults. There are several examples of shared-memory runtimes, where task faults within parallel regions have been detected and corrected [17,20].

Process Faults: A process may also fail, which leads to the termination of all underlying tasks. For example, a node crash can lead to a process failure. In such a scenario, a process cannot detect its failure; however, in a distributed run, another process may detect the failure, and trigger a recovery strategy across all processes. A recovery strategy in this case may rely on one of two redundancy techniques: checkpoint/restart or replication.

System Faults: On the last level, a distributed system execution may fail in cases of severe faults like switch failure, or power outage. In this case, a failure cannot be detected. No recovery strategy can be applied in such scenarios.

4 Classification

Table 1 classifies the existing task-parallel APIs according to the API taxonomy (see Sect. 2), however with additional clarifications. First, for an API to support a given feature, this API must not require the user to resort to third party libraries or implementation-specific details of the API. For instance, some APIs offer arbitrary task graphs via manual task reference counting [21]. This does not qualify as support in our classification. Second, all APIs shown as featuring task cancellation do so in a non-preemtive manner due to the absence of OS-level preemption capabilities.

Some entries require additional clarification. In C++ STL, we consider the entity launched by `std::async` to represent a task. Also, while StarPU offers shared memory parallelism, it is capable of generating MPI communication from a given task graph and data distribution [8], hence it is marked with explicit support for distributed memory using a message-based communication model. Furthermore, PaRSEC includes both a task-based runtime that works on user-specified task graph and data distribution information, as well as a compiler that accepts serial input and generates this data. As the latter is limited to loops, we only consider the runtime in this work.

Several observations can be made from the data presented in Table 1. First, all APIs with distributed memory support also allow task partitioning and support heterogeneity in some form. APIs offering implicit distributed memory support employ a global address space. Second, among APIs lacking distributed memory, only OmpSs offers resilience (via its Nanos++ runtime), and distributed memory APIs only recently started to include resilience support [3] – likely driven by the continuous increase in machine sizes and hence decreased mean-time-between-failures. Finally, some form of heterogeneity support is provided in almost all modern APIs, though it often requires explicit heterogeneous task provisioning by the programmer.

Table 2 provides the corresponding classification with respect to the runtime system and its subcomponents (see Sect. 3). It is worth mentioning that there are various contributions extending runtime features, but these contributions are not part of the main release yet. We do not consider such extended features in our taxonomy. For instance, recent work in X10 [12] extends the X10 scheduler with distributed work-stealing algorithms across nodes; however, we classify X10 as not (yet) having a distributed scheduler. The same applies to StarPU and OmpSs. Namely, new distributed-memory scheduling policies are

Table 1. Feature comparison of APIs for task parallelism.

	Technological Readiness	Distributed Memory	Heterogeneity	Worker Management	Task Partitioning	Work Mapping	Synchronization	Resilience Management	Communication Model	Result Handling	Graph Structure	Task Cancellation	Implementation Type
C++ STL	9	×	×	i	×	i/e	e	×	smem	i/e	dag	×	Library
TBB	8	×	×	i	×	i	i	×	smem	i	tree	✓	
HPX	6	i	e	i	✓	i/e	e	×	pgas	e	dag	✓	
Legion	4	i	e	i	✓	i/e	e	×	pgas	e	tree	×	
PaRSEC	4	e	e	i	×	i/e	i	✓	msg	e	dag	✓	
OpenMP	9	×	i	e	×	i	i/e	×	smem	i	dag	✓	Extension
Charm++	6	i	e	i	✓	i/e	e	✓	pgas	i/e	dag	×	
OmpSs	5	×	i	i	×	i	i/e	✓	smem	i	dag	×	
StarPU	5	e	e	i	✓	i/e	e	×	msg	i	dag	×	
Cilk Plus	8	×	×	i	×	i	e	×	smem	i	tree	×	Lang.
Chapel	5	i	i	i	✓	i/e	e	×	pgas	i	dag	×	
X10	5	i	i	i	✓	i/e	e	✓	pgas	i	dag	×	

Table 2. Feature comparison of runtimes for task parallelism.

	Data Distribution	Scheduling on shared memory	Scheduling on distr. memory	Performance Monitoring	Fault Tolerance	Target Architecture
OpenMP	×	m	×	off/on	×	sm
TBB	×	ws	×	off	×	sm
Cilk Plus	×	ws	×	off	×	sm
StarPU	e	m	×	off/on	×	d
OmpSs	i	m	×	off/on	tf	d
Charm++	i	m	m	off/on	pf	d
X10	i	ws	×	off	pf	d
Chapel	i	ws	×	off	pf	d
HPX	i	ws	×	off/on	×	d
ParSEC	e	m	l	off	tf	d
Legion	i	ws	ws	off/on	pf	d

i	implicit
e	explicit
m	multiple (incl. ws)
l	limited
ws	work stealing
tf	task faults
pf	process faults
sm	shared memory
d	distributed memory
on	online use
off	offline (post-mortem) use

being developed for both runtimes, however, they are not part of their main release yet[5]. Also, for Chapel, X10, and HPX, there is automatic data distribution support (runtime feature); however, these runtimes require explicit work mapping in distributed memory environments (API feature).

[5] We received feedback from their developers.

Most of the runtime systems have similarities in scheduling within a single shared-memory node and work-stealing is the most common method of scheduling. On the other hand, there is no established method for inter-node scheduling. For instance, ParSEC [9] only provides a limited inter-node scheduling based on remote completion notifications, while Legion uses distributed work-stealing.

5 Conclusions

The shift in HPC towards emerging task-based parallel programming paradigms has led to a broad ecosystem of different task-based technologies. With such diversity, and some degree of isolation between individual communities of developers, there is a lack of documentation and common classification, thus hindering researchers to have a complete view of the field. In this paper, we provide an initial attempt to establish a common taxonomy and provide the corresponding categorization for many existing task-based programming environments.

We divided our taxonomy into two broad categories: *API characteristics*, which define how the programmer interacts with the system; and *many-task runtime systems*, classifying the underlying technologies. For the latter, we analyze the types of scheduling policies and goals supported, online and offline performance monitoring integration, as well as the level of resilience and detection provided for task, process and system faults.

Acknowledgement. This work was partially supported by the AllScale project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671603.

References

1. Bauer, M.E.: Legion: programming distributed heterogeneous architectures with logical regions. Ph.D. thesis, Stanford University (2014)
2. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM (JACM) **46**(5), 720–748 (1999)
3. Cunningham, D.: Resilient X10: efficient failure-aware programming. In: Proceedings of PPoPP 2014, pp. 67–80. ACM (2014)
4. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
5. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
6. Chamberlain, B.L., et al.: Parallel programmability and the chapel language. Int. J. HPC Appl. **21**(3), 291–312 (2007)
7. Augonnet, C., Thibault, S., Namyst, R.: Automatic calibration of performance models on heterogeneous multicore architectures. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009. LNCS, vol. 6043, pp. 56–65. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14122-5_9

8. Augonnet, C., et al.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput.: Pract. Exp. **23**(2), 187–198 (2011)
9. Bosilca, G., et al.: PaRSEC: exploiting heterogeneity to enhance scalability. Comput. Sci. Eng. **15**(6), 36–45 (2013)
10. Kaiser, H., et al.: HPX: a task based programming model in a global address space. In: PGAS 2014, p. 6. ACM (2014)
11. Kasim, H., March, V., Zhang, R., See, S.: Survey on parallel programming model. In: Cao, J., Li, M., Wu, M.-Y., Chen, J. (eds.) NPC 2008. LNCS, vol. 5245, pp. 266–275. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88140-7_24
12. Paudel, J., et al.: On the merits of distributed work-stealing on selective locality-aware tasks. In: 42nd International Conference on Parallel Processing, pp. 100–109, October 2013
13. Planas, J., et al.: Self-adaptive OmpSs tasks in heterogeneous environments. In: IPDPS 2013, pp. 138–149. IEEE (2013)
14. Meyer, J.C., et al.: Implementation of an energy-aware OmpSs task scheduling policy. http://www.prace-ri.eu/IMG/pdf/wp88.pdf. Accessed 02 May 2017
15. Huck, K., et al.: An early prototype of an autonomic performance environment for exascale. In: Proceedings of ROSS13, p. 8. ACM (2013)
16. Huck, K., et al.: An autonomic performance environment for exascale. Supercomput. Frontiers Innov. **2**(3), 49–66 (2015)
17. Subasi, O., et al.: NanoCheckpoints: a task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In: PDP 2015, pp. 99–102 (2015)
18. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of OOPSLA 2005, pp. 519–538. ACM (2005)
19. Blumofe, R.D., et al.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
20. Hukerikar, S., et al.: Opportunistic application-level fault detection through adaptive redundant multithreading. In: Proceedings of HPCS 2014, pp. 243–250 (2014)
21. General Acyclic Graphs of Tasks in TBB. https://software.intel.com/en-us/node/506110. Accessed 02 May 2017
22. Hoemmen, M., Heroux, M.A.: Fault-tolerant iterative methods via selective reliability. In: Proceedings of SC 2011, p. 9. IEEE Computer Society (2011)
23. Kale, L.V., Krishnan, S.: Charm++: a portable concurrent object oriented system based on c++. In: OOSPLA 1993, pp. 91–108. ACM (1993)
24. Robison, A.D.: Composable parallel patterns with intel cilk plus. Comput. Sci. Eng. **15**(2), 66–71 (2013)
25. Willhalm, T., Popovici, N.: Putting Intel threading building blocks to work. In: IWMSE08, pp. 3–4. ACM (2008)

Workshop on PGAS Programming

Interoperability of GASPI and MPI in Large Scale Scientific Applications

Dana Akhmetova[1], Luis Cebamanos[2], Roman Iakymchuk[1](✉), Tiberiu Rotaru[3], Mirko Rahn[3], Stefano Markidis[1], Erwin Laure[1], Valeria Bartsch[3], and Christian Simmendinger[4]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
`{danaak,riakymch,markidis,erwinl}@kth.se`
[2] EPCC, The University of Edinburgh, Edinburgh, UK
`l.cebamanos@epcc.ed.ac.uk`
[3] Fraunhofer ITWM, Kaiserslautern, Germany
`{tiberiu.rotaru,mirko.rahn,valeria.bartsch}@itwm.fraunhofer.de`
[4] T-Systems Solutions for Research, Frankfurt, Germany
`christian.simmendinger@t-systems-sfr.com`

Abstract. One of the main hurdles of a broad distribution of PGAS approaches is the prevalence of MPI, which as a de-facto standard appears in the code basis of many applications. To take advantage of the PGAS APIs like GASPI without a major change in the code basis, interoperability between MPI and PGAS approaches needs to be ensured. In this article, we address this challenge by providing our study and preliminary performance results regarding interoperating GASPI and MPI on the performance crucial parts of the Ludwig and iPIC3D applications. In addition, we draw a strategy for better coupling of both APIs.

Keywords: Interoperability · GASPI · MPI · Ludwig · iPIC3D Halo exchange

1 Introduction

The Message Passing Interface (MPI) has been considered the de-facto standard for writing parallel programs for clusters of computers for more than two decades. Although the API has become very powerful and rich, having passed through several major revisions, new alternative models that are taking into account modern hardware architectures have evolved in parallel. Such a model is the *Global Address Space Programming Interface (GASPI)* [9], with *GPI-2* (www.github.com/cc-hpc-itwm/GPI-2) representing an open source implementation of the GASPI standard.

The GASPI standard promotes the use of *one-sided communication*, where one side, the initiator, has all the relevant information for performing the data movement. The benefit of this is decoupling the data movement from the synchronization between processes. It enables the processes to put or get data from remote memory, without engaging the corresponding remote process, or having a synchronization point for every communication request. However, some form of synchronization is still needed in order to allow the remote process to be notified upon the completion of an operation. In addition, GASPI provides what is known as weak synchronization primitives which update a notification on the remote side. The notification semantics is complemented with routines that wait for the update of a single or a set of notifications. GASPI allows for a thread-safe handling of notifications, providing an atomic function for resetting a local notification. The notification procedures are one-sided and only involve the local process.

Thus, there is a potential of enhancing applications' performance by shifting to one-sided communication like in GASPI. There are two possibilities for such shift: 1. Rewriting large legacy MPI codes to use a different inter-node programming model is, in many cases, highly labor intensive and, therefore, not appealing to developers; 2. Replacing MPI with another API – such as GASPI – only in performance critical parts of those codes is an attractive solution from a practical perspective, but this requires both APIs to interoperate effectively and efficiently on sharing communication and on data management. In this article, we address the latter and aim to study *interoperability* of GASPI and MPI in order to allow for incremental porting of applications. GPI-2 supports [5] this interoperability with MPI in a so-called mixed-mode, where the MPI and GASPI interfaces can be mixed in a simple way. As a case study, we consider two large-scale scientific applications: iPIC3D [7] (see Sect. 3) – an implicit Particle-In-Cell code for space weather simulations; Ludwig [3] (see Sect. 4) – a large scale Lattice-Boltzmann code for complex fluids. We collect the preliminary performance results for both applications (see Sect. 5). Furthermore, we derive a strategy for enhancing the MPI and GASPI coupling using so-called shared notifications (see Sect. 2) and provide evidences that this strategy is beneficial (see Sect. 5) on simple operations such as Allreduce.

2 A Strategy to Better Interoperate GASPI and MPI

Scientific applications may use MPI features – such as MPI derived data types – not known in the GASPI specification. Due to the fact that GASPI does not support the derived data types the interoperability between MPI and GASPI within a program using MPI derived data types lacks ease of use, because the data of each local process on a node have to be packed, sent, and, then, unpacked.

To mitigate the adverse effect of MPI derived data types on the MPI plus GASPI interoperability so-called shared notifications have been recently implemented in the GPI implementation of the GASPI standard. This feature allows a smoother interoperability between a flat MPI code with shared windows and GASPI. With shared notifications a GASPI memory segment is shared between all processes local to a node. GASPI segments can be created with user allocated memory, e.g. using MPI shared windows in an MPI plus GASPI mixed program. Instead of implicit (via derived data types) or explicit packing/

unpacking of communication data, application can share information about node local data layout, structure, and computational state. As all node-local processes can access this shared data, the node local explicit ghost cell exchanges in applications can be replaced with the corresponding state notifications, where the required data can be directly read from the neighboring processes based on previously exchanged information of data layout and type. We believe that the correspondingly required programming interface can be generic and – for node local exchanges – common for both MPI and GASPI. The interface will require an allocation of a shared memory segment across node-local processes. It will require a universally acceptable format for sharing of process local data layouts and corresponding data offsets. It will require the ability to automatically detect whether or not a neighboring process is node-local; the latter information can be used to signal node-local readiness for the ghost-cell exchange or to perform explicit packing and/or unpacking into/from linear communication buffers for remote nodes. The interface will also require the ability to trigger node-local notifications in shared memory. This will include required memory fences between neighboring node local processes. Last not least – by using shared notifications – the interface becomes able to aggregate data for remote nodes and to perform one single write to the other node (for all local processes on that node) and notify all remote local processes in one step. As all remote processes can detect and access this common buffer, each remote process/rank can retrieve the required partial data for its ghost cell exchange. The ongoing, but converging, development of this generic interface will facilitate the interoperability of MPI and GASPI significantly.

3 iPIC3D: Implicit Particle-in-Cell Code

iPIC3D is a Particle-in-Cell (PIC) code for the simulation of space plasmas in space weather applications during the interaction between the solar wind and the Earth's magnetic field. The magnetosphere is a large system with many complex physical processes, requiring realistic domain sizes and billions of computational particles. The numerical discretization of Maxwell's equations and particle equations of motion is based on the implicit moment method that allows simulations with large time steps and grid spacing still retaining the numerical stability. Plasma particles from the solar wind are mimicked by computational particles. At each computational cycle, the velocity and the location of each particle are updated, the current and charge densities are interpolated to the mesh grid, and Maxwell's equations are solved. Figure 1 depicts these computational steps in iPIC3D.

iPIC3D is parallelized using domain decomposition and message-passing communications: an iPIC3D simulation is being run on a number of processors and on a network of cells, so each processor handles a number of cells. However, at certain intervals, each processor must find out the values of the cells adjacent to those in its own domain. The procedure of finding these values out is called *halo exchange*. To achieve the full 3D halo exchange, the standard approach of shifting the relevant data in each co-ordinate direction in turn is adopted. This

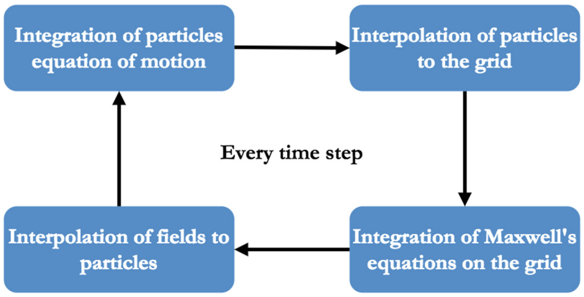

Integration of particles equation of motion
Interpolation of particles to the grid
Every time step
Integration of Maxwell's equations on the grid
Interpolation of fields to particles

Fig. 1. Structure of the iPIC3D code.

involves extensive communication between processes and requires appropriate synchronization – a receive in the first co-ordinate direction must be complete before a send in the second direction involving relevant data can take place, and so on. Note that only "outgoing" elements of the distribution need to be sent at each edge. In the particle mover part hundreds of particles per cell are constantly moved, resulting in billions of particles in large-scale simulations. All these particles are completely independent from each other, which ensures very high scalability. MPI communication at this stage is only required to transfer some of the particles from one cell or a subdomain to its neighbor.

The iPIC3D MPI communication is dominated by non-blocking point-to-point communication, occurring from communication of particles and ghost cells among neighboring processes (halo exchange), and by global reductions resulting from solving two linear systems every simulation time step. In order to reduce the communication burden in iPIC3D, we aim at replacing the MPI communication with the GASPI asynchronous one-sided communication on the communication critical parts of the code such as halo exchange in the field solver and with the GASPI reduction communication in the iPIC3D linear solver.

Implementation Highlights. The main halo exchange routine uses non-blocking MPI and MPI derived datatypes. MPI derived datatypes allow us to specify non-contiguous data in a convenient manner and yet treat it as if it was contiguous. GASPI requires the creation and later use of the so-called GASPI segments. In the case of iPIC3D, there is one GASPI segment per plane and direction. As there are three planes and two directions per plane, iPIC3D will require six different GASPI segments. The size of the segments is defined as twice the size of buffer to be sent as we will use the same segment to send and receive data from the neighbor subdomains. As iPIC3D uses MPI datatypes, complex data layouts, it is necessary to unpack the MPI datatypes and copy the data contiguously into a GASPI segment. Once the data has been sent and notified, we need to put the data back from the GASPI segment to the original buffer to be able to continue with the execution of iPIC3D.

To implement the halo exchange with GASPI, firstly the field values belonging to the boundary are being copied to the local GASPI segment. Secondly, segments of neighbors are being read to get their ghost cells and copied to the local segment. The local copy does not require a barrier: each process writes to its neighbor process' segment directly and sends a notification to that process in order to notify that data writing has accomplished. The remote process does not know that another process writes something into its memory and will not wait for when data writing ends, until it receives a notification from its neighbor. The remote process checks for locally posted notifications to get the information about changes related to a segment. Once a notification arrives, the process starts to work with data related to that particular notification.

In addition, the MPI reduction operations were replaced with the GASPI communication in the linear solvers (CG and GMRes) to calculate the inner products and the norm of vectors located on different processes.

4 The Ludwig Application

Ludwig [3] is a versatile code for the simulation of Lattice-Boltzmann models in 3D on cubic lattices. Some of the problems that could be simulated with Ludwig include detergency, mesophase formation in amphiphiles, colloidal suspensions, and liquid crystal flows. Broadly, the code is intended for complex fluid problems at low Reynolds numbers, so there is no consideration of turbulence, high Mach number flows, high density ratio flows, and so on. Ludwig uses an efficient domain decomposition algorithm, which employs the Lattice-Boltzmann method to iterate the solution on each subdomain. The domain decomposition is carried out by splitting a three dimensional lattice into smaller lattices on subdomains and exchanging information with adjacent subdomains [4]. For each iteration, Ludwig uses MPI for communications with adjacent subdomains using *halo exchange* [2].

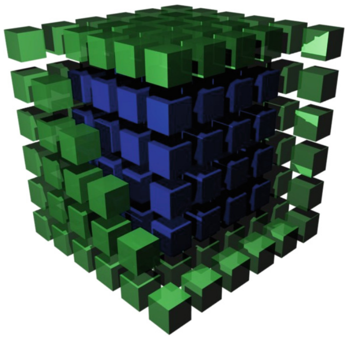


Fig. 2. Lattice subdomain where the internal section represents the real lattice and the external region the halo sites.

In the original implementation of the Ludwig halo exchange, the number of messages sent and received by each MPI process is reduced as much as possible. Each subdomain needs to exchange data with its 26 neighbors in three directions to continue with the solution of the problem. This means that synchronization between the different planes is required. To coordinate the solution, communication between adjacent subdomains is required after each iteration. This is done by creating halos around the dimensions of the subdomain, i.e. extending the dimension of the subdomain by one lattice point in each direction as depicted in Fig. 2. After each time step, MPI processes will have to communicate a 2D plane of m velocities to their adjacent MPI processes. Since each plane shares some sites with the other planes, the exchange of information in each direction should be synchronized before continuing with the execution.

GASPI promotes the use of one-sided communication, where the initiator has all the relevant information for performing the data movement. This idea decouples the data movement from the synchronization between processes and it is especially relevant in applications that rely on continuous halo communications between neighbors. We aim at reducing the synchronization between subdomains by porting Ludwig's main halo exchange routines form MPI to GASPI.

Implementation Highlights. The halo exchange routine responsible for exchanging data between neighbor subdomains uses non-blocking MPI and MPI derived datatypes. MPI derived datatypes allow us to specify non-contiguous data in a convenient manner and yet treat it as if it was contiguous.

GASPI requires the creation and later on use of what is known as GASPI segments. A GASPI segment is window of memory allocated to be used with the GASPI model. In our case we have created one GASPI segment per plane and direction. Therefore, since we have three planes and two directions per plane, we will require six different GASPI segments. This number of GASPI segments is sufficient for each subdomain to communicate its faces with its immediate neighbors in the 3D space. The size of the segments is defined as twice the size of buffer to be sent since we will use the same segment to send and receive data from neighbor subdomains.

Listing 1.1. GASPI pointers to GASPI segments in the YZ plane.

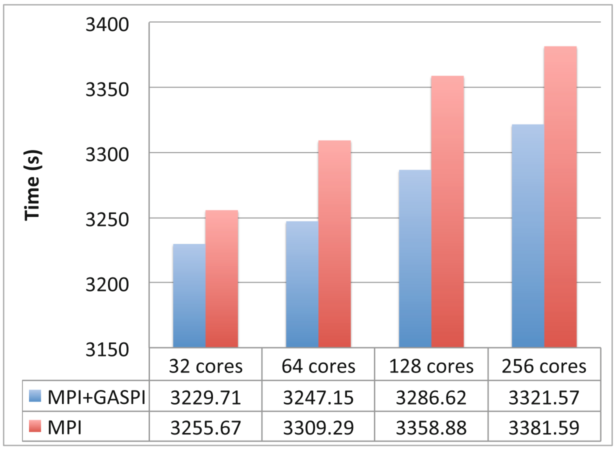
```
int YZ_size = lb->ndist*NVEL*ny*nz;

/* Segment size is exactly twice the size of the buffer.*/
const gaspi_size_t seg_size = 2 * YZ_size * sizeof(double);

 /* segment ids */
const gaspi_segment_id_t seg_id_YZ_L = 0;
const gaspi_segment_id_t seg_id_YZ_R = 1;
gaspi_pointer_t gptr_YZ_L, gptr_YZ_R;

/* pointer to the right */
GASPIERROR(gaspi_segment_ptr(seg_id_YZ_L, &gptr_YZ_L));
double* ptr_YZ_L = (double*)gptr_YZ_L;

/* pointer to the left */
GASPIERROR(gaspi_segment_ptr(seg_id_YZ_R, &gptr_YZ_R));
double* ptr_YZ_R = (double*)gptr_YZ_R;
```

For purposes of clarity, Listing 1.1 shows the GASPI pointer creation only in the YZ plane. For instance, in the YZ plane, each created segment is assigned with an independent id number. Hence, the data is already contiguous in memory and, therefore, a simple copy directly from the buffer that contains the data to a GASPI segment is straightforward. However, since Ludwig uses MPI datatypes, more complicated layouts of the data exist for other planes and it is necessary to unpack the MPI datatypes and copy the data contiguously into a GASPI segment. Once the data has been sent and notified we need to recover the data back from the GASPI segment to the original buffer to be able to continue with the normal execution of Ludwig.

5 Performance Results

iPIC3D. We performed our tests on the Beskow supercomputer (Cray XC40) equipped with two 16-core @ 2.3 GHz Intel Haswell-EP processors. To compare the original version of the iPIC3D code with the new, GASPI-based, version, we used a standard simulation cases called Geospace Environment Modeling (GEM) Reconnection Challenge that is adapted to the Earth's magnetotail reconnection [1,6]. In addition, we used two different simulation cases, namely field- and particle-dominated, with a fixed number of iterations (20) in the field solver.

Figure 3 shows the results of the weak scaling tests for one of the iPIC3D simulations. Three-dimensional decomposition of MPI processes on X-, Y- and Z-axes was used, resulting in different topologies of MPI processes. For this particle dominated Magnetosphere 3D simulation on 64 cores ($4 \times 4 \times 4$ MPI processes $\times$ 4 OpenMP threads), 27×106 particles and $30 \times 30 \times 30$ cells were used, and the simulation size increased proportionally to the number of processes.



	32 cores	64 cores	128 cores	256 cores
🟦 MPI+GASPI	3229.71	3247.15	3286.62	3321.57
🟥 MPI	3255.67	3309.29	3358.88	3381.59

Fig. 3. Weak scaling results for the GEM 3D simulation of the particle-mover dominated regime of iPIC3D on Beskow.

For this simulation test case, the new version, based on GASPI, is slightly faster (by 1–2%) on different number of cores. The challenge of a successful porting of iPIC3D to GASPI depends on the optimal utilization of one-sided communication mechanism to achieve performance gain and scalability on pre-Exascale supercomputers. GASPI provides the one-sided communication that facilitates asynchronous procedures between processes. However, this requires the local processes to manage the communication in an optimized way to maximum the overlapping of communication and computation. The trade-off between asynchronicity and data synchronization requires further investigation.

Ludwig. A set of performance tests were carried out on ARCHER, a Cray XC30 system equipped with two 12-core @ 2.7 GHz Intel Ivy Bridge processors. All simulations were executed five times on fully populated nodes, i.e. using 24 MPI/GASPI processes per node.

The time to transfer a message depends on the network latency and bandwidth. The latency is independent of the size of the message being sent, but dependent of the MPI implementation and network use. Figure 4 shows the measured bandwidth against the message size using Cray MPI. The bandwidth is low at very small message sizes because the time spent to send each message is dominated by the latency. As soon as the message size is increased over 0.2 MBytes, the bandwidth quickly rises to the maximum allowed by the fabric interconnect. We have also measured the amount of data required to be sent and received from each process at the end of each iteration in 192^3 lattice size, as represented in Fig. 5.

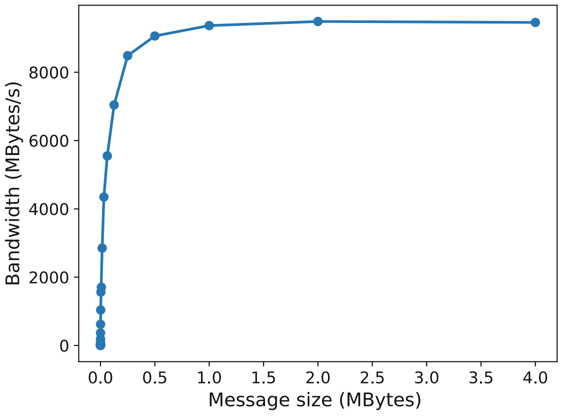


Fig. 4. Bandwidth and message size on ARCHER, using the OSU benchmarks [8].

Figure 6 shows the strong scaling results of running Ludwig on up to 3,072 processes on ARCHER. The total time that Ludwig spends on the main stepping loop is represented in Fig. 6a, showing small difference in performance between the pure MPI version and the MPI + GASPI version of Ludwig; the performance overhead is negligible with less than 1000 processes. When narrowing our focus to the halo exchange (see Fig. 6b), which is one of the key components in the main stepping loop, we can see that this performance penalty is low for a small processes count, but it grows as the number of processes is increased. This is probably due to the fact that the bandwidth is at its best in that region as Fig. 4 indicates Thus, there is a direct connection between the overhead in the halo exchange and the total loop. Nevertheless, given the performance benefits of one-sided communication in GASPI[1], we attribute this performance penalty

[1] http://www.gpi-site.com/gpi2/benchmarks/.

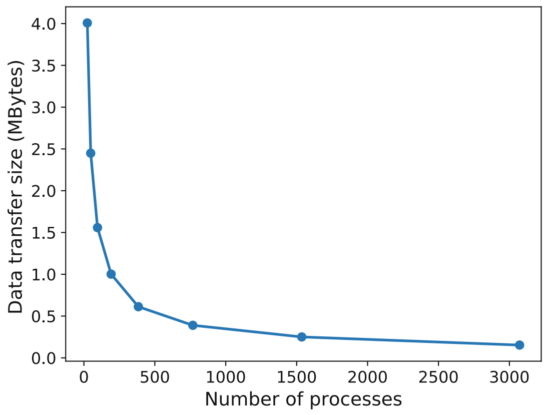
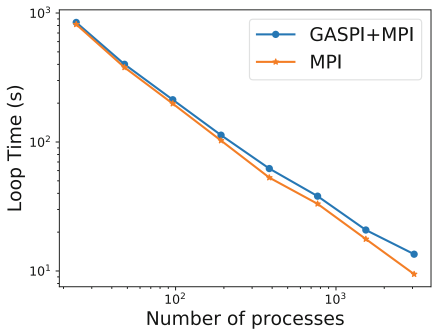
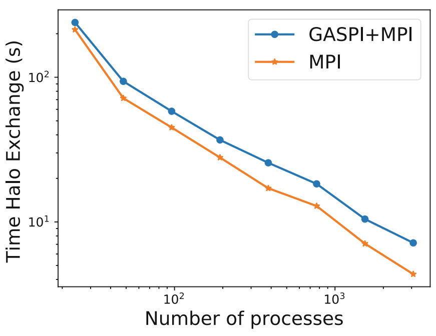


Fig. 5. Data transfer size by each process at the end of 192^3 lattice size simulation.



(a) Total loop time.



(b) Total halo exchange time.

Fig. 6. Strong scaling results of Ludwig for a 192^3 lattice size on ARCHER.

to tedious process of unpacking and packing back and forth between the MPI datatypes and the GASPI segments.

Shared Window Communication in GASPI. In order to validate this new programming paradigm of shared notifications in GASPI, we have implemented an equivalent to the MPI Allreduce for large messages. The implementation makes substantial use of pipelined rings. The algorithm consists of two stages. In the first stage, each of the N nodes performs a reduction of $1/N$ of the dataset (via the pipelined ring). In the second stage, the partial result from each node is broadcasted to the other nodes (again in the pipelined ring) such that after the broadcast all nodes have access to the complete reduced dataset.

In order to split the reduction and communication loads across all processes, each of the N parts is again subdivided into at least M parts (where M is the number of processes per node) such that there are at least $N \times M$ messages in the ring at any point in time. The GASPI shared notification model allows

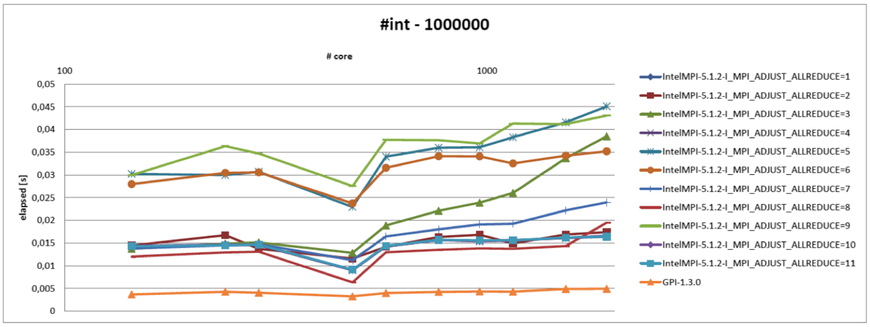


Fig. 7. Performance results of the pipelined ring implementation of Allreduce.

any process to detect any of these $N \times M$ incoming asynchronous and one-sided notified messages, to reduce and forward them along the pipelined ring. Figure 7 shows a comparison of Allreduce implemented on top of GASPI shared windows against various Allreduce MPI low-level implementations in Intel MPI 5.1.2. Those are 1. Recursive doubling; 2. Rabenseifner's; 3. Reduce + Bcast; 4. Topology aware Reduce + Bcast; 5. Binomial gather + scatter; 6. Topology aware binominal gather + scatter; 7. Shumilin's ring; 8. Ring; 9. Knomial; 10. Topology aware SHM based flat; 11. Topology aware SHM based Knomial. Some of these implementations feature an optimal bandwidth term (Ring based or Rabenseifner's), however they are not able to leverage pipelining as efficiently as the high-level GASPI Implementation. The main problem here is that the underlying MPI point-to-point low-level frameworks (such as e.g. UCX) are not able to make efficient use of notified communication either.

6 Conclusions

The original versions of both iPIC3D and Ludwig – like many other MPI applications – use MPI datatypes. That soon became a problem while interoperating with GASPI since GASPI works on segments of data. This means that we had to unpack the data from the MPI datatypes, copy them to a GASPI segment, send them, and, then, unpack the data. We believe this packing-unpacking was the major burden for the applications' performance.

In order to improve the interoperability with a flat MPI programming model, GASPI has introduced a novel allocation policy for segments where data and GASPI notifications can be shared across multiple processes on a single node. To that end, any incoming one-sided GASPI notification will be visible node-locally across all node-local ranks. The shared notifications should be used with GASPI segments that are employing shared memory, such as MPI windows, provided by the applications under the interoperability mode.

We are currently developing a generic interface which can make use of these shared memory segments for the specific purpose of ghost cell exchanges.

The developed interface will not only facilitate the interoperability of MPI plus GASPI significantly, but it will also substantially enrich the programming paradigm of MPI shared windows.

Acknowledgement. This work was funded by EU H2020 Research and Innovation programme through the INTERTWinE project (no. 671602). The simulations were performed on resources provided by SNIC at PDC-HPC, KTH.

References

1. Birn, J., Hesse, M.: Geospace environment modeling (GEM) magnetic reconnection challenge: resistive tearing, anisotropic pressure and hall effects. JGR: Space Phys. **106**(A3), 3737–3750 (2001)
2. Davidson, E.: Message-passing for Lattice Boltzmann. Master's thesis, EPCC, The University of Edinburgh, Scotland, UK (2008)
3. Desplat, J.C., Pagonabarraga, I., Bladon, P.: LUDWIG: a parallel Lattice-Boltzmann code for complex fluids. Comput. Phys. Commun. **134**(3), 273–290 (2001)
4. Gray, A., Hart, A., Henrich, O., Stratford, K.: Scaling soft matter physics to thousands of graphic processing units in parallel. IJHPCA **29**(3), 274–283 (2015)
5. Machado, R., Rotaru, T., Rahn, M., Bartsch, V.: Guide to porting MPI applications to GPI-2. Technical report, Fraunhofer ITWM (2015)
6. Markidis, S., Henri, P., Lapenta, G., Rönnmark, K., Hamrin, M., Meliania, Z., Laure, E.: The fluid-kinetic particle-in-cell method for plasma simulations. J. Comput. Phys. **271**, 415–429 (2014)
7. Markidis, S., Lapenta, G., et al.: Multi-scale simulations of plasma with iPIC3D. Math. Comput. Simul. **80**(7), 1509–1519 (2010)
8. MVAPICH. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/benchmarks/
9. Simmendinger, C., Rahn, M., Gruenewald, D.: The GASPI API: a failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures. In: Resch, M., Bez, W., Focht, E., Kobayashi, H., Patel, N. (eds.) Sustained Simulation Performance 2014, pp. 17–32. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-10626-7_2

Evaluation of the Parallel Performance of the Java and PCJ on the Intel KNL Based Systems

Marek Nowicki[1], Łukasz Górski[1,2], and Piotr Bała[2](✉)

[1] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Chopina 12/18, 87-100 Toruń, Poland
faramir@mat.umk.pl

[2] Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw, Pawińskiego 5a, 02-106 Warsaw, Poland
{lgorski,bala}@icm.edu.pl

Abstract. In this paper, we present performance and scalability of the Java codes parallelized on the Intel KNL platform using Java and PCJ Library. The parallelization is performed using PGAS programming model with no modification to Java language nor Java Virtual Machine. The obtained results show good overall performance, especially for parallel applications. The microbenchmark results, compared to the C/MPI, show that PCJ communication efficiency should be improved.

Keywords: Parallel computing · Multicore · PCJ · PGAS · KNL

1 Introduction

Recent developments in hardware lead to multinode computers with nodes equipped with a couple of multicore processors. Therefore developers have to deal with the node parallelism ranging a hundred cores. Recent developments are pushing this limit even further. Intel KNL (*Knights Landing*) processor is equipped with the 64–72 cores depending on the model, each of them can run 4 threads using Hyper-threading technology. This leads to a couple of hundreds thread parallelism in the single processor. Keeping in mind that typical node can be equipped with more than one processor we have to face the situation where thousands of threads are running in parallel on each node.

1.1 Programming Paradigms

The hardware changes force software to be written in a multi-threaded manner to take full advantages of the current hardware. The current standard for parallel programming is the MPI (*Message Passing Interface*) which has been created decades ago. However, MPI is still under development as in 2015 MPI-3.1 standard was released. The MPI has its implementation for programming languages: C, C++, and Fortran. There are also wrappers-bindings for other programming languages like Python or Java, but under the cover, they are using libraries written in C.

For the multicore nodes, in order to achieve good performance, MPI is often connected with OpenMP which leads to the parallelization on different levels and makes programmers work more difficult. Therefore there is strong interest in new programming paradigms which will be easier to use. PGAS (Partitioned Global Address Space) [2] is one of most promising ones.

In the PGAS paradigm, there are three basic operations: *synchronisation*, *put* and *get*. *Synchronisation* (sometimes called *barrier*) allows ensuring that every processor reaches the expected execution point before continuing execution. *Put* is used for changing shared variable value of another processor while *get* is used for retrieving shared variable value of another processor.

1.2 Parallel Programming in Java

Java is considered as a higher level programming language. In the TIOBE index [1] it is most popular programming language over the world. Designed for wide range of devices, Java has been developed with the concurrency in mind. From the very beginning, there was `java.lang.Thread` class representing native operating system threads or `synchronized` keyword for denoting the critical section in the code. The version 1.5 of the Java Platform added Concurrency Utilities (under *JSR 166: Concurrency Utilities*) and the latest version of the Java platform at the time of writing this paper, Java SE 8, introduces the possibility to process data using parallel streams.

Despite the fact, the Java is concurrency-oriented language, the available Java Virtual Machines (JVM) allow to run concurrent application only on a single workstation, single computational node. There were attempts to create distributed Java Virtual Machines [3,4], that would be visible to the user as single JVM, but be transparently running on multinode systems, but there were problems with scalability and narrow applicability.

There are libraries and frameworks for Java that are trying to solve this issue. There are well-known frameworks like Apache Hadoop or Apache Spark for processing data in the map-reduce model. However, the model is not well suitable for many problems. Sometimes it is very hard, or even impossible to adopt problem to process it using map-reduce model. There is also Akka toolkit for the Java that uses actor model for processing data in parallel. However, the data stored by actors is immutable, and concurrency is based on passing messages between actors that are working transparently on the same or different nodes.

This paper describes a PCJ library for the Java language [5] which is a Java library for parallel computing using PGAS programming model. The main focus is on the performance evaluation of PCJ using Intel KNL processors. The paper is organized as follows: the PCJ library is described in the Sect. 2. Section 3 provides basic information on Intel KNL platform. The results are described in Sect. 4. Paper concludes with related work and conclusions.

2 PCJ Library

The PCJ Library [5–7] uses PGAS paradigm for parallel processing. It helps to perform parallel calculations on single or multinode architecture. The library is OpenSource (BSD license), and its source code is available on GitHub [8].

In this paper, we use the newest version of the PCJ library – version 5 (PCJ5). Compared to the older versions, this one provides modified API, but the most revolutionary changes were made in the PCJ engine which is hidden from the user.

In particular, the *custom class loaders* have been removed, so PCJ threads on the same JVM can easily share the state of static fields. The *deserializers* threads (one for each PCJ thread on the JVM) that were responsible for deserializing incoming data have been removed. Instead of them, there are multiple threads (*workers*) that process messages from its arrival to sending, if necessary, a notification. There are also changes in communication protocol: internal messages that are sent between PCJ nodes (between JVMs that take part of parallel execution) are modified. The each message type has common only one field (byte) indicating a type of the message. Any additional data is associated with the specified message type.

3 Knights Landing Architecture

The Intel KNL architecture was launched during Summer 2016. It is a manycore system composed of numerous computing CPU cores (at least 64 cores). Each core is an Intel Airmont (Atom) core having four threads each. The cores are organized into tiles each containing two cores sharing a 1 MB L2 cache. The tiles are connected to each other with a mesh. From the memory point of view, in addition to the traditional DDR4 memory, the device is equipped with the Multi-Channel DRAM (MCDRAM) memory which is a high bandwidth (about 4x more than DDR4), but low capacity (16 GB) memory.

KNL supports legacy x86 instructions. In addition, it introduces (as its predecessor) the AVX-512 instruction set which provides support for 512-bit-wide vector instructions. Tiles, memory controllers, I/O controllers and other chip components are interconnected through a 2D mesh. The mesh supports the MESIF cache coherent protocol.

The processor used for the experiments in this paper is the Intel Xeon Phi model 7250 (KNL). The CPU has 68 cores, running at 1.40 GHz. The machine is equipped with 96 GB of RAM per CPU. The installed OS is a Scientific Linux 7.2 and the kernel version 3.10.0-327.10.1.el7.x86_64 and Java 1.8.0_60. Intel MPI 2017 has been used to launch parallel applications.

4 Results

We have performed a number of performance and scalability tests. The results are presented below for the microbenchmark such as ping-pong and broadcast and for the selected applications with the different execution profile.

All application have been run using 1 or 2 Intel KNL nodes provided by the Rescale. The jobs were submitted using Rescale web interface. Example execution script is presented in the Listing 1.

```
export JAVA_HOME=/home/rescale/shared/program/java/jre1.8.0
    _60
export PATH=$PATH:$JAVA_HOME/bin

mpirun hostname > nodes.all
uniq nodes.all > nodes.uniq

mpirun -hostfile nodes.uniq -ppn 1 \
   java -cp .:PCJ-5.0.3.jar:HelloWorld.jar HelloWorld
```

Listing 1. Example script to run PCJ HelloWorld Application on Intel KNL provided by Rescale. Please note that Java JRE is not included in the default path and has been added explicitly.

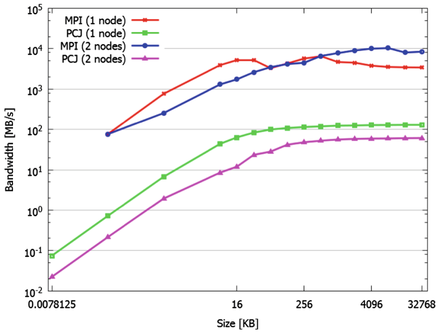
4.1 Microbenchmarks

First tests that typically are performed on the new cluster are ping-pong and broadcast. Those tests measure the performance of simplest interprocess communication to assess possible maximum communication rate on the machine.

Ping-pong. We have measured the bandwidth for sending an array of `double` elements ranged from 1 element (8 bytes) up to 4.194.304 elements (32 MB). We have done 5 tests, each sending 100 times the array and calculating average time necessary to finish the sending loop. The best time (the lowest value) is taken as a result. The JVM was warmed up.

The results obtained for the ping-pong is presented on Fig. 1. The figure compares the performance of MPI and PCJ on the same machine and between nodes using sending data in blocking manner, that is sending next block of data is performed after receiving confirmation that previous part has arrived at the destination. The bandwidth for PCJ is about 25 times lower than for MPI on 1 node. That is due to the fact, the messages in the PCJ has to be serialized, even when the sender and receiver are on the same node, to have a complete copy (clone) of the data. Comparing results on 2 nodes shows that the bandwidth of MPI is 100 times higher than PCJ. Additionally, the usage of TCP sockets in the PCJ has decreased bandwidth by the factor of 2. However, the bandwidth on MPI increased by the factor of 2 on intranode communication for large messages.

Figure 1 presents the bandwidth for benchmark made using only the PCJ on 1 and 2 nodes. In addition to blocking method, the non-blocking method was used – the next block of data is sent just after previous data was transfer to send. The confirmation of correctly received message is not used here. The performance of sending the data using 1 and 2 nodes are in principle the same in this situation.



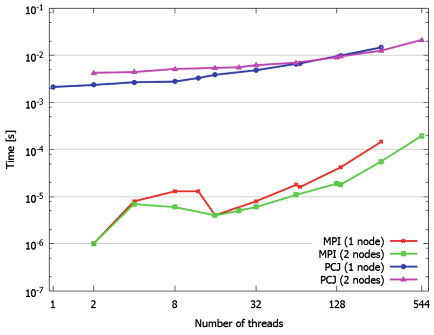
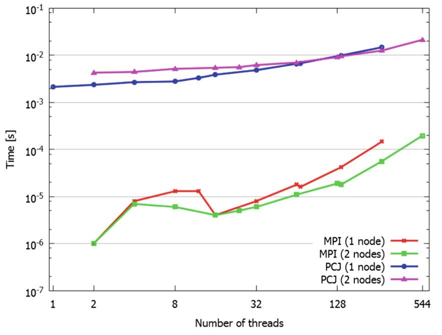


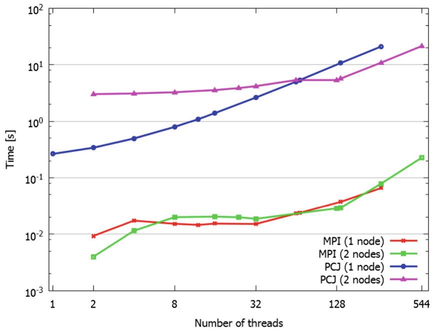
Fig. 1. Ping-pong on 1 and 2 nodes using MPI and PCJ (left). For PCJ results implemented with blocking and non-blocking communication are presented (right).

Broadcast. Broadcast benchmark measures time needed for broadcasting value from one selected thread to all threads. Like in ping-pong benchmark, the message size is calculated as a length in bytes of the data without adding a size of any header. Similarly, the data sent in this benchmark was an array of `double` elements ranged from 1 element (8 bytes) to 4.194.304 elements (32 MB). The benchmark method was invoked 100 times and the average time necessary to finish broadcasting (with notification) was calculated as a result.

The charts presented in Fig. 2 show the time needed to perform broadcast to a number of threads ranged from 1 to 544 transferring 8 bytes and 32 MB. The time spent on the broadcast operation using PCJ in average is about 100 times longer than for MPI both for the smaller and bigger array.



1 element array (8 B)



4.194.304 elements array (32 MB)

Fig. 2. Broadcast on 1 and 2 nodes using MPI and PCJ

Figure 3 depicts the bandwidth of the broadcast operation depending on the message size. The value is calculated as an amount of bytes necessary to transfer data to each of 544 threads. The bandwidth stabilizes both for MPI and PCJ when the message size is bigger than 16 KB. Regardless of the message size, the MPI bandwidth is in average 100 times higher than PCJ.

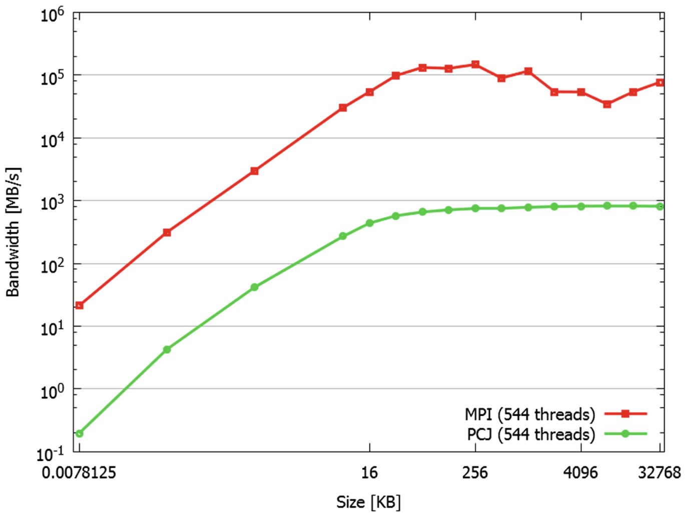


Fig. 3. Broadcast on 2 nodes (544 threads) using MPI and PCJ

4.2 RayTracer

RayTracer is popular performance test based on the rendering of the 3-dimensional scene using ray tracing. The PCJ version of the application is based on the example included in the Java Grande Forum Benchmark [9]. The scene consists of 64 balls placed in a grid 4 × 4 × 4 and 5 light sources. The parallelization of the code is based on the work distribution. While the scene is copied to the processors, each trace can be processed independently on the designated core. The work is distributed by image rows, and each PCJ thread executes every `nprocess = PCJ.threadCount()` row. The benchmark is computationally intensive and has one communication point at the end of calculation where the reduction occurs – every thread sends calculated data to *thread 0*.

The PCJ implementation has been compared to the C++ implementation which was directly translated from the Java code. The reference code has been improved by the adjustment of the keywords and by the optimization of the syntax. We have decided to use C++ code as the reference to utilizing as much as possible object oriented features of the original Java Grande Forum Benchmark. The C++ code was compiled with `-O2` flag.

Figure 4 presents the performance of rendering the 2500 × 2500 pixels scene. The performance results were gained using a different amount of threads on 2 nodes. The efficiency of PCJ version of the benchmark is from 15% up to 75% (in average 45%) higher than MPI version.

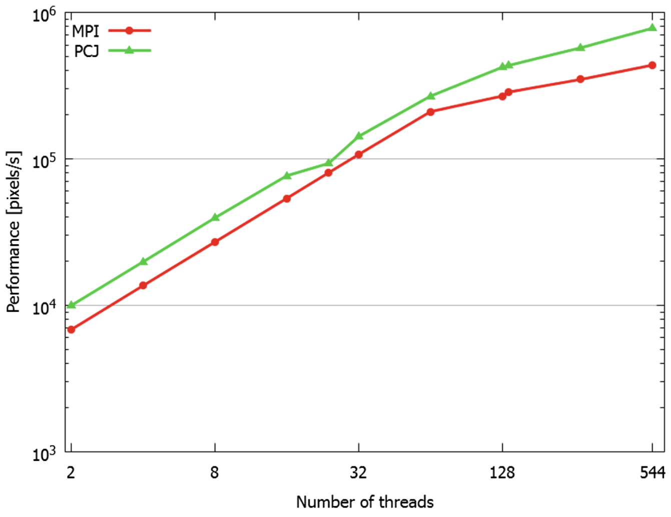


Fig. 4. RayTracer on 2500×2500 pixels scene using MPI and PCJ

4.3 Genetic Algorithm

The PCJ implementation of evolutionary algorithm has been used to find minimum of two-dimensional Rosenbrock function as defined in the CEC'14 Benchmark Suite [10] and used in our other evaluation papers [11]:

$$f_4(x, y) = (1 - x)^2 + 100(y - x^2)^2 \tag{1}$$

f_4's minimum lies in a deep, parabolic valley and the problem is generally hard for numerical solvers to approach. For testing purposes, to further increase problem's difficulty level, we have decided to deviate from the standard range of function's parameters values ($x, y \in [-2.048, 2.048]$) and have chosen significantly larger parameter space instead ($x, y \in [-20, 20]$).

The optimization problem was solved using the well-known differential evolution algorithm. It is a metaheuristic that has proven to be a solid candidate for the task of optimization problem-solving. It uses a set of nature-inspired operations (mutations, crossing-over) to generate candidate vectors that approximate optimal problem solution. A detailed description of the algorithm and its implementation is given in [11]. For the evolution, we have used 40 000 generations with 300 individuals at each PCJ thread (island model). Accurate results could have been achieved with the use of much smaller population data; however effective running times of milliseconds achieved in that scenario would not have allowed reaching conclusive performance results. Two migration policies were tested; migration either occurred every generation (continuous migration) or every 30 generations, thus limiting bandwidth use.

The scalability results are presented for the weak case, eg. the number of individuals and populations for each thread is the same. With the increased

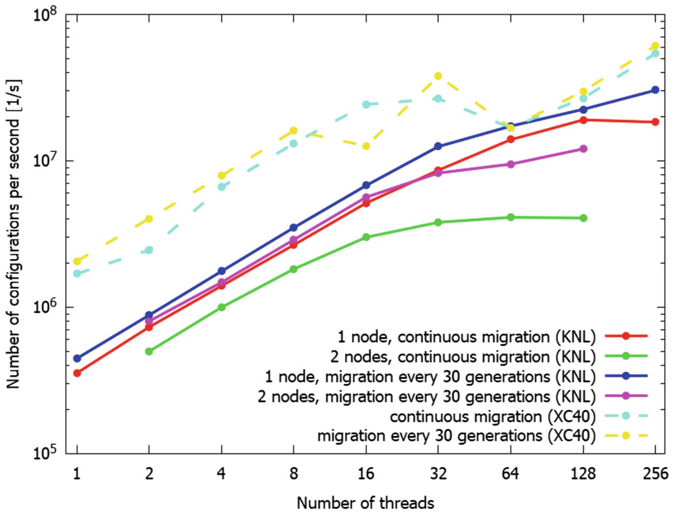


Fig. 5. Parallel differential evolution - Rosenbrock function

number of threads the total size of the population increases. The number of configurations tested in the unit of time for the different number of PCJ threads presented in the Fig. 5 shows that, generally speaking, multicore architecture is successfully exploited. It is a well known fact that PGAS languages solve shared-memory scalability problems best with the use of communication-limiting algorithms [12]. This phenomenon is exhibited by comparing performance results of 1-node and 2-node runs (internode communication cost is factored out in the former case) and in case of communication-limiting migration strategy. The performance results for Cray XC40 are given as reference (dashed lines).

4.4 Performance Analysis

The presented performance results show that PCJ scales well on the KNL hardware. The speedup is usually close to the ideal one, especially for the thread number fitting to the single CPU size (68 in this case). The good scalability is visible for the example applications such as raytracing and genetic algorithm. In both cases PCJ implementation benefits from asynchronous communication and overlap between calculations and communication. The slowdown visible for larger number of threads is caused by the small size of the data used for experiments.

Good scalability but with the communication slower than in the case of C/MPI is visible for the results of microbenchmarks such as ping-pong and broadcast. This comes from the fact, that PCJ is using Java Concurrency and socket for the exchange data between PCJ threads. This requires copying of the data from the arrays to the byte arrays and vice versa. This operation is time consuming and decreases performance even if communication is performed within single CPU.

The communication between KNL CPU's is performed in PCJ using sockets, while MPI is capable to use OmniPath interconnect which allows for lower latency and much higher bandwidth. The work on using OmniPath with PCJ library is ongoing.

5 Related Work

Intel KNL processor has been designed to provide significant computational performance from a single chip. Therefore it is of strong interest to the HPC community. Different performance evaluations have been performed, however, due to small Java popularity in the high performance computing such data is not yet available for this particular language.

In particular, the Mantevo suite of mini applications and NAS Parallel Benchmarks are used to analyze the behavior of very different application kernels, from molecular dynamics to CFD mini-applications [13]. Initial scalability results show promise for all the applications considered, however still there is need for improvements.

Performance and scalability results for particle discrete event simulation on KNL processor show that within a single KNL processor, up to 2X performance improvement can be achieved compared to commodity Xeon multicore processors [14]. The performance scales well with the best results achieved when thread affinity is assigned, CPU cores are evenly loaded, cache sharing is exploited and communication is limited to small clusters of cores.

6 Conclusions

Presented results confirm that Java applications can run successfully on the Intel KNL architecture benefiting form the large number of cores available. In particular, PCJ library allows for easy development of the scalable parallel applications running on the multiple CPUs.

However, the efficiency of the applications developed with PCJ library can be improved by reducing data copying in the application and by development of the dedicated communication mechanisms based on the OmniPath interface.

Acknowledgment. The authors would like to thank CHIST-ERA consortium for financial support under HPDCJ project (Polish part funded by NCN grant 2014/14/Z/ST6/00007).

The performance tests have been performed using Rescale Inc. computational facilities.

References

1. TIOBE Index—TIOBE - The Software Quality Company. https://www.tiobe.com/tiobe-index/. Accessed 18 May 2017
2. De Wael, M., Marr, S., De Fraine, B., Van Cutsem, T., De Meuter, W.: Partitioned global address space languages. ACM Comput. Surv. **47**(4), 62 (2015). https://doi.org/10.1145/2716320
3. Zhu, W., Wang, C.L., Lau, F.C.M.: Lightweight transparent Java thread migration for distributed JVM. In: 2003 International Conference on Parallel Processing (ICPP 2003), pp. 465–472. IEEE (2003)
4. Zhu, W., Wang, C.-L., Lau, F.C.M.: JESSICA2: a distributed Java virtual machine with transparent thread migration support. In: IEEE International Conference on Cluster Computing, pp. 381–388. IEEE (2002)
5. PCJ Home. http://pcj.icm.edu.pl. Accessed 18 May 2017
6. Nowicki, M., Bała, P.: Parallel computations in Java with PCJ library. In: Smari, W.W., Zeljkovic, V. (eds.) 2012 International Conference on High Performance Computing and Simulation (HPCS), pp. 381–387. IEEE (2012)
7. Nowicki, M., Górski, Ł., Grabarczyk, P., Bała, P.: PCJ - Java library for high performance computing in PGAS model. In: Smari, W.W., Zeljkovic, V. (eds.) 2014 International Conference on High Performance Computing and Simulation (HPCS), pp. 202–209. IEEE (2014)
8. https://github.com/hpdcj/pcj. Accessed 30 Mar 2017
9. Java Grande Project: benchmark suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite. Accessed 19 Jan 2014
10. Liang, J.J., Qu, B.Y., Suganthan, P.N.: Problem definitions and evaluation criteria for the CEC 2014 special session and competition on single objective real-parameter numerical optimization. Technical report 201311, Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou, China (2014). http://bee22.com/resources/Liang%20CEC2014.pdf
11. Górski, Ł., Rakowski, F., Bała, P.: Parallel differential evolution in the PGAS programming model implemented with PCJ Java library. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9573, pp. 448–458. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32149-3_42
12. Da Costa, G., Fahringer, T., Gallego, J.A.R., Grasso, T., Hristov, A., Karatza, H., Lastovetsky, A., Marozzo, F., Petcu, D., Stavrinides, G.: Exascale machines require new programming paradigms and runtimes. Supercomput. Front. Innov. **2**(2), 627 (2015)
13. Rosales, C., Cazes, J., Milfeld, K., Gómez-Iglesias, A., Koesterke, L., Huang, L., Vienne, J.: A comparative study of application performance and scalability on the Intel knights landing processor. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) ISC High Performance 2016. LNCS, vol. 9945, pp. 307–318. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_22
14. Williams, B., Ponomarev, D., Abu-Ghazaleh, N., Wilsey, P.: Performance characterization of parallel discrete event simulation on knights landing processor. In: Proceedings of ACM SIGSIM International Conference on Principles of Advanced Discrete Simulation (2017)

Fault-Tolerance Mechanisms for the Java Parallel Codes Implemented with the PCJ Library

Michał Szynkiewicz[1,2](✉) and Marek Nowicki[1]

[1] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Chopina 12/18, 87-100 Toruń, Poland
{michalsz,faramir}@mat.umk.pl

[2] Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw, Pawińskiego 5a, 02-106 Warsaw, Poland

Abstract. Parallel programs run on multiple processor systems with hundreds and thousands of cores. Because of a large number of nodes, failure can happen quite often, sometimes within hours. This makes fault-tolerance a crucial concern for nowadays HPC solutions.

PCJ library is one of the most successful Java solutions allowing to parallelize large scale computations up to thousands of cores. This paper describes a minimal overhead failure discovery and mitigation mechanisms introduced to the PCJ library.

Our implementation provides a programmer with a basic functionality to detect a node failure. Java exception mechanism and local node monitoring are used to detect execution problems. The problems are presented to the programmer through easy to use extensions. In the result, the programmer does not have to deal with low-level programming.

The detailed scenario how to recover from the failure has to be decided and implemented by the programmer.

Keywords: Java · Fault tolerance · High performance computing
Partitioned Global Address Space

1 Introduction

Because of a large number of nodes, the time between failures of modern computational systems systems ranges from hours to even minutes [1]. Resilience is *the ability of a system to maintain state awareness and an accepted level of operational normalcy in response to disturbances, including threats of an unexpected and malicious nature* [2]. It is a major roadblock for parallel programming on current and future HPC systems.

Applications are becoming more complex and consist of an increasingly large number of distinct modules. Data assimilation, simulation, and analysis are coupled into complex workflows. Furthermore, the need to reduce communication, allow asynchrony, and tolerate failures results in more complex algorithms.

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 298–307, 2018.
https://doi.org/10.1007/978-3-319-78054-2_28

In addition to the progress in the hardware, it turns out that traditional programming models such as MPI and OpenMP are not enough and programmers are looking for new tools, new languages and new paradigms that are better-suited to exploit parallelism [5]. Languages adhering to Partitioned Global Address Space (PGAS) paradigm are very promising in that context. In PGAS languages, the memory is divided among all the threads of execution. The PGAS languages have been successfully used for parallelization of numerous HPC applications for which proper handling of fault tolerance becomes important.

This paper deals with resilience for parallel Java applications built using Parallel Computing in Java (PCJ) library [7,8]. Here we understand resilience as the techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults. The PCJ library is built upon Partitioned Global Address Space Model and has been successfully used to parallelize selected scientific applications on the HPC systems with thousands of cores. The paper presents an extension to the library which allows for the fault-tolerant execution of applications.

2 Related Work

In this section, we present selected solutions developed for the PGAS programming paradigm. The overview of other solutions can be found elsewhere (eg. [3]).

2.1 Checkpointing for OpenSHMEM

OpenSHMEM is a PGAS implementation that provides bindings to C and FORTRAN. The fault tolerance strategy chosen for OpenSHMEM is the checkpoint-restart strategy [6]. In this model the `shmem_checkpoint_all()` method is used to copy all shared (symmetric) data to another node. The method needs to be called on all nodes. Then, in the case of a failure, the failed nodes can be restarted and the job of all nodes can be started from the latest checkpoint. The main disadvantage is the cost of synchronization of all nodes on every checkpoint which grows at least logarithmically with the number of nodes.

2.2 Resilience for X10

The X10 programming language [9] has been developed as a simple, clean and practical programming model for large scale computation. It uses PGAS programming model organized around the notion of *places* and *asynchrony*. The fault tolerance policy for X10 language has been recently developed and called Resilient X10 [10]. The Resilient X10 precisely defines the semantics of termination detection in the presence of failures. This makes it possible to continue executing in spite of failures without reverting to a check-point while preserving the execution order of all surviving tasks.

The Resilient X10 introduces resilient storage that allows access to nodes data even if the node has failed and enables the programmer to handle failures by throwing *DeadPlaceException* when a node fails.

The Resilient X10 has been used in three ways. One of them is replaying from disk which is appropriate when there is a large amount of immutable state that can be recovered from the original input file. The second one, decimation, is appropriate for applications where an approximate result is acceptable. Finally, if there is a large amount of mutable state then a resilient store can be used to allow a state to be checkpointed in memory at neighboring nodes.

3 PCJ Library

The PCJ library [11] is a Java library for parallel and distributed computing based on the PGAS programming model. The outstanding benefits of the library are, among others, the ability to use pure Java language to implement HPC algorithms and easy shipment. Programs that use the library use a standard javac compiler and can leverage full support from Java IDEs. Moreover, the library is shipped as a single jar file and does not depend on any third party libraries, which makes deployment of programs trivial.

The library provides methods to perform basic operations like synchronization of tasks, getting and putting values in an asynchronous one-sided way. The library offers methods for creating groups of tasks, broadcasting, and monitoring variables. The PCJ has the ability to work on the multinode multicore systems hiding details of inter- and intra-node communication.

The smallest unit of computation in PCJ is called a *thread*. Each *thread* has its own local address space and shared address space. The owner *thread* accesses local and global address spaces variables as usual Java fields. Multiple *threads* may be run on one physical machine, within a single Java Virtual Machine (JVM). A JVM that is running group of *threads* is called a *node*. PCJ distinguishes *node 0* which is the start node for all the computations.

The PCJ library provides methods for one-sided asynchronous data transfer between nodes e.g. *put(), getFutureObject()*, synchronous data transfer (e.g. *get()* as well as methods for execution synchronization (*barrier()*). Below we present functionality important for the resilience implementation. The detailed description of the PCJ library can be found elsewhere [8].

The described changes are implemented and tested in PCJ version 4.1.

3.1 Shared Memory in the PCJ

Variables that should be put into shared address space have to be annotated with `@Shared` annotation. The shared fields are accessible from other threads with the help of *put()* and *get()* methods:

- `get(threadId, variableName)` and `getFutureObject(threadId, variableName)` are used to read the value of the variable `variableName`,

- `put(threadId, variableName, value)` is used for asynchronous update of the variable `value` in a given thread,
- `broadcast(variableName, value)` is for the asynchronous update of the variable `variableName` with the given value. The operation is performed for all threads. In order to maximize performance, the `broadcast` does not update the variable value one by one but uses communication over a binary tree. The underlying data transfer is organized in a balanced binary tree which means that the update will reach each node in at most *O(log n)* steps (n is the number of nodes used for execution).

Since the above methods are asynchronous, the `waitFor(variableName)` and `monitor(variableName)` methods are used to lock the current thread until other thread updates given variable on this node.

3.2 Synchronization in the PCJ

Threads synchronization in the PCJ is realized by the `barrier()` method. The barrier can be made two threads or all of them. In the second case, each task has to call `barrier` method. The barrier is managed by the *node 0* and is realized in two steps. First, each node informs the *node 0* that it has reached the `barrier` and starts waiting. When *node 0* gathers barrier confirmations from all nodes, it releases the barrier by broadcasting a specific message to all nodes.

4 Fault Tolerant Version of the PCJ Library

PCJ library has notions of node and thread. Fault tolerant extensions focus on nodes.

A node is considered dead if it is not accessible to the other nodes. If a node is found dead, all the threads running on the node are assumed dead.

The main concerns for the fault tolerance extensions for the PCJ library were to minimize the performance overhead and to preserve the compatibility with the original library.

The idea behind the changes was to provide a set of features that would allow a programmer to implement a dedicated recovery solution for the program. The reconfiguration of PCJ internal structures is done automatically and the failures are presented in a convenient manner, but the way to handle them has to be decided by the programmer.

This allowed providing a solution that is flexible and has a minimal performance overhead. In a way, it resembles the User Level Failure Mitigation for the MPI [4].

The implementation relies on an assumption that *node 0* never dies. Although it limits the fault-tolerance of the solution, it is a reasonable compromise. *node 0* is the place where the execution control is performed. A probability of a failure of a selected node is much smaller than a probability of a failure among all nodes.

4.1 The Fault Tolerance API

PCJ version with fault tolerance extensions is backward compatible. Existing programs can be run with it without any modifications.

The existing API is extended with:

- `PCJ.getFailedThreadIds()` method, which returns the identifiers of failed threads. Upon failure, all threads keep their identifiers. The identifiers of failed threads are stored internally by the library and exposed to the programmer via the aforementioned method.
- `NodeFailedException` exception, which can be thrown by any of PCJ methods which rely on the inter-node communication. That is, among others:
  - The `get`/`getFutureObject` methods throw the exception when they try to access data from a failed node. In the case of `get`, the exception is thrown right away. In the case of `getFutureObject`, the exception might be thrown right away or when a program tries to read the result of the future object, depending on the situation.
  - The `waitFor` method throws an exception on any node failure. `waitFor` is used to wait for an update of a variable and PCJ does not have any information which node should update the variable. Once the exception is thrown, the programmer may use `getFailedThreadIds()` method to check if the particular thread is still alive. Then the `waitFor` can be invoked again.
  - The `put` method throws an exception on an attempt to update a variable on a node that failed. The exception is thrown immediately after the method is executed.

  It is a runtime exception. This means the programs written for the previous version of the library do not have to be modified to add a `try-catch` blocks or `throws` declarations to.

4.2 Implementation

(i) Failure detection. Failure detection mechanism is hybrid. It consists of failure propagation of communication exceptions and local node monitoring by heartbeat.

Communication error propagation. TCP is the underlying protocol used by PCJ library. We rely on the reliable nature of the TCP protocol, and we assume that failure to communicate with some node means that the node is dead. I.e., when an IOException is thrown by Java on TCP communication, the error is caught and the target node is assumed failed.

What is important is that the error is not exposed directly to the program. In the case of methods that return an error on a node failure, the execution of the method is paused and only after the appropriate message comes from the *node 0*, the execution is resumed and the exception is thrown. This way we make sure that the error is not reported twice and that the execution is performed in a consistent state.

Heartbeat monitoring. It may happen that a node failure happens when a program does not perform any communication. E.g. during a `barrier` operation. In this case, it is not enough to propagate the communication errors. To handle this situation, heartbeat monitoring has been added.

The PCJ library uses a number of internal messages to perform synchronization and communication between nodes. This list has been extended with a PING message used to monitor if nodes are alive.

To ensure scalability of the heartbeat monitoring, the nodes are monitored locally. As mentioned in Sect. 3.1, PCJ nodes are organized in a fully balanced binary tree.

Heartbeat mechanism leverages the tree structure in the following manner: Each node (but *node 0*) sends a heartbeat message to its parent in the tree (see Fig. 1). Each node, but the leaf nodes, is responsible for monitoring its children nodes. If a parent node does not get a heartbeat message from its child for a long (configurable) time, it assumes the child node is dead.

In this way, we achieve a scalable monitoring mechanism, similar to the ring of observers described in [4].

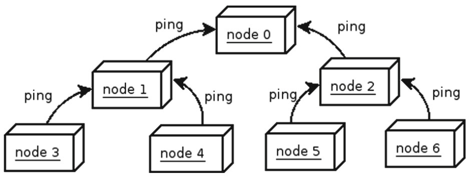


Fig. 1. Node monitoring. Nodes send heartbeat (PING) messages to their parents.

(ii) Reconfiguration. When a node has failed, it has to be removed from all internal PCJ structures. This includes fixing the communication tree described in Sect. 3.1.

Fixing of the internal structures is implemented in the following fashion: Let *node k* denote the failed node. When *node i* discovers that *node k* has failed, it informs *node 0* about the failure. Then the following actions are performed:

1. *Node 0* removes the failed node from the configuration - not to try to send any data to it.
2. If there is a barrier in progress, *node 0* removes the *node k* from the barrier. If *node k* is the last node the barrier is waiting for - the barrier is released.
3. The communication tree is updated. To keep the communication tree fully balanced *node k* is replaced in the communication tree with the rightmost leaf node in the deepest layer. The operation is depicted in Fig. 2. To implement this action, a new message, `MessageNodeRemoved`, was added. Resilient broadcasting is used to deliver the message in a reliable and scalable fashion.

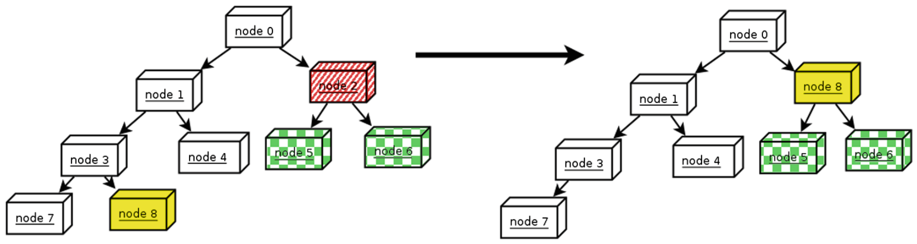


Fig. 2. A schematic view of reconfiguration of the node tree after a node failure.

(iii) Replaying communication. As stated above, the node failure can cause loss of underlying communication. The `barrier` is a good example of such situation.

As described in Sect. 3.2, when the barrier is reached by all threads, *node 0* broadcasts a message that releases the barrier to all nodes. When *node k* fails during broadcast and it has *node i* and *node j* as children, *node i* and *node j* will not get this message and will hang.

To prevent this situation *node l*, which is established a new communication parent for *node i* and *node j*, has to replay the communication. Since a node can never know in advance when it will become a communication parent, every node stores all broadcast communication that it gets. Messages are stored for a configurable amount of time and evicted when the time passes.

When *node i* and *node j* are attached to *node l*, *node l* sends all broadcast messages to *node i* and *node j*. It might happen that a message was already processed on e.g. *node i*. Each message is given a unique identifier. Based on this identifier, the PCJ figures out if the message has been already processed or not. If it has not, the message is processed and sent further in the communication tree. If it has, the message is only passed through.

(iv) Presenting the errors to the user. When `MessageNodeRemoved` is received by a node, it may have no means to present this information right away to the program. E.g. the program may be performing computations and not invoking any PCJ methods at the time.

However, in some cases, such error may have an impact on how the program should work in the future. That is why a failure register had to be introduced.

When a node is informed about another node's failure, it puts this information to the register. Then when a user performs an operation that throws an error on a node failure or asks for failed threads directly, the information about the failure is presented and the register is cleared.

5 Evaluation

In order to evaluate the fault tolerant version of the PCJ library, we have performed performance tests. The tests were performed on 256 nodes of the Okeanos system [12]. The programs were using 256 PCJ nodes, one thread on each node, to maximize the (negative) impact of node monitoring mechanisms.

The tests were performed for the non-fault tolerant version of PCJ, fault tolerant version with no errors during execution and for the fault tolerant version with 1 and 2 node failures. The test used the default settings for node pinging interval (500 ms) and node timeout (5 s).

The test results are presented in Fig. 3.

The first test periodically invoked the barrier operation for 100 000 times. The execution time has been recorded. Without failures, the fault-tolerant version of the library was 3 s slower (158,618 s vs 162,701 s). This means around 2.5% overhead. The overhead is caused by the increased communication between nodes and synchronization on access to node's configuration that had to be introduced to implement fault tolerance extensions. A single failure added 2.5 s to the execution, i.e. the execution time increased by another 2%. When two failures occurred, the overhead was comparable to the single failure. It is probably due to the fact that synchronizing with one node less is slightly faster.

When a node fails, its parent is waiting for the heartbeat messages from that node, and if it does not get it, it assumes that the node is dead. The performance results show that node failure has a small impact on the performance - it did not exceed 3 s in both cases.

Next test was an implementation of π calculation as the integral value. This algorithm required an extra effort to make it fault-tolerant. The chosen solution is based on a queue. At first, the whole interval is divided among all nodes. If some of the nodes fail before the end of the calculations, their work is divided between the nodes that are still alive. Without any node failures, the version with fault tolerance extensions performed as good as the non-fault tolerant version. With node failures, the additional cost was proportionate to the amount of work that had to be replayed.

A similar experiment has been performed for a simple application, namely evaluation of the π using Monte Carlo method. When an error occurs, the failing node is eliminated from the calculations. The final value of the π is calculated based on the data calculated by the threads which finished calculations successfully. The result is, therefore, less accurate. However, the loss of accuracy is minimal, especially for a large number of threads.

The results show that performance overhead of the extensions is negligible.

6 Future Work

Two most important features that we plan to provide in the future are respawning failed threads and manual checkpointing.

Currently, the fault tolerant PCJ does not handle the situation when failed node's job has to be taken into account to calculate the results. The programmer has to implement the code to replay the work on a working node himself. We plan to add a possibility to create new threads on existing nodes to replace the failed threads.

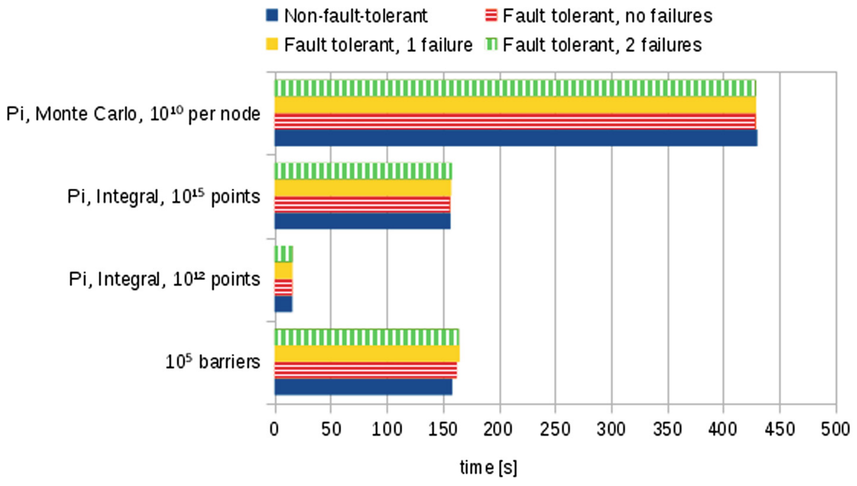


Fig. 3. Performance comparison of the barrier operation and *pi* estimation using integral and Monte Carlo method. The execution time is presented for the following situations: non-fault tolerant PCJ library, fault tolerant with no node failures, fault tolerant with a single node failure and fault tolerant with two node failures.

To replay work from the other node, either on an existent thread or on a new thread it is necessary to start from a state that was already achieved by the failed thread. This requires checkpointing and support for restoring the state of a failed node. Which can be implemented for example through a fault tolerant storage. The data will be shared across physical nodes and will be accessible even if some node failed.

7 Conclusions

In this paper, we have addressed the important problem of the resilience of the large-scale parallel programs. We have designed and implemented fault tolerant version of the PCJ library for the parallel calculations with Java.

The resilience extensions provide a programmer with the basic functionality which allows detecting node failure. For this purposes, the Java exception mechanism is used which allows to detect execution problems and present it to the programmer. The programmer can uptake proper actions to continue program execution. The detailed scenario how to recover from the failure has to be decided and implemented by the programmer. The resilience implementation relies on the assumption that *node 0* does not die, therefore, it can handle the reconfiguration after a node failure. This limitation can be removed later on.

We have performed performance tests of the fault tolerant version of the PCJ library showing that overhead is small and does not exceed 5–6% of the total execution time in the case of heavy communication between nodes. In the example applications, the overhead is significantly reduced.

The use of the fault tolerant version of the PCJ library is simple and requires small but straightforward changes in the application code. At the same time programmer is given high flexibility in the implementation of fault handling.

Acknowledgements. This work has been performed using the PL-Grid infrastructure. The authors would like to thank CHIST-ERA consortium for financial support under HPDCJ project (Polish part funded by NCN grant 2014/14/Z/ST6/00007).

References

1. Xie, X., Fang, X., Hu, S., Wu, D.: Evolution of supercomputers. Front. Comput. Sci. China **4**(4), 428–436 (2010)
2. Rieger, C.G., Gertman, D.I., McQueen, M.A.: Resilient control systems: next generation design research. In: Proceedings of the 2nd Conference on Human System Interactions, HSI 2009, Piscataway, NJ, USA, pp. 629–633. IEEE Press (2009)
3. Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., Snir, M.: Toward exascale resilience: 2014 update. Supercomput. Front. Innov. **1**(1), 5–28 (2014)
4. Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., Dongarra, J.: Failure detection and propagation in HPC systems. In: SC 2016 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Article No. 27 (2016)
5. Ricci, A., Yonezawa, A.: Away from the sequential paradigm tarpit: modelling and programming with actors, concurrent objects and agents. In: Proceedings of the Second International Workshop on Combined Object-Oriented Modelling and Programming Languages, pp. 1–6. ACM (2013)
6. Hao, P., Shamis, P., Venkata, M.G., Pophale, S., Welch, A., Poole, S., Chapman, B.: Fault Tolerance for OpenSHMEM, PGAS/OUG14. http://nic.uoregon.edu/pgas14/oug_submissions/oug2014_submission_12.pdf. Accessed 28 May 2017
7. Nowicki, M., Bała, P.: Parallel computations in Java with PCJ library. In: Smari, W.W., Zeljkovic, V., (eds.) International Conference on High Performance Computing and Simulation (HPCS), pp. 381–387. IEEE (2012)
8. Nowicki, M., Górski, Ł., Grabarczyk, P., Bała, P.: PCJ - Java library for high performance computing in PGAS model. In: Smari, W.W., Zeljkovic, V., (eds.) International Conference on High Performance Computing and Simulation (HPCS), pp. 202–209. IEEE (2014)
9. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOP-SLA 2005, pp. 519–538 (2005)
10. Cunningham, D., Grove, D., Herta, B., Iyengar, A., Kawachiya, K., Murata, H., Saraswat, V., Takeuchi, M., Tardieu, O.: Resilient X10. Efficient failure-aware programming. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 67–80. ACM (2014)
11. http://pcj.icm.edu.pl. Accessed 4 Apr 2017
12. https://kdm.icm.edu.pl/kdm/Okeanos/en. Accessed 28 May 2017

Exploring Graph Analytics with the PCJ Toolbox

Roxana Istrate(✉), Panagiotis Kl. Barkoutsos, Michele Dolfi, Peter W. J. Staar, and Costas Bekas

IBM Research - Zurich Research Laboratory, Rüschlikon, Switzerland
{roi,bpa,dol,taa,bek}@zurich.ibm.com

Abstract. Graph analysis is an intrinsic tool embedded in the big data domain. The demand in processing of bigger and bigger graphs requires highly efficient and parallel applications. In this work we explore the possibility of employing the new PCJ library for distributed calculations in Java. We apply the toolbox to sparse matrix matrix multiplications and the k-means clustering problem. We benchmark the strong scaling performance against an equivalent C++/MPI implementation. Our benchmarks found comparable good scaling results for algorithms using mainly local point-to-point communications, and exposed the potential for logarithmic collective operations directly available in the PCJ library. Further more, we also experienced an improvement of development time to solution, as a result of the high level abstractions provided by Java and PCJ.

Keywords: PCJ · MPI · PGAS · Java · C++ · SPMM · Sparse K-means · Graph analytics

1 Introduction

Graphs are modelling tools used in numerous domains such as biology, computer science, mathematics, physics and many others. Their properties can reveal relationships, similarities, or discrepancies between graph items, which would be inaccessible with no structural data representation. In computer science, graphs are used to represent networks of communication, flow of computation, and data organization. In natural language processing, dependency graphs provide simple descriptions of the grammatical relations in a sentence. These descriptions help people without linguistic expertise disambiguate the meaning of the sentence when they are only interested in extracting textual relations.

At the core of graph analytics are path analysis, community analysis, and centrality analysis algorithms. Path analysis refers to determining the shortest distance between two nodes in a graph. For example, this algorithm is employed when performing route optimisation in smart cities. Community analysis finds groups of people in a social network that share similar interests. One widely known use case is the “other customers were also interested in” syntagm. Centrality analysis focuses on identifying the most influential node in a network. This can indicate the most dominant person in a social network, the highest accessed web page, the most inspiring scientific article.

Since most of the graph analytics applications are data hungry and data sizes are growing at unprecedented pace, focusing our attention towards distributed frameworks comes as a natural consequence. To easily port old algorithms from single- to multi-node implementations, a distributed framework should allow the programmer to focus on algorithm specific challenges, by hiding details such as communication, synchronization, fault tolerance. In this way the time to solution can be considerably shortened.

Implementation of graph analytics on distributed machines via high-level programming languages is intriguing. The high computational requirements of these algorithms, make the implementation more suitable for powerful, low-level and fast languages like C/C++ employing message passing interface (MPI) libraries for the distributed communication. However, the evolution of heterogeneous computing systems redirects development to large shared memory architectures. The PGAS (Partioned Global Address Space) [8] is the natural extension of this paradigm on multiple nodes. Thus combining portable programming languages, such as Java, with these communication patterns and evaluating the performance is a very interesting point of research.

Java is one of the leading programming languages used in academia and industry. The major aspects of portability and versatility, as well as the large ecosystem of tools and libraries that makes it a complete framework, are some factors that make Java an appealing language for implementation of graph analytics. Performing parallel and distributed operations using multiple processes on the same or multiple hosts has been substantially simplified by the PCJ Java library [6]. The library implements PGAS model without modifying the language syntax, making it easy to use. Being highly scalable, the programmer is more involved in the algorithmic part of the program, achieving higher code reliability and faster development.

1.1 PCJ Library and Related Work

PCJ is an actively developed library which allows programming in multi-node systems connected using standard TCP sockets [1]. The distribution model used is based on the PGAS model, allowing only one-sided communication (which in a general case is easier in comparison with double-sided message parsing of MPI) with all communication details hidden. The PGAS paradigm for distributed computing creates dedicated areas for storage for each thread and the programmer has the option to partially (or even fully) share information among threads.

PCJ is based on standard distribution of Java (requiring at least Java SE8 for the latest PCJ 5 version) and no further libraries are required. Programming-wise PCJ provides the basic tools for distributed tasks such as synchronization tasks, asynchronous one-sided communication, broadcasting etc. Every task translates into a PCJ thread, having by default only local private variables. PCJ threads share variables using a PCJ specific annotation syntax.

The first performance evaluation of the PCJ library for graph analytics problems benchmarked two of the Graph500 kernels: generate and compress a graph into sparse structures (kernel 1), and compute the level-synchronous BFS search of some random vertices (kernel 2) [9,10]. In the two studies the authors compare the performance of the PCJ library against existing efficient MPI implementations, obtaining good results. The performance discrepancy against the MPI application was attributed to the different communication paradigms employed by the two solutions.

A recent publication by Nowicki et al. [7] has shown the potential of the PCJ library to scale on more than 100'000 cores for a 2D stencil code based on the Game of Life. The techniques employed for this project have provided substantial input for the improvement of the of the library functionality.

In this work we continue the series of graph analytics benchmarks. We will focus on the PCJ implementation of the Sparse Matrix Matrix multiplication (SPMM) kernel and the k-means clustering algorithm. The former is used for complex graph calculations, such as for centrality analytics [2,3,11], while the latter is an essential ingredient for unsupervised clustering analytics.

The choice of the two problems is motivated by their different communication patterns. With the first example we focus on point-to-point local communications to the neighbouring processes, whereas the second application targets many global collective reductions enforcing higher synchronization. Further more, in order to obtain a direct comparison between the Java/PCJ and the C++/MPI application, we implement the same algorithm with both programming environments.

This paper is organized as follows: the next section introduces the algorithms used for the implementation of the two graph analytics kernels. Results of the scaling behavior and the comparison between the two implementations is presented in Sect. 3. Finally conclusions of this work and future outlook are highlighted.

2 Algorithms

For both proposed applications we describe its standard sequential algorithms and how these have been distributed to the multi-node architecture. Based on the algorithmic complexity and the communication patterns, we estimate the scaling of the maximum speedup, given its problem size.

2.1 SPMM

Standard algorithm. The SPMM kernel is given by the $C = AB$ equation, where A is an $n * m$ matrix, B is an $m * k$ matrix, and C is an $n * k$ matrix. The equation is identical to the more common dense matrix matrix kernel, but the matrix A composition and data representation are completely different. While in dense matrix matrix multiplication, the frequency of zero elements in the A matrix is low, in the sparse kernel most of the A matrix elements are zero. Because storing the entire A matrix is wasteful, we adopted the CSR encoding [12]. CSR consumes $O(nnz)$ memory, as compared to $O(n * m)$ required by the full matrix storage, where nnz denotes the number of non-zero elements. Furthermore, we will present the case where the matrices do not fit the memory of a single compute node, and the regular multiplication has to be adapted to work in a distributed environment.

Distributed version. Before detailing the distribution algorithm, it is worth mentioning that no matrix is fully stored in only one compute node during the multiplication process.

The distribution algorithm begins by dividing the A, B and C matrices along rows. In our experiments we considered B matrices with many more rows than columns ($m \gg k$), thus making the blocking along columns unfeasible from the memory limitations point of view. For simplicity reasons, we choose the number of blocks to be equal to the number of processes and the size of each block to be the same among processes. In case of unbalanced sparsity of the A matrix, or in case of heterogeneous hardware, these constraints can be easily adapted without affecting the rest of the algorithm.

Since the multiplication between two matrices involves dot products between each row of matrix A with each column of matrix B, and since matrix B is distributed among rows, this implies that by the end of the multiplication, each node needs access to all the other remote B blocks. In the naive approach the compute nodes are arranged in a mesh topology, where each node communicates with each other node to obtain the remote B blocks. With a simple trick, we reduce the network topology to a list, where each node sends its local B block to the next node in the list, and receives another B block from the previous node in the list. If we consider the list of nodes $\{N_0, N_1 \ldots, N_{p-1}\}$, we say that N_0's previous node is node N_{p-1}, and N_{p-1}'s next node is N_0. By cycling around p times the B blocks, each node computes p local multiplications between its local block A, the currently received block B and it accumulates the result in the local block C. The final result is distributed across the nodes, and obtaining the result matrix involves a gather operation.

By evaluating the complexity and the communication requirements of the distributed algorithm we estimate the maximum theoretical scaling. In the case of SPMM kernel, the scaling for the distributed algorithm is $\mathcal{O}\left(\frac{\mathrm{nnz}}{2n}\right)$, where nnz is the number of non-zero elements and n is the matrix size.

2.2 *k*-means Clustering

Standard algorithm. k-means is a clustering algorithm that groups n data points into k clusters. Considering the points $\{P_1, P_2, \ldots, P_n\}$ and the clusters $\{C_1, C_2, \ldots, C_k\}$, point P_i belongs to cluster $C = \mathrm{argmin}_k \|P_i - m_{C_k}\|$, where m_{C_k} is the mean of cluster C_k, and $\|P_i - m_{C_k}\|$ denotes the Euclidean distance between the point P and the centroid of the cluster m_{C_k}.

The algorithm initiates by randomly choosing k centroids from the set of points. It continues by assigning each point to the cluster of the nearest centroid. Mathematically, this means partitioning the points according to the Voronoi diagram generated by the centroids. After the points were divided in k clusters, the centroid position is recalculated as the center of the points assigned to each of the centroids of the previous step of the algorithm. Then the points are reassigned and the steps are iteratively repeated until the centroids do not change position or a maximum number of iterations is reached [5].

k-means belongs to the NP-Hard class of problems, which means that the optimal solution can not be reached in polynomial time, but for a large number of iterations the result can be driven to an approximate solution of a certain error.

Distributed version. The distribution algorithm begins by dividing the group of points among the available compute nodes. For simplicity reasons, we considered a balanced division, but this constraint can be easily modified without affecting the rest of the algorithm. Each compute node, besides the local points, owns a copy of the list of current centroids and begins by assigning every local point to its closest centroid. The master node gathers the summaries from all the other nodes, computes the new centroids, and broadcasts the centroids to the rest of the cluster.

It is worth mentioning that the summaries send by each node to the master node represent compressed information in the size of the number of centroids, not in the number of local points. Therefore, the communication requirements are upper limited by O(k). Because no compute node stores at any point in the algorithm the entire dataset of points, the algorithm can be scaled up to extremely large datasets.

The complexity and the computational intensity of the distributed algorithm indicate the theoretical scaling of the problem. For distributed k-means the expected scaling is $\mathcal{O}\left(\frac{n}{p}\right)$, with n the number of data points and p the number of processes.

3 Results

In this section, we present the performance analysis of the two applications, *i.e.*, SPMM and k-means, using the two different environment, Java/PCJ and C++/MPI, that were introduced in the previous sections. In order to better understand the scaling features and to provide a fair comparison, we don't rely on any pre-existing library and we implement the same version of the algorithm by exploiting just the *out-of-the-box* features provided by PCJ and MPI, respectively.

Our benchmarks present comparison between the two frameworks in terms of speedup with increasing the number of processes, speedup when varying the problem size, and actual time to solution.

Benchmarks are performed on an IBM POWER8 cluster. Each node is provisioned with highly balanced bandwidth between all the computational components that are optimized for data intensive analytics.

We developed CPU-based codes that we scale up to four nodes, for a total of 80 physical cores. The resource allocation is scheduled in a sequential order, *i.e.*, first we bind the processes to all the cores of a single node before starting to use the second node[1]. We have used the 64-bit version of the Java OpenJDK 1.8.0 with PCJ 5.0.4 and GCC 5.3.1 with OpenMPI 2.1.1. The C++ code is compiled with the `-O3` optimization flag.

3.1 Datasets

To simplify the scale up of the problem size during the benchmark process, we rely on synthetic data which is generated in a distributed way upon initialization. However, our example applications are already capable to operate on real datasets, *e.g.* the classification datasets available in [4].

Randomly initialized matrices. For the SPMM benchmarks we build randomly initialized sparse matrices with a homogeneously distributed density. For each matrix block we initialize a random number generator with a unique seed on the corresponding execution process. For the input dense vector v we initialize all entries to 1.

Random clusters. The input of the k-means application is a set of homogeneously distributed points in the d-dimensional feature space. After initialization, each executing process is generating its own independent list of points. The initial position of the centroids is generated only by the master process and broadcasted to all processes.

3.2 Benchmarks of the Algorithms

Figure 1 provides the scaling results for the SPMM problem. The matrix size and its density have been scaled up to $m = 96\text{'}000$ with a density varying from $d = 0.2$ to $d = 0.6$, corresponding to ca. 80 GB of memory needed only to store the matrix A. The MPI benchmark shows that the application has the potential to scale for the given system sizes, even though, for the smallest dataset and a large number of processes, one observes a small decrease in the scaling, as expected from the analytic algorithm performance. By increasing the problem size the communication cost can be better overlapped with the larger computation. At a first impression, the PCJ results do not seem to provide the expected scaling. Further investigation shows that the performance degradation is not linked to the PCJ framework, but with the difficulty of benchmarking short operations within a JVM. For a large number of processes, a single step in the distributed multiplication runs for less than 100 ms. Effectively, internal Java mechanisms

[1] This is specially relevant in the one-dimensional domain decomposition used in the SPMM algorithm.

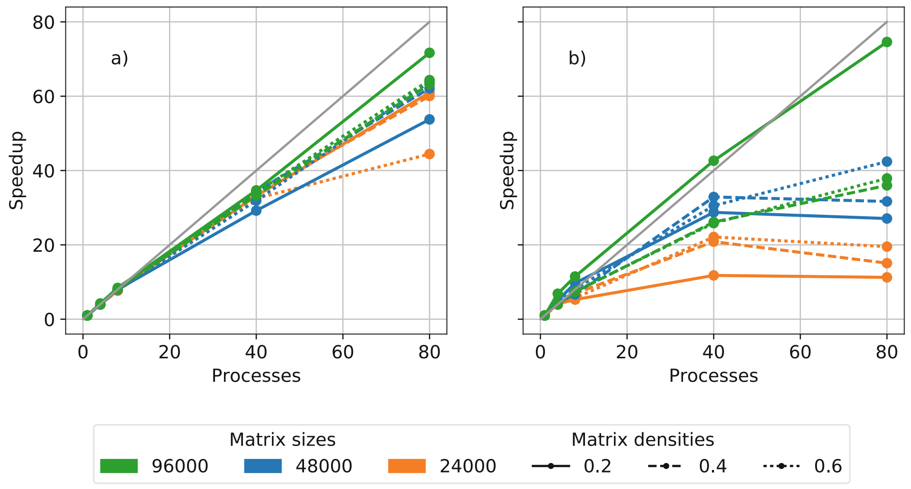


Fig. 1. SPMM strong scaling for multiple matrix sizes and densities. (a) Results using the C++/MPI application. (b) Results using the Java/PCJ application. Note that the Java benchmarks are affected by timing issues (see main discussion).

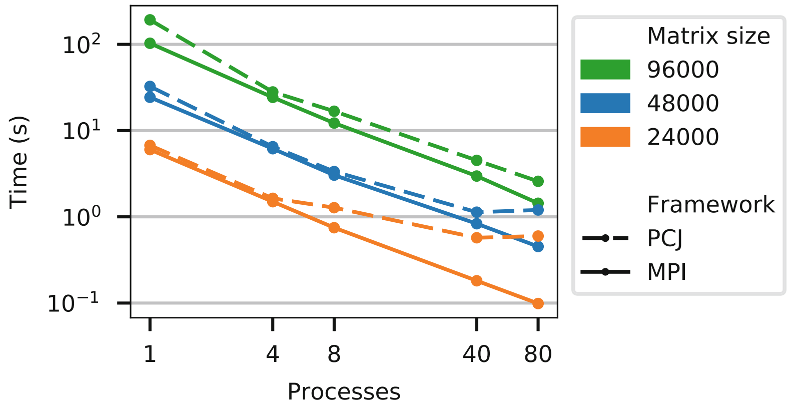


Fig. 2. SPMM - time to solution comparison between PCJ and MPI. The matrices involved in the multiplication have a constant density of 0.2. This plot is in log-log scale.

such as the garbage collection and the just-in-time compiler bring a significant overhead. This is even clearer when analyzing the absolute timings (Fig. 2). The super-scaling observed for the largest matrix $m = 96'000$ and $d = 0.2$ is due to cache optimization. Because of a warm up execution, the matrix blocks are already loaded into the faster cache memory.

The scaling results for the k-means application are presented in Fig. 3. The number of points in the dataset varies from 2.4 M to 480 M, each having $n_f = 2$

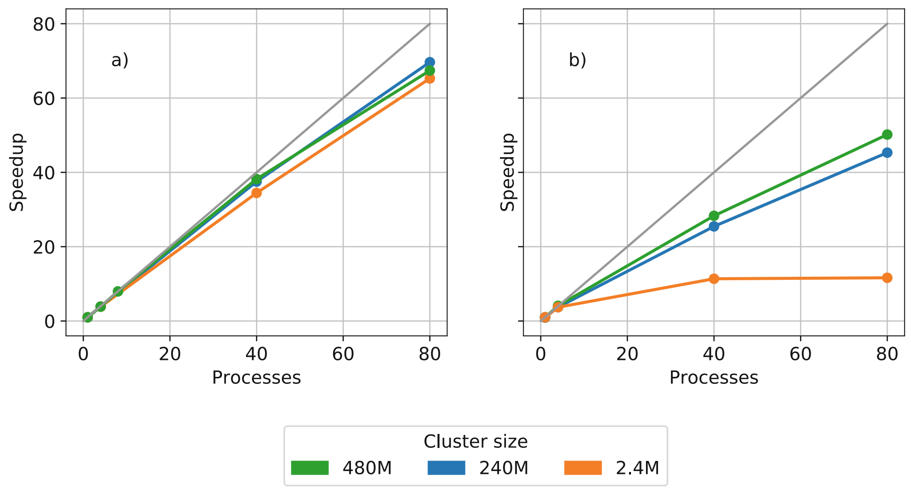


Fig. 3. k-means strong scaling for various number of points n and fixed number of features $n_f = 2$. (a) Results using the C++/MPI application. (b) Results using the Java/PCJ application.

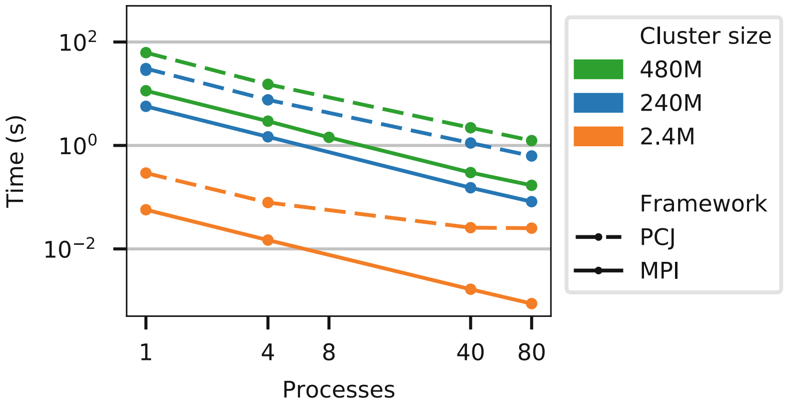


Fig. 4. k-means - comparison of the time per iteration between PCJ and MPI. This plot is in log-log scale.

features and three centroids. The algorithm shows a close to optimal scaling for the MPI implementation, whereas the small system sizes show performance degradation for the PCJ scaling. The latter is an expected result, because MPI profits from the logarithmic collective reductions, which are not currently exposed in the PCJ API.

Similar results are obtained when comparing the absolute timings for the time to solution (Figs. 2 and 4). For large problem sizes, the scaling of both frameworks is very similar, but for smaller problems we encounter timing issues, because of which the measured time of SPMM with PCJ saturates at the constant overhead caused by Java internal mechanisms. As expected, the low level optimizations in the C++ codes achieve 2–3 faster execution compared to the Java implementation. The stronger difference in the timings of the k-means application reflects the (current) different algorithmic complexity.

4 Conclusions and Outlook

In this work we implement and benchmark two graph analytic kernels. The results show a promising strong scaling for the Java/PCJ implementation that, for larger system sizes, reaches the same performance as the low-level C++/MPI application. Within the context of the two different kernels we find that the PCJ library provides excellent *out-of-the-box* performance for local communications, whereas the global communications pattern would highly profit from generic collective communications employing logarithmic complexity. Regarding absolute timings the Java code is found to be between 2–3 times slower than the C++ version. This difference mostly originates from the inner kernel functions and not from the communication side. The gap is expected to be much smaller if both implementation would execute the inner kernels on modern accelerators.

However, the large benefit of the PCJ library lies in the reduced development effort as a result of the abstraction mechanisms available in the Java language. As an insight, it took very little effort for a first-time user to extend a basic PCJ skeleton code into a fully working parallel application. Given the easy integration into existing frameworks and applications used both in academia and industry, the library has the potential to provide a valid alternative to low-level C++ codes.

The further development of the demonstrative benchmark applications is the evolution into production application that would, e.g., serve a distributed graph database. A graph is efficiently described by its adjacency matrix A, *i.e.*, a sparse matrix with $A_{ij} \neq 0$ when there is an edge between the node i and the node j. Our benchmark targets already the analysis of large graphs that don't fit on the memory of a single machine. This is for example the case of typical social network graphs with billions of nodes and edges as well as a representation of the links between webpages [4].

Complex graph analytics, such as a centrality analysis, or simpler traversal operations, such as retrieving neighbouring nodes, are based on the SPMM kernel, hence the current benchmark applications provide already the core of a scalable graph database implemented using the PCJ library.

Acknowledgements. The authors wish to thank Piotr Bała and Marek Nowicki for driving the development of the PCJ library and for fruitful discussions and debugging. This work was partial supported by the CHIST-ERA consortium.

References

1. Parallel computing in Java. https://pcj.icm.edu.pl
2. Estrada, E.: Subgraph centrality in complex networks. Phys. Rev. E **71**(5), 056103 (2005)
3. Estrada, E., et al.: Network properties revealed through matrix functions. SIAM Rev. **52**(4), 696–714 (2010)
4. Leskovec, J., Krevl, A.: Snap datasets: Stanford large network dataset collection (2014). http://snap.stanford.edu/data
5. Lloyd, S.: Least squares quantization in PCM. IEEE Trans. Inf. Theory **28**(2), 129–137 (1982)
6. Nowicki, M., Górski, Ł., Grabrczyk, P., Bala, P.: PCJ - Java library for high performance computing in PGAS model. In: 2014 International Conference on High Performance Computing Simulation (HPCS), pp. 202–209, July 2014. https://doi.org/10.1109/HPCSim.2014.6903687
7. Nowicki, M., Bzhalava, D., Bała, P.: Massively parallel sequence alignment with BLAST through work distribution implemented using PCJ library. In: Ibrahim, S., Choo, K.-K.R., Yan, Z., Pedrycz, W. (eds.) ICA3PP 2017. LNCS, vol. 10393, pp. 503–512. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65482-9_36
8. Ropo, M., Westerholm, J., Dongarra, J. (eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03770-2. ISBN: 978-3-642-03769-6
9. Ryczkowska, M., Nowicki, M., Bala, P.: The performance evaluation of the Java implementation of Graph500. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 221–230. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_21
10. Ryczkowska, M., Nowicki, M., Bała, P.: Level-synchronous BFS algorithm implemented in Java using PCJ library. In: 2016 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 596–601 (2016)
11. Staar, P.W.J., Barkoutsos, P.K., Istrate, R., Malossi, A.C.I., Tavernelli, I., Moll, N., Giefers, H., Hagleitner, C., Bekas, C., Curioni, A.: Stochastic matrix-function estimators: scalable big-data kernels with high performance. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 812–821 (2016). https://doi.org/10.1109/IPDPS.2016.34
12. Tinney, W.F., Walker, J.W.: Direct solutions of sparse network equations by optimally ordered triangular factorization. Proc. IEEE **55**(11), 1801–1809 (1967)

Big Data Analytics in Java with PCJ Library: Performance Comparison with Hadoop

Marek Nowicki[1], Magdalena Ryczkowska[1,2], Łukasz Górski[1,2], and Piotr Bala[2](✉)

[1] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Chopina 12/18, 87-100 Torun, Poland
faramir@mat.umk.pl

[2] Interdisciplinary Centre for Mathematical and Computational Modeling, University of Warsaw, Pawinskiego 5a, 02-106 Warsaw, Poland
{lgorski,gdama,bala}@icm.edu.pl

Abstract. The focus of this article is to present Big Data analytics using Java and PCJ library. The PCJ library is an award-winning library for development of parallel codes using PGAS programming paradigm. The PCJ can be used for easy implementation of the different algorithms, including ones used for Big Data processing. In this paper, we present performance results for standard benchmarks covering different types of applications from computational intensive, through traditional map-reduce up to communication intensive. The performance is compared to one achieved on the same hardware but using Hadoop. The PCJ implementation has been used with both local file system and HDFS. The code written with the PCJ can be developed much faster as it requires a smaller number of libraries used. Our results show that applications developed with the PCJ library are much faster compare to Hadoop implementation.

Keywords: Big Data · Java · Parallel computing · Hadoop

1 Introduction

The concept of big data has been around for years but nowadays organizations understand that if they capture all the data that streams into their businesses, they can apply analytics and get significant value from it. The new benefits that big data analytics brings to the table are speed and efficiency. Whereas a few years ago one would have gathered information, run analytics and unearthed information that could be used for future decisions, today there is a strong need for immediate decisions. Such approach requires parallel processing of data necessary to provide results in short time.

For the analysis of large-scale data one can use sampling, data condensation, density-based approaches, grid-based approaches, divide and conquer, incremental learning, and distributed computing [1]. The focus on big data analytics

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 318–327, 2018.
https://doi.org/10.1007/978-3-319-78054-2_30

resulted in the development of dedicated algorithms and software tools. The most popular is MapReduce algorithm [2] which became synonymous with big data processing. Different flavors of it are available over Apache Hadoop [3] being at the core of big data processing. It is important to note that the primary Hadoop MapReduce application programming interfaces (APIs) are mainly called from Java. In addition, advanced skills are needed for development and maintenance.

Recently, the new approach to the parallel processing has been presented by the PCJ library [4,5]. It is Java library which allows for easy development of highly scalable applications using PGAS (Partitioned Global Address Space) programming paradigm. Although the main focus of PCJ library was on HPC systems, it can be also used for big data processing.

The aim of this paper is performance comparison of the Hadoop and PCJ software frameworks for the standard benchmarks running on the same infrastructure. The selected benchmarks cover different types of applications from computational intensive, through traditional map-reduce up to communication intensive. The experimental setup is described in the Sect. 2. The Sect. 3 presents details of the performance evaluation and provides results. Next section compares coding effectiveness for Hadoop and PCJ and the following section presents related work. The paper concludes with the conclusions and summary.

2 Experimental Setup

The experimental results for Hadoop and PCJ implementations were obtained on the same hardware system. The strong (fixed global problem size) and weak (the same size of the problem executed by all threads) executions are used depending on the tested algorithm.

2.1 Hardware

For the tests, a dedicated instance of the Hadoop cluster has been used. It consisted of 68 server (computing) nodes, 3 namenodes, and 2 management nodes. All nodes were equipped with the two, 24 core Intel(R) Xeon(R) CPU E5-2680 v3 processors running at 2.50 GHz. Nodes were running CentOS Linux release 7.2.1511. The InfiniBand interconnect was used, the storage was HDFS filesystem with 5.1 TB capacity. The cluster was used exclusively for tests and no other application was running.

2.2 Software

The Cloudera CDH 5.7.5 open source platform distribution, including Apache Hadoop, was installed on the cluster. By integrating Hadoop with other critical open source projects, CDH is a functionally advanced system that helps to perform end-to-end Big Data workflows and minimize administration effort necessary to set up the system. The PCJ library version 5.0.1 has been used to implement word count and π estimator. The BFS algorithm was implemented and tested using PCJ version 4.1. For PCJ and Hadoop, the same version of Java was selected. The 64-bit 1.8.0 version from Oracle has been used. The PCJ applications were started using `PCJ.deploy()` method, which executes ssh command to start an application on computing nodes.

3 Results

The parallel workload can consists of different types of applications (e.g. I/O-intensive, data-intensive and CPU-intensive) [6] which may benefit differently from software platforms such as Hadoop and PCJ. CPU-intensive applications devote most execution time to computational requirements and typically require small volumes of data. CPU-intensive applications executed in parallel can have different characteristics depends on the amount of communication performed. There are applications with the low communication profile such as trivially parallel, or with the significant communication such as graph processing. The I/O-intensive applications require large volumes of data and devote most processing time to I/O. Such applications do not have or have only a few computations and contain only pure write/read operations.

It is important to gain a deep understanding of each type of applications because it can provide guidance to decide the best parallelization platform for a given application to maximize performance and scalability. In this paper, we mainly focus on investigating the performance and scalability of different types of applications parallelization with the Hadoop (MapReduce) and PCJ software platforms.

3.1 Data Intensive

WordCount is a simple piece of code that demonstrates core Hadoop features and basics of programming in MapReduce paradigm. Test program reads an input file line-by-line and counts the number of unique words occurring in each line. Reduction steps gather all computed partial results. Combiners may be used together with Reducers to facilitate in-memory communication and increase overall performance. In the end, a mapping between all the unique words in whole text and number of their occurrences are emitted. No prior text transformations are performed (for example, stop word list and stemming are not utilized), so - depending on input formatting and used word division algorithms - glyphs like punctuation marks and their combinations might be considered a unique word; the same goes for different grammatical forms of the same word.

PCJ code mimics the structure of Hadoop implementation. The word calculations are divided into two steps. Mapping phase utilities the same word-division code that was provided with our Hadoop implementation. Partial results are stored in a shared global variable, unique to every thread of execution. After this phase, a reduction occurs with thread 0 chosen as a root. No overlap between two phases is facilitated. The reduction policies are a major contribution to the overall scalability results and in the case of PCJ three distinct policies were implemented:

- *Reduction 1* - traditional hypercube reduction scheme, suitable for thread counts that are a power of two.
- *Reduction 2* - a 2-step reduction scheme; first step consists of intra-node reduction, in second step thread 0 collects partial results from remote computation nodes.
- *Reduction 3* - a 2-step reduction scheme in which all threads affiliated with **node** 0 performed remote reduction; remote computation nodes were assigned to node 0 threads on a round-robin fashion; after this step intra-node 0 reduction was performed.

Two novels were chosen as a textual corpus for the text. We have used an UTF-8 encoded plain text English translation of Lev Tolstoy's *War and Peace* [7], a file of 3.3 MB, and lesser-known, but nevertheless considerate in length plain ISO 8859-1 encoded text of original French version of Georges de Scudéry's *Artamène ou le Grand Cyrus* [8], one of the longest novels ever written, totaling in 10 MB file size. Different encodings were accounted for in the code. Whilst data sizes itself are quite small, they have been the basis for the weak scalability testing, thus forming a sizable dataset for larger numbers of threads, reaching 52 GB in case of former file and 174 GB for the latter. Other researchers have reached conclusive results in the past using much smaller sets, of 1 GB and 8 GB [9].

During the weak scalability testing, each file was replicated n times, for $n \in \{1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384\}$. This allowed to affiliate each PCJ thread with one input file and was decisive for a number of map operations executed by Hadoop. HDFS filesystem was used in all Hadoop tests, whilst PCJ performance was tested against the node local filesystems as well as HDFS. PCJ threads were distributed so that the smallest number of computing nodes was used, as determined by available hardware threading capabilities. Each PCJ thread read its own copy of input data.

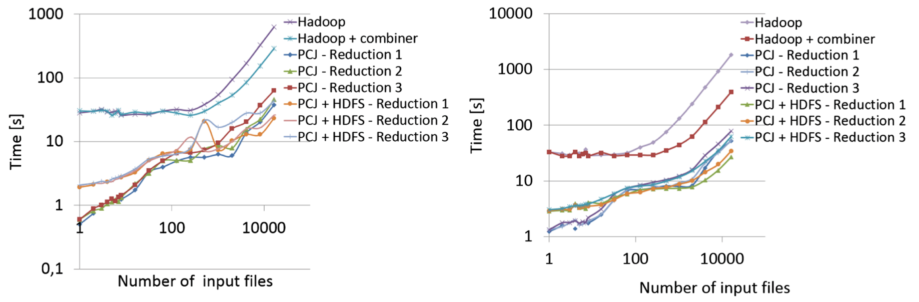


Fig. 1. WordCount weak scalability for 3.3 MB (left) and 10 MB (right) input files. Data for various Hadoop and PCJ implementations are presented.

Timing results for different input size are presented in the Fig. 1. Timings encompass the execution time of `job.waitForCompletion()` method in the case of

Hadoop and measure the total time that thread 0 spent in mapping and reduce phase in the case of PCJ. Input data copying times (from local storage to HDFS and to remote nodes in non-HDFS PCJ version) are not accounted for in the timings.

Generally, PCJ exhibits great scalability and first symptoms of a slowdown are barely visible only when available CPUs are oversubscribed (i.e. when a number of threads is larger than 3264). Overlapping of mapping and reduction steps, as well as the usage of combiners, did not allow Hadoop to achieve similar results and first symptoms of resource exhaustion are exhibited when 1024 input files are processed. On the other hand, the performance of HDFS (excluding the data copying times) was exemplary in the tests and allowed PCJ + HDFS version of the code to achieve even better results. In the case of this implementation, HDFS allowed achieving about two-fold speedup of mapping phase when compared to non-HDFS code (reduction phase times remained constant as they are not IO-bound).

3.2 Computational Intensive

For the computationally intensive workload, we have used a map-reduce program to estimate the value of π using quasi-Monte Carlo method [10]. Mapper generates points in a unit square and then counts points inside/outside of the inscribed circle of the square. Reducer accumulates points inside/outside results from the mappers. Hadoop π example uses Halton sequence which perfectly matches this scenario. It has no duplicate numbers in the sequence and sequence can be as long as needed. The performance data is generated for week scaling where each mapper is using its own part of the sequence of the defined length.

The PCJ implementation follows the same scheme. The work is distributed among PCJ threads and after calculations, the results are summed up over all threads using asynchronous get method.

The performance results were gathered in the weak scaling mode, i.e. with the constant number of generated point for each PCJ thread or Hadoop mapper. The execution time for the 1,000,000 points are presented in the Fig. 2. In the case of Hadoop implementation, the total execution time which accounts for job creation and execution has to be considered, especially for a larger number of jobs.

Both implementations show very good scaling up to 64 PCJ threads/Hadoop mappers which refer to the execution of the single PCJ thread or mapper job on each hardware node. For the higher number of threads, more than one thread/job is run on the single node and reduction time increases due to the more communication performed. For a larger number of threads, the parallel efficiency starts to decrease for both implementations. Depends on the number of threads used, the PCJ implementation is 300–450 times faster than Hadoop.

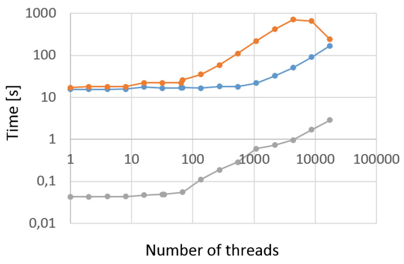


Fig. 2. Execution time for estimate of the value of π in the case of weak scaling. The time for PCJ (lower) and Hadoop (upper) is presented. The total time (execution and job creation) for Hadoop is also shown (top).

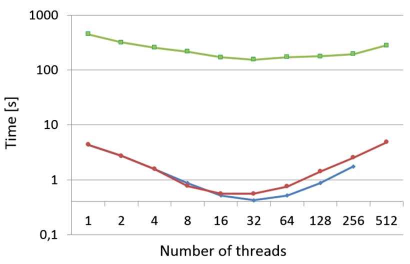


Fig. 3. BFS strong scalability for graph consisting of 2^{22} nodes. Execution time for Hadoop (top) and for PCJ implementations are presented. The red and blue lines are for 8 and 4 threads running on the single physical node.

3.3 Communication Intensive

The important example of the communication intensive algorithm is graph processing, which is used in many fields of science such as sociology, risk prediction or biology. It poses numerous challenges especially for large graphs which have to be processed on multicore systems.

Most of the tools for graph processing is using traditional programming languages such as C/C++. However, the growing adoption of Java as a programming language for the data analytics opens requirement for new scalable solutions. The parallel execution in Java is based on the `Thread` class or fork-join framework available since Jave SE7. All these features can be used within single Java Virtual Machine, which limits parallelization capabilities to the single shared memory node which is not enough for large problems.

BFS as one of the most important graph algorithms has been widely studied. The main idea of our BFS implementation is based on MPI reference simple approach of Graph500 benchmark [11] with synchronization after each level, which has been closely examined in [12]. Most of the algorithms based on the level-synchronized BFS, adopt the idea to either specific programming model or to the environment and present some optimizations to improve performance [13].

Recently there has been developed PGAS (Partitioned Global Address Space) version of the level-synchronous BFS (Breadth First Search) algorithm and it has been implemented in Java using PCJ library which allows running graph processing on multiple nodes [14]. The implementation is based on 1D partitioning: all vertices and edges of the original graph are distributed so that each PCJ thread owns N/p vertices and its incident edges (p is a number of processors and N is a number of vertices in a graph). The distribution of vertices is realized by blocks.

Hadoop implementation of the BFS algorithm is based on [15] The search starts from the root node and the neighboring nodes are visited until there are no more possible nodes to visit. One way of performing the BFS is by coloring the nodes and traversing according to the color of the nodes. There are three possible colors for the node: unvisited, visited and finished. Before the start of the traversal, nodes are colored indicating that all the nodes are unvisited. The source node is colored as visited which indicates that its neighbors should be processed. All the nodes adjacent to a visited node that are unvisited are changed to be visited. The originally visited node is then colored that all its neighbors are visited and the processing of the node is finished. The process continues until there are no more visited nodes to process in the graph. Each iteration can use the previous iteration's output as its input. This kind of iterative MapReduce is useful for applications including graph problems such as parallel breadth-first search. The iteration proceeds using a simple loop condition. Satisfaction of the loop condition can be determined by the mapper, the reducer or the driver.

In order to speed up processing, the `CustomWritable` class was used to read graph data as objects instead of extracting necessary information from strings, which was used in the original code.

Sample graphs (in the form of edge tuple list) - based on Kronecker Graph model together with BFS source vertices - used in performance tests have been generated from Graph Generator of the Graph 500 benchmark. The performance has been tested on graphs of $SCALE = 22$ with $edgefactor = 6$ ($SCALE$ is the logarithm base two of the number of vertices and $edgefactor$ is the ratio of the graph's edge count to its vertex count). In the case of Hadoop, before BFS computation, the generated binary file with edge tuple list had been converted to the adjacency list and split into a different number of binary Hadoop sequence files (`org.apache.hadoop.io.SequenceFile`). The conversion was not accounted for final results.

The execution time presented in the Fig. 3 shows that both implementations show similar scalability - up to 32 threads. For the larger number of threads, the communication time starts to dominate and execution time increases. There is a large difference in the total execution time: the PCJ implementation is more than 100 times faster for the whole range of the threads used.

4 Coding Effectiveness

Coding effectiveness is in general subjective measure and depends significantly on the programmer's skills and experience. However, we can set up a number of commonly used metrics which can be useful for code comparison. One of them is a number of libraries imported for the application for Hadoop and PCJ implementations respectively. The number of Java classes required by PCJ is significantly smaller. The number of lines of the code for the PCJ is usually larger due to the fact that PCJ implementation contains *explicite* code for reduction of the results, while in the Hadoop this task is realized by the classes imported from the library.

The number of PCJ methods used for implementation of the highly scalable code is relatively small and for simple codes does not extend 20. Even for large codes such as BFS, the number of lines of the code which contain calls to PCJ library is less than 10%. This data confirms that PCJ library is efficient and easy to use tool for development of highly scalable applications of different type. Moreover, the parallelization is not limited to the map-reduce scheme and programmer can easily implement any parallel algorithm with few lines of code.

There is, of course, a trade-off between programming the application almost from scratch and framework approach. The framework offers programmer number of tools which simplify application development. The price to pay is the adaptation of the application to an available framework which might not be easy nor efficient. Resulting code, usually smaller compare to programming from scratch and very often difficult to read.

An opposite approach makes fundaments for the PCJ library. The role of the programmer is work with the algorithm, decide on the most efficient parallelization strategy and then use relatively simple tools to implement his ideas in the computer readable form. This might be more difficult for a non-experienced user but leads to more efficient and scalable code. This is especially important for parallel programming where most of the work has to be performed on the algorithm rather than coding itself.

5 Related Work

The performance of Hadoop and Apache Spark attracts significant attention as both frameworks are widely used in Big Data processing [16]. In particular, an improvement of Hadoop performance by the in-memory processing was the motivation for the development of the Apache Spark framework. In results, the 3–4 times better performance compare to Hadoop has been reported [17,18].

A number of optimizations has been performed for Hadoop. Some of them are based on the parameters tuning, other deal with the adaptation of the hardware. For example, to solve the defect of storage of small files, they are merged into a single file which is then stored in the Hadoop filesystem [19]. Another solution is in-storage computing based on the offloading some data processing performed by mapper to the ISC device which allows for about 2 times faster execution [20].

Recent research has focused on the integration of Hadoop and HPC cluster, particularly in the use of the HPC file system replacing HDFS in the Hadoop framework, e.g., [21]. Researchers also tried to seek for substitution for HDFS looking for modern distributed file systems such as GoogleFS [22], PVFS [23] and OrangeFS [24].

6 Conclusions

We have presented Hadoop and PCJ implementation details of the selected applications with the different characteristics: data-intensive, computational and communication intensive. We have compared the performance of the example applications such as word count, π estimation using quasi-Monte Carlo method and BFS search using the level synchronous parallel algorithm. In all cases, PCJ library provides code which scales well and runs faster compare to the Hadoop. The LOC for the PCJ is reasonable, the number of imported libraries is significantly reduced compare to Hadoop and the fraction of the code which contains calls to PCJ library is less than 10%. In many cases, the computational kernel remains the same as for the sequential execution. Presented results confirm that implementations based on PCJ library are much faster than Hadoop ones, even for typical Map-Reduce applications.

In this paper, the Hadoop implementation of selected algorithms was used. The processing can speed up 3–4 times using Apache Spark [17] which does not change the main outcome of the presented work since PCJ implementation is much faster than Hadoop ones. The comparison of the PCJ library with the Apache Spark is ongoing.

Parallel applications using PCJ library can be easy deployed on any infrastructure with Java 8 installed. The deployment is much easier than for Hadoop infrastructure. The Java/PCJ can be therefore used as an interesting solution for HPC and Big Data types of applications on different hardware platforms. It is efficient and easy to use tool which can be used to implement highly scalable parallel algorithms of a various type including, but not limiting to the map-reduce schema.

Acknowledgments. The authors would like to thank CHIST-ERA consortium for financial support under HPDCJ project. The Polish contribution is financed through NCN grant 2014/14/Z/ST6/00007. The performance tests have been performed using ICM University of Warsaw computational facilities.

References

1. Tsai, C.-W., Lai, C.-F., Chao, H.-C., Vasilakos, A.V.: Big data analytics: a survey. J. Big Data **2**, 21 (2015)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In Proceedings of OSDI (2004)
3. Apache Hadoop. http://hadoop.apache.org/. Accessed 20 Sept 2017
4. http://pcj.icm.edu.pl. Accessed 20 Sept 2017
5. Nowicki, M., Górski, Ł., Grabarczyk, P., Bała, P.: PCJ - Java library for high performance computing in PGAS model. In: Smari, W.W., Zeljkovic, V. (eds.) 2014 International Conference on High Performance Computing and Simulation (HPCS), pp. 202–209. IEEE (2014)
6. Li, Z., Shen, H., Ligon, W.B., Denton, J.: An exploration of designing a hybrid scale-up/out hadoop architecture based on performance measurements. IEEE Trans. Parallel Distrib. Syst. **99**, 1–1 (2016)
7. Tolstoy, L.: War and Peace. Random House, Newyork (2016)
8. de Scudéry, M.: Artamène ou le grand Cyrus (1972)

9. Ibrahim, S., Jin, H., Lu, L., Qi, L., Wu, S., Shi, X.: Evaluating mapreduce on virtual machines: the hadoop case. In: Jaatun, M.G., Zhao, G., Rong, C. (eds.) CloudCom 2009. LNCS, vol. 5931, pp. 519–528. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10665-1_47
10. https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/examples/PiEstimator.html. Accessed 10 Feb 2017
11. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the graph 500. Cray Users' Group (CUG) **19**, 45–74 (2010)
12. Ueno, K., Suzumura, T.: Highly scalable graph search for the Graph500 benchmark, In: Proceedings of the 21st International ACM Symposium on High-Performance Parallel and Distributed Computing, pp. 149–160 (2012)
13. Buluc, A., Madduri, K.: Parallel breadth-first search on distributed memory systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011)
14. Ryczkowska, M., Nowicki, M., Bała, P.: Level-synchronous BFS algorithm implemented in Java using PCJ library. In: 2016 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, pp. 596–601 (2016)
15. https://hadooptutorial.wikispaces.com/Iterative+MapReduce+and+Counters. Accessed 21 Mar 2017
16. Li, Z., Shen, H., Denton, J., Ligon, W.: Comparing application performance on HPC-based hadoop platforms with local storage and dedicated storage. In: 2016 IEEE International Conference on Big Data (Big Data), pp. 233–242 (2016)
17. Augustine, D.P., Raj, P.: Performance evaluation of parallel genetic algorithm for brain MRI segmentation in hadoop and spark. Indian J. Sci. Technol. (2016). http://www.indjst.org/index.php/indjst/article/view/91373. Accessed 24 Mar 2017
18. Islam, N.S., Wasi-ur-Rahman, M., Lu, X., Panda, D.K.D.K.: Efficient data access strategies for Hadoop and Spark on HPC cluster with heterogeneous storage. In: 2016 IEEE International Conference on Big Data (Big Data), Washington, DC, pp. 223–232 (2016)
19. He, H., Du, Z., Zhang, W., Chen, A.: Optimization strategy of Hadoop small file storage for big data in healthcare. J. Supercomput. **72**, 3696–3707 (2016)
20. Park, D., Wang, J., Kee, Y.S.: In-storage computing for Hadoop mapreduce framework: challenges and possibilities. IEEE Trans. Comput. **PP**(99), 1–1 (2016)
21. Maltzahn, C., Molina-Estolano, E., Khurana, A., Nelson, A., Brandt, S., Weil, S.: Ceph as a scalable alternative to the Hadoop Distributed File System. The USENIX Mag. **4**(35), 518–529 (2010)
22. Ghemawat, S., Gobioff, H., Leung, S.-T.: The google file system. In Proceedings of ACM SOSP (2003)
23. Carsn, P.H., Ligon, W.B., Ross, R.B., Thakur, R.: PVFS: a parallel file system for linux clusters (2000)
24. Yang, S., Ligon, W., Quarles, E.: Scalable distributed directory implementation on orange file system. In: Proceedings of SNAPI (2011)

Performance Comparison of Graph BFS Implemented in MapReduce and PGAS Programming Models

Magdalena Ryczkowska[1,2](✉) and Marek Nowicki[2]

[1] Interdisciplinary Centre for Mathematical and Computational Modeling, University of Warsaw, Pawinskiego 5a, 02-106 Warsaw, Poland
gdama@icm.edu.pl

[2] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Chopina 12/18, 87-100 Torun, Poland
{gdama,faramir}@mat.umk.pl

Abstract. Computations based on graphs are very common problems but complexity, increasing size of analyzed graphs and a huge amount of communication make this analysis a challenging task. In this paper, we present a comparison of two parallel BFS (Breath-First Search) implementations: MapReduce run on Hadoop infrastructure and in PGAS (Partitioned Global Address Space) model. The latter implementation has been developed with the help of the PCJ (Parallel Computations in Java) - a library for parallel and distributed computations in Java. Both implementations realize the level synchronous strategy - Hadoop algorithm assumes iterative MapReduce jobs, whereas PCJ uses explicit synchronization after each level. The scalability of both solutions is similar. However, the PCJ implementation is much faster (about 100 times) than the MapReduce Hadoop solution.

Keywords: High performance computing · Hadoop · MapReduce
PGAS · Parallel and distributed computation
Performance evaluation · Parallel graph algorithms · Java

1 Introduction

The demand on increasingly faster data processing resulted in creating dedicated tools and algorithms. One of the most widespread ideas focused on big data analysis is MapReduce [1] model together with open-source Apache Hadoop platform [2]. Hadoop application programming interfaces are mainly based on Java language. The big advantage of this tool is fault tolerance and ability to keep thousands of terabytes of data on distributed file system. All those features make MapReaduce and Hadoop an interesting solution. However, gaining high performance in some sort of problems might be a huge challenge.

One of the new, promising solutions for parallel and distributed computations in Java language is PCJ (Parallel Computations in Java) library [3–5]. PCJ is a library that allows developing applications in pure Java language, which does not require any language extensions or special compiler. It is based on the PGAS (Partitioned Global Address Space) model, with all communication details like threads administration or network programming hidden. The communication in the model is one-sided. All those features make programming simpler together with high performance preserved.

PCJ is still being upgraded. At the beginning PCJ was written using Java 7, later it was upgraded to use Java 8. Recently a new version 5 of the library has been released, with code fixes and performance boost. Currently, work is being carried out on further important features like fault tolerance.

The goal of this paper is to compare performance comparison of the BFS algorithm on Hadoop and using PCJ library, both running on the same infrastructure. This paper is constructed as follows: Sect. 2 contains information about BFS Hadoop implementation. Next Section contains general remarks about how BFS was implemented using PCJ library. Section 4 contains comparison results of BFS algorithm implementation using PCJ and Hadoop MapReduce. The paper concludes with final remarks and future plans in Sect. 5.

2 BFS - Hadoop Implementation

The idea of the Hadoop implementation of the BFS algorithm assumes iterative MapReduce jobs, where the same Mapper and Reducer run multiple times [6]. The only difference is that in each iteration the previous iteration's output is used as input.

In the algorithm, each vertex has its own status (color) which indicates whether it was visited or not. There are three possible statuses: white (unvisited), gray (visited) and black (finished). BFS traversal proceeds according to those colors. The search starts from a randomly sampled source vertex of the graph, which at the beginning has a gray color. All other vertices are white. The gray color indicates that this vertex is visited and its neighbors should be processed. All vertices that are adjacent to the gray vertex are becoming visited and gain gray color. Whereas, the original gray vertex is colored black, what indicates that all its neighbors are already visited. When there are no other gray vertices, the BFS algorithm stops.

In the first MapReduce iteration, vertices with distance (understood as the shortest path connecting two vertices) one from a source are being explored. In the i-th iteration, new vertices i steps away from the source are becoming visited.

Each time single MapReduce iteration finishes, to check whether next iteration should proceed, a simple condition: `numberOfGrayVertices` $>$ `0` is verified. This is realized via MapReaduce counter defined as an *enum* type, as in the Listing 1.

```
public static enum NextLevel {
    numberOfGrayVertices
}
```

Listing 1. Number of gray vertices - counter.

The information about all vertices of the graph is kept in the `CustomWritable` class (Listing 2). This class contains data about the list of adjacent vertices, the distance from the source, the color of the vertex and the predecessor (parent). By default, each vertex except source is unvisited (white) and its distance is set to `Long.MAX_VALUE`. The getter and setter methods are omitted.

The input is a binary sequence file, where vertex information is kept in the following format: `color distanceFromSource parent adjacencyList`. For example for source vertex: `G 0 MAX 2,4,5,` and for regular vertex: `W MAX MAX 2,4,67,`.

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.Writable;

public class CustomWritable implements Writable {

    private long parent;

    private long[] adjacencyList;

    // W, G, B
    private char color;

    private long distance;

    public CustomWritable() {
        this.color = 'W';
        this.distance = Long.MAX_VALUE;
        this.parent = Long.MAX_VALUE;
        this.adjacencyList = new long[0];
    }

    public CustomWritable(char color, long distance, long
        parent) {
        this.color = color;
        this.distance = distance;
        this.parent = parent;
        this.adjacencyList = new long[0];
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        color = in.readChar();
        distance = in.readLong();
        parent = in.readLong();
        int length = in.readInt();
        adjacencyList = new long[length];
        for (int i = 0; i < length; i++) {
            adjacencyList[i] = in.readLong();
        }
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeChar(color);
        out.writeLong(distance);
        out.writeLong(parent);
        out.writeInt(adjacencyList.length);
        for (long l : adjacencyList) {
            out.writeLong(l);
        }
    }

    // getters and setters
}
```

Listing 2. Class responsible for keeping information about single vertex.

Class with whole BFS algorithm extends `Configured` class and implements `Tool` interface [10]. Each single MapReduce iteration job is defined as in the Listing 3.

```
job.setMapOutputKeyClass(LongWritable.class);
job.setMapOutputValueClass(CustomWritable.class);
job.setOutputKeyClass(LongWritable.class);
job.setOutputValueClass(CustomWritable.class);
job.setInputFormatClass(SequenceFileInputFormat.class);
job.setOutputFormatClass(SequenceFileOutputFormat.class);
```

Listing 3. MapReduce job configuration in each iteration.

`Map` class extends the `Mapper` class specifying the parameters as the types of the input key - LongWritable, input value - CustomWritable, output key - LongWritable and output value - CustomWritable (Listing 4). All information about vertices is written as key-value pairs. Key is vertex unique identification. If vertex v has a gray color, for all its adjacent vertices as a value: color (gray), new distance and parent is emitted. New vertex distance from the source is set to be the distance of its parent incremented by one. The original vertex v becomes black. At the end, for vertex v emit: color, distance, parent and adjacency list.

```
public static class Map extends Mapper<LongWritable,
    CustomWritable, LongWritable, CustomWritable> {

  public void map(LongWritable key, CustomWritable v, Context
    context) throws IOException, InterruptedException {
    if (v.getColor() == 'G') {
      for (final long v2Id : v.getAdjacencyList()) {
        CustomWritable v2 = new CustomWritable('G', v.
            getDistance() + 1, key.get());
        context.write(new LongWritable(v2Id), v2);
      }
      v.setColor('B');
    }
    context.write(key, v);
  }
}
```

Listing 4. Hadoop BFS Map class.

Reduce class combines all the information for a single vertex identificator (Listing 5). The following values are determined and emitted from the reducer function to the output file: the complete list of adjacent vertices, the minimum distance from the source together with parent, the darkest color. If there are more gray nodes, the counter is incremented by one.

```
public static class Reduce extends Reducer<LongWritable,
    CustomWritable, LongWritable, CustomWritable> {

  public void reduce(LongWritable key, Iterable<
      CustomWritable> values, Context context) throws
      IOException, InterruptedException {

    final CustomWritable reduced = new CustomWritable('W',
        Integer.MAX_VALUE, Integer.MAX_VALUE);
    for (CustomWritable v : values) {
      if (v.getAdjacencyList().length > 0) {
        reduced.setAdjacencyList(v.getAdjacencyList());
      }
        if (v.getDistance() < reduced.getDistance()) {
        reduced.setDistance(v.getDistance());
        reduced.setParent(v.getParent());
      }
        if (reduced.getColor() > v.getColor()) {
        // save the darkest color
        reduced.setColor(v.getColor());
      }
    }
    context.write(key, reduced);
    if (reduced.getColor() == 'G')
```

```
        context.getCounter(NextLevel.numberOfGrayVertices).
            increment(1L);
    }
}
```

Listing 5. Hadoop BFS Reduce class.

3 BFS - PCJ Implementation

The BFS implementation, used for performance tests in this paper, is a part of a recent implementation of Graph500 benchmark in PGAS model using PCJ library. More detailed description can be found in [7,8]. Below there are presented only general information.

In the implementation the level-synchronous BFS strategy is used, what means that all vertices at a distance k form source vertex are visited before vertices at distance $k+1$. Because the input graph is static, it is kept in distributed CSR format in which every PCJ task holds its own subset of vertices and its adjacent edges in two arrays. The distribution of vertices among PCJ tasks is realized by 1D partitioning of the graph.

Each PCJ task keeps two queues. Both queues hold only local vertices owned by specific PCJ task. The first queue keeps vertices processed at the current level (from distance k from source vertex), whereas the second queue holds vertices that are within one vertex away from the source vertex and should be visited at the next level (distance $k+1$). The result - predecessor array for BFS result tree - is also distributed in the way that each PCJ task holds part of the array, only for its local vertices.

The BFS traversal starts from a random, source vertex sampled from a graph, put in the first (current level) queue of the owner task. The communication between PCJ tasks occurs when vertex u adjacent to the vertex v from the first queue does not belong to the task performing this check. Proper message is constructed and sent to the owner task of vertex u. At the end of each level, all tasks exchange information about a number of vertices in the second queue to check if BFS algorithm should be stopped. Otherwise, search continues to the next level.

In the implementation, many optimizations have been used. Instead of sending single messages, data is accumulated in array buffers, which is important as PCJ library supports sending arrays with provided indexes. This allows to split messages at the time of creation and minimize the overhead connected with starting the communication. Used bitmap for checking whether the vertex has already been visited, allows reducing the number of sent messages between tasks (each task keeps a vector of bits, set to 1 if the vertex is visited, 0 otherwise). Another important feature is overlapping the communication and computation - while waiting for communication coming to an end, the task uses this time to process data that has already been received.

4 Results

4.1 Data

Experimental graphs used in performance tests together with BFS source vertices have been taken from Kronecker Generator of the Graph500 benchmark [9]. Generated graphs have the edge tuple list form, so in the case of Hadoop implementation, this edge tuple list had to be transformed to the following input data: `vertexId color distanceFromSource parent adjacencyList`.

At the beginning, the generated graph file has been converted to the number of distributed text files holding proper BFS input data. The program required three parameters: a path to the edge list file (file with sources has the same path and name, but ends with '-root'), the name of generated file and number of generated files parts. A further step was to create a binary Hadoop sequence file (`org.apache.hadoop.io.SequenceFile`) from a text file to read graph data as objects instead of extracting necessary information from strings. We compare BFS performance both for sequence files and for text files. Neither the conversion nor uploading files to HDFS were accounted for final results.

The performance has been tested on graphs of $SCALE \in \{18, 19, 20, 21, 22\}$ with $edgefactor = 16$ ($SCALE$ is the logarithm base two of the number of vertices and edgefactor is the ratio of the graph's edge count to its vertex count).

4.2 Environment

The results both for Hadoop and PCJ are obtained using the same hardware system. Hadoop cluster consisted of 68 computing nodes, 3 namenodes, and 2 management nodes. All nodes have two 24 core Intel(R) Xeon(R) CPU E5-2680 v3 processors with 2.50 GHz and the InfiniBand interconnect. Nodes were running CentOS Linux release 7.2.1511. The storage was HDFS filesystem with 5.1 TB capacity. The Cloudera CDH 5.7.5 open source platform distribution, including Apache Hadoop, was installed.

Java version used for both Hadoop and PCJ tests was Oracle 64-bit 1.8.0. The PCJ library version 4.1 has been used. In the tests, only the strong scalability has been examined.

4.3 Performance

Figure 1 (left chart) shows comparison of Hadoop BFS algorithm in two variants: where graph is read and written as a plain text file or a binary Hadoop sequence file (`org.apache.hadoop.io.SequenceFile`) which allows to read graph data as objects instead of extracting necessary information from strings. The sequence implementation is faster than the text one. The greater the graph, the difference between sequence and text version is bigger.

The execution time presented in the Figs. 1 and 2 (right chart) shows that both implementations show similar scalability - up to 32 threads for graphs of

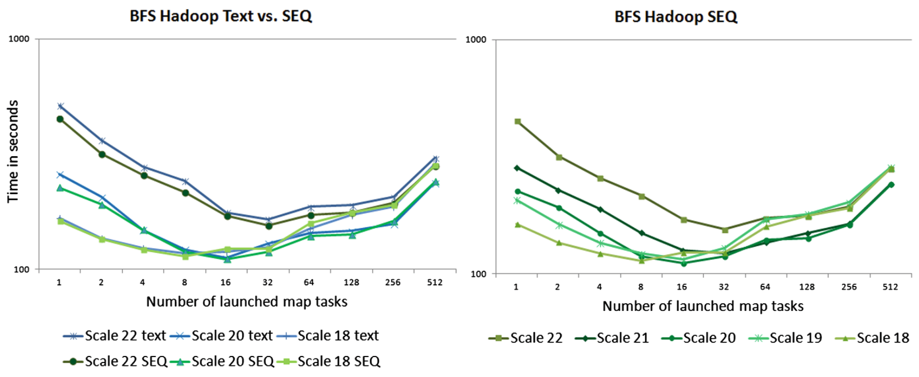


Fig. 1. BFS on Hadoop for text and binary sequence files as input (left). BFS execution time for Hadoop (with sequence files) for graphs of $SCALE \in \{18, 19, 20, 21, 22\}$ (right).

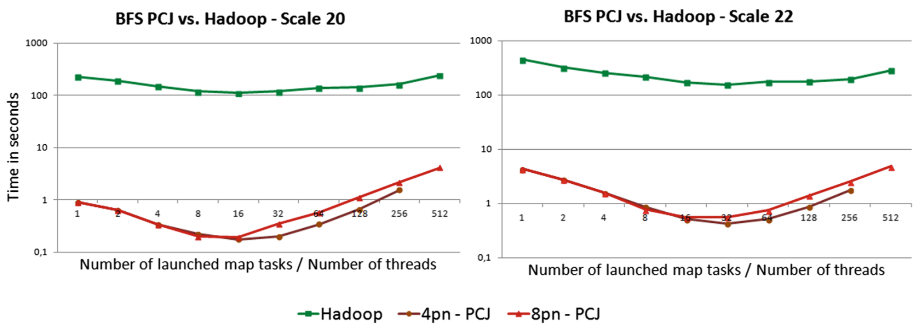


Fig. 2. BFS execution time for Hadoop (with sequence files) and PCJ (with 4 or 8 threads per node - marked as 4pn and 8pn) for graphs $SCALE = 20$ and $SCALE = 22$.

$SCALE = 22$. The bigger the graph, the scalability is better. In PCJ implementation for the larger number of tasks, the communication time starts to dominate and execution time increases. There is a huge difference in the total execution time: namely the PCJ implementation is more than 100 times faster for the whole range of the tasks used. Slightly better outcomes for PCJ imlementation have been gained for 4pn (4 threads per node) configuration over 8pn (8 threads per node).

Figure 3 shows an execution time box chart with informations about 1st quartile, median, third quartile together with minimum and maximum value for PCJ implementation ($SCALE = 22$) with 4pn and 8pn. We can see that, the more number of threads take part in the algorithm the bigger difference starts to arise between median and the maximum value especially for 8 thread per node configuration. Minimum value however, is approximately close to the median for whole range of the threads used both for 4pn and 8pn.

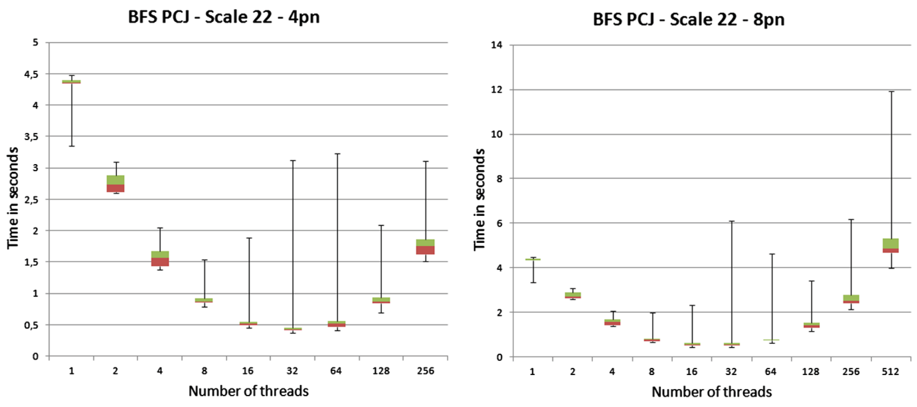


Fig. 3. Box plot for PCJ execution time (with 4 or 8 threads per node - marked as 4pn and 8pn) for graph of $SCALE = 22$.

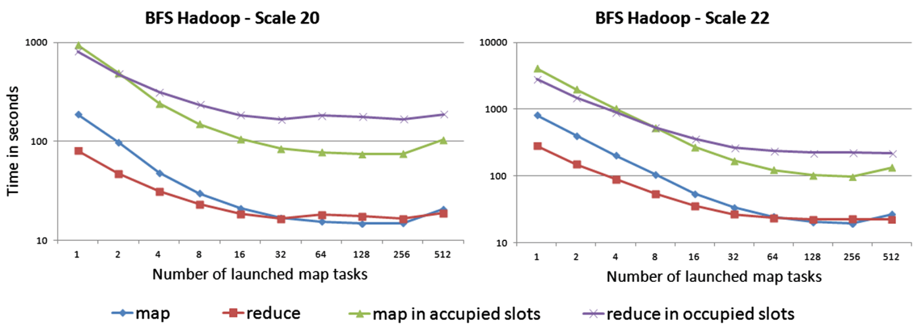


Fig. 4. BFS Hadoop execution time for operations: map, reduce, map in occupied slots, reduce in occupied slots.

5 Conclusions and Future Work

We have presented Hadoop MapReduce and PCJ implementation details of the level synchronous parallel BFS algorithm and introduced the performance comparison of those two solutions. The BFS algorithm coded in the PGAS model in Java with PCJ library is much faster (about 100 times) than the Hadoop implementation. The scalability of both implementations is similar.

As graph processing is not a typical Map-Reduce application, we currently are focused on comparing PCJ implementation of BFS with selected tools strictly dedicated to graph problems like Apache Giraph (open-source counterpart to Pregel, built on top of Apache Hadoop) and GraphX (Apache Spark's API for graphs and graph-parallel computations).

Acknowledgments. This work has been performed using the PL-Grid infrastructure. The authors would like to thank CHIST-ERA consortium for financial support under HPDCJ project (Polish part funded by NCN grant 2014/14/Z/ST6/00007).

References

1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
2. Apache Hadoop. http://hadoop.apache.org/. Accessed 20 May 2017
3. Nowicki, M., Bała, P.: Parallel computations in Java with PCJ library. In: Smari, W.W., Zeljkovic, V. (eds.) 2012 International Conference on High Performance Computing and Simulation (HPCS), pp. 381–387. IEEE (2012)
4. Nowicki, M., Bała, P.: PCJ - new approach for parallel computations in Java. In: Manninen, P., Öster, P. (eds.) PARA 2012. LNCS, vol. 7782, pp. 115–125. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36803-5_8
5. http://pcj.icm.edu.pl. Accessed 25 May 2017
6. https://hadooptutorial.wikispaces.com/Iterative+MapReduce+and+Counters. Accessed 20 Mar 2017
7. Ryczkowska, M., Nowicki, M., Bala, P.: The performance evaluation of the Java implementation of Graph500. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 221–230. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_21
8. Ryczkowska, M., Nowicki, M., Bała, P.: Level-synchronous BFS algorithm implemented in Java using PCJ library. In: Proceedings of the 2016 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, pp. 596–601 (2016)
9. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the Graph500. Cray User's Group (CUG) (2010)
10. White, T.: Hadoop: The Definitive Guide, 4th edn. O'Reilly, Sebastopol (2015)
11. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Mol. Biol. **215**(3), 403–410 (1990)
12. Suzumura, T., Ueno, K., Sato, H., Fujisawa, K., Matsuoka, S.: Performance characteristics of Graph500 on large-scale distributed environment. In: Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC), pp. 149–158 (2011)
13. Ueno, K., Suzumura, T.: Highly scalable graph search for the Graph500 benchmark, In: Proceedings of the 21st International ACM Symposium on High-Performance Parallel and Distributed Computing, pp. 149–160 (2012)
14. Buluc, A., Madduri, K.: Parallel breadth-first search on distributed memory systems. In: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011)
15. Berrendorf, R., Makulla, M.: Level-synchronous parallel breadth-first search algorithms for multicore and multiprocessor systems. In: FC 2014, pp. 26–31 (2014)

Minisymposium on HPC Applications in Physical Sciences

Efficient Parallel Generation of Many-Nucleon Basis for Large-Scale *Ab Initio* Nuclear Structure Calculations

Daniel Langr[1](✉), Tomáš Dytrych[2], Tomáš Oberhuber[3], and František Knapp[4]

[1] Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 16000 Praha, Czech Republic
daniel.langr@fit.cvut.cz
[2] Nuclear Physics Institute, Czech Academy of Sciences, Řež 130, 25068 Řež, Czech Republic
[3] Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, Břehová 7, 11519 Praha, Czech Republic
[4] Faculty of Mathematics and Physics, Charles University, Ke Karlovu 3, 12116 Praha, Czech Republic

Abstract. We address the problem of generating a many-nucleon basis for *ab initio* nuclear structure modeling, which quickly becomes a significant runtime bottleneck for large model spaces. We first analyze the original basis generation algorithm, which does not employ multi-threading parallel paradigm. Based on the analysis, we propose and empirically evaluate a new efficient scalable basis generation algorithm. We report a reduction of basis generation runtime by a factor of 42 on the Blue Waters supercomputer and by two orders of magnitude on our test-bed computer system with Broadwell CPUs.

Keywords: Ab initio · Basis generation · Many-nucleon basis · Nuclear structure · Parallel algorithm

1 Introduction

Understanding the origin, structure, and phases of hadronic matter is key to comprehending the evolution of the universe. To fully achieve this, we need to model the complex dynamics of atomic nuclei that control a vast array of astrophysical phenomena and are often found key to understanding processes in extreme environments, from stellar explosions to the interior of nuclear reactors.

Over the past two decades, *ab initio* approaches to nuclear structure and reactions have considerably advanced our understanding and capability of achieving first-principles descriptions of light nuclei [6,7,11,12]. To extend the reach of *ab initio* methods towards heavier nuclei, we have developed a novel method dubbed *symmetry-adapted no-core shell model* (SA-NCSM) [1] and implemented it as a highly scalable computer code `LSU3shell`[1] [2]. Our approach is to solve the

[1] https://sourceforge.net/projects/lsu3shell/ (In the time of writing this paper, latest updates were included in the `LSU3develop` repository branch).

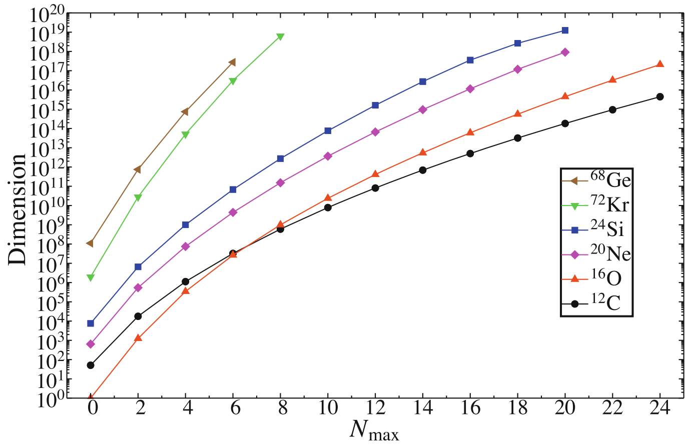


Fig. 1. Dimension of basis as a function of *Nmax* for sample atomic nuclei.

Schrödinger equation for a many-nucleon quantum system interacting via realistic interactions that are tied to the underlying quark/gluon considerations. The solution to this problem is achieved by finding eigenstates and eigenvalues of the nuclear Hamiltonian, which is computed in a *many-nucleon basis* that spans a physically relevant subspace of the nuclear Hilbert space, the so-called *model space*.

In particular, we consider many-nucleon basis states of a fixed parity, consistent with the Pauli principle, and limited by a many-body basis cutoff *Nmax*. This cutoff is defined as the maximum number of harmonic oscillator quanta allowed in a many-nucleon basis state above the minimum for a given nucleus. The dimension of basis grows combinatorially as a function of *Nmax* and the number of nucleons, which is illustrated in Fig. 1.

The SA-NCSM further organizes *Nmax* model space according to U(3) and SU(2) symmetries inherent to the low-energy nuclear dynamics [3–5,13]. This step introduces *selection rules* that allows us to reduce an *Nmax* model space to a smaller number of physically relevant configurations based on their deformations and proton, neutron, and total intrinsic spins.

In the original version of `LSU3shell`, the construction of many-nucleon basis spanning a given model space turns out to be a major bottleneck for large *Nmax* cutoffs and medium mass nuclei. For example, to generate basis of ^{20}Ne in *Nmax* = 12 takes hours on commodity HPC hardware. In this paper, we first analyze the original algorithm and identify its inefficiencies. Based on this analysis, we then propose a new optimized scalable parallel basis generation algorithm and prove its superiority in the experimental study.

2 Analysis

`LSU3shell` functionality is divided into the following 3 phases:

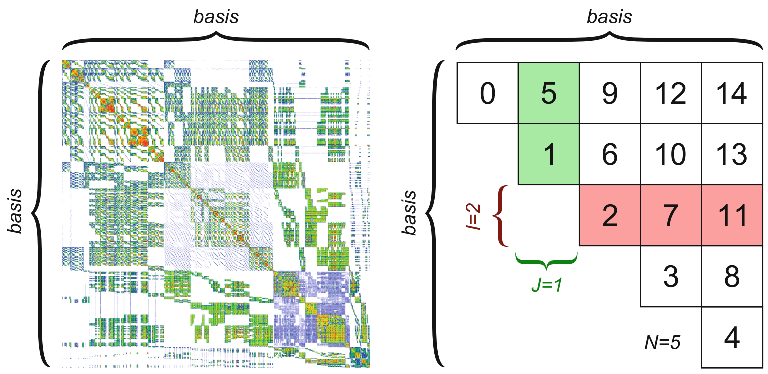


Fig. 2. Sparsity pattern of a Hamiltonian matrix H (*left*). Mapping of MPI processes to the upper triangular part of H for $N = 5$ (*right*).

1. Generation of a many-nucleon basis that spans a given model space.
2. Construction of a Hamiltonian matrix H in this basis.
3. Finding few lowest-lying eigenpairs of H using the Lanczos method.

The code is written mostly in C++ and built upon hybrid MPI+OpenMP parallel programming model. Due to the Hermiticity and sparsity of H, only nonzero elements from a triangular part need to be stored in memory. `LSU3shell` maps MPI processes to the upper triangular part of H in a two-dimensional checkerboard fashion (see Fig. 2). The basis states are split into N blocks and each process is then assigned Ith and Jth block for rows and columns of H, respectively. The total number of MPI processes is equal to $nprocs = N \cdot (N + 1)/2$. Processes with MPI ranks $0, \ldots, N-1$ are mapped to diagonal blocks of H; we therefore call these processes *diagonal*. Remaining processes are mapped to the non-diagonal blocks of H.

To ensure load-balanced computation of H, `LSU3shell` assigns basis states to N blocks in a round robin fashion [8]. Consequently, the order of basis states is a function of N and the rows and columns of H are permuted/reshuffled accordingly.

For large-scale runs of `LSU3shell`, we use the Blue Waters supercomputer. Its non-GPU part consists of 22640 Cray XE6 computational nodes. Each node consists of 16 AMD Bulldozer cores, which are exposed as 2 virtual cores each under certain conditions (so-called clustered multi-threading; CMT). Since `LSU3shell` mostly meet these CMT conditions, we typically use 1 MPI process per node which is then split into 32 threads in performance-critical program parts. According to the above introduced matrix-processes mapping, we can utilize up to 22366 nodes which corresponds to $N = 211$ (the implementation of Lanczos method requires N to be an odd number [10]). MPI processes are then split into 715712 OpenMP threads.

Algorithm 1. `Reshuffle`(*modelSpace*,*N*,*K*)

```
Input: modelSpace: given model space
Input: N: number of basis blocks, i.e., diagonal MPI processes
Input: K: index of basis block computed by actual MPI process
Output: dims[], pnBasisIpIn[], wpn[], blockEnd[], firstStateInBlock[]: arrays with
        calculated basis data
1  $Nmax, i_p^{\max}, i_n^{\max} \leftarrow$ function-of-$modelSpace$
2  $maxStatesInBlock \leftarrow 0$
3  $numberOfStates \leftarrow 0$
4  $ipInPair \leftarrow 0$
5  $dims[0 : N-1] \leftarrow 0$
6  for $i_p \leftarrow 0$ to $i_p^{\max}$ do
7      for $i_n \leftarrow 0$ to $i_n^{\max}$ do
8          $Nhw \leftarrow$ function-of-$(i_p, i_n)$
9          if $Nhw > Nmax$ then break
10         $NhwSubspace \leftarrow$ function-of-$(modelSpace, Nhw)$
11         if $NhwSubspace$ does not belong to $modelSpace$ then continue
12         $K' \leftarrow$ function-of-$(ipInPair, N)$
13         if $NhwSubspace$ contains allowed spins then
                   /* current $(i_p, i_n)$ pair is valid but "process-global" */
14             calculate $data$ for current $(i_p, i_n)$
15             $dims[K'] \leftarrow dims[K']$ + function-of-$data$
16             if $K' = K$ then
                       /* current $(i_p, i_n)$ pair is valid and "process-local" */
17                 append $(i_p, i_n)$ at the end of $pnBasisIpIn[]$
18                 append (function-of-$data$) at the end of $wpn[]$
19                 append (function-of-$data$) at the end of $blockEnd[]$
20                 append $numberOfStates$ at the end of $firstStateInBlock[]$
21                 $numberOfStates \leftarrow numberOfStates$ + function-of-$data$
22             $ipInPair \leftarrow ipInPair + 1$
```

2.1 Basis Generation

Previously, only the matrix construction and eigensolver `LSU3shell` phases were parallelized within MPI processes by OpenMP threading. This is convenient for small model spaces, i.e., for light nuclei and small values of *Nmax*. However, with the transition to medium-mass nuclei and larger values of *Nmax*, the sequential generation of the basis becomes a significant runtime bottleneck.

The vast majority of the basis generation runtime is spent within a routine (C++ class member function) called `Reshuffle`; we show its pseudocode in Algorithm 1. This routine generates the process-local basis block according to parameters K and N. Each MPI process generally needs to call this routine twice while setting its I and J as arguments for the K routine parameter.

`Reshuffle` iterates over all possible pairs of i_p and i_n indices, which are pointers into the tables of irreducible proton and neutron representations. Some of these pairs are filtered out by the *selection rules*; see lines 9, 11, and 13 of Algorithm 1. We call the pairs that are not filtered-out *valid*. After applying the selection rules, the algorithm decides which valid pairs belong to which MPI processes (line 12). If the pair belongs to the current MPI process (line 16), we call it "process-local". Valid process-local (i_p, i_n) pairs are stored into *pnBasisIpIn*[] array and the corresponding calculated basis data are appended at the end of arrays *wpn*[], *blockEnd*[], and *firstStateInBlock*[]. Moreover, there is an additional array *dims*[] of the size of N calculated by all processes; this array contains dimension of each block of basis states. Total dimension of the entire model space is a sum of all elements of this array.

In the pseudocode, we omit computational details which are irrelevant to the problem described in this paper. To denote functional dependencies between particular algorithm parts, we use the simplified syntax with the "function-of-" prefix, which generally means that some data are calculated in dependence on another data. We also use the square brackets suffix "[]" to denote that some data represents an array. Such arrays are generally considered to be dynamic, i.e., they are empty at algorithm input and can grow in size by appending elements to them (in the code, these arrays are C++ vector containers).

3 Parallelization and Optimization of `Reshuffle`

Two main causes of inefficiency may be observed in Algorithm 1. First, the array *dims*[] is redundantly calculated by all MPI processes. To evaluate elements of *dims*[], some *data* need to be calculated first for each valid (i_p, i_n) pair (line 14), which is a costly operation. We therefore proposed an alternative solution, where only the diagonal processes evaluate their contribution to the *dims*[] array, i.e., *dims*$[K]$, and the whole array is finally reduced and distributed to all processes by `MPI_Allreduce` communication operation. The proposed solution allows all processes to calculate *data* only for their local (i_p, i_n) pairs.

Second, there is no threading in Algorithm 1. One might observe that the whole iterative process (lines 6 and 7) is inherently sequential. Namely, the development of each iteration depends on the outcome of all the previous iterations for the following reasons:

1. The distribution of valid (i_p, i_n) pairs to processes is a function of *ipInPair* (line 12), which represents its order among all valid pairs (line 22). There is no way how to find out the value of *ipInPair* for particular (i_p, i_n) directly.
2. The calculated basis data are *appended* into corresponding arrays (lines 17–20). With direct loop OpenMP parallelization, the order of insertions of elements into these arrays would therefore change, which is not acceptable.

We proposed and introduce here a parallelization of the `Reshuffle` routine presented by Algorithm 2. This algorithm iterates over (i_p, i_n) pairs three times, however, the outer loops over i_p indices are performed in parallel by all OpenMP threads (lines 3, 12, and 26). Our solution is generic such that it allows to use OpenMP dynamic loop scheduling, which was found necessary for balanced computational load among threads. Algorithm 2 works as follows:

Algorithm 2. `ParallelOptimizedReshuffle`(*modelSpace*,N,K)

```
1  . . .                                                              /* initialization */
2  $ipInPairsForIp[0 : i_p^{\max}] \leftarrow 0$
3  for $i_p \leftarrow 0$ to $i_p^{\max}$ do in parallel                                /* first loop */
4      for $i_n \leftarrow 0$ to $i_n^{\max}$ do
5          . . .                                          /* lines 8–11 in Algorithm 1 */
6          if $NhwSubspace$ does not contains allowed spins then continue
                                  /* current $(i_p, i_n)$ is valid and “process-global” */
7          $ipInPairsForIp[i_p] \leftarrow ipInPairsForIp[i_p] + 1$
8  $firstIpInPairForIp \leftarrow$ parallel exclusive prefix sum over $ipInPairsForIp$
9  $localIpInPairsForIp[0 : i_p^{\max}] \leftarrow 0$
10 $localWpnsForIp[0 : i_p^{\max}] \leftarrow 0$
11 $locNumStatesForIp[0 : i_p^{\max}] \leftarrow 0$
12 for $i_p \leftarrow 0$ to $i_p^{\max}$ do in parallel                               /* second loop */
13     $ipInPair \leftarrow firstIpInPairForIp[i_p]$
14     for $i_n \leftarrow 0$ to $i_n^{\max}$ do
15         . . .                                   /* check conditions as in first loop */
16         $K' \leftarrow$ function-of-$(ipInPair, N)$
17         $ipInPair \leftarrow ipInPair + 1$
18         if $K' \neq K$ then continue
                                   /* current $(i_p, i_n)$ is valid and “process-local” */
19         $localIpInPairsForIp[i_p] \leftarrow localIpInPairsForIp[i_p] + 1$
20         $localWpnsForIp[i_p] \leftarrow localWpnsForIp[i_p] +$ function-of-$(i_p, i_n)$
21         $locNumStatesForIp[i_p] \leftarrow locNumStatesForIp[i_p] +$ function-of-$data$
22 $localFirstIpInPairForIp \leftarrow$ parallel excl. prefix sum over $localIpInPairsForIp$
23 $localFirstWpnForIp \leftarrow$ parallel exclusive prefix sum over $localWpnsForIp$
24 $locNumStatesBeforeIp \leftarrow$ parallel exclusive prefix sum over $locNumStatesForIp$
25 resize $pnBasisIpIn$, $firstStateInBlock$, $wpn$, and $blockEnd$ properly
26 for $i_p \leftarrow 0$ to $i_p^{\max}$ do in parallel                                /* third loop */
27     $ipInPair \leftarrow firstIpInPairForIp[i_p]$
28     $localIpInPair \leftarrow localFirstIpInPairForIp[i_p]$
29     $wpnIndex \leftarrow localFirstWpnForIp[i_p]$
30     $numberOfStates \leftarrow locNumStatesBeforeIp[i_p]$
31     for $i_n \leftarrow 0$ to $i_n^{\max}$ do
32         . . .                       /* check conditions as in first and second loop */
33         $pnBasisIpIn[localIpInPair] \leftarrow (i_p, i_n)$
34         $firstStateInBlock[localIpInPair] \leftarrow numberOfStates$
35         calculate $data$ for current $(i_p, i_n)$
36         if process rank $< N$ then $dims[K] \leftarrow dims[K]+$ function-of-$data$
37         $wpnCount \leftarrow$ function-of-$(i_p, i_n)$
38         write function-of-$data$ to $wpn[wpnIndex : wpnIndex + wpnCount - 1]$
39         $wpnIndex \leftarrow wpnIndex + wpnCount$
40         $blockEnd[localIpInPair] \leftarrow$ function-of-$(i_p, i_n)$
41         $numberOfStates \leftarrow numberOfStates+$ function-of-$data$
42         $localIpInPair \leftarrow localIpInPair + 1$
43 reduce $dims$ across all processes (by using MPI_Allreduce)
```

1. Within the first loop, the number of valid (i_p, i_n) pairs for each i_p is stored into the temporary *ipInPairsForIp*[] array (line 7). Next, the exclusive parallel prefix sum is run over this array resulting in a new temporary array *firstIpInPairForIp*[]. The element *firstIpInPairForIp*$[i_p]$ therefore equals the number of valid (i_p, i_n) pairs for all $0 \leq i'_p < i_p$.
2. Within the second loop, we can now evaluate *ipInPair* for each i_p independently (and therefore concurrently; lines 13 and 17). In this loop, the number of valid process-local (i_p, i_n) pairs for each i_p is stored into the temporary array *localIpInPairsForIp*[] (line 19). Similarly, the number of *wpn*[] elements and the number of states for each i_p are stored into temporary arrays *localWpnsForIp*[] and *locNumStatesForIp*[], respectively (lines 20 and 21). As a next step, parallel prefix sums over these arrays are performed (lines 22–24). The results of these prefix sums then allows to properly resize the resulting basis arrays (line 25) and to find out, for each i_p, where to store generated basis data into them.
3. Within the third loop, the basis data for process-local (i_p, i_n) pairs are finally calculated and stored into corresponding arrays (lines 33–41).

The additional advantage of the proposed solution is that data are not appended into arrays; instead, they are written at already-known positions. This avoids memory reallocations, which, when performed frequently, might considerably hinder scalability in multi-threaded programs.

Finally, to prevent redundant calculations within loops, we integrated several software cache data structures into the code. These are not shown in Algorithm 2 due to text space limitations; however, they also contribute to the higher efficiency of the new version of basis generation procedure. These caches are mostly thread-local and are implemented by arrays (C++ vectors) or binary search trees (C++ maps).

4 Experiments

We carried out experiments to compare the performance of the original and new versions of basis generation procedure. For these experiments, we utilized our test-bed machine that contained two 10-core Intel Xeon CPUs with Broadwell microarchitecture and 128 GB of memory. For measurements, we used `LSU3shell` in a so-called *simulation mode*; it allows to run only a single MPI process, which generates its local basis block based on chosen N, I, and J parameters. Since the basis generation is generally very well balanced across MPI processes, such an approach provides runtimes similar as if the full basis would be generated on a hypothetical HPC system consisting of nodes identical with our test-bed machine.[2]

[2] Note that the simulation mode does not reflect the runtime of the `MPI_Allreduce` communication operation. However, such a reduction of a small array is generally very fast. For instance, on Blue Waters it takes up to few seconds even if majority of the nodes are involved [9].

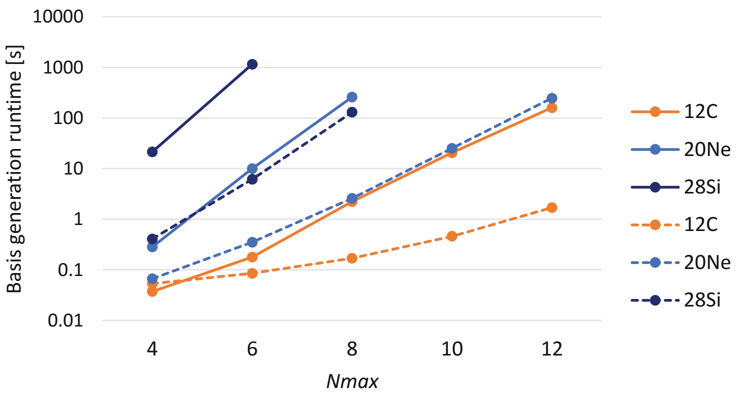


Fig. 3. Runtimes of original (*solid lines*) and optimized (*dashed lines*) basis generation for various nuclei and different values of $Nmax$, measured for 20 OpenMP threads and $N = 211$.

First, we compared basis generation runtimes for different nuclei as a function of increasing $Nmax$. The results shown in Fig. 3 indicate, that the new optimized parallel variant of `Reshuffle` routine reduced the basis generation runtime by a factor of around 100 on a 20-core machine.

In the second experiment, we measured the basis generation runtime as a function of growing N. The results are shown in Fig. 4 for the original `Reshuffle` version and the optimized version with 1 and 20 OpenMP threads. The original version was obviously insensitive to N, which may be attributed to the redundant calculation of the whole $dims[]$ array by all MPI processes. On the contrary, the new version reduced the runtime significantly even for a single thread, which was caused by all the proposed optimizations except of multi-threading. The parallelization itself then additionally reduced the runtime approximately by a factor of 8 utilizing the 20-core machine. We attribute such a relatively low parallel efficiency to the limits of memory bandwidth together with the effects of non-uniform memory architecture (NUMA).

Note that for the single-threaded run with $N = 1$, the whole basis was generated by a single MPI process. Even though the optimized basis generation iterated over (i_p, i_n) indices 3 times, its runtime was only slightly higher than the runtime of the original version.

Finally, we compared the basis generation runtime in a large-scale `LSU3shell` run on Blue Waters (22366 nodes, $N = 211$) for ^{20}Ne and $Nmax = 10$. It was reduced from 2738 to 65 s, i.e., 42 times (matrix construction took 2934 s and eigensolver 237 s). Such a speedup might seem low in comparison with the test-bed machine, however, we need to take into account the following facts:

1. In our experience, the architecture of Cray XE6 nodes is considerably less powerful than the Broadwell-based test-bed system, both in terms of CPU power and memory bandwidth.

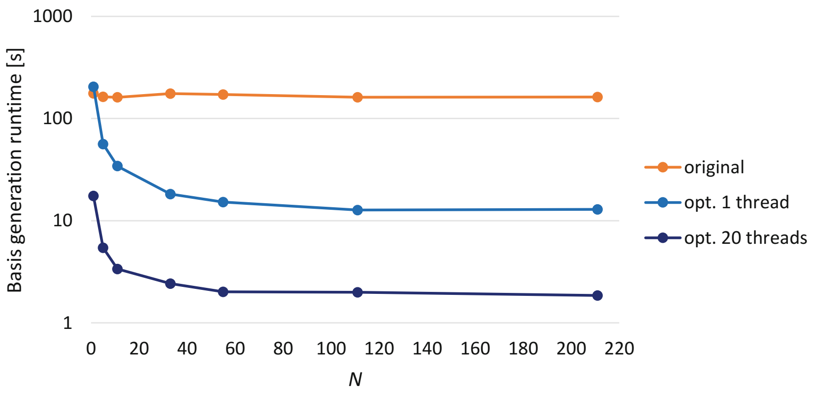


Fig. 4. Basis generation runtime as a function of growing N, measured for ^{12}C nucleus, $Nmax = 12$, and $N \in \{1, 5, 11, 33, 55, 111, 211\}$.

2. AMD CMT brings only slight improvement for `LSU3shell`; one therefore should view Blue Waters nodes more as 16-core rather than 32-core shared-memory machines.
3. In contrast to the simulation mode, the runtime on Blue Waters additionally includes the `MPI_Allreduce` communication operation.

5 Conclusions

The contribution of this paper is a new efficient scalable algorithm for generation of many-nucleon basis in large-scale *ab initio* nuclear structure calculations. Our implementation based on the hybrid MPI+OpenMP parallel programming model reduced the basis generation runtime around 100 times on a commodity 20-core machine and 42 times in a production large-scale run on Blue Waters. The proposed algorithm eliminates the basis-generation bottleneck that have heretofore hindered applications of SA-NCSM approach for *ab initio* modeling of important collective and cluster nuclear states in medium- and light-mass nuclei.

Acknowledgements. This work was supported by the Czech Science Foundation under Grant No. 16-16772S. This work is also part of the "Collaborative Research: Innovative ab initio symmetry-adapted no-core shell model for advancing fundamental physics and astrophysics" PRAC allocation support by NSF (award number ACI-1516338), and is part of the Blue Waters sustained-petascale computing project, which is supported by NSF (awards OCI-0725070 and ACI-1238993) and the state of Illinois (Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications). The authors acknowledge support from J.P. Draayer from the Louisiana State University, P. Tvrdík from the Czech Technical University in Prague, P. Vrchota from Výzkumný a zkušební letecký ústav, a.s., and M. Pajr from IHPCI.

References

1. Dytrych, T., Launey, K., Draayer, J., Maris, P., Vary, J., Saule, E., Catalyurek, U., Sosonkina, M., Langr, D., Caprio, M.: Collective modes in light nuclei from first principles. Phys. Rev. Lett. **111**(25), 252501 (2013). https://doi.org/10.1103/PhysRevLett.111.252501
2. Dytrych, T., Maris, P., Launey, K., Draayer, J., Vary, J., Langr, D., Saule, E., Caprio, M., Catalyurek, U., Sosonkina, M.: Efficacy of the SU(3) scheme for ab initio large-scale calculations beyond the lightest nuclei. Comput. Phys. Commun. **207**, 202–210 (2016). https://doi.org/10.1016/j.cpc.2016.06.006
3. Elliott, J.P.: Collective motion in the nuclear shell model. I. Classification schemes for states of mixed configurations. Proc. Roy. Soc. Lond. A: Math. Physi Eng. Sci. **245**(1240), 128–145 (1958). https://doi.org/10.1098/rspa.1958.0072
4. Elliott, J.P.: Collective motion in the nuclear shell model. II. The introduction of intrinsic wave-functions. Proc. Roy. Soc. Lond. A: Math. Phys. Eng. Sci. **245**(1243), 562–581 (1958). https://doi.org/10.1098/rspa.1958.0101
5. Elliott, J.P., Harvey, M.: Collective motion in the nuclear shell model. III. The calculation of spectra. Proc. Roy. Soc. Lond. A: Math. Phys. Eng. Sci. **272**(1351), 557–577 (1963). https://doi.org/10.1098/rspa.1963.0071
6. Epelbaum, E., Krebs, H., Lee, D., Meißner, U.G.: Ab initio calculation of the Hoyle state. Phys. Rev. Lett. **106**, 192501 (2011). https://doi.org/10.1103/PhysRevLett.106.192501
7. Hagen, G., Papenbrock, T., Hjorth-Jensen, M.: Ab initio computation of the 17F proton halo state and resonances in A = 17 nuclei. Phys. Rev. Lett. **104**, 182501 (2010). https://doi.org/10.1103/PhysRevLett.104.182501
8. Kleinrock, L.: Computer Applications. Queueing Systems, vol. 2, 1st edn. Wiley-Interscience, Hoboken (1976)
9. Langr, D., Tvrdík, P., Šimeček, I., Dytrych, T.: Downsampling algorithms for large sparse matrices. Int. J. Parallel Prog. **43**(5), 679–702 (2014). https://doi.org/10.1007/s10766-014-0315-8
10. Maris, P., Sosonkina, M., Vary, J.P., Ng, E., Yang, C.: Scaling of ab-initio nuclear physics calculations on multicore computer architectures. Procedia Comput. Sci. **1**(1), 97–106 (2010). https://doi.org/10.1016/j.procs.2010.04.012
11. Navrátil, P., Vary, J.P., Barrett, B.R.: Properties of 12C in the ab initio nuclear shell model. Phys. Rev. Lett. **84**, 5728–5731 (2000). https://doi.org/10.1103/PhysRevLett.84.5728
12. Quaglioni, S., Navrátil, P.: Ab initio many-body calculations of n-3H, n-4He, p-3,4He, and n-10Be scattering. Phys. Rev. Lett. **101**, 092501 (2008). https://doi.org/10.1103/PhysRevLett.101.092501
13. Rosensteel, G., Rowe, D.J.: Nuclear Sp(3, R) model. Phys. Rev. Lett. **38**, 10–14 (1977). https://doi.org/10.1103/PhysRevLett.38.10

Parallel Exact Diagonalization Approach to Large Molecular Nanomagnets Modelling

Michał Antkowiak(✉)

Faculty of Physics, Adam Mickiewicz University, ul. Umultowska 85, 61-614 Poznań, Poland
antekm@amu.edu.pl

Abstract. The exact diagonalization method is used to calculate the energy levels of ring-shaped molecular nanomagnets of different sizes and spin numbers. Two-level hybrid parallelization is used to increase the efficiency and obtain the optimally balanced workload. The results of the successful runs of our application on two Tier-0 supercomputers are presented with emphasis on the satisfactory speedup obtained by threading the diagonalization process.

Keywords: Molecular nanomagnets · Exact diagonalization
High Performance Computing

1 Introduction

Molecular nanomagnets based on transition metal ions have been very intensively investigated [8]. Their popularity is mostly due to the fact that quantum phenomena characteristic for a single molecule (like, e.g., quantum tunnelling or step like field dependence of magnetisation) can be observed in bulk samples. It is possible because nanomolecules are magnetically shielded from each other by organic ligands and the dominant interactions are those within the molecule. There are also expectations that this kind of materials may find application in quantum computing [4,9,16,18] and information storage [17].

A large family of molecular nanomagnets comprises ring-shaped molecules. Most of them contain even number of antiferromagnetically interacting ions. Only recently the first odd membered antiferromagnetic molecules have been reported [5,6,11,12,19]. They are especially interesting because of magnetic frustration which is expected to appear in this kind of materials.

Precise determination of the energy structure of ring-shaped molecular nanomagnets is necessary to allow the calculations of the state dependent properties such as local magnetisations or correlations [1,2,7,13,14]. An ideal tool for fulfilling this task is the exact diagonalization (ED) of Hamiltonian matrix [3,15]. In this paper we present the testing results of the new version of our ED application. We check the ability of our code to efficiently calculate the energy structure of large spin systems taking advantage of the modern supercomputing facilities.

2 Physical Setup

The ring-shaped molecular nanomagnets can be modelled using the following Heisenberg Hamiltonian:

$$\mathcal{H} = \sum_{j=1}^{n} \left(J_j \boldsymbol{S}_j \cdot \boldsymbol{S}_{j+1} + D_j (S_j^z)^2 - g_j \mu_{_B} \boldsymbol{B} \cdot \boldsymbol{S}_j \right), \qquad (1)$$

where j denotes the positions of magnetic ions within a ring, J_j are the nearest-neighbor exchange integrals between sites j and $j+1$, $\boldsymbol{S}_j$ is the spin operator of the spin S of site j, D_j is the single-ion anisotropy of site j, $\boldsymbol{B}$ is the external magnetic field, g_j is the corresponding Landé factor and μ_B stands for the Bohr magneton. We assume the periodic boundary conditions $(j + n \equiv j)$ because of the ring geometry of the molecule.

3 Exact Diagonalization Technique and HPC Environment

To obtain the precise values of the magnetic properties of the model we use the ED technique. The results obtained by this method are numerically accurate, but a major constraint and challenge is the exponential increase of the size of the matrix defined by $(2S + 1)^n$, where n stands for the size of the system. It is very helpful to exploit the symmetry of a given compound. If the magnetic field is oriented along the z axis, the Hamiltonian takes a quasi-diagonal form (see Fig. 1) in the basis formed by eigenvectors of the total spin projection S^z and can be divided into a number of submatrices labelled by quantum number M and the symmetry of the eigenstates.

We used the MPI [23] library to parallelize the processes of the diagonalization of separate submatrices. For the most efficient use of computing time of all processes we implemented the *Longest Processing Time* algorithm [10]. In the final version of our code we applied ScaLAPACK library [22] which not only accelerates the diagonalization process, but also allows to parallelize the diagonalization of a single submatrix over all the computational cores at a single node with shared memory.

The access for two Tier-0 European supercomputers was awarded under the PRACE Preparatory Access: *Hazel Hen* located at the Stuttgart Supercomputing Center (HLRS) [20] and *MareNostrum* at the Barcelona Supercomputing Center (BSC) [21]. *Hazel Hen* is composed of 7712 compute nodes with a total of 185088 Intel Haswell E5-2680 v3 2.5 GHz compute cores. Each node has 128 GB DDR4 memory at its disposal. *MareNostrum* consists of 3056 IBM DataPlex DX360M4 compute nodes, for a total of 48896 physical Intel Sandy Bridge cores running at 2.6 GHz. The most of the nodes (2752) have 32 GB of DDR3 memory each, however there are 128 nodes with 64 GB and another 128 with 128 GB available. 42 nodes are heterogeneous and consist of both Intel Sandy Bridge and Xeon Phi processors and have 64 GB of memory each. The peak performance of the computers is 7.4 Pflops and 1.1 Pflops respectively.

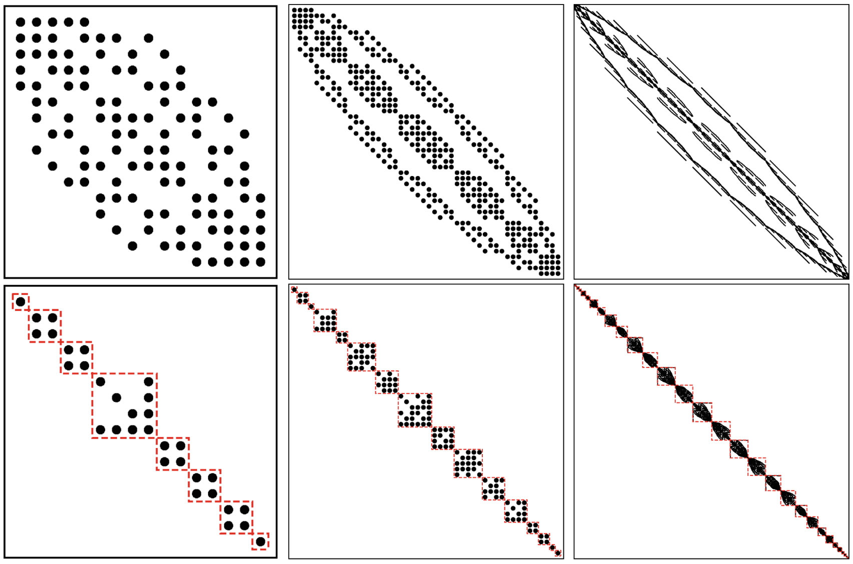


Fig. 1. Structure of the Hamiltonian matrix for chosen small ring-shaped spin systems varying in dimension from 16 (left) to 12288 (right), in simple vector basis (top) and exploiting the symmetry (bottom).

4 Results

We run a set of tests on both computers to check the scalability and performance of the new version of our application. The main task was to check the efficiency of the two-level hybrid parallelization (blocks in matrix representation and SMP parallelization of math kernels to solve the eigenvalue problem for a given block) applied for large systems.

We started with the $S = 3/2$, $n = 10$ model (representing the Cr_{10} molecule) with 4^{10} dimensional spin space. Exploiting the symmetry of the system we obtain 60 submatrices from which the dimension of the largest one is 58152 and it takes 25 GB of memory to store it in double precision. We run the tests for the 8 and 16 SMP threads and without threading on both supercomputers. The results are shown in Fig. 2. We were not able to compute the largest matrices without threading due to time or memory limits, therefore in those cases we use parallelized runs as a reference point for further analysis. We calculate the speedup using the equation:

$$u = \frac{t_{\text{ref}}}{t} p_{\text{ref}}, \tag{2}$$

where t_{ref} is the time of the calculation for the reference point i.e. the one with the lowest number of threads used, t is the time of calculation for the current point and p_{ref} is the number of threads for the reference point. We also show the efficiency using following formula:

$$E = \frac{u}{p}, \tag{3}$$

where p is number of threads used. For small matrix (2245) the sequential run (in the meaning of threading) is used as a reference point. On *Hazel Hen* computer efficiency for 8 and 16 threads stays at the level of 0.8 whereas on *MareNostrum* it drops down below 0.5. Although the sequential execution time is shorter for the latter one (4.7 s comparing to *Hazel Hen's* 6.7 s) it is longer for 16 threads version (0.61 s and 0.53 s respectively). For the largest matrix in this model we compare only the versions for 8 and 16 threads and we notice a small decrease of efficiency to 0.96. It takes about 15300 s to calculate the 58152 matrix on *Hazel Hen* whereas the execution time on *MareNostrum* fluctuates strongly varying between 15300 s and 22200 s.

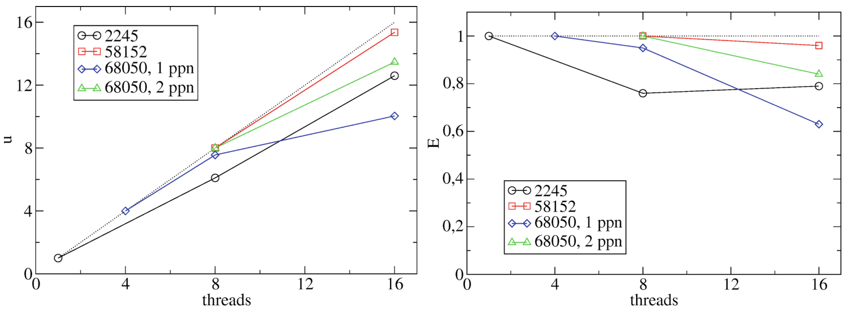


Fig. 2. Speedup u and efficiency E of the SMP parallelization of the diagonalization process for different matrix sizes (denoted in the legend). The point for the lowest number of used threads is always taken as a reference point. For the 68050 matrix the results for 8 threads were obtained using one and two processes per node showing significant difference in performance.

Another system tested on *Hazel Hen* was the $S = 5/2$, $n = 8$ model with 6^8 state space and 80 submatrices, with the dimension of the largest equal 68050 (35 GB). In that case the efficiency for 8 threads is 0.95 for 4 threads as a reference point, for 16 threads however we notice significant drop to 0.63. This may be caused by more intensive access to the shared memory. For 8 threads half of the cores stay idle comparing to 16 threads version. We performed the run for 8 threads in which 2 processes per node were used (16 cores per node) and we noticed the significant drop down of the performance. Using this point as a reference we obtained the efficiency of 0.84 for 16 threads (see Fig. 2).

The system with the same dimensional space as the $S = 3/2$, $n = 10$ is the $S = 1/2$, $n = 20$ model (2^{20}). In this case however the distribution of the submatrices is less favourable (see Fig. 3): 40 blocks of which the dimension of the largest one is 92504 (64 GB). We were able to calculate all the energy levels of the system only when 16 threads were applied. In that case it took over 17 h to diagonalize the largest submatrix. The workload for this example is not even (see Fig. 4) because of the significant dominance of the two largest submatrices diagonalization time over the others.

The test of the $S = 9/2$, $n = 6$ model (10^6) on the *MareNostrum* showed slightly better performance than the other one discussed earlier for this computer. For the largest matrix (27626) and the reference point for 4 threads the efficiency obtained was 0.85 for 8 and 0.63 for 16 threads which is only slightly lower than the results for $S = 5/2$, $n = 8$ model on *Hazel Hen*. We have also

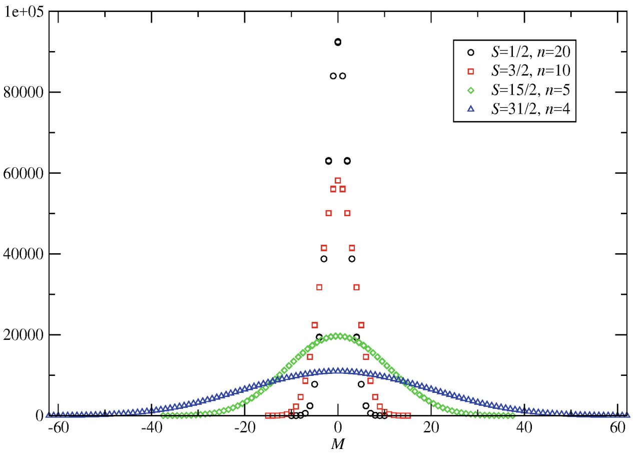


Fig. 3. Sizes of the submatrices (labelled by quantum number M) for systems with different spins S and numbers of sites n but with common state space equal 1048576. With increasing S the number of submatrices increases, however the size of the largest submatrices decreases significantly.

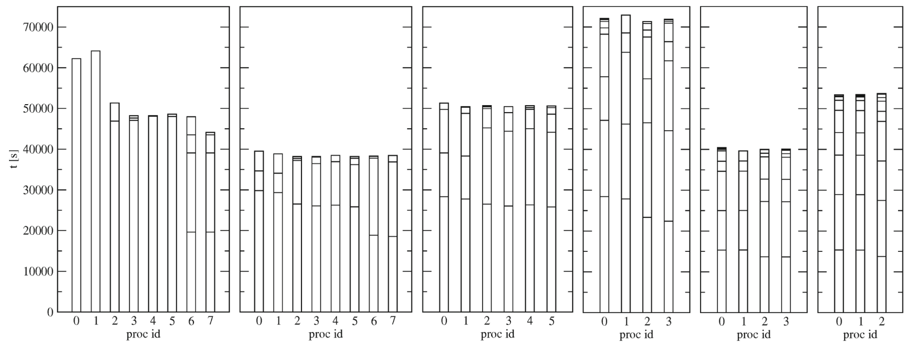


Fig. 4. Computing time balance for different systems, number of processes and threads. First diagram concerns the $S = 1/2$, $n = 20$ using 16 threads, next three - $S = 3/2$, $n = 10$ using 8 threads and last two - $S = 3/2$, $n = 10$ using 16 threads. Uneven workload in the first diagram is caused by the small number of submatrices and large differences in their size. For $S = 3/2$ significant computing time decrease may be noticed when the number of threads increases.

tested the feature of running multiple parameter calculations (so called task farming) for smaller systems (such as $S = 3/2$, $n = 8$ which represents the Cr_8 molecule). We successfully run the application using 300 processes each using 4 threads obtaining well balanced workload over 75 nodes of the *MareNostrum*.

5 Conclusions

Our application was tested on two Tier-0 supercomputers proving usefulness in the computation of energy levels of the models of ring-shaped molecular nanomagnets. We are able to obtain well balanced workload among the assigned computing resources although it is more difficult to achieve it for smaller spin numbers because of the uneven distribution of the Hamiltonian submatrices. Using two-level hybrid parallelization we are able to achieve reasonable efficiency, moreover for large spin systems it is necessary to use threading to fit into time limits of given machine. SMP parallelization is also recommended when running to many processes on one node could exceed the memory limits and another option is to waste the resources by leaving some cores idle. Useful method of obtaining efficient workload for large sets of smaller spin systems is using the task farming.

Above tests were run successfully on both tested supercomputers. For large systems the *Hazel Hen* appeared to have a slightly better performance. With faster one-level parallelization runs the *MareNostrum* proved its ability in calculating smaller models. The results obtained during the Preparatory Access would be useful for Project Access application.

Acknowledgments. We acknowledge PRACE for awarding us access to resource MareNostrum based in Spain at Barcelona Supercomputing Center as well as Hazel Hen in Germany at High Performance Computing Center Stuttgart.

References

1. Antkowiak, M., Kozłowski, P., Kamieniarz, G.: Zero temperature magnetic frustration in nona-membered s = 3/2 spin rings with bond defect. Acta Phys. Pol. A **121**, 1102–1104 (2012)
2. Antkowiak, M., Kozłowski, P., Kamieniarz, G., Timco, G., Tuna, F., Winpenny, R.: Detection of ground states in frustrated molecular rings by in-field local magnetization profiles. Phys. Rev. B **87**, 184430 (2013)
3. Antkowiak, M., Kucharski, Ł., Kamieniarz, G.: Genetic algorithm and exact diagonalization approach for molecular nanomagnets modelling. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 312–320. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_29
4. Ardavan, A., Rival, O., Morton, J., Blundell, S., Tyryshkin, A., Timco, G., Winpenny, R.: Will spin-relaxation times in molecular magnets permit quantum information processing? Phys. Rev. Lett. **98**, 057201 (2007)

5. Baker, M., Timco, G., Piligkos, S., Mathieson, J., Mutka, H., Tuna, F., Kozłowski, P., Antkowiak, M., Guidi, T., Gupta, T., Rath, H., Woolfson, R., Kamieniarz, G., Pritchard, R., Weihe, H., Cronin, L., Rajaraman, G., Collison, D., McInnes, E., Winpenny, R.: A classification of spin frustration in molecular magnets from a physical study of large odd-numbered-metal, odd electron rings. P. Natl. Acad. Sci. USA **109**(47), 19113–19118 (2012)
6. Cador, O., Gatteschi, D., Sessoli, R., Barra, A.L., Timco, G., Winpenny, R.: Spin frustration effects in an oddmembered antiferromagnetic ring and the magnetic Möbius strip. J. Magn. Magn. Mater. **290–291**, 55 (2005)
7. Florek, W., Kaliszan, L.A., Jaśniewicz-Pacer, K., Antkowiak, M.: Numerical analysis of magnetic states mixing in the heisenberg model with the dihedral symmetry. In: EPJ Web of Conferences, vol. 40, p. 14003 (2013). https://doi.org/10.1051/epjconf/20134014003
8. Gatteschi, D., Sessoli, R., Villain, J.: Molecular Nanomagnets. Oxford University Press, Oxford (2006)
9. Georgeot, B., Mila, F.: Chirality of triangular antiferromagnetic clusters as qubit. Phys. Rev. Lett. **104**, 200502 (2010)
10. Graham, R.: Bounds of multiprocessing timing anomalies. SIAM J. Appl. Math. **17**, 416–429 (1969)
11. Hoshino, N., Nakano, M., Nojiri, H., Wernsdorfer, W., Oshio, H.: Templating odd numbered magnetic rings: oxovanadium heptagons sandwiched by β-cyclodextrins. J. Am. Chem. Soc. **131**, 15100 (2009)
12. Kamieniarz, G., Florek, W., Antkowiak, M.: Universal sequence of ground states validating the classification of frustration in antiferromagnetic rings with a single bond defect. Phys. Rev. B **92**, 140411(R) (2015)
13. Kamieniarz, G., Kozłowski, P., Antkowiak, M., Sobczak, P., Ślusarski, T., Tomecka, D., Barasiński, A., Brzostowski, B., Drzewiński, A., Bieńko, A., Mroziński, J.: Anisotropy, geometric structure and frustration effects in molecule-based nanomagnets. Acta Phys. Pol. A **121**, 992–998 (2012)
14. Kozłowski, P., Antkowiak, M., Kamieniarz, G.: Frustration signatures in the anisotropic model of a nine-spin $s = 3/2$ ring with bond defect. J. Nanopart. Res. **13**(11), 6093–6102 (2011)
15. Kozłowski, P., Musiał, G., Antkowiak, M., Gatteschi, D.: Effective parallelization of quantum simulations: nanomagnetic molecular rings. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013. LNCS, vol. 8385, pp. 418–427. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55195-6_39
16. Lehmann, J., Gaita-Ariño, A., Coronado, E., Loss, D.: Spin qubits with electrically gated polyoxometalate molecules. Nature Nanotech. **2**, 312 (2007)
17. Mannini, M., Pineider, F., Sainctavit, P., Danieli, C., Otero, E., Sciancalepore, C., Talarico, A., Arrio, M.A., Cornia, A., Gatteschi, D., Sessoli, R.: Magnetic memory of a single-molecule quantum magnet wired to a gold surface. Nature Mat. **8**, 194 (2009)
18. Timco, G., Carretta, S., Troiani, F., Tuna, F., Pritchard, R., Muryn, C., McInnes, E., Ghirri, A., Candini, A., Santini, P., Amoretti, G., Affronte, M., Winpenny, R.: Engineering the coupling between molecular spin qubits by coordination chemistry. Nature Nanotech. **4**, 173–178 (2009)
19. Yao, H., Wang, J., Ma, Y., Waldmann, O., Du, W., Song, Y., Li, Y., Zheng, L., Decurtins, S., Xin, X.: An iron(III) phosphonate cluster containing a nonanuclear ring. Chem. Commun. **16**, 1745–1747 (2006)

20. Cray XC40 (Hazel Hen). https://www.hlrs.de/en/systems/cray-xc40-hazel-hen/
21. MareNostrum. https://www.bsc.es/innovation-and-services/supercomputers-and-facilities/marenostrum
22. ScaLAPACK – Scalable Linear Algebra PACKage. http://www.netlib.org/scalapack/
23. The Message Passing Interface (MPI) Standard. http://www.mcs.anl.gov/research/projects/mpi/

Application of Numerical Quantum Transfer-Matrix Approach in the Randomly Diluted Quantum Spin Chains

Ryszard Matysiak[1(✉)], Philipp Gegenwart[2,3], Akira Ochiai[4], and Frank Steglich[2]

[1] Institute of Engineering and Computer Education, University of Zielona Góra, ul. prof. Z. Szafrana 4, 65-516 Zielona Góra, Poland
r.matysiak@iibnp.uz.zgora.pl
[2] Max Planck Institute for Chemical Physics of Solids, 01187 Dresden, Germany
[3] Experimental Physics VI, Center for Electronic Correlations and Magnetism, University of Augsburg, 86159 Augsburg, Germany
[4] Center for Low Temperature Science, Tohoku University, Sendai 980-8578, Japan

Abstract. The description of the numerical method of simulation based on the quantum transfer-matrix (QTM) approach is presented for diluted spin $S = 1/2$ chains. Modification of the extrapolation technique has been used to obtain better accuracy of numerical results. The simulations have been performed using the $S = 1/2$ antiferromagnetic Heisenberg model with the transverse staggered field and a uniform magnetic field perpendicular to the staggered field applicable for the diluted compound $(Yb_{1-x}Lu_x)_4As_3$. In the model calculations the fixed microscopic parameters established earlier for the pure system have been assumed and the random impurity distribution has been considered. The experimental field-dependent specific heat of the polydomain diluted $(Yb_{1-x}Lu_x)_4As_3$ sample is compared with that calculated using the HPC resources and providing additional verification of both the QTM method and the physical model.

Keywords: Quantum transfer-matrix method
Segmented Heisenberg antiferromagnet · One-dimensional spin chains

1 Introduction

One-dimensional systems have attracted the interest of physicists and chemists for more then three decades. The theory of ideally uniform $S = 1/2$ antiferromagnetic Heisenberg chain in the magnetic field is well established and the properties observed in real systems are usually well described. A new class of rare-earth compounds like Yb_4As_3 have become the focus of attention. At high-temperatures ($T > 295$ K), Yb_4As_3 is a homogeneous intermediate valent (IV) metal with a cubic crystal structure. The Yb ions reside statistically on four equivalent families of chains along the space diagonals of a cube [1]. At low-temperatures the crystal $\mathrm{Yb_4As_3}$ shrinks along the $\langle 111\rangle$ direction getting a trigonal structure and the $\mathrm{Yb^{3+}}$ ions form a one-dimensional spin $S = 1/2$ chain along the $\langle 111\rangle$ direction. The remaining Yb ions occupy nonmagnetic divalent states. The neutron scattering experiments on pure $\mathrm{Yb_4As_3}$ have confirmed that the excitation spectrum is well described by the one-dimensional $S = 1/2$ isotropic Heisenberg model [2] in the absence of magnetic field. The interchain interactions are small and ferromagnetic, leading to a spin-glass freezing at low temperatures [3] which are below the region analyzed for the diluted samples. The system has attracted a lot of interest due to its striking quantum properties such as the energy gap formation in the magnetic field [4] and Bose-glass effects recently observed [5].

To simulate the finite-temperature properties of the pure $\mathrm{Yb_4As_3}$ and the diluted $\mathrm{(Yb_{1-x}Lu_x)_4As_3}$ systems we consider the $S = 1/2$ anisotropic Heisenberg model with the antisymmetric Dzyaloshinskii–Moriya interaction [6,7]:

$$\begin{aligned}\mathcal{H} = -\Bigg\{ J\sum_{i=1}^{L}\Big[\hat{\mathcal{S}}_i^z\hat{\mathcal{S}}_{i+1}^z + \cos(2\theta)\left(\hat{\mathcal{S}}_i^x\hat{\mathcal{S}}_{i+1}^x + \hat{\mathcal{S}}_i^y\hat{\mathcal{S}}_{i+1}^y\right)\Big] + \\ + J\sin(2\theta)\sum_{i=1}^{L}(-1)^i\left(\hat{\mathcal{S}}_i^x\hat{\mathcal{S}}_{i+1}^y - \hat{\mathcal{S}}_i^y\hat{\mathcal{S}}_{i+1}^x\right) + g_\perp\mu_B B\sum_{i=1}^{L}\hat{\mathcal{S}}_i^x\Bigg\}, \qquad (1)\end{aligned}$$

where L is the number of spins in the chain, J denotes the nearest–neighbour interaction constant, B is the external magnetic field and g is the gyromagnetic ratio.

The Dzyaloshinskii–Moriya interaction is eliminated by rotating the spins in the x–y plane by the angle θ [8]:

$$\begin{aligned}\hat{\mathcal{S}}_i^x &= \cos(\theta)\mathcal{S}_i^x + (-1)^i\sin(\theta)\mathcal{S}_i^y \\ \hat{\mathcal{S}}_i^y &= -(-1)^i\sin(\theta)\mathcal{S}_i^x + \cos(\theta)\mathcal{S}_i^y \\ \hat{\mathcal{S}}_i^z &= \mathcal{S}_i^z \end{aligned} \qquad (2)$$

Then the model is mapped onto

$$\mathcal{H} = -J\sum_{i=1}^{L}\mathbf{S}_i\mathbf{S}_{i+1} - g_\perp\mu_B B^x\sum_{i=1}^{L}S_i^x - g_\perp\mu_B B_s^y\sum_{i=1}^{L}(-1)^i S_i^y, \qquad (3)$$

where $B^x = B\cos(\theta)$, $B_s^y = B\sin(\theta)$ and B is the uniform external magnetic field perpendicular to the one–dimensional spin–chain. If the magnetic field B is applied along the spin–chain we replace $\theta = 0$ and $g_\perp = g_\parallel$.

Using hamiltonian (3) we can describe thermodynamic properties of the pure and diluted system both in the absence and the presence of external magnetic field. In the model (3) all the parameters are fixed and the values arise from the earlier studies [8–10]. The g factors for the applied field along the directions parallel and perpendicular to the spin chain amount to $g_\parallel = 3.0$ and $g_\perp = 1.3$, the exchange coupling $J/k_B = -28$ K and the transformation angle corresponds to the value $\tan(\theta) = 0.19$.

2 Description of the Model and the Simulation Technique

To characterize the finite-temperature properties of one-dimensional systems we need to calculate the free energy $\mathcal{F} = -k_B T \ln(\mathcal{Z})$ which is related to the partition function $\mathcal{Z}$ defined as:

$$\mathcal{Z} = \mathbf{Tr} e^{-\beta \mathcal{H}}\,, \tag{4}$$

where $\beta = 1/(k_B T)$. The values of matrix elements of $e^{-\beta \mathcal{H}}$ cannot be found exactly for large L because of noncommuting operators in $\mathcal{H}$ so we look for the systematic approximants $\mathcal{Z}_m$ to the partition function $\mathcal{Z}$, where m is the natural number (the Trotter number). In the framework of the transfer-matrix method [11,12], first we divide the Hamiltonian (3) into two noncommuting parts $\mathcal{H}^{odd}$, $\mathcal{H}^{even}$:

$$\mathcal{H} = \mathcal{H}^{odd} + \mathcal{H}^{even} = (\mathcal{H}_{1,2} + \ldots + \mathcal{H}_{L-1,L}) + (\mathcal{H}_{2,3} + \ldots + \mathcal{H}_{L,1}) \tag{5}$$

each part defined by the commuting spin-pair operators $\mathcal{H}_{i,i+1}$:

$$\mathcal{H}_{i,i+1} = -J\mathbf{S}_i\mathbf{S}_{i+1} - \frac{1}{2} g_\perp \mu_B \left[B^x \left(S_i^x + S_{i+1}^x \right) + (-1)^i B_s^y \left(S_i^y + S_{i+1}^y \right) \right]. \tag{6}$$

For the infinite chains in the limit $L \to \infty$ the partition function $\mathcal{Z}_m$ is equal to the highest eigenvalue [12–16] of the global transfer matrix $\mathcal{W}$ [17]:

$$Z_m = Tr\,(\mathcal{W})^L\,, \tag{7}$$

where

$$\mathcal{W} = \prod_{r=1}^{2m} \mathcal{L}_{r,r+1} = (\mathcal{P}\mathcal{L}_{1,2})^{2m}\,. \tag{8}$$

The local transfer matrix $\mathcal{L}$ whose elements depend on $\mathcal{V}$ elements is defined as:

$$\langle S_{r,i}^z S_{r+1,i}^z \mid \mathcal{L}_{r,r+1} \mid S_{r,i+1}^z S_{r+1,i+1}^z \rangle = \langle S_{r,i}^z S_{r,i+1}^z \mid \mathcal{V}_{i,i+1} \mid S_{r+1,i}^z S_{r+1,i+1}^z \rangle\,, \tag{9}$$

where $\mathcal{V}_{i,i+1} = e^{-\beta \mathcal{H}_{i,i+1}/m}$ and the shift operator $\mathcal{P}$:

$$\mathcal{P} \equiv \sum_{S_1^z} \ldots \sum_{S_{2m}^z} \mid S_2^z S_3^z \ldots S_{2m-1}^z S_{2m}^z S_1^z \rangle \langle S_1^z S_2^z S_3^z \ldots S_{2m}^z \mid. \tag{10}$$

In the absence of magnetic field, the numerical calculations can be performed using the pure $S = 1/2$ isotropic Heisenberg model:

$$\mathcal{H} = -J \sum_{i=1}^{L} \mathbf{S}_i \mathbf{S}_{i+1}. \tag{11}$$

To confirm the reliability of the model (11), the QTM numerical results are compared with the Bethe ansatz (BA) results [18] which is an exact method and then to establish the value of the exchange coupling J, the simulation results are compared with the experimental specific heat data for $\mathrm{Yb_4As_3}$ compound [9]. The values of others parameters in Hamiltonian (3) are estimated from comparision with field-dependent specific heat experimental results of $\mathrm{Yb_4As_3}$ [9].

In the presence of the external magnetic field applied perpendicular to the spin-chain in the model (3), the system is nonuniform because of induced staggered field, so we need define the partition function accordingly. Then the m-th approximant $\mathcal{Z}_m$ of the partition function $\mathcal{Z}$ is related to the two global transfer matrix $\mathcal{W}_1$ and $\mathcal{W}_2$:

$$Z_m = Tr\left(\mathcal{W}_1\mathcal{W}_2\right)^{L/2}, \text{ where } \mathcal{W}_1 = \left(\mathcal{P}^2\mathcal{L}_{1,2}\right)^m, \mathcal{W}_2 = \left(\mathcal{P}^2\mathcal{L}_{2,3}\right)^m \tag{12}$$

To calculate the partition function for finite chains we need to define two vectors which act in a Hilbert space $\mathcal{H}^{2m}$ [11,17,19]:

$$\mid a\rangle = \sum_{\{S^z\}}\prod_{r=1}^{2m}\delta_{S^z_{2r-1},S^z_{2r}} \mid S^z_1 \ldots S^z_{2m}\rangle\,, \tag{13}$$

$$\mid b\rangle = \sum_{\{S^z\}}\prod_{r=1}^{2m}\delta_{S^z_{2r},S^z_{2r+1}} \mid S^z_1 \ldots S^z_{2m}\rangle\,. \tag{14}$$

Then the m-th approximant of the partition function is different for odd and even number of sites in the chains:

$$\mathcal{Z}_m = \langle b \mid (\mathcal{W}_1\mathcal{W}_2)^{(L-1)/2} \mid a\rangle \quad \text{for} \quad \text{odd} \quad L\,, \tag{15}$$

$$\mathcal{Z}_m = \langle b \mid (\mathcal{W}_1\mathcal{W}_2)^{L/2} \mid a\rangle \quad \text{for} \quad \text{even} \quad L\,. \tag{16}$$

Taking the quantum limit $m \to \infty$ in (12), the partition function $\mathcal{Z}$ can be estimated and the corresponding thermodynamic function can be calculated. In order to improve the accuracy of the extrapolation for low temperatures, we have calculated the specific heat using the extrapolation polynomial of the degree k $(k = 1, \ldots, k_{max})$ in $1/m^2$.

$$C_k\left(\frac{1}{m^2}\right) = \sum_{j=0}^{k} a_j \cdot \left(\frac{1}{m^2}\right)^j. \tag{17}$$

The approximants C_m correspond to $m_{min} \leq m \leq m_{max}$. For practical reasons, in our procedure $k_{max} \leq 10$. The value of the highest Trotter index m_{max} is fixed and amounts to 14 or 15 in low temperatures and 13 in high temperatures. The value m_{min} is subject to variation in the region $2 \leq m_{min} \leq m_{max} - 1$.

The extrapolation procedure starts with $m_{min} = 2$ and is continued till $m = m_{max} - 1$. In each step the number of fitted points n $(n = m_{max} - m_{min} + 1)$ is fixed and the extrapolations are performed with polynomials of the degree k $(1 \leq k \leq n - 1$, but not more than 10). In this way for a given field and

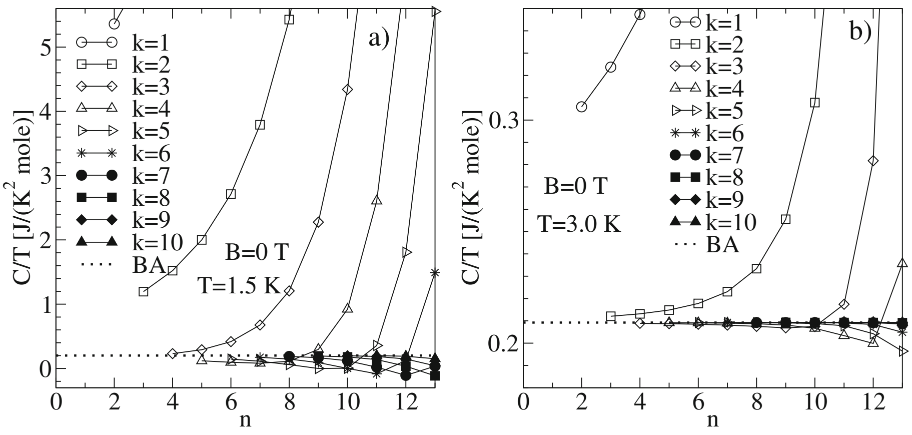


Fig. 1. The extrapolated values of specific heat versus the number of points n for which the polynomials are constructed. Each particular plot corresponds to the polynomial of a given degree k. The figures (a) and (b) have been drawn for infinite chains and $B = 0$ T with $T = 1.5$ K and $T = 3$ K, respectively. The QTM results are presented by symbols in reference to the Bethe ansatz results (dotted line).

temperature we obtain a set of extrapolated values for different values of n and k and we can present the variation of the data with n for the fixed degree k of the polynomial.

The results of the analysis of the extrapolated specific heat values according to the above procedure are shown in Fig. 1. To check the accuracy of the extrapolations performed for infinite chains, the plots were referred to the value obtained on the basis of the Bethe ansatz approach [18] which is shown as a dotted line in Fig. 1. As demonstrated, the convergence depends significantly on the degree k of the polynomial. The convergence of the extrapolated values is much better for higher temperatures and implies higher accuracy of the numerical estimates.

To describe the magnetic specific heat of diluted $(Yb_{1-x}Lu_x)_4As_3$ we need to calculate the contribution C_L of a finite chain with L sites and to find the probability distribution. Assuming the uniform distribution of non–magnetic Lu–ions among the chains, each site in the Yb^{3+}–chain is randomly occupied by a magnetic ion with a probability $p = 1 - x$. The probability of finding a chain with L sites is $p^L(1-p)^2$. The number of L–chains is $n_L = Np^L(1-p)^2$ ($N \to \infty$ is the total chain length and is much larger than the cluster length) and the total number of all L-chains is given by the following sum:

$$n_t = \sum_{L=1}^{\infty} n_L = N \sum_{L=1}^{\infty} p^L(1-p)^2 = N(1-p)^2 \sum_{L=1}^{\infty} p^L = N(1-p)p. \qquad (18)$$

Finally, we obtain the specific heat per spin:

$$C = x \sum_{L=1}^{\infty} \omega_L C_L, \tag{19}$$

where the probability distribution of chains with L sites [20]:

$$\omega_L = p^{L-1}(1-p). \tag{20}$$

For each temperature we have calculated within the QTM technique the specific heat $C(L)$ for $L \leq 30$. Our specific heat results for two temperatures ($T = 7$ and $T = 14$ K) and two configurations of magnetic field are shown in Fig. 2. The open symbols represent the specific heat for various numbers of sites L. The filled symbols represent specific heat data whose we have obtained using QTM technique for infinite chains. Those results are consistent with exact Bethe ansatz results [18]. For sufficiently large $L > L_0$ we can estimate the specific heat by the linear function and finally, specific heat for whole range of L [21]:

$$C = x^2 \cdot \sum_{L=1}^{L_0} (1-x)^L \cdot C(L)L + C(L > L_0). \tag{21}$$

We emphasize that the value L_0 exceeds the sizes where the domain of the exact diagonalization technique is applicable [21,22]. Moreover, the computational complexity of the QTM estimates C_L increases lineary with L which is additional advantage with respect to other techniques [14].

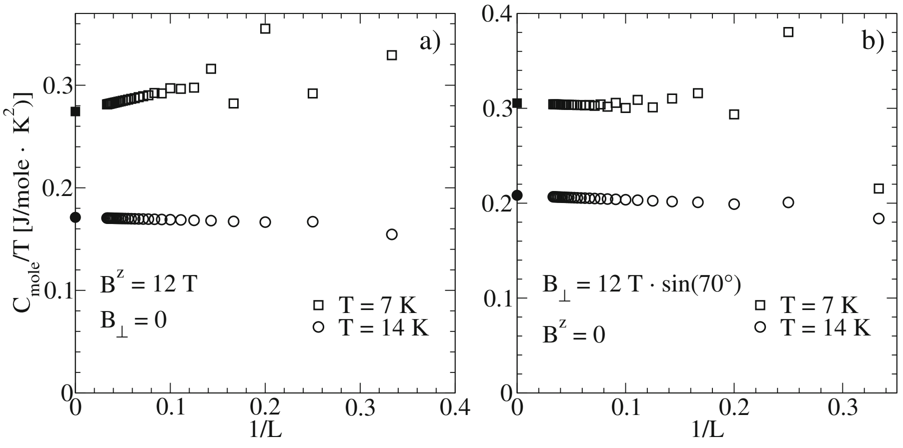


Fig. 2. Size dependence of specific heat calculated for finite segments with different number of sites L (open symbols). The filled symbols are the simulation results corresponding to the infinite chains. The external magnetic field is applied along the spin chain (a) and perpendicular to the spin chain (b).

3 The Results of the Numerical Simulations and Discussion

The QTM method described has been tested with respect to convergence in the Trotter index m and the size L, and has been applied to simulation via the Heisenberg model (3) of the experimental field-dependent specific heat data for diluted $(Yb_{1-x}Lu_x)_4As_3$ system of particular interest [5]. In order to calculate magnetic specific heat we assumed that 25% of the domains in the poly-domain single cristal sample were oriented in parallel and 75% were oriented perpendicular to the direction of the applied field. The effective magnetic field $B_{eff} = B\sin(70°)$ is assumed to be oriented in the direction perpendicular to the chain [5]. Finally, the numerical specific heat result is given by:

$$C(T,B) = 0.75 \cdot C_{\perp}(T, B_{eff}) + 0.25 \cdot C_{\parallel}(T,B)\,. \tag{22}$$

The results of simulations and experimental data are shown in the Fig. 3 for the concentration of impurities $x = 3\%$ and for two different values of magnetic field $B = 6$ T and $B = 15$ T. The experimental data supplement those published before [5] and were measured using the same protocol.

As demonstrated in Fig. 3, the reliability of the simulation method and the model applied have been strongly verified. The simulations require the HPC recourses as far as the temporal and memory complexity are concerned. They can be efficiently parallelized [17,23–25] as the traces in (7) and (12) refer to

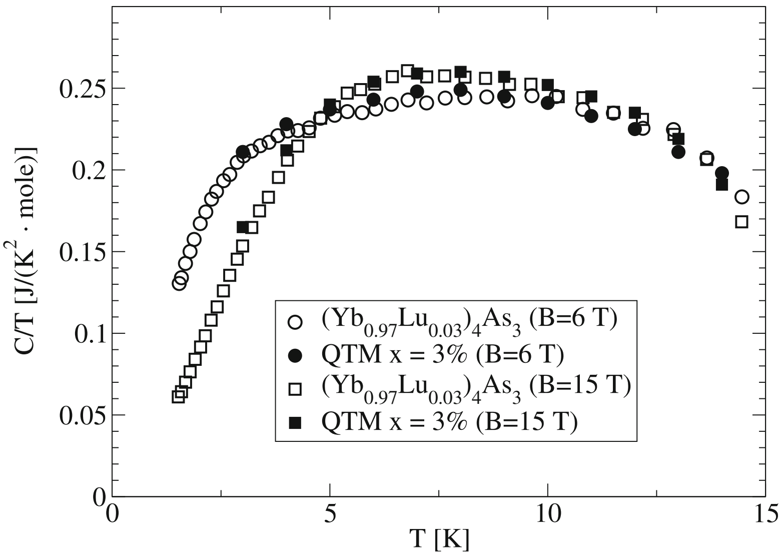


Fig. 3. Comparision between experiment and numerical results for the diluted samples subject to an applied field. The field applied is $B = 6$ T and 15 T. The concentration of nonmagnetic impurities $x = 3\%$.

the independent vectors in the Hilbert space. The finite size contributions C_L, depending on temperature and field, can be also evaluated for each L separately, i.e. in parallel.

In conclusion, we have presented the numerical QTM approach to characterize the finite temperature magnetic properties of the diluted $(Yb_{1-x}Lu_x)_4As_3$ system. We have successfully compared the results of our QTM simulations with the experimental findings. We have enhanced evidence that the spin model worked out for the pure compound Yb_4As_3 can also explain the specific heat results for the diluted systems, using their random distribution.

Acknowledgement. We thank for an access to the HPC resources in PSNC Poznań (Poland).

References

1. Köppen, M., Lang, M., Helfrich, R., Steglich, F., Thalmeier, P., Schmidt, B., Wand, B., Pankert, D., Benner, H., Aoki, H., Ochiai, A.: Phys. Rev. Lett. **82**, 4548 (1999)
2. Kohgi, M., Iwasa, K., Mignot, J.-M., Ochiai, A., Suzuki, T.: Phys. Rev. B **56**, R11388 (1997)
3. Schmidt, B., Aoki, H., Cichorek, T., Custers, J., Gegenwart, P., Kohgi, M., Lang, M., Langhammer, C., Ochiai, A., Paschen, S., Steglich, F., Suzuki, T., Thalmeier, P., Wand, B., Yaresko, A.: Phys. B **300**, 121 (2001)
4. Kohgi, M., Iwasa, K., Mignot, J.-M., Fak, B., Gegenwart, P., Lang, M., Ochiai, A., Aoki, H., Suzuki, T.: Phys. Rev. Lett. **86**, 2439 (2001)
5. Kamieniarz, G., Matysiak, R., Gegenwart, P., Ochiai, A., Steglich, F.: Phys. Rev. B **94**, 100403(R) (2016)
6. Oshikawa, M., Ueda, K., Aoki, H., Ochiai, A., Kohgi, M.: J. Phys. Soc. Jpn. **68**, 3181 (1999)
7. Shiba, H., Ueda, K., Sakai, O.: J. Phys. Soc. Jpn. **69**, 1493 (2000)
8. Shibata, N., Ueda, K.: J. Phys. Soc. Jpn. **70**, 3690 (2001)
9. Matysiak, R., Kamieniarz, G., Gegenwart, P., Ochiai, A.: Phys. Rev. B **79**, 224413 (2009)
10. Iwasa, K., Kohgi, M., Gukasov, A., Mignot, J.-M., Shibata, N., Ochiai, A., Aoki, H., Suzuki, T.: Phys. Rev. B **65**, 052408 (2002)
11. Delica, T., Leschke, H.: Phys. A **168**, 736 (1990)
12. Kamieniarz, G., Bieliński, M., Renard, J.-P.: Phys Rev. B **60**, 14521 (1999)
13. Kamieniarz, G., Matysiak, R., D'Auria, A.C., Esposito, F., Esposito, U.: Phys. Rev. B **56**, 645 (1997)
14. D'Auria, A.C., Esposito, U., Esposito, F., Gatteschi, D., Kamieniarz, G., Wałcerz, S.: J. Chem. Phys. **109**, 1613 (1998)
15. Barasiński, A., Kamieniarz, G., Drzewiński, A.: Comput. Phys. Commun. **182**, 2013 (2011)
16. Barasiński, A., Kamieniarz, G., Drzewiński, A.: Phys. Rev. B **86**, 214412 (2012)
17. Kamieniarz, G., Matysiak, R.: Comput. Mater. Sci. **28**, 353 (2003)
18. Johnston, D.C., Kremer, R.K., Troyer, M., Wang, X., Klümper, A., Bud'ko, S.L., Panchula, A.F., Canfield, P.C.: Phys. Rev. B **61**, 9558 (2000)
19. Kamieniarz, G., Matysiak, R.: J. Comput. Appl. Math. **189**, 471 (2006)
20. Asakawa, H., Matsuda, M., Minami, K., Yamazaki, H., Katsumata, K.: Phys. Rev. B **57**, 8285 (1998)

21. Matysiak, R., Gegenwart, P., Ochiai, A., Antkowiak, M., Kamieniarz, G., Steglich, F.: Phys. Rev. B **88**, 224414 (2013)
22. Kamieniarz, G., Matysiak, R., D'Auria, A.C., Esposito, F., Benelli, C.: Eur. Phys. J. B **23**, 183 (2001)
23. Kamieniarz, G., Matysiak, R., D'Auria, A.C., Esposito, F., Benelli, C.: Application of parallel computing in the transfer — matrix simulations of the supramolecules Mn_6 and $Ni_1$2. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 502–509. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-48086-2_55
24. Kamieniarz, G., Matysiak, R.: Deterministic large-scale simulations of the low-dimensional magnetic spin systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2003. LNCS, vol. 3019, pp. 1091–1098. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24669-5_141
25. Antkowiak, M., Kucharski, Ł., Matysiak, R., Kamieniarz, G.: Comput. Methods Sci. Technol. **22**, 87 (2016)

Minisymposium on High Performance Computing Interval Methods

A New Method for Solving Nonlinear Interval and Fuzzy Equations

Ludmila Dymova and Pavel Sevastjanov(✉)

Institute of Computer and Information Sciences,
Czestochowa University of Technology, Dabrowskiego 73,
42-201 Czestochowa, Poland
{dymowa,sevast}@icis.pcz.pl

Abstract. In this paper, a new concept called "interval extended zero" method which recently was used for solving interval and fuzzy linear equations is adapted to the solution of nonlinear interval and fuzzy equations. The known "test" example of quadratic fuzzy equation is used to perform the advantages of a new method. In this example, only the positive solution can be obtained using known methods, whereas generally a negative fuzzy root can exits too. The sources of this problem are clarified. It is shown that opposite to the known methods, a new approach makes it possible to get both the positive and negative solutions of quadratic fuzzy equation. Generally, the developed method can be applied for solving a wide range of nonlinear interval and fuzzy equations if some initial constraints on the bounds of solution are known.

Keywords: Interval nonlinear equation · Fuzzy nonlinear equation

1 Introduction

Although the problem of solving nonlinear interval and fuzzy equations is of perennial interest [1–6,8,11,13,14], to date there are no universal methods for solving such equations proposed in the literature. Therefore, this problem is now open.

There are many different numerical methods proposed in the literature for solving interval and fuzzy equations including such complicated as Neural Net solutions [4,5] and fuzzy extension of Newton method [1,2], but only particular solutions valid in specific conditions were obtained. For example, only a positive root of the quadratic fuzzy equation have been obtained in [1,3], although a negative solution can exist too.

To alleviate these problems in the case of linear interval and fuzzy equations, in [15,16] we proposed a new "interval extended zero" method. It was presented earlier as a useful heuristic [9] which makes it possible to solve the system of linear interval equations. In the current paper, we show that "interval extended zero" method may be successfully used for solving nonlinear interval and fuzzy equations. Using the same example as in [1,3], we get not only the positive fuzzy solution of quadratic fuzzy equation, but the negative too.

The rest of the paper is set out as follows. Section 2 is devoted to presentation of "interval extended zero" method in the case of quadratic interval and fuzzy equations. Section 3 concludes with some remarks.

2 Fuzzy Solutions of Nonlinear Interval and Fuzzy Equations

The general approach described in previous section can be adapted for solving nonlinear equations. The method we develop in this section can be applied for solving a wide range of nonlinear interval and fuzzy equations if some initial constraints on the bounds of solutions are known.

Nevertheless, to present our method more transparent, we consider the well known example of quadratic fuzzy equation [1,3] that factually can be treated as the "test" task:

$$ax^2 + bx = c, \tag{1}$$

where $a = (3, 3, 4, 5), b = (1, 2, 3), c = (1, 1, 2, 3)$ are trapezoidal and triangular fuzzy numbers (see Fig. 1). In [1,3], the positive fuzzy solution of Eq. (1) with these fuzzy parameters was obtained (see Fig. 2). Although it is stated in [1,3] that Eq. (1) have no a negative fuzzy root, we obtain such root. Moreover, using the results of our analysis in [15,16], we clarify the origins of the problem the authors of [1,3] faced with.

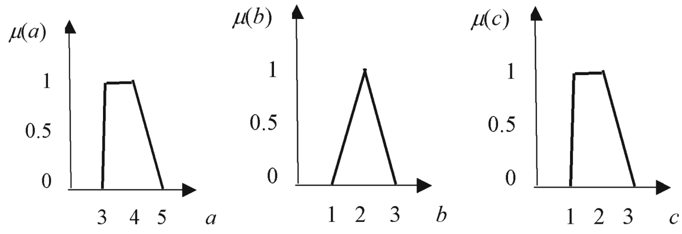


Fig. 1. Fuzzy parameters of Eq. (1)

As we prefer to use the α-cut representation of fuzzy numbers, fuzzy Eq. (1) is decomposed to the set of interval equations on the corresponding α-cuts. Obviously, when dealing with Eq. (1), on the lowest α-cut, i.e., for $\alpha = 0$, we get

$$[3, 5]x^2 + [1, 3]x = [1, 3]. \tag{2}$$

Consider the case of $[x] > 0$, i.e., $\underline{x}, \overline{x} > 0$. Then from Eq. (2) we obtain

$$3\underline{x}^2 + \underline{x} = 1, 5\overline{x}^2 + 3\overline{x} = 3$$

and finally we obtain the approximate formal (algebraic) solution $\underline{x} = 0.4343, \overline{x} = 0.5307$.

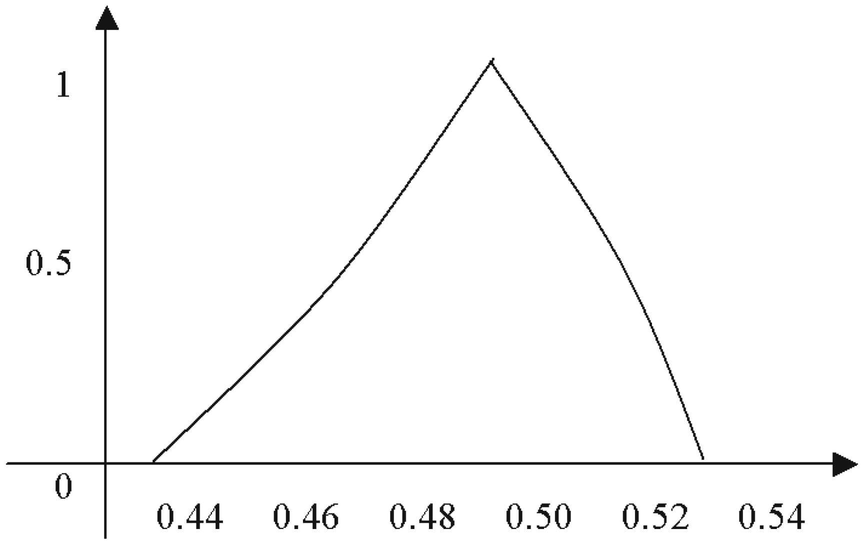


Fig. 2. Positive fuzzy root of Eq. (1) obtained in [1, 4]

Nevertheless, in the assumption of negative $x < 0$, i.e., $\underline{x}, \overline{x} < 0$, from Eq. (2) we get

$$3\underline{x}^2 + 3\underline{x} = 1, 5\overline{x}^2 + \overline{x} = 3$$

and "...$\underline{x} \cong -0.629, \overline{x} \cong -0.98$ and therefore the negative root does not exist" [1].

To clarify the origins of this problem, let as consider the simplest interval linear equation $[a]x = [b]$, where $[a]$ and $[b]$ are intervals. Using classical interval arithmetic rules [12], from this equation we get $[\underline{ax}, \overline{ax}] = [\underline{b}, \overline{b}]$ and finally: $\underline{x} = \frac{\underline{b}}{\underline{a}}$, $\overline{x} = \frac{\overline{b}}{\overline{a}}$.

Consider some examples.

For $[a] = [3, 4]$, $[b] = [1, 2]$ from $\underline{x} = \frac{\underline{b}}{\underline{a}}$, $\overline{x} = \frac{\overline{b}}{\overline{a}}$ we get $\underline{x} = 0.333, \overline{x} = 0.5$, for $[a] = [1, 2]$, $[b] = [3, 4]$ we get $\underline{x} = 3, \overline{x} = 2$, for $[a] = [3, 4]$, $[b] = [0.7, 0.8]$ we get $\underline{x} = 0.23, \overline{x} = 0.2$.

It is seen that interval equation $[a]x = [b]$ often have only inverted formal interval solution, i.e., such that $\underline{x} > \overline{x}$. Obviously, in the case of the degenerated $[b]$, i.e., $\underline{b} = \overline{b}$ only inverted solutions can be obtained. Of course in the frameworks of directed interval arithmetic, modal interval arithmetic or the extended interval arithmetic developed by Kaucher [10], inverted interval solutions make a sense from purely mathematical point of view. But generally it is hard to interpret inverted intervals in economic or mechanic terms.

It is seen that exact correct (noninverted) formal solutions of interval equation $[a]x = [b]$ exist only in some special conditions. Therefore, only what we can say is that the interval equation in the form of $[a]x = [b]$ is not a reliable representation of the interval equation if we are looking for approximate formal (algebraic) solution. On the other hand, the united solution set (often called simply solution set), tolerable solution set and controllable solution set [17] can be analyzed, but this is out of scope of the current paper.

We can see that Eq. (2) has the structure similar to that of $[a]x = [b]$ which is an unreliable representation of the interval equation problem if we aim to obtain an approximate noninverted formal solution. As with lowering of width of the right hand side of $[a]x = [b]$ this equation can provide inverted interval roots, we can expect such results from nonlinear Eq. (2) as well. For example, changing $c = (1, 1, 2, 3)$ by the more narrow value $c^1 = (1, 1.5, 2)$, instead of (2) we get the equation $[3, 5]x^2 + [1, 3]x = [1, 2]$ and finally $3\underline{x}^2 + \underline{x} = 1, 5\overline{x}^2 + 3\overline{x} = 2$. The positive roots of these equations are $\underline{x} = 0.4343, \overline{x} = 0.4$. So we have inverted interval solution, $\underline{x} > \overline{x}$.

To avoid above problems, at first we represent Eq. (1) on each α-cut in the form of interval equation $[a]x^2 + [b]x - [c] = 0$.

In the spirit of "interval extended zero" method described in [15,16], we represent Eq. (1) in the following form:

$$[\underline{a}, \overline{a}][\underline{x}, \overline{x}]^2 + [\underline{b}, \overline{b}][\underline{x}, \overline{x}] - [\underline{c}, \overline{c}] = [-y, y], \tag{3}$$

where y is the undefined parameter (see [15,16]) and index α is omitted for the simplicity. Using conventional interval arithmetic rules, from Eq. (3) we get

$$[\underline{ax} + \underline{b}, \overline{ax} + \overline{b}][\underline{x}, \overline{x}] - [\underline{c}, \overline{c}] = [-y, y]. \tag{4}$$

Firstly, consider the case of positive interval root of Eq. (4), i.e., $\underline{x}, \overline{x} > 0$. Then from (4) we obtain

$$\underline{ax}^2 + \underline{bx} - \overline{c} = -y, \overline{ax}^2 + \overline{bx} - \underline{c} = y. \tag{5}$$

The sum of Eq. (5) results in

$$\underline{ax}^2 + \underline{bx} - \overline{c} + \overline{ax}^2 + \overline{bx} - \underline{c} = 0. \tag{6}$$

As in the case of real valued a, b, c, the positive root of (1) is presented by the expression $x = \frac{-b+\sqrt{b^2+4ac}}{2a}$, the "natural constraints" on the positive interval solution of (6) can be represented as follows:

$$x_{min} = \frac{-\overline{b} + \sqrt{\underline{b}^2 + 4\underline{ac}}}{2\overline{a}}, x_{max} = \frac{-\underline{b} + \sqrt{\overline{b}^2 + 4\overline{ac}}}{2\underline{a}}. \tag{7}$$

Similar to the case of linear interval equation (see [15,16]) we consider the real valued (degenerated) solution of Eq. (6), x_m, as the natural top bound for positive $\underline{x}$, i.e., $\underline{x} \leq x_m$ and bottom bound for positive $\overline{x}$, i.e., $x_m \leq \overline{x}$. For the case of $\underline{x} = \overline{x} = x_m$ from (6) we get

$$x_m = \frac{-(\underline{b} + \overline{b}) + \sqrt{(\underline{b} + \overline{b})^2 + 4(\underline{a} + \overline{a})(\underline{c} + \overline{c})}}{2(\underline{a} + \overline{a})}. \tag{8}$$

Equation (6) with described above constraints $x_{\min} \leq \underline{x} \leq x_m, x_m \leq \overline{x} \leq x_{\max}$ is a typical Constraint Satisfaction Problem [7] and its interval solution can be obtained. From Eq. (6) we get the expressions

$$\underline{x} = \underline{f}(\overline{x}) = \frac{-\underline{b} + \sqrt{\underline{b}^2 + 4\underline{a}(\underline{c} + \overline{c} - \overline{a}\overline{x}^2 - \overline{b}\overline{x})}}{2\underline{a}}, \overline{x} = \overline{f}(\underline{x}) = \frac{-\overline{b} + \sqrt{\overline{b}^2 + 4\overline{a}(\underline{c} + \overline{c} - \underline{a}\underline{x}^2 - \underline{b}\underline{x})}}{2\overline{a}}.$$

Generally, the interval solution of above constraint satisfaction problem can be represented as follows:

$$[\underline{x}] = [x_{\min}, x_m] \cap [\underline{x}_1^*, \underline{x}_2^*], [\overline{x}] = [x_m, x_{\max}] \cap [\overline{x}_1^*, \overline{x}_2^*], \tag{9}$$

where

$\underline{x}_1^* = \min \underline{f}(\overline{x}),\ \underline{x}_2^* = \max \underline{f}(\overline{x})\ (x_m \leq \overline{x} \leq x_{\max});$
$\overline{x}_1^* = \min \overline{f}(\underline{x}),\ \overline{x}_2^* = \max \overline{f}(\underline{x})\ (x_{\min} \leq \underline{x} \leq x_m).$

It is easy to see that in our case

$$\underline{x}_1^* = \frac{-\underline{b} + \sqrt{\underline{b}^2 + 4\underline{a}(\underline{c} + \overline{c} - \overline{a}x_{\max}^2 - \overline{b}x_{\max})}}{2\underline{a}}, \underline{x}_2^* = \frac{-\underline{b} + \sqrt{\underline{b}^2 + 4\underline{a}(\underline{c} + \overline{c} - \overline{a}x_m^2 - \overline{b}x_m)}}{2\underline{a}},$$
$$\overline{x}_1^* = \frac{-\overline{b} + \sqrt{\overline{b}^2 + 4\overline{a}(\underline{c} + \overline{c} - \underline{a}x_m^2 - \underline{b}x_m)}}{2\overline{a}}, \overline{x}_2^* = \frac{-\overline{b} + \sqrt{\overline{b}^2 + 4\overline{a}(\underline{c} + \overline{c} - \underline{a}x_{\min}^2 - \underline{b}x_{\min})}}{2\overline{a}}.$$

From (9) we obtain the following interval solution

$$[\underline{x}] = [\underline{x}_{\min}, \underline{x}_{\max}], [\overline{x}] = [\overline{x}_{\min}, \overline{x}_{\max}], \tag{10}$$

where $\underline{x}_{\min} = \max(x_{\min}, \underline{x}_1^*)$, $\underline{x}_{\max} = \min(x_m, \underline{x}_2^*)$, $\overline{x}_{\min} = \max(x_m, \overline{x}_1^*)$, $\overline{x}_{\max} = \min(x_{\max}, \overline{x}_2^*)$.

As in the linear case (see [15,16]), substituting the widest possible interval solution $[\underline{x}_{\min}, \overline{x}_{\max}]$ into Eq. (4) we get the maximal value of y, i.e., $y_{\max}$, and substituting in this equation the shortness possible solution $[\underline{x}_{\max}, \overline{x}_{\min}] = [x_m, x_m]$ we obtain $y_{\min}$. As in the linear case, the formal interval solution (10) factually represents the continuous set of nested interval solutions of Eq. (4) and we can use the expression (see [15,16]) $\eta = 1 - \frac{y - y_{min}}{y_{max} - y_{min}}$ to calculate the values of y on the η-cuts. For η rising from 0 to 1 using the above expression for η we get the values of y and substituting them into (5) we obtain the set of interval solutions $[\underline{x}, \overline{x}]_\eta$ on the corresponding η-cuts. In Fig. 3, the positive fuzzy solution for the lowest α-cut ($a = [3, 5]$, $b = [1.3]$, $c = [1, 3]$) is presented.

Using the proposed method, the negative root ($\underline{x}, \overline{x} < 0$) of fuzzy Eq. (4) can be obtained as well. For this case we get the following set of expressions:

$$\underline{a}\overline{x}^2 + \overline{b}\underline{x} - \overline{c} = -y, \overline{a}\underline{x}^2 + \underline{b}\overline{x} - \underline{c} = y. \tag{11}$$

$$\underline{a}\overline{x}^2 + \overline{b}\underline{x} - \overline{c} + \overline{a}\underline{x}^2 + \underline{b}\overline{x} - \underline{c} = 0. \tag{12}$$

$$x_m = \frac{-(\underline{b} + \overline{b}) - \sqrt{(\underline{b} + \overline{b})^2 + 4(\underline{a} + \overline{a})(\underline{c} + \overline{c})}}{2(\underline{a} + \overline{a})}. \tag{13}$$

$$x_{min} = \frac{-\overline{b} - \sqrt{\overline{b}^2 + 4\overline{ac}}}{2\underline{a}}, x_{max} = \frac{-\underline{b} - \sqrt{\underline{b}^2 + 4\underline{ac}}}{2\overline{a}}. \tag{14}$$

$$\underline{x} = \underline{f}(\overline{x}) = \frac{-\overline{b} - \sqrt{\overline{b}^2 + 4\overline{a}(\underline{c} + \overline{c} - \underline{a}\overline{x}^2 - \underline{b}\overline{x})}}{2\overline{a}},$$
$$\overline{x} = \overline{f}(\underline{x}) = \frac{-\underline{b} - \sqrt{\underline{b}^2 + 4\underline{a}(\underline{c} + \overline{c} - \overline{a}\underline{x}^2 - \overline{b}\underline{x})}}{2\overline{a}}. \tag{15}$$

$$[\underline{x}] = [x_{\min}, x_m] \cap [\underline{x}_1^*, \underline{x}_2^*], [\overline{x}] = [x_m, x_{\max}] \cap [\overline{x}_1^*, \overline{x}_2^*], \tag{16}$$

where $\underline{x}_1^* = \min \underline{f}(\overline{x})$, $\underline{x}_2^* = \max \underline{f}(\overline{x})$ $(x_m \leq \overline{x} \leq x_{\max})$; $\overline{x}_1^* = \min \overline{f}(\underline{x})$, $\overline{x}_2^* = \max \overline{f}(\underline{x})$ $(x_{\min} \leq \underline{x} \leq x_m)$.

The numerical algorithm we have used to obtain the negative root is similar to that we have presented above for the positive root. The negative fuzzy solution for the lowest α-cut $(a = [3, 5], b = [1.3], c = [1, 3])$ is presented in Fig. 3.

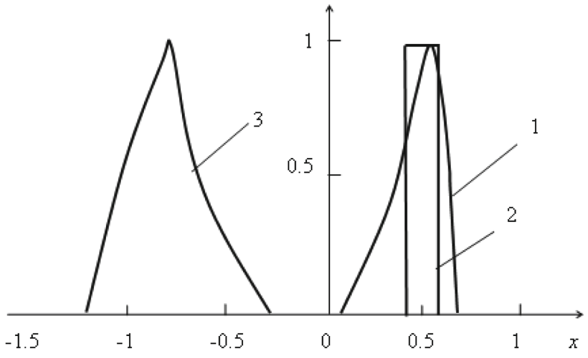


Fig. 3. The fuzzy roots of quadratic interval equation: 1, 3-fuzzy roots obtained with use of “interval extended zero” method, 2-interval solution from [1, 4]

It is seen that our positive fuzzy solution in the considered example is wider than the interval solution obtained in [1,3]. Nevertheless, it does not mean that the results from [1,3] are more “true” since the methods proposed in [1,3] do not provide negative fuzzy roots. Besides, our results may be substantially shortened using the reduction of fuzzy solution to interval one with a help of defuzzification procedure (see [15,16]).

The negative fuzzy solution presented in Fig. 3 is obtained for the lowest α-cut $(\alpha = 0)$. To get the complete fuzzy solution of (11), the fuzzy solutions for other α-cuts $(0 < \alpha \leq 1)$ should be obtained using the algorithm described above. The positive solutions obtained for $\alpha = 0$, $\alpha = 0.5$ and $\alpha = 1$ are presented in Fig. 4. For different α-cuts we have fuzzy solutions with different supports and peaks. As a fuzzy value can be represented by the disjunction of its α-cuts, we

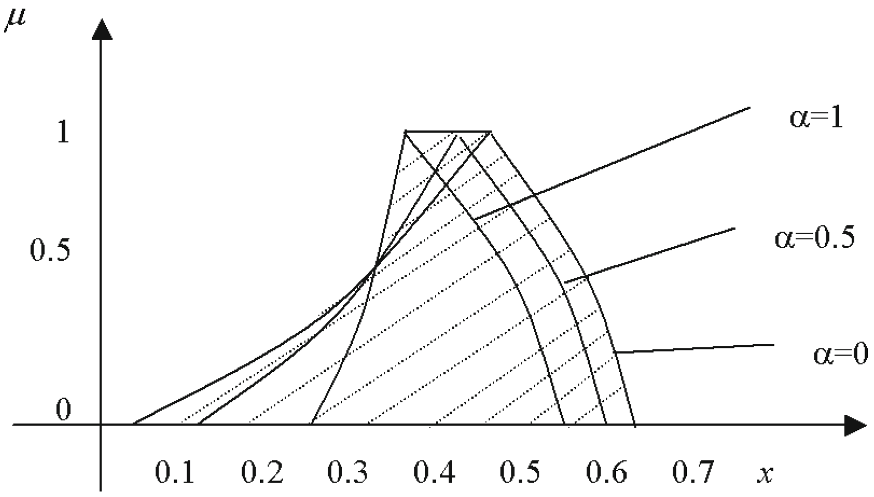


Fig. 4. The positive fuzzy root of Eq. (1)

treat the shaded area in Fig. 4 as the final fuzzy solution. It is interesting that opposite to the result of [1,3] (see Fig. 2) it has the trapezoidal form and this seems quite natural since some parameters of fuzzy equation (a and c in Eq. (1)) are trapezoidal fuzzy values too.

The numerical algorithm we have used to obtain the negative root is similar to that we have presented above for the positive root. The result is presented in Fig. 5. The resulting negative fuzzy root has the triangular form, whereas the positive root (see Fig. 4) is of trapezoidal type. This fact is a consequence of the special form of trapezoidal fuzzy parameters a and c (see Fig. 1), which have no fuzzy left parts.

It is seen that the proposed method allows us to get positive and negative approximate formal fuzzy solutions of interval quadratic and fuzzy equations, whereas the known approaches do not provide negative solutions. In the case of considered quadratic fuzzy equation, from Eq. (5) we have obtained the expressions $\underline{x} = \underline{f}(\overline{x})$, $\overline{x} = \overline{f}(\underline{x})$ simplifying the analysis, but in the general case of nonlinear fuzzy equation $F(x) = 0$, such expressions can not be always obtained. Therefore, generally the algorithm of solving nonlinear fuzzy equation (the solution can be qualified as an approximate formal (algebraical) solution) can be presented as follows:

1. Split out the nonlinear fuzzy equation $F(x) = 0$ into the set of α-cuts. For each α-cut accomplish the steps 2–7.
2. Obtain $[F(\underline{x}, \overline{x})] = [-y, y]$ and

$$f_1(\underline{x}, \overline{x}) = -y, f_2(\underline{x}, \overline{x}) = y. \tag{17}$$

These expressions are similar to (5). Finally from (17) obtain

$$g(\underline{x}, \overline{x}) = f_1(\underline{x}, \overline{x}) - f_2(\underline{x}, \overline{x}) = 0$$

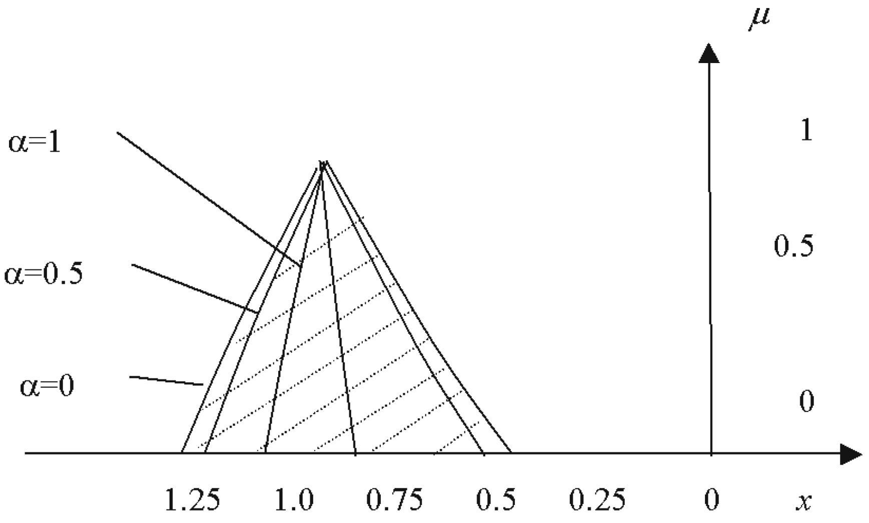


Fig. 5. The negative fuzzy root of Eq. (1).

(This equation is the analog of Eq. (6)).

3. Obtain x_m as the numerical solution of $g(\underline{x}, \overline{x}) = 0$.
4. Define $x_{\min}$ and $x_{\max}$ as the natural constraints like in the case of quadratic equation or as the external constraints originated from the mechanical or economical features of the considered problem.
5. Let $\underline{x}(\overline{x})$ be a numerical solution of $g(\underline{x}, \overline{x}) = 0$ for given $\overline{x}$ and $\overline{x}(\underline{x})$ be a numerical solution of $g(\underline{x}, \overline{x}) = 0$ for given $\underline{x}$. Then obtain
$\underline{x}_1^* = \min \underline{x}(\overline{x})$, $(\overline{x} \in [x_m, x_{\max}])$, $\underline{x}_2^* = \max \underline{x}(\overline{x})$, $(\overline{x} \in [x_m, x_{\max}])$,
$\overline{x}_1^* = \min \overline{x}(\underline{x})$, $(\underline{x} \in [x_{\min}, x_m])$, $\overline{x}_2^* = \max \overline{x}(\underline{x})$, $(\underline{x} \in [x_{\min}, x_m])$,
$[\underline{x}] = [x_{\min}, x_m] \cap [\underline{x}_1^*, \underline{x}_2^*]$, $[\overline{x}] = [x_m, x_{\max}] \cap [\overline{x}_1^*, \overline{x}_2^*]$,
$[\underline{x}] = [\underline{x}_{\min}, \underline{x}_{\max}]$, $[\overline{x}] = [\overline{x}_{\min}, \overline{x}_{\max}]$,
where $\underline{x}_{\min} = \max(x_{\min}, \underline{x}_1^*)$, $\underline{x}_{\max} = \min(x_m, \underline{x}_2^*)$, $\overline{x}_{\min} = \max(x_m, \overline{x}_1^*)$, $\overline{x}_{\max} = \min(x_{\max}, \overline{x}_2^*)$.
6. Substituting the widest possible interval solution $[\underline{x}_{\min}, \overline{x}_{\max}]$ into (17) obtain $y_{\max}$ and substituting in this equation the shortness solution $[\underline{x}_{\max}, \overline{x}_{\min}]$ obtain $y_{\min}$ (usually $\underline{x}_{\max} = \overline{x}_{\min} = x_m$).
7. Introduce the set of η-cuts as follows:

$$\eta = 1 - \frac{y - y_{\min}}{y_{\max} - y_{\min}}. \tag{18}$$

For each η-cut $(0 \leq \eta \leq 1)$ from (18) obtain y, substitute it in (17) and obtain the numerical solution of nonlinear system on the η-cut: $[\underline{x}, \overline{x}]_\eta$.

To obtain the complete solution of initial nonlinear fuzzy equation $F(x) = 0$, the steps 2–7 should be repeated for all α-cuts and solutions obtained on the α-cuts should be disjointed into the final solution.

3 Conclusion

The aim of this paper is to present an extension of the so called "interval extended zero" proposed in [15,16], to the case of nonlinear interval equations. The key idea of this method is the treatment of "interval zero" as an interval symmetrical with respect to 0. It is shown that such approach is a direct consequence of interval subtraction operation. It is shown that the method provides a fuzzy solution of nonlinear interval and fuzzy equations. It is important that opposite to the known approaches, the method makes it possible to get both the positive and negative fuzzy solutions of interval and fuzzy quadratic equation. It is shown that the proposed method may be used for the solution of more complicated fuzzy nonlinear equations and the corresponding general algorithm is presented as well.

Acknowledgement. The research has been supported by the grant financed by National Science Centre (Poland) on the basis of decision number DEC-2013/11/B/ST6/00960.

References

1. Abbasbandy, S., Asady, B.: Newton's method for solving fuzzy nonlinear equations. Appl. Math. Comput. **159**, 349–356 (2004)
2. Abbasbandy, S.: Extended Newton's method for a system of nonlinear equations by modified Adomian decomposition method. Appl. Math. Comput. **170**, 648–656 (2005)
3. Buckley, J.J., Qu, Y.: Solving linear and quadratic fuzzy equations. Fuzzy Sets Syst. **38**, 43–59 (1990)
4. Buckley, J.J., Eslami, E.: Neural net solutions to fuzzy problems: the quadratic equation. Fuzzy Sets Syst. **86**, 289–298 (1997)
5. Buckley, J.J., Eslami, E., Hayashi, Y.: Solving fuzzy equations using neural nets. Fuzzy Sets Syst. **86**, 271–278 (1997)
6. Chang, J.-C., Chen, H., Shyu, S.-M., Lian, W.-C.: Fixed-point theorems in fuzzy real line. Comput. Math. Appl. **47**, 845–851 (2004)
7. Cleary, J.C.: Logical arithmetic. Future Comput. Syst. **2**, 125–149 (1987)
8. Dubois, D., Prade, H.: Operations on fuzzy numbers. J. Syst. Sci. **9**, 613–626 (1978)
9. Dymova, L., Gonera, M., Sevastianov, P., Wyrzykowski, R.: New method for interval extension of Leontief's input-output model with use of parallel programming. In: Proceedings of the International Conference on Fuzzy Sets and Soft Computing in Economics and Finance, (FSSCEF), St. Petersburg, Russian, pp. 549–556 (2004)
10. Kaucher, E.: Interval analysis in the extended interval space IR. In: Alefeld, G., Grigorieff, R.D. (eds.) Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis). Computing Supplementum, vol. 2, pp. 33–49. Springer, Vienna (1980). https://doi.org/10.1007/978-3-7091-8577-3_3
11. Kawaguchi, M.F., Da-Te, T.: A calculation method for solving fuzzy arithmetic equations with triangular norms. In: Proceedings of 2nd IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), San Francisco, pp. 470–476 (1993)
12. Moore, R.E.: Interval Analysis. Prentice-Hall, Englewood Cliffs (1966)

13. Nieto, J.J., Rodríguez-López, R.: Existence of extremal solutions for quadratic fuzzy equations. Fixed Point Theory Appl. **3**, 321–342 (2005)
14. Nieto, J.J., Rodríguez-López, R.: Contractive mapping theorems in partially ordered sets and applications to ordinary differential equations. Order **22**, 223–239 (2005)
15. Sevastjanov, P., Dymova, L.: Fuzzy solution of interval linear equations. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 1392–1399. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68111-3_147
16. Sevastjanov, P., Dymova, L.: A new method for solving interval and fuzzy equations: linear case. Inf. Sci. **17**, 925–937 (2009)
17. Shary, S.P.: A new technique in systems analysis under interval uncertainty and ambiguity. Reliab. Comput. **8**, 321–418 (2002)

Role of Hull-Consistency in the HIBA_USNE Multithreaded Solver for Nonlinear Systems

Bartłomiej Jacek Kubica(✉)

Department of Applied Informatics, Warsaw University of Life Sciences,
ul. Nowoursynowska 159, 02-776 Warsaw, Poland
bartlomiej_kubica@sggw.pl

Abstract. This paper considers incorporating a hull-consistency enforcing procedure in an interval branch-and-prune method. Hull-consistency has been used with interval algorithms in several solvers, but its implementation in a multithreaded environment is non-trivial. We describe arising issues and discuss the ways to deal with them. Numerical results for some benchmark problems are presented and analyzed.

Keywords: Nonlinear equations systems · Interval computations
Hull consistency · Multithreading · Solver

1 Introduction

In a series of papers, including [15,16,19,20] the author considered an interval solver for nonlinear systems – targeted mostly at underdetermined equations systems – and its shared-memory parallelization (see also references in [19] for the author's other papers). The solver described in these papers is called HIBA_USNE (Heuristical Interval Branch-and-prune Algorithm for Underdetermined and well-determined Systems of Nonlinear Equations) and is currently available from the author's ResearchGate profile under the GPL license [6].

In none of these papers (and in none of previous versions of HIBA_USNE), hull-consistency has been used.

2 Generic Algorithm

HIBA_USNE uses interval methods. They are based on interval arithmetic operations and basic functions operating on intervals instead of real numbers (so that result of an operation on numbers always belongs to the result of operation on intervals that contain the numerical inputs). We shall not define interval operations here; the interested reader is referred to several papers and textbooks, e.g., [12,13].

The solver is based on the branch-and-prune (B&P) schema that can be expressed by pseudocode presented in Algorithm 1.

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 381–390, 2018.
https://doi.org/10.1007/978-3-319-78054-2_36

Algorithm 1. Interval branch-and-prune algorithm

```
Require: $L, \mathsf{f}, \varepsilon$
1: {$L$ – the list of initial boxes, often containing a single box $\mathbf{x}^{(0)}$}
2: {$L_{ver}$ – verified solution boxes, $L_{pos}$ – possible solution boxes}
3: $L_{ver} = L_{pos} = \emptyset$
4: $\mathbf{x}$ = pop ($L$)
5: loop
6:    process the box $\mathbf{x}$, using the rejection/reduction tests
7:    if ($\mathbf{x}$ does not contain solutions) then
8:       discard $\mathbf{x}$
9:    else if ($\mathbf{x}$ is verified to contain a segment of the solution manifold) then
10:      push ($L_{ver}$, $\mathbf{x}$)
11:   else if (the tests resulted in two subboxes of $\mathbf{x}$: $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$) then
12:      $\mathbf{x} = \mathbf{x}^{(1)}$
13:      push ($L$, $\mathbf{x}^{(2)}$)
14:      cycle loop
15:   else if ($\mathrm{wid}\,\mathbf{x} < \varepsilon$) then
16:      push ($L_{pos}$, $\mathbf{x}$) {The box $\mathbf{x}$ is too small for bisection}
17:   if ($\mathbf{x}$ was discarded or $\mathbf{x}$ was stored) then
18:      if ($L == \emptyset$) then
19:         return $L_{ver}, L_{pos}$ {All boxes have been considered}
20:      $\mathbf{x}$ = pop ($L$)
21:   else
22:      bisect ($\mathbf{x}$), obtaining $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$
23:      $\mathbf{x} = \mathbf{x}^{(1)}$
24:      push ($L$, $\mathbf{x}^{(2)}$)
```

The "rejection/reduction tests", mentioned in the algorithm are described in previous papers (specifically [19]), i.e.:

- switching between the componentwise Newton operator (for larger boxes) and Gauss-Seidel with inverse-midpoint preconditioner, for smaller ones,
- a heuristic to choose whether to use or not the BC3 algorithm [19],
- a heuristic to choose when to use bound-consistency [20],
- sophisticated heuristics to choose the bisected component [16,19],
- an additional second-order approximation procedure [18],
- an initial exclusion phase of the algorithm (deleting some regions, not containing solutions) – based on Sobol sequences [17,19].

Other possible variants (see, e.g., [15]) are not going to be considered.

3 Hull-Consistency

Hull-consistency (also known under the name of 2B-consistency) has been used in several interval programs over the years; see, e.g., [7,8]. It can be defined as follows.

Definition 1. *A box* $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^T$ *is hull-consistent with respect to a constraint* $c(x_1, \ldots, x_n)$*, iff:*

$$\forall i\ \mathbf{x}_i = \square\{s \in \mathbf{x}_i \mid \exists x_1 \in \mathbf{x}_1, \cdots \exists x_{i-1} \in \mathbf{x}_{i-1}, \exists x_{i+1} \in \mathbf{x}_{i+1} \cdots \exists x_n \in \mathbf{x}_n \\ c(x_1, \ldots, x_{i-1}, s, x_{i+1}, \ldots, x_n)\}\ .$$

Following [14], the symbol "□" denotes the interval hull.

Other words, $\mathbf{x}$ is hull-consistent iff for each i we can find two points x^a and x^b, satisfying the property c, for which $x_i^a = \underline{x}_i$ and $x_i^b = \overline{x}_i$.

Now, let us describe, how to check if a box is hull-consistent and how to enforce hull-consistency on a box.

3.1 Algorithms for Enforcing Hull-Consistency

For simple constraints, checking and/or enforcing hull-consistency is relatively simple.

As a simple example, let us consider an equation $x_1 + x_2 - 3 = 0$. By obvious symbolic transformations, we obtain formulae for both variables that can be used to obtain their consistent domains:

$$\mathbf{x}_1 = 3 - \mathbf{x}_2 \text{ and} \\ \mathbf{x}_2 = 3 - \mathbf{x}_1.$$

Using the above consistency operators, we can simply check consistency for any box or compute its sub-box containing all consistent values. For instance, a box $[-4, 2] \times [-2, 4]$ is not hull-consistent, but it can be reduced to the hull consistent one, by applying:

$$\mathbf{x}_1 = \mathbf{x}_1 \cap (3 - \mathbf{x}_2) = [-4, 2] \cap [-1, 5] = [-1, 2], \\ \mathbf{x}_2 = \mathbf{x}_2 \cap (3 - \mathbf{x}_1) = [-2, 4] \cap [1, 7] = [1, 4].$$

This box is hull-consistent indeed, as points $(-1, 4)$ and $(1, 2)$ are solutions of the initial constraint $x_1 + x_2 - 3 = 0$.

However, for a more sophisticated constraint, obtaining a consistent box is not as straightforward. Let us consider the constraint:

$$x_1^3 + x_1^2 - \exp(x_2) = 0. \tag{1}$$

Again, by relatively simple symbolic transformations we can extract x_2 from Eq. (1), but not x_1. The solution is to decompose such an equation into primitive ones, by adding additional variables and apply hull-consistency to such a decomposed system. For the constraint (1), we could obtain:

$$t_1 - x_1^3 = 0, \\ t_2 - x_1^2 = 0, \\ t_3 - t_1 - t_2 = 0, \\ t_4 - \exp(x_2) = 0, \\ t_3 - t_4 = 0.$$

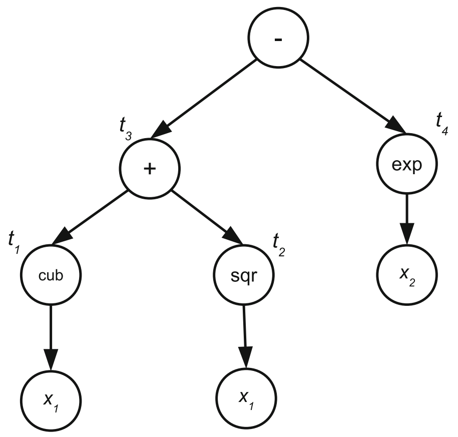


Fig. 1. Expression tree of constraint (1)

The algorithm HC4 [7] (cf. also [11]) performs such a decomposition, creating a tree of the initial constraint, where a variable corresponds to each node: By traversing the tree *forward* and *backward*, we enforce hull-consistency on subsequent variables (Fig. 1).

3.2 ADHC Implementation

The ADHC library [5] (Algorithmic Differentiation and Hull Consistency enforcing), developed by the author, contains procedures for constructing the expression tree and for the HC4 algorithm.

Thanks to the virtues of C++ template metaprogramming, the same source code can be used to generate binary procedures computing function values, gradients and Hesse matrices, and to generate the procedure creating the expression tree, in the form of a dynamic data structure.

4 Hull-Consistency Vs Multithreading

Since the very beginning (cf. [15]) the HIBA_USNE solver has been implemented as parallel. The early version has been parallelized using OpenMP, but then the author switched to Intel TBB (Threading Building Blocks [3]). Parallelization of the HIBA_USNE solver, i.e., of Algorithm 1, is done on several levels. Firstly, operations on different boxes form different tasks that can be executed by different threads.

Also, some of the procedures applied on a single box are parallel. Such a concurrent implementations has been particularly useful for the procedure enforcing bound-consistency [20], but enforcing box-consistency (see, e.g., [8]) can be parallelized, also – and such version is applied at least for the initial box.

Parallel implementation of the HC4 algorithm is also possible, but it does not seem worthwhile. The cost of enforcing hull-consistency is far smaller than box-consistency (which, in particular, requires computing derivatives – at least for BC3 and BC4 algorithms; cf. [7].

Hence, the HC4 implementation we use in the current version of the solver (Beta 2.5; cf. Sect. 5). Still, it is not easy to implement the HC4 algorithm in a MT-safe (multithreaded-safe) manner. The procedure requires the expression tree representation. There are, in general, three possibilities:

– there is a shared expression tree and access to it is synchronized,
– there is a shared expression tree, but domains of variables associated to each node are thread-specific,
– each thread has its own copy of the expression tree, to compute the domains of variables for various boxes.

The first approach seems absolutely unacceptable for a solver that is supposed to be scalable with the number of threads. The second one seems interesting, but is somewhat cumbersome to implement. Also, it might result in suboptimal cache usage as domains of each variable will have to be placed outside the node of the expression tree. The third approach is currently implemented in HIBA_USNE. It uses some memory, as each of the threads has a separate copy of the data structure (and this might become an issue for higher number of threads, e.g., on the MIC architecture, where 240 threads can work in parallel), but, in our experiments, is seems to be acceptable.

5 Computational Experiments

Numerical experiments have been performed on a machine with two Intel Xeon E5-2695 v2 processors (2.4 GHz). Each of them has 12 cores and on each core two hyper-threads (HT) can run. So, $2 \times 12 \times 2 = 48$ HT can be executed in parallel. The machine runs under control of a 64-bit GNU/Linux operating system, with the kernel 3.10.0-123.e17.x86_64 and glibc 2.17. They have non-uniform turbo frequencies from range 2.9–3.2 GHz.

As there have been other users performing their computations also, we limited ourselves to using 24 threads only.

The Intel C++ compiler ICC 15.0.2 has been used.

The solver has been written in C++, using the C++11 standard. The C-XSC library (version 2.5.4) [2] was used for interval computations. The parallelization was done with the packaged version of TBB 4.3 [3].

The following test problems have been considered: two underdetermined ones: 5R planar and Puma7, and six well-determined: Brent10, BT50 (Broyden-tridiagonal), BB30 (Broyden-banded), BB24-mod, Transistor, EF200 (Extended-Freudenstein). Their formulation (and used accuracies) has been described in [19,20] and references therein. Function BB24-mod is the Broyden-banded function BB24 minus 1; such a minor modification results in a much harder problem. It is worth noting that it was the function Broy$N$-mod that was used in previous papers ([15,19,20], etc.) under the name of the Broyden-banded function.

Here we give used accuracies:

- 5R planar: $\varepsilon = 0.02$,
- Puma7: $\varepsilon = 0.05$,
- Brent10: $\varepsilon = 10^{-7}$,
- BT50: $\varepsilon = 10^{-6}$,
- BB30, BB24-mod: $\varepsilon = 10^{-6}$,
- Transistor: $\varepsilon = 10^{-8}$,
- EF200: $\varepsilon = 10^{-6}$.

The following algorithm versions have been considered:

- "Beta 2.0" – HIBA_USNE Beta 2.0, using box and bound-consistency, but no hull-consistency,
- "HC only" – hull-consistency used instead of box-consistency and 3B consistency, instead of bound-consistency,
- "Beta 2.5" – HIBA_USNE Beta 2.5, combining box and hull-consistency, in a manner similar to BC4 [7]: algorithm HC4 is used always and BC3 is applied after it, but only if there is more than one occurrence of the variable in the formula for the constraint.

Also, please note, execution times of parallel programs are to some extent random. We try to present median results, but please note all of them may vary in a few-seconds interval.

The following notation is used in the tables:

- fun.evals, grad.evals, Hesse evals – numbers of functions evaluations, functions' gradients and Hesse matrices evaluations (in the interval automatic differentiation arithmetic),
- bisecs – the number of boxes bisections,
- preconds – the number of preconditioning matrix computations (i.e., performed Gauss-Seidel steps),
- Sobol excl. – the number of boxes to be excluded generated by the initial exclusion phase,
- Sobol resul. – the number of boxes resulting from the exclusion phase (cf. [17,19]),
- bc3 – the number of calls of bc3revise; see [19],
- hc – the number of calls of hc_enforce,
- 3B/bnd.cons. – the number of calls to the procedure enforcing a higher-order consistency, i.e., – depending on the algorithm variant – bound-consistency, 3B consistency or a mixed one (when BC4 is used),
- pos.boxes, verif.boxes – number of elements in the computed lists of boxes containing possible and verified solutions,
- Leb.pos., Leb.verif. – total Lebesgue measures of both sets,
- time – computation time in seconds.

Table 1. Computational results for the “Beta 2.0” solver version

Problem	5R planar	Puma7	Brent10	BT50	BB30	BB24-mod	Transistor	EF200
fun. evals	215,370,202	24,716,399	9,288,556	546,517,232	48,025,431	2,560,784,337	177,883,219	13,531,560
grad.evals	53,540,850	25,687,861	6,479,385	139,774,706	3,929,878	288,405,687	19,786,538	1,300,186
Hesse evals	307,648	357	264,252	30	30	91,564	127,749	200
bisections	5,445,519	1,718,076	50,299	617057	21,644	3,903,618	33,246	1,300
preconds	10,056,243	3,206,635	78,424	221427	26,019	6,225,697	67,554	2
bc3.rev	86,889,730	1,039,374	2,218,450	241,468,708	23,193,747	1,190,987,170	84,359,918	1,454,871
hc	—	—	—	—	—	—	—	—
3B/bnd.cons	110,745	222	95	0	48	8,190	10,408	0
pos.boxes	1,878,238	681,004	401	2	0	0	0	0
verif.boxes	3,458	202,694	820	1	1	1	1	1
Leb.poss	0.000333	3e-47	3e-83	1e-323	0.0	0.0	0.0	0.0
Leb.verif	1e-6	3e-11	1e-82	5e-324	3e-12	3e-233	2e-102	5r-324
time	54	14	11	209	11	662	32	39

Table 2. Computational results for the “HC only” algorithm version

Problem	5R planar	Puma7	Brent10	BT50	BB30	BB24-mod	Transistor	EF200
fun. evals	34,874,569	20,989,100	898,053	279,863,240	65,789,226	n/a	2,615,238	11,031,042
grad.evals	35,558,735	23,491,133	1,696,949	310,595,033	88,392,235	n/a	3,637,807	1,080,256
Hesse evals	682,103	820	295,851	82,063	222	n/a	789,087	200
bisections	5,777,688	1,603,978	56,435	3,694,559	1,472,865	n/a	138,951	1,801
preconds	10,486,933	2,997,809	87,508	3,570,485	2,190,128	n/a	286,216	3
bc3.rev	15	21	1,092	1,056	4,656	n/a	52	2,400
hc	282,907	1,406	4,141	8,756,612	2,577,003	n/a	298,050	4,388
3B/bnd.cons	313,406	452	1,268	1	863,681	n/a	181,261	0
pos.boxes	1,872,259	627,067	421	0	0	n/a	0	0
verif.boxes	4,098	203,577	816	2	1	n/a	1	1
Leb.poss	0.000323	1e-47	8e-84	0.0	0.0	n/a	0.0	0.0
Leb.verif	2e-6	1e-12	5e-72	1e-323	5e-324	n/a	1e-112	5e-324
time	41	13	3	192	698	$> 3,600$	16	9

Table 3. Computational results for the “Beta 2.5” solver version

Problem	5R planar	Puma7	Brent10	BT50	BB30	BB24-mod	Transistor	EF200
fun. evals	173,706,494	28,125,076	9,348,171	118,071	39,817,448	2,343,834,369	80,230,271	12,156,051
grad.evals	47,579,449	31,068,946	6,414,680	118,555	3,075,583	140,805,784	10,551,334	1,161,508
Hesse evals	333,876	637	260,631	60	270	94,705	318,036	200
bisections	5,471,725	2,124,805	50,193	1	16,574	1,265,552	53,239	1,298
preconds	10,087,309	3,947,726	78,336	6	19,441	1,771,075	100,787	2
bc3.rev	67,838,381	235,058	2,288,678	29,147	19,327,021	1,142,463,509	39,421,250	762,070
hc	94,045	1,282	713	325	31,932	2,471,674	110,407	3,314
3B/bnd.cons	117,320	393	99	1	65	6,465	48,197	0
pos.boxes	1,868,601	846,350	419	2	0	0	0	0
verif.boxes	3,415	247,477	820	0	1	1	1	1
Leb.poss	0.000334	2e-47	2e-82	1e-323	0.0	0.0	0.0	0.0
Leb.verif	1e-6	1e-12	7e-65	0.0	5e-7	4e-9	3e-118	5e-324
time	53	17	11	< 1	9	390	21	29

For comparison, let us consider some results, obtained using another solver, *Realpaver* [1] – a mature interval solver that can be considered the current state-of-the-art:

- 5R-planar – 17 min (for `Bisection precision = 2.0`, much less accurate than the presented solver) and did not cover the whole solution set ("Property: non reliable process (some solutions may be lost)").
- Brent10 – 55 sec to find all solutions (1065); parameter `-number 2000` must be set to loose no solution.
- Transistor – 30 sec to find the solution for the default setting.

6 Analysis of the Results

Replacing box- with hull-consistency resulted in a minor speedup, for 5R-planar and Puma7 problems and a major one for Brent10, Transistor and Extended-Freudenstein200 (see Tables 1 and 2. Hence for problems BT50, BB30 and BB24-mod, we obtained a significant slowdown.

Combining both consistencies (Table 3) resulted in reasonable runtimes for all problems. The time for problems BT50 and BB24-mod have been particularly good – better than for any of the previous algorithm versions. Unfortunately, the speedup for Brent10 and EF200 problems, that had been observed for the "HC only" version, has not been preserved. The author has not managed to design a better heuristic.

As for *Realpaver* – our solver performed better on all problems; in earlier versions (e.g., [20]), it had been outperformed for problems, where hull-consistency was very efficient, like the Transistor problem.

7 Conclusions

We investigated incorporating of a hull-consistency enforcing procedure to the interval nonlinear systems solver. Contrary to author's earlier fears (see [19], Sect. 3), we managed to implement this function in a MT-safe and MT-efficient (yet not parallelized itself) manner.

In general, trying to replace box- with hull-consistency is often very worthwhile, but there are significant exceptions to this rule; in our experiments hull-consistency turned out to be inefficient on various instances of the Broyden function: BT50, BB30, BB24-mod.

Enforcing hull-consistency is less computationally intensive than box-consistency, but the reduction of the box diameter is usually smaller. An exception to this rule are constraints, where a variable occurs only once; in such cases hull-consistency is definitely superior to box-consistency. This is consistent with results obtained by other researchers, e.g., [10]. Reasonable results have been obtained for the algorithm version, combining hull- and box-consistency enforcing procedures. Unfortunately, these results, while acceptable, are significantly worse than using "HC only", for some problems. As designing a better heuristic seems difficult, using machine learning might be a proper direction [9].

Acknowledgments. The author is grateful to Roman Wyrzykowski (Częstochowa University of Technology) and the team of the MICLAB project [4], for providing the great machine with Xeon and Xeon Phi processors, on which the computations have been performed.

References

1. Realpaver: Nonlinear constraint solving and rigorous global optimization (2014). http://pagesperso.lina.univ-nantes.fr/info/perso/permanents/granvil/realpaver/
2. C++ eXtended Scientific Computing library (2015). http://www.xsc.de
3. Intel TBB (2015). http://www.threadingbuildingblocks.org
4. MICLAB project (2015). http://miclab.pl
5. ADHC, C++ library (2017). https://www.researchgate.net/publication/316610415_ADHC_Algorithmic_Differentiation_and_Hull_Consistency_Alfa-05
6. HIBA_USNE, C++ library (2017). https://www.researchgate.net/publication/316687827_HIBA_USNE_Heuristical_Interval_Branch-and-prune_Algorithm_for_Underdetermined_and_well-determined_Systems_of_Nonlinear_Equations_-_Beta_25
7. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising hull and box consistency. In: International Conference on Logic Programming, pp. 230–244. The MIT Press (1999)
8. Benhamou, F., McAllester, D., Hentenryck, P.V.: CLP (intervals) revisited. In: Logic Programming, Proceedings of the 1994 International Symposium, pp. 124–138. The MIT Press (1994)
9. Goualard, F., Jermann, C.: A reinforcement learning approach to interval constraint propagation. Constraints **13**(1–2), 206–226 (2008)
10. Granvilliers, L.: On the combination of interval constraint solvers. Reliable Comput. **7**(6), 467–483 (2001)
11. Granvilliers, L., Benhamou, F.: Progress in the solving of a circuit design problem. J. Global Optim. **20**(2), 155–168 (2001)
12. Hansen, E., Walster, W.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (2004)
13. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
14. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis. Vychislennyie Tiehnologii (Comput. Technol.) **15**(1), 7–13 (2010)
15. Kubica, B.J.: Interval methods for solving underdetermined nonlinear equations systems. Reliable Comput. **15**, 207–217 (2011)
16. Kubica, B.J.: Tuning the multithreaded interval method for solving underdetermined systems of nonlinear equations. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7204, pp. 467–476. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31500-8_48
17. Kubica, B.J.: Excluding regions using Sobol sequences in an interval branch-and-prune method for nonlinear systems. Reliable Comput. **19**(4), 385–397 (2014)

18. Kubica, B.J.: Using quadratic approximations in an interval method for solving underdetermined and well-determined nonlinear systems. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013. LNCS, vol. 8385, pp. 623–633. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55195-6_59
19. Kubica, B.J.: Presentation of a highly tuned multithreaded interval solver for underdetermined and well-determined nonlinear systems. Numer. Algorithms **70**(4), 929–963 (2015)
20. Kubica, B.J.: Parallelization of a bound-consistency enforcing procedure and its application in solving nonlinear systems. J. Parallel Distrib. Comput. **107**, 57–66 (2017)

Parallel Computing of Linear Systems with Linearly Dependent Intervals in MATLAB

Ondřej Král(✉) and Milan Hladík

Department of Applied Mathematics, Faculty of Mathematics and Physics, Charles University, Malostranské nám. 25, 118 00 Prague, Czech Republic
ondrejkral@seznam.cz, hladik@kam.mff.cuni.cz

Abstract. We implemented several known algorithms for finding an interval enclosure of the solution set of a linear system with linearly dependent interval parameters. To do that we have chosen MATLAB environment with use of INTLAB and VERSOFT libraries. Because our implementation is tested on Toeplitz and symmetric matrices, among others, there is a problem with a sparsity. We introduce straightforward format for representing such matrices, which seems to be almost as effective as the standard matrix representation but with less memory demands. Moreover, we take an advantage of Parallel Computing Toolbox to enhance the performance of implemented methods and to get more insights on how the methods stands in a scope of a tightness-performance ratio. The contribution is a time-tightness performance comparison of such methods, memory efficient representation and an exploration of explicit parallelization impact.

Keywords: Interval system · Linear dependency · Parallelization MATLAB · INTLAB

1 Introduction

Intervals are a natural way to express uncertainty in a real data or to process a continuum of values [4,7]. Well-defined operations over them are forming foundation for interval analysis. We focus only on one part which is to solve systems of linear equations with some interval coefficients in a manner of finding as tight as possible interval containing whole solution set. In addition we are using an explicit information of coefficients' linear dependencies to get better results.

1.1 Notation

First, we introduce some basic notation. The set of all intervals over real numbers is denoted by $\mathbb{IR}$, and the set of $m \times n$ interval matrices by $\mathbb{IR}^{m\times n}$. For the definition of interval arithmetic see, e.g., [4,7]. For an interval $\mathbf{x} = [\underline{x}, \overline{x}] \in \mathbb{IR}$, we use $\underline{x}$ and $\overline{x}$ for its lower and upper bound, respectively. Now we can define the midpoint, the radius, the magnitude and the absolute value of an interval $\mathbf{x} \in \mathbb{IR}$ as follows

$$\begin{aligned}
\mathbf{x}^c &:= \frac{1}{2}(\overline{x} + \underline{x}), \\
\mathbf{x}^\Delta &:= \frac{1}{2}(\overline{x} - \underline{x}), \\
\mathrm{mag}(\mathbf{x}) &:= \max_{x \in \mathbf{x}}\{|x|\}, \\
|\mathbf{x}| &:= \{|x|; x \in \mathbf{x}\}.
\end{aligned}$$

Linear system with linearly dependent interval parameters is denoted as

$$A(\mathbf{p})x = b(\mathbf{p}),$$

which is an abbreviation for the following family of parametric linear systems

$$\Sigma_{k=1}^m A^k p_k x = \Sigma_{k=1}^m b^k p_k,\ p \in \mathbf{p}$$

where $\mathbf{p}$ is an interval vector that we call *vector of parameters*, A^k (b^k) is a square matrix (column vector) indicating the coefficients of parameter $\mathbf{p}_k$ in the linear system. The united solution set is defined as

$$\Sigma := \{x \in \mathbb{R}^n;\ \exists p \in \mathbf{p} : A(p)x = b(p)\}.$$

For a real matrix $A \in \mathbb{R}^{n \times n}$ the spectral radius is denoted as $\rho(A)$.

2 Methods and Implementation

The solution set Σ has a complicated structure and there is no closed-form characterization known [9]. Even for specific case with symmetric matrices, the known description is rather long [6]. That is why we seek for an outer approximation in the simple form of an interval vector. There are many methods known; see, e.g., [2,3,5,14].

2.1 Bauer-Skeel, Hansen-Bliek-Rohn Bounds and Their Refinement

Bauer-Skeel [1,16] bound is generalized by Hladík [3] as follows.
If $A(p^c)$ is regular, let us denote

$$\begin{aligned}
M &:= \Sigma_{k=1}^m p_k^\Delta |A(p^c)^{-1} A^k|, \\
x^* &:= A(p^c)^{-1} b(p^c).
\end{aligned}$$

If $\rho(M) < 1$, then

$$\begin{aligned}
&[x^* - (I - M)^{-1} \Sigma_{k=1}^m p_k^\Delta |A(p^c)^{-1}(A^k x^* - b^k)|, \\
&\ x^* - (I - M)^{-1} \Sigma_{k=1}^m p_k^\Delta |A(p^c)^{-1}(A^k x^* - b^k)|]
\end{aligned}$$

is an interval enclosure of Σ.

Hansen-Bliek-Rohn bound [10] is also generalized by Hladík [3]. If $A(p^c)$ is nonsingular, denote

$$\begin{aligned} M^* &:= (I - \Sigma_{k=1}^m p_k^\Delta |A(p^c)^{-1} A^k|)^{-1}, \\ x^* &:= A(p^c)^{-1} b(p^c), \\ x^0 &:= M^* |x^*| + \Sigma_{k=1}^m p_k^\Delta M^* |A(p^c)^{-1} b^k|. \end{aligned}$$

If $\rho(M) < 1$, then the solution set is included in the interval $\mathbf{x}$ defined component-wise as

$$\begin{aligned} \overline{x}_i &:= \max\{\tilde{x}_i, v_i \tilde{x}_i\}, \\ \underline{x}_i &:= \min\{\hat{x}_i, v_i \hat{x}_i\}, \end{aligned}$$

where

$$\begin{aligned} \tilde{x}_i &= x_i^0 + (x_i^* - |x_i^*|) m_{ii}^*, \\ \hat{x}_i &= -x_i^0 + (x_i^* + |x_i^*|) m_{ii}^*, \\ v_i &= 1/(2m_{ii}^* - 1). \end{aligned}$$

These bounds can be refined [3] by checking the sign of the following expression

$$A(p^c)^{-1}(A^k \mathbf{x} - b^k),$$

and removing absolute value in the inequality

$$(|A(p^c)^{-1}(A^k \mathbf{x} - b^k)|)_i \le (|A(p^c)^{-1} A^k| |\mathbf{x} - x^*| + |A(p^c)^{-1}(A^k x^* - b^k)|)_i,$$

where $\mathbf{x}$ is the resulting enclosure of Σ computed by some of the previous methods, or any other. This is what we refer as a refinement of the Bauer-Skeel and Hansen-Bliek-Rohn method.

2.2 Residual Form

Defining a new variable vector $y \in \mathbb{R}^n$ and setting

$$\begin{aligned} x^* &:= A(p^c)^{-1} b(p^c), \\ x &:= x^* + y, \end{aligned}$$

leads to a new form of the linear system of parametric equations

$$A(\mathbf{p}) y = b(\mathbf{p}) - A(\mathbf{p}) x^*,$$

or,

$$(\Sigma_{k=1}^m p_k A^k) y = \Sigma_{k=1}^m p_k (b^k - A^k x^*),\ p \in \mathbf{p}.$$

We solve this system directly by method proposed by Skalna [15] or by Rump's epsilon-inflation [13, Algorithm 10.7].

Skalna's method assumes some preconditioner $C \in \mathbb{R}^{n\times n}$ and $x^0 \in \mathbb{R}^n$. Denote

$$D := \text{mag}(I - \Sigma_{k=1}^{m}(CA^k)\mathbf{p}_k).$$

If $\rho(D) < 1$, we have an enclosure

$$\Sigma \subseteq x^0 + (I \ - D)^{-1}\text{mag}(\Sigma_{k=1}^{m} C(b^k - A^k x^0)\mathbf{p}_k)[-1, 1].$$

The best results are obtained for $C := A(p^c)^{-1}$, $x^0 := A(p^c)^{-1}b(p^c)$ [15] and we also use this approach.

Rump's epsilon-inflation is an algorithm that works with non-parametric system. As input we've chosen

$$\mathbf{A} := \Sigma_{k=1}^{m}\mathbf{p}_k(A(p^c)^{-1}A^k)\ ,\ \mathbf{b} := \Sigma_{k=1}^{m}\mathbf{p}_k(A(p^c)^{-1}(b^k - A^k x^*))$$

Because of sub-distributive property of intervals, we might get better results. Original algorithm can be written in the following pseudo-code for some small $d, e \in \mathbb{R}$, $d, e > 0$:

```
$C := (A^c)^{-1}$
$x_s := Cb^c$
$\mathbf{z} := C(\mathbf{b} - \mathbf{A}x_s)$
$\mathbf{R} := I - C\mathbf{A}$
**repeat**
    $\mathbf{y} := \mathbf{x}^k \cdot [1 - d, 1 + d] + [-e, e]$
    $\mathbf{x}^{k+1} := \mathbf{z} + \mathbf{R}\mathbf{y}$
**until** $\mathbf{z} + \mathbf{R}\mathbf{y} \subset \text{int}\ \mathbf{y}$ **or** $\textit{iteration} > 15$
```

2.3 Exploiting Monotonicity

Another known approach is exploiting monotonicity of the solution set with respect to some parameters [8]. The derivative of our linear system by k-th parameter yields

$$A(\mathbf{p})\frac{\partial x}{\partial p_k} = b^k - A^k x.$$

To obtain the enclosure of $\dfrac{\partial x}{\partial p_k}$, $p \in \mathbf{p}$ one must resolve the system and get the result $\mathbf{x}^*$, then use it in the equation above as $x := \mathbf{x}^*$ and find an enclosure $\mathbf{d}$ of the solutions set such that $\{\dfrac{\partial x}{\partial p_k} | p \in \mathbf{p}\} \subseteq \mathbf{d}$. Now we introduce new vectors of parameters $\underline{p}^i$ and $\overline{p}^i$ for each x_i as follows

$$\underline{p}_k^i = \begin{cases} \underline{p}_k, & \text{if } \underline{d}_i \geq 0, \\ \overline{p}_k, & \text{if } \overline{d}_i \leq 0, \end{cases} \qquad \overline{p}_k^i = \begin{cases} \overline{p}_k, & \text{if } \underline{d}_i \geq 0, \\ \underline{p}_k, & \text{if } \overline{d}_i \leq 0, \end{cases}$$

$$\text{otherwise } \underline{p}_k^i = \overline{p}_k^i = \mathbf{p}_k.$$

In the case when all signs are determined, we can obtain the hull of the solution set by solving $2n$ linear systems in the form

$$\mathbf{x}_i = [(A(\underline{p}^i)^{-1}b(\underline{p}^i))_i, (A(\overline{p}^i)^{-1}b(\overline{p}^i))_i].$$

In our implementation we reduce parameters with determined signs to point values, leave others unchanged and do not iterate the process.

2.4 Implementation

Each of the above mentioned methods has been vectorized. A parallelization of our MATLAB code is done in two ways. Many vectorized operations are executed in multiple kernel threads. The other way is to use MATLAB *workers* (instances of MATLAB) and *parfor* (parallel loop) construct in order to speed up execution of the same and independent code for different parameters (SPMD). Here we take an advantage of the so called *reduction variables*, which allow us to share a memory between workers, and *sliced variables*, which reduce amount of the data transferred to workers at the initialization (dependency representation of a parameter is transferred only to the worker where it's processed). The representation of dependencies is done in a format of triples (row, column, value) each time represented parameter occurs in the matrix. Then these triples are stored to a *cell array.* This is what we call *cell representation.* By *matrix representation*, we mean representing a matrix of coefficients (A^k) by 2D array. For symmetric and Toeplitz matrices up to dimension 100×100 it was tested that there is no significant performance overhead when we expand cell representation to a matrix representation during the computation (the delay was lower then 1%). But there is significant memory saving and therefore faster initial broadcasting of the data. The following table shows differences in Toeplitz matrix representation memory consumption:

Dimension	Cell	Array
50	69.69 kb	1.91 Mb
100	256.41 kb	15.26 Mb

Therefore we decided to use cell representation in the following testing.

3 Results

Let us compare the tightness of the computed enclosures. We calculate average width over all discussed dimensions and use it as a measure of tightness. Then we compute the average tightness over 10 random Toeplitz and symmetric systems where all intervals in vector of parameters have same fixed radius and their centers are uniformly generated from range $[-10, 10]$. If we relativize this result

Toeplitz system dimension	100	Relative tightness		
Radius of interval parameters	0.05	0.1	0.5	1
Generalized Bauer-Skeel	1.00000	1.00000	1.00000	1.00000
Refinement of BS	0.99769	0.99537	0.97778	0.95896
Generalized Hansen-Bliek-Rohn	1.44822	1.44721	1.45232	1.14435
Refinement of HBR	1.00557	1.00766	1.04910	1.08704
Residual form, Rump's alg.	1.00077	1.00132	1.00131	1.01332
Residual form, Skalna's method	1.00000	1.00000	1.00000	1.00000
Monotonicity approach	0.99251	0.98515	0.92802	0.86494

Symmetric system dimension	100	Relative tightness		
Radius of interval parameters	0.05	0.1	0.5	1
Generalized Bauer-Skeel	1.00000	1.00000	1.00000	1.00000
Refinement of BS	0.99912	0.99814	0.99103	0.98398
Generalized Hansen-Bliek-Rohn	1.09541	1.09559	1.09645	1.09423
Refinement of HBR	1.00059	1.00107	1.00629	1.01412
Residual form, Rump's alg.	1.00073	1.00128	1.00129	1.01341
Residual form, Skalna's method	1.00000	1.00000	1.00000	1.00000
Monotonicity approach	0.99254	0.98517	0.92932	0.87542

to one particular method, for example Bauer-Skeel method, we obtain following tables for Teoplitz and symmetric systems (in this order):

The generalized Bauer-Skeel and the residual form with Skalna's method are mathematically equivalent. When we increase width of the intervals in vector of parameters, monotonicity approach and generalized Bauer-Skeel are performing better than others.

3.1 Execution Time

Now we are interested more in how much faster we can compute these enclosures and whether there are methods that are less precise but faster than precise ones. We tested these implementations on AMD Opteron(tm) Processor 6134 with 16 cores, 64 GB RAM, Linux, MATLAB R2016b with the packages INTLAB V9 [12] and VERSOFT V10 [11]. Tested dimensions are $5, 10, 25, 50, 100$. Based on their speed and precision, methods can be naturally divided into three groups as one can see in the table below. These values are execution times in seconds with single-worker computation:

First group includes fast methods: the generalized Bauer-Skeel, Hansen-Bliek-Rohn and the residual forms. Then there are refinement methods and finally most precise but slowest monotonicity approach.

Dimension: 100	Toeplitz	Symmetric
Generalized Bauer-Skeel	0.658	16.931
Refinement of BS	18.10	487.18
Generalized Hansen-Bliek-Rohn	0.533	12.894
Refinement of HBR	6.454	333.62
Residual form, Rump's alg.	0.445	11.385
Residual form, Skalna's method	0.443	11.399
Monotonicity approach	113.9	2779.3

We also noticed a strange behaviour. We started with one worker with one computational thread and then we increase number of threads by 1 up to 6 by use of deprecated *maxNumCompThreads*() function. Suprisingly, the performance suffered with each additional thread. Cause of this problem is unknown, might be tied to a processor cache.

Workers. We are adding more workers from 1 up to 6 and measure performance speed-up. In contrast to adding new threads, there is visible and significant performance improvement. For lower dimensions up to 10, more workers are not so beneficial, 6 workers even slowing the computations, possibly because there is an initial data broadcast overhead. Major improvements start around dimension 50. For dimension 100 and 6 workers, methods from the first group run 2-times faster and methods from the second group even 4-times faster. The monotonicity approach speed-up is the biggest, about 6-times faster. Typical curve of such an improvement looks like Fig. 1, where x-axis is number of workers and y-axis is computation time. This particular curve is for the Bauer-Skeel refinement (on the left) and the monotonicity approach (on the right) over a Toeplitz system of dimension 100.

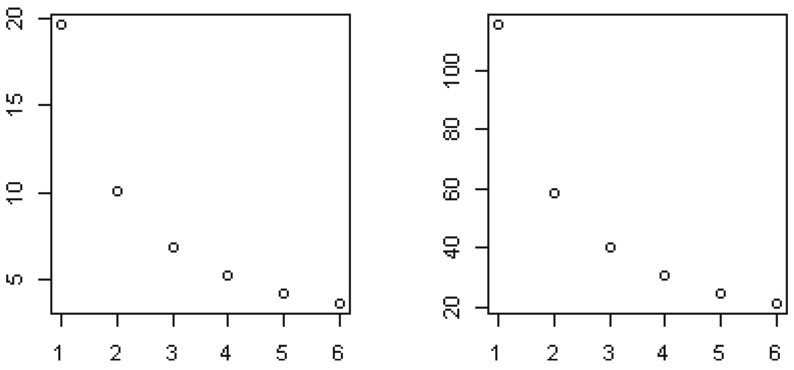


Fig. 1. Execution time (y-axis) with respect to number of workers (x-axis). Bauer-Skeel refinement (on the left) and the monotonicity approach (on the right).

Thus the best improvement is for the methods that do some nontrivial operations with each parameter in one iteration of a for-cycle. On the other hand, methods that just do some fast matrix computation for each parameter profit less, beacause of implicit parallelization.

For s symmetric system, there is an increase of parameters, in dimension 100 up to 5000. In this case, speed-up ratio of each method is not so different, roughly 5-times faster for 6 workers which is closer to the theoretical 6-times speed-up.

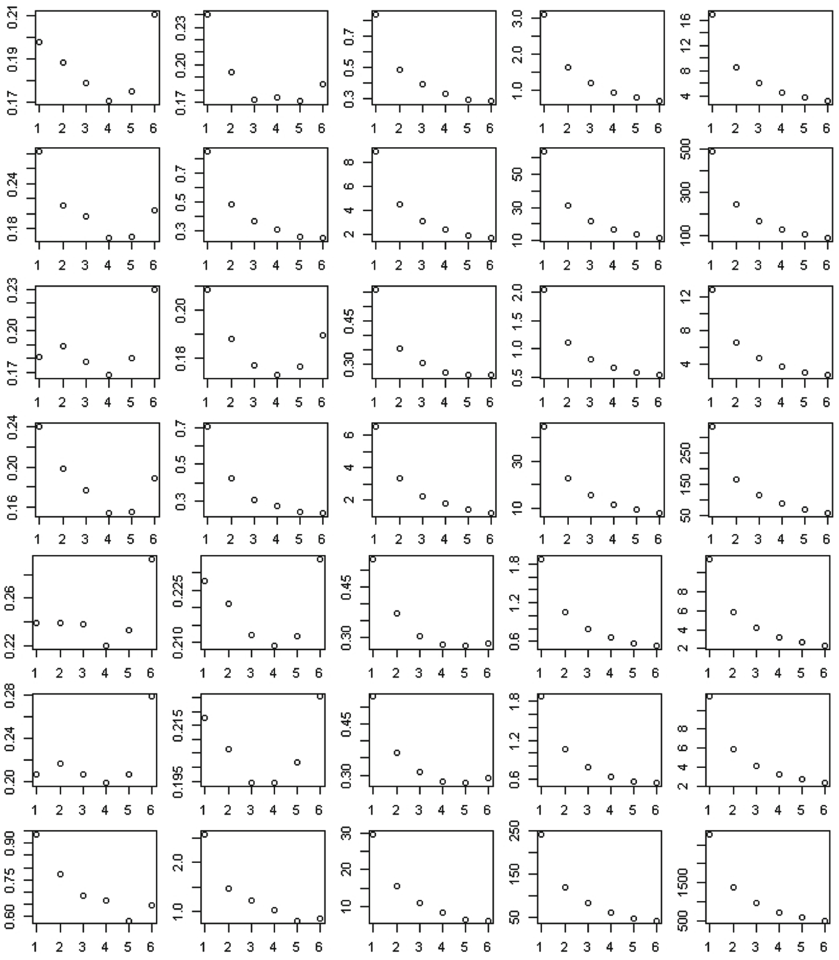
3.2 Conclusion

Comparison of relative tightness and of single-worker execution times help us develop a basic of understanding, which methods are better (For example Skalna's method on residual form is faster and even yields better results than refinement of generalized Hansen-Bliek-Rohn method.) and how to classify methods into some reasonable tightness-performance groups. Simple memory saving representation might be beneficial with increasing number of workers and dimension, where initial broadcast of variables slowing down the computation. With large number of parameters in symmetric systems (5000), tightness-performance groups hold. When this number is getting lower (200), as for Toeplitz systems, the monotonicity approach and refinement methods are speeding-up more than others, so they can be an interesting alternative for fast methods. Explicit parallelization with workers might have a negative effect for lower dimensions (10).

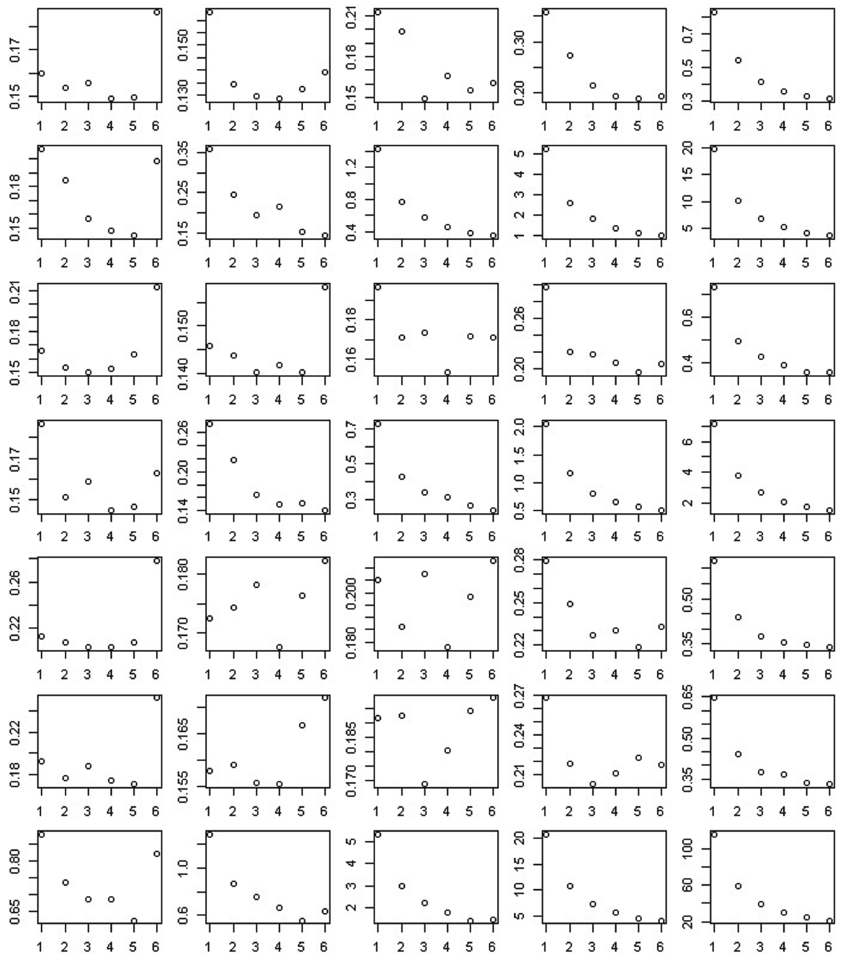
3.3 Table of Results

The columns correspond to dimensions $5, 10, 25, 50, 100$ (in this order). The rows correspond to the methods in the order used in the previous tables. It's visualization of execution time in seconds (y-axis) depending on number of workers (x-axis).

Symmetric



Toeplitz



Acknowledgments. The authors were supported by the Czech Science Foundation Grant P402-13-10660S.

References

1. Bauer, F.L.: Genauigkeitsfragen bei der Lösung linearer Gleichungssysteme. ZAMM **46**(7), 409–421 (1966)
2. Hladík, M.: Optimal preconditioning for the interval parametric Gauss–Seidel method. In: Nehmeier, M., Wolff von Gudenberg, J., Tucker, W. (eds.) SCAN 2015. LNCS, vol. 9553, pp. 116–125. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31769-4_10
3. Hladík, M.: Enclosures for the solution set of parametric interval linear systems. Int. J. Appl. Math. Comput. Sci. **22**(3), 561–574 (2012)

4. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001). https://doi.org/10.1007/978-1-4471-0249-6
5. Kolev, L.V.: Improvement of a direct method for outer solution of linear parametric systems. Reliable Comput. **12**(3), 193–202 (2006)
6. Mayer, G.: Three short descriptions of the symmetric and of the skew-symmetric solution set. Linear Algebra Appl. **475**, 73–79 (2015)
7. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
8. Popova, E.D.: Computer-assisted proofs in solving linear parametric problems. In: 12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics, SCAN 2006, Conference Post-Proceedings, Duisburg, Germany, 26–29 September, p. 35. IEEE Computer Society Press (2006)
9. Popova, E.D.: Solvability of parametric interval linear systems of equations and inequalities. SIAM J. Matrix Anal. Appl. **36**(2), 615–633 (2015)
10. Rohn, J.: Cheap and tight bounds: the recent result by E. Hansen can be made more efficient. Interval Comput. **4**(13–21), 2 (1993)
11. Rohn, J.: VERSOFT: Verification software in MATLAB/INTLAB, version 10 (2009). http://www.nsc.ru/interval/Programing/versoft/
12. Rump, S.M.: INTLAB - INTerval LABoratory. In: Csendes, T. (ed.) Developments in Reliable Computing, pp. 77–104. Kluwer Academic Publishers, Dordrecht (1999). http://www.ti3.tu-harburg.de/rump/
13. Rump, S.M.: Verification methods: rigorous results using floating-point arithmetic. Acta Numer. **19**, 287–449 (2010)
14. Skalna, I.: A method for outer interval solution of systems of linear equations depending linearly on interval parameters. Reliable Comput. **12**(2), 107–120 (2006)
15. Skalna, I.: Enclosure for the solution set of parametric linear systems with non-affine dependencies. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011. LNCS, vol. 7204, pp. 513–522. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31500-8_53
16. Skeel, R.D.: Scaling for numerical stability in Gaussian elimination. J. ACM **26**(3), 494–526 (1979)

What Decision to Make in a Conflict Situation Under Interval Uncertainty: Efficient Algorithms for the Hurwicz Approach

Bartłomiej Jacek Kubica[1], Andrzej Pownuk[2](✉), and Vladik Kreinovich[2]

[1] Department of Applied Informatics, Warsaw University of Life Sciences, ul. Nowoursynowska 159, 02-776 Warsaw, Poland
bartlomiej.jacek.kubica@gmail.com

[2] Computational Science Program, University of Texas at El Paso, El Paso, TX 79968, USA
{ampownuk,vladik}@utep.edu

Abstract. In this paper, we show how to take interval uncertainty into account when solving conflict situations. Algorithms for conflict situations under interval uncertainty are known under the assumption that each side of the conflict maximizes its worst-case expected gain. However, it is known that a more general Hurwicz approach provides a more adequate description of decision making under uncertainty. In this approach, each side maximizes the convex combination of the worst-case and the best-case expected gains. In this paper, we describe how to resolve conflict situations under the general Hurwicz approach to interval uncertainty.

Keywords: Interval uncertainty · Conflict situation
Hurwicz approach

1 Conflict Situations Under Interval Uncertainty: Formulation of the Problem and What Is Known so Far

How conflict situations are usually described. In many practical situations – e.g., in security – we have conflict situations in which the interests of the two sides are opposite. For example, a terrorist group wants to attack one of our assets, while we want to defend them.

To fully describe such a situation, we need to describe:

- for each possible strategy i of one side and
- for each possible strategy j of the other side,

R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10778, pp. 402–411, 2018.
https://doi.org/10.1007/978-3-319-78054-2_38

what will be the resulting gain u_{ij} to the first side (negative if it is a loss), and the gain v_{ij} to the other side. A conflict situation is when we cannot improve v without worsening u.

An example of a conflict situation in a *zero-sum game*, when the gain of one side is the loss of another side, i.e., when $v_{ij} = -u_{ij}$; see, e.g., [9].

While zero-sum games are a useful approximation, they are not always a perfect description of the situation. For example, the main objective of the terrorists may be publicity. In this sense, a small attack in the country's capital may not cause much damage but it will bring them a lot of media attention, while a more serious attack in a remote location may be more damaging to the country, but not as media-attractive. To take this difference into account, we need, for each pair of strategies (i, j), to describe both:

– the gain u_{ij} of the first side and
– the gain v_{ij} of the second side.

In this general case, we do not necessarily have $v_{ij} = -u_{ij}$ [9].

How to describe this problem in precise terms. It is a well-known fact that in conflict situations, instead of following one of the deterministic strategies, it is beneficial to select a strategy at random, with some probability. For example, if we only have one security person available and two objects to protect, then we have two deterministic strategies:

– post this person at the first objects and
– post him/her at the second object.

If we exactly follow one of these strategies, then the adversary will be able to easily attack the other – unprotected – object. It is thus more beneficial to every time flip a coin and assign the security person to one of the objects at random. This way, for each object of attack, there will be a 50% probability that this object will be defended.

In general, each corresponding strategy of the first side can be described by the probabilities $p_1, \ldots, p_n$ of selecting each of the possible strategies, so that

$$\sum_{i=1}^{n} p_i = 1. \tag{1.1}$$

Similarly, the generic strategy of the second side can be described by the probabilities $q_1, \ldots, q_m$ for which

$$\sum_{j=1}^{m} q_j = 1. \tag{1.2}$$

If the first side selects the strategy $p = (p_1, \ldots, p_n)$ and the second side selects the strategy $q = (q_1, \ldots, q_m)$, then the expected gain of the first side is equal to

$$g_1(p, q) = \sum_{i=1}^{n} \sum_{j=1}^{m} p_i \cdot q_j \cdot u_{ij}, \tag{1.3}$$

while the expected gain of the second side is equal to

$$g_2(p,q) = \sum_{i=1}^{n} \sum_{j=1}^{m} p_i \cdot q_j \cdot v_{ij}. \tag{1.4}$$

Based on this, how can we select a strategy? It is reasonable to assume that once a strategy is selected, the other side knows the corresponding probabilities – simply by observing the past history. So, if the first side selects the strategy p, the second side should select a strategy for which, under this strategy of the first side, their gain is the largest possible, i.e., the strategy $q(p)$ for which

$$g_2(p,q(p)) = \max_q g_2(p,q). \tag{1.5}$$

In other words,

$$q(p) = \arg\max_q g_2(p,q). \tag{1.6}$$

Under this strategy of the second side, the first side gains the value $g_1(p,q(p))$. A natural idea is to select the strategy p for which this gain is the largest possible, i.e., for which

$$g_1(p,q(p)) \to \max_p, \text{ where } q(p) \overset{\text{def}}{=} \arg\max_q g_2(p,q). \tag{1.7}$$

Similarly, the second side select a strategy q for which

$$g_2(p(q),q) \to \max_q, \text{ where } p(q) \overset{\text{def}}{=} \arg\max_p g_1(p,q). \tag{1.8}$$

Towards an algorithm for solving this problem. Once the strategy p of the first side is selected, the second side selects q for which its expected gain $g_2(p,q)$ is the largest possible.

The expression $g_2(p,q)$ is linear in terms of q_j. Thus, for every q, the resulting expected gain is the convex combination

$$g_2(p,q) = \sum_{j=1}^{m} q_j \cdot q_{2j}(p) \tag{1.9}$$

of the gains

$$g_{2j}(p) \overset{\text{def}}{=} \sum_{i=1}^{n} p_i \cdot v_{ij} \tag{1.10}$$

corresponding to different deterministic strategies of the second side. Thus, the largest possible gain is attained when q is a deterministic strategy.

The j-th deterministic strategy will be selected by the second side if its gain at this strategy are larger than (or equal to) gains corresponding to all other deterministic strategies, i.e., under the constraint that

$$\sum_{i=1}^{n} p_i \cdot v_{ij} \geq \sum_{i=1}^{n} p_i \cdot v_{ik} \tag{1.11}$$

for all $k \neq j$.

For strategies p for which the second side selects the j-th response, the gain of the first side is

$$\sum_{i=1}^{n} p_i \cdot u_{ij}. \tag{1.12}$$

Among all strategies p with this "j-property", we select the one for which the expected gain of the first side is the largest possible. This can be found by optimizing a linear function under constraints which are linear inequalities – i.e., by solving a *linear programming* problem. It is known that for linear programming problems, there are efficient algorithms; see, e.g., [6].

In general, we thus have m options corresponding to m different values $j = 1, \ldots, m$. Among all these m possibility, the first side should select a strategy for which the expected gain is the largest possible. Thus, we arrive at the following algorithm.

An algorithm for solving the problem. For each j from 1 to m, we solve the following linear programming problem:

$$\sum_{i=1}^{n} p_i^{(j)} \cdot u_{ij} \to \max_{p_i^{(j)}} \tag{1.13}$$

under the constraints

$$\sum_{i=1}^{n} p_i^{(j)} = 1, \;\; p_i^{(j)} \ge 0, \;\; \sum_{i=1}^{n} p_i^{(j)} \cdot v_{ij} \ge \sum_{i=1}^{n} p_i^{(j)} \cdot v_{ik} \text{ for all } k \ne j. \tag{1.14}$$

Out of the resulting m solutions $p^{(j)} = \left(p_1^{(j)}, \ldots, p_n^{(j)}\right)$, $1 \le j \le m$, we select the one for which the corresponding value $\sum\limits_{i=1}^{n} p_i^{(j)} \cdot u_{ij}$ is the largest.

Comment. Solution is simpler in zero-sum situations, since in this case, we only need to solve one linear programming problem; see, e.g., [9].

Need for parallelization. For simple conflict situations, when each side has a small number of strategies, the corresponding problem is easy to solve.

However, in many practical situations, especially in security-related situations, we have a large number of possible deterministic strategies of each side. This happens, e.g., if we assign air marshals to different international flights. In this case, the only way to solve the corresponding problem is to perform at least some computations in parallel.

Good news is that the above problem allows for a natural parallelization: namely, all m linear programming problems can be, in principle, solved on different processors. Not so good news is that, once we get to the linear programming problems, while we can improve them somewhat using parallelization, these problems are P-hard, i.e., provably the hardest to parallelize efficiently; see, e.g., [8].

Need to take uncertainty into account. The above description assumed that we know the exact consequence of each combination of strategies. This is rarely the case. In practice, we rarely know the exact gains u_{ij} and v_{ij}. At best, we know the *bounds* on these gains, i.e., we know:

- the interval $[\underline{u}_{ij}, \overline{u}_{ij}]$ that contains the actual (unknown) values u_{ij}, and
- the interval $[\underline{v}_{ij}, \overline{v}_{ij}]$ that contains the actual (unknown) values v_{ij}.

It is therefore necessary to decide what to do in such situations of interval uncertainty.

How interval uncertainty is taken into account now. In the above description of a conflict situation, we mentioned that when we select the strategy p, we maximize the worst-case situation, i.e., the smallest possible gain $g_1(p, q)$ under all possible actions of the second side. It seems reasonable to apply the same idea to the case of interval uncertainty, i.e., to maximize the smallest possible gain $g_1(p, q)$ over all possible strategies of the second side *and* over all possible values $u_{ij} \in [\underline{u}_{ij}, \overline{u}_{ij}]$.

For some practically important situations, efficient algorithms for such worst-case formulation have indeed been proposed; see, e.g., [3].

Need for a more adequate formulation of the problem. In the case of adversity, it makes sense to consider the worst-case scenario: after all the adversary wants to minimize the gain of the other side.

However, in case of interval uncertainty, using the worst-case scenario may not be the most adequate idea. The problem of decision making under uncertainty, when for each alternative a, instead of the exact value $u(a)$, we only know the interval $[\underline{u}(a), \overline{u}(a)]$ of possible values of the gain, has been thoroughly analyzed.

It is known that in such situations, the most adequate decision strategy is to select an alternative a for which the following expression attains the largest possible value:

$$u^H(a) \stackrel{\text{def}}{=} \alpha \cdot \overline{u}(a) + (1 - \alpha) \cdot \underline{u}(a), \tag{1.15}$$

where $\alpha \in [0, 1]$ describes the decision maker's attitude; see, e.g., [1,4,5]. This expression was first proposed by the Nobelist Leonid Hurwicz and is thus, known as the Hurwicz approach to decision making under interval uncertainty.

In the particular case of $\alpha = 0$, this approach leads to optimizing the worst-case value $\underline{u}(a)$, but for other values α, we have different optimization problems.

What we do in this paper. In this paper, we analyze how to solve conflict situations under this more adequate Hurwicz approach to decision making under uncertainty.

In this analysis, we will assume that each side knows the other's parameter α, i.e., that both sides know the values α_u and α_v that characterize their decision making under uncertainty. This can be safely assumed since we can determine these values by analyzing past decisions of each side.

2 Conflict Situation Under Hurwicz-Type Interval Uncertainty: Analysis of the Problem

Once the first side selects a strategy, what should the second side do? If the first side selects the strategy p, then, for each strategy q of the second side, the actual (unknown) gain of the second side is equal to $\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{m} p_i \cdot q_j \cdot v_{ij}$. We do not know the exact values v_{ij}, we only know the bounds $\underline{v}_{ij} \le v_{ij} \le \overline{v}_{ij}$. Thus, once:

- the first side selects the strategy p and
- the second side selects the strategy q,

the gain of the second side can take any value from

$$\underline{g}_2(p, q) = \sum_{i=1}^{n}\sum_{j=1}^{m} p_i \cdot q_j \cdot \underline{v}_{ij} \tag{2.1}$$

to

$$\overline{g}_2(p, q) = \sum_{i=1}^{n}\sum_{j=1}^{m} p_i \cdot q_j \cdot \overline{v}_{ij}. \tag{2.2}$$

According to Hurwicz's approach, the second side should select a strategy q for which the Hurwicz combination

$$g_2^H(p, q) \stackrel{\text{def}}{=} \alpha_v \cdot \overline{g}_2(p, q) + (1 - \alpha_v) \cdot \underline{g}_2(p, q) \tag{2.3}$$

attains the largest possible value.

Substituting the expressions (2.1) and (2.2) into the formula (2.3), we conclude that

$$g_2^H(p, q) = \sum_{i=1}^{n}\sum_{j=1}^{m} p_i \cdot q_j \cdot v_{ij}^H, \tag{2.4}$$

where we denoted

$$v_{ij}^H \stackrel{\text{def}}{=} \alpha_v \cdot \overline{v}_{ij} + (1 - \alpha_v) \cdot \underline{v}_{ij}. \tag{2.5}$$

Thus, once the first side selects its strategy p, the second side should select a strategy $q(p)$ for which the corresponding Hurwicz combination $g_2^H(p, q)$ is the largest possible, i.e., the strategy $q(p)$ for which

$$g_2^H(p, q(p)) = \max_q g_2^H(p, q). \tag{2.6}$$

In other words,

$$q(p) = \arg\max_q g_2^H(p, q). \tag{2.7}$$

Based on this, what strategy should the first side select? Under the above strategy $q = q(p)$ of the second side, the first side gains the value

$$g_1(p, q(p)) = \sum_{i=1}^{n}\sum_{j=1}^{m} p_i \cdot q_j \cdot u_{ij}. \tag{2.8}$$

Since we do not know the exact values u_{ij}, we only know the bounds $\underline{u}_{ij} \le u_{ij} \le \overline{u}_{ij}$, we therefore do not know the exact gain of the first side. All we know is that this gain will be between

$$\underline{g}_1(p, q(p)) = \sum_{i=1}^{n} \sum_{j=1}^{m} p_i \cdot q_j \cdot \underline{u}_{ij} \tag{2.9}$$

and

$$\overline{g}_1(p, q(p)) = \sum_{i=1}^{n} \sum_{j=1}^{m} p_i \cdot q_j \cdot \overline{u}_{ij}. \tag{2.10}$$

According to Hurwicz's approach, the first side should select a strategy p for which the Hurwicz combination

$$g_1^H(p, q) \overset{\text{def}}{=} \alpha_u \cdot \overline{g}_1(p, q(p)) + (1 - \alpha_u) \cdot \underline{g}_1(p, q(p)) \tag{2.11}$$

attains the largest possible value.

Substituting the expressions (2.9) and (2.10) into the formula (2.11), we conclude that

$$g_1^H(p, q) = \sum_{i=1}^{n} \sum_{j=1}^{m} p_i \cdot q_j \cdot u_{ij}^H, \tag{2.12}$$

where we denoted

$$u_{ij}^H \overset{\text{def}}{=} \alpha_u \cdot \overline{u}_{ij} + (1 - \alpha_u) \cdot \underline{u}_{ij}. \tag{2.13}$$

What strategy should the second side select? Thus, the first side will select the strategy p for which this Hurwicz combination is the largest possible, i.e., for which

$$g_1^H(p, q(p)) \to \max_p, \text{ where } q(p) \overset{\text{def}}{=} \arg\max_q g_2^H(p, q). \tag{2.14}$$

Similarly, the second side select a strategy q for which

$$g_2^H(p(q), q) \to \max_q, \text{ where } p(q) \overset{\text{def}}{=} \arg\max_p g_1^H(p, q). \tag{2.15}$$

We thus reduce the interval-uncertainty problem to the no-uncertainty case. One can easily see that the resulting optimization problem is exactly the same as in the no-uncertainty case described in Sect. 1, with the gains u_{ij}^H and v_{ij}^H described by the formulas (2.13) and (2.5).

Thus, we can apply the algorithm described in Sect. 1 to solve the interval-uncertainty problem.

3 Algorithm for Solving Conflict Situation Under Hurwicz-Type Interval Uncertainty

What is given. For every deterministic strategy i of the first side and for every deterministic strategy j of the second side, we are given:

– the interval $[\underline{u}_{ij}, \overline{u}_{ij}]$ of the possible values of the gain of the first side, and
– the interval $[\underline{v}_{ij}, \overline{v}_{ij}]$ of the possible values of the gain of the second side.

We also know the parameters α_u and α_v characterizing decision making of each side under uncertainty.

Preliminary step: forming appropriate combinations of gain bounds. First, we compute the values

$$u^H_{ij} \stackrel{\text{def}}{=} \alpha_u \cdot \overline{u}_{ij} + (1 - \alpha_u) \cdot \underline{u}_{ij} \tag{3.1}$$

and

$$v^H_{ij} \stackrel{\text{def}}{=} \alpha_v \cdot \overline{v}_{ij} + (1 - \alpha_v) \cdot \underline{v}_{ij}. \tag{3.2}$$

Main step. For each j from 1 to m, we solve the following linear programming problem:

$$\sum_{i=1}^{n} p^{(j)}_i \cdot u^H_{ij} \to \max_{p^{(j)}_i} \tag{3.3}$$

under the constraints

$$\sum_{i=1}^{n} p^{(j)}_i = 1, \quad p^{(j)}_i \geq 0, \quad \sum_{i=1}^{n} p^{(j)}_i \cdot v^H_{ij} \geq \sum_{i=1}^{n} p^{(j)}_i \cdot v^H_{ik} \text{ for all } k \neq j. \tag{3.4}$$

Final step. Out of the resulting m solutions $p^{(j)} = \left(p^{(j)}_1, \ldots, p^{(j)}_n\right)$, $1 \leq j \leq m$, we select the one for which the corresponding value

$$\sum_{i=1}^{n} p^{(j)}_i \cdot u^H_{ij} \tag{3.5}$$

is the largest.

Comment. In view of the fact that in the no-uncertainty case, zero-sum games are easier to process, let us consider zero-sum games under interval uncertainty. To be more precise, let us consider situations in which possible values v_{ij} are exactly values $-u_{ij}$ for possible u_{ij}:

$$[\underline{v}_{ij}, \overline{v}_{ij}] = \{-u_{ij} : \underline{u}_{ij} \in [\underline{u}_{ij}, \overline{u}_{ij}]\}. \tag{3.6}$$

One can easily see (see, e.g., [2,7]) that this condition is equivalent to

$$\underline{v}_{ij} = -\overline{u}_{ij} \text{ and } \overline{v}_{ij} = -\underline{u}_{ij}. \tag{3.7}$$

In this case, we have

$$v_{ij}^H = \alpha_v \cdot \overline{v}_{ij} + (1 - \alpha_v) \cdot \underline{v}_{ij} = \alpha_v \cdot (-\underline{u}_{ij}) + (1 - \alpha_v) \cdot (-\overline{u}_{ij}), \tag{3.8}$$

and thus,

$$v_{ij}^H = -((1 - \alpha_v) \cdot \overline{u}_{ij} + \alpha_v \cdot \underline{u}_{ij}). \tag{3.9}$$

By comparing this expression with the formula (3.1) for u_{ij}^H, we can conclude that the resulting game is zero-sum (i.e., $v_{ij}^H = -u_{ij}^H$) only when $\alpha_u = 1 - \alpha_v$.

In all other cases, even if we start with a zero-sum interval-uncertainty game, the no-uncertainty game to which we reduce that game will *not* be zero-sum – and thus, the general algorithm will be needed, without a simplification that is available for zero-sum games.

4 Conclusion

In this paper, we show how to take interval uncertainty into account when solving conflict situations.

Algorithms for conflict situations under interval uncertainty are known under the assumption that each side of the conflict maximizes its worst-case expected gain. However, it is known that a more general Hurwicz approach provides a more adequate description of decision making under uncertainty. In this approach, each side maximizes the convex combination of the worst-case and the best-case expected gains.

In this paper, we describe how to resolve conflict situations under the general Hurwicz approach to interval uncertainty.

Acknowledgments. This work was supported in part by the National Science Foundation grants HRD-0734825 and HRD-1242122 (Cyber-ShARE Center of Excellence) and DUE-0926721, and by an award "UTEP and Prudential Actuarial Science Academy and Pipeline Initiative" from Prudential Foundation.

The authors are thankful to the anonymous referees for valuable suggestions.

References

1. Hurwicz, L.: Optimality Criteria for Decision Making Under Ignorance. Cowles Commission Discussion Paper, Statistics, No. 370 (1951)
2. Jaulin, L., Kiefer, M., Dicrit, O., Walter, E.: Applied Interval Analysis. Springer, London (2001). https://doi.org/10.1007/978-1-4471-0249-6
3. Kiekintveld, C., Islam, M.T., Kreinovich, V.: Security fame with interval uncertainty. In: Ito, T., Jonker, C., Gini, M., Shehory, O. (eds.) Proceedings of the Twelfth International Conference on Autonomous Agents and Multiagent Systems AAMAS 2013, Saint Paul, Minnesota, 6–10 May 2013, pp. 231–238 (2013)
4. Kreinovich, V.: Decision making under interval uncertainty (and beyond). In: Guo, P., Pedrycz, W. (eds.) Human-Centric Decision-Making Models for Social Sciences. SCI, vol. 502, pp. 163–193. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-39307-5_8

5. Luce, R.D., Raiffa, R.: Games and Decisions: Introduction and Critical Survey. Dover, New York (1989)
6. Luenberger, D.G., Ye, Y.: Linear and Nonlinear Programming. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-18842-3
7. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
8. Sipser, M.: Introduction to Theory of Computation. Thomson Course Technology, Boston (2012)
9. Tadelis, S.: Game Theory: An Introduction. Princeton University Press, Princeton (2013)

Practical Need for Algebraic (Equality-Type) Solutions of Interval Equations and for Extended-Zero Solutions

Ludmila Dymova[1], Pavel Sevastjanov[1], Andrzej Pownuk[2](✉), and Vladik Kreinovich[2]

[1] Institute of Computer and Information Science, Czestochowa University of Technology, Dabrowskiego 73, 42-200 Czestochowa, Poland
sevast@icis.pcz.pl

[2] Computational Science Program, University of Texas at El Paso, El Paso, TX 79968, USA
{ampownuk,vladik}@utep.edu

Abstract. One of the main problems in interval computations is solving systems of equations under interval uncertainty. Usually, interval computation packages consider united, tolerance, and control solutions. In this paper, we explain the practical need for *algebraic* (equality-type) solutions, when we look for solutions for which both sides are equal. In situations when such a solution is not possible, we provide a justification for extended-zero solutions, in which we ignore intervals of the type $[-a, a]$.

Keywords: Interval equations · Extended zero
Nonlinear equations systems · Uncertainty · Algebraic solution

1 Practical Need for Solving Interval Systems of Equations: What Is Known

Need for data processing. In many practical situations, we are interested in the values of quantities $y_1, \ldots, y_m$ which are difficult – or even impossible – to measure directly. For example, we can be interested in a distance to a faraway star or in tomorrow's temperature at a certain location.

Since we cannot measure these quantities directly, to estimate these quantities we must:

- find easier-to-measure quantities $x_1, \ldots, x_n$ which are related to y_i by known formulas $y_i = f_i(x_1, \ldots, x_n)$,
- measure these quantities x_j, and

– use the results $\widetilde{x}_j$ of measuring the quantities x_j to compute the estimates for y_i:

$$\widetilde{y}_i = f(\widetilde{x}_1, \ldots, \widetilde{x}_n).$$

Computation of these estimates is called *indirect measurement* or *data processing.*

Need for data processing under uncertainty. Measurements are never 100% accurate. Hence, the measurement result $\widetilde{x}_j$ is, in general, different from the actual (unknown) value x_j of the corresponding quantity; in other words, the measurement errors $\Delta x_j \stackrel{\text{def}}{=} \widetilde{x}_j - x_j$ are, in general, different from 0.

Because of the non-zero measurement errors, the estimates $\widetilde{y}_i$ are, in general, different from the desired values y_i. It is therefore desirable to know how accurate are the resulting estimates.

Need for interval uncertainty and interval computations. The manufacturer of the measuring instrument usually provides us with an upper bound Δ_j on the measurement error: $|\Delta x_j| \le \Delta_j$; see, e.g., [8]. If no such upper bound is known, i.e., if the reading of the instrument can be as far away from the actual value as possible, then this is not a measuring instrument, this is a wild-guess-generator.

Sometimes, we also know the probabilities of different values Δx_j within this interval; see, e.g., [8,15]. However, in many practical situations, the upper bound is the only information that we have [8]. In this case, after we know the result $\widetilde{x}_j$ of measuring x_j, the only information that we have about the actual (unknown) value x_j is that this value belongs to the interval $[\underline{x}_j, \overline{x}_j]$, where $\underline{x}_j \stackrel{\text{def}}{=} \widetilde{x}_j - \Delta_j$ and $\overline{x}_j \stackrel{\text{def}}{=} \widetilde{x}_j + \Delta_j$.

In this case, the only thing that we can say about each value $y_i = f_i(x_1, \ldots, x_n)$ is that this value belongs to the range

$$\{f_i(x_1, \ldots, x_n) : x_1 \in [\underline{x}_1, \overline{x}_1], \ldots, x_n \in [\underline{x}_n, \overline{x}_n]\}.$$

Computation of this range is one of the main problems of *interval computations*; see, e.g., [3,6].

Sometimes, we do not know the exact dependence. The above text described an ideal case, when we know the exact dependence $y_i = f_i(x_1, \ldots, x_n)$ between the desired quantities y_i and the easier-to-measure quantities x_j. In practice, often, we do not know the exact dependence. Instead, we know that the dependence belongs to a finite-parametric *family* of dependencies, i.e., that

$$y_i = f_i(x_1, \ldots, x_n, a_1, \ldots, a_k)$$

for some parameters $a_1, \ldots, a_k$.

For example, we may know that y_i is a linear function of the quantities x_j, i.e., that $y_i = c_i + \sum\limits_{j=1}^{n} c_{ij} \cdot x_j$ for some coefficients c_i and c_{ij}.

The presence of these parameters complicates the corresponding data processing problem. Depending on what we know about the parameters, we have different situations.

Simplest situation, when we know the exact values of all the parameters. The simplest situation is when we know the exact values of these parameters. In this case, the dependence of y_i on x_j is known, and we have the same problem of computing the range as before.

Specific case: control solution. Sometimes, not only we *know* the values a_ℓ of these parameters, but we can also *control* these values, by setting them to any values within certain intervals $[\underline{a}_\ell, \overline{a}_\ell]$. By setting the appropriate values of the parameters, we can change the values y_i. This possibility naturally leads to the following problem:

– we would like the values y_i to be within some given ranges $[\underline{y}_i, \overline{y}_i]$; for example, we would like the temperature to be within a comfort zone;
– in this case, it is desirable to find the range of possible values of x_j for which, by applying appropriate controls $a_i \in [\underline{a}_\ell, \overline{a}_\ell]$, we can place the values y_i within these intervals.

In the degenerate case, when all the intervals for y_i and a_ℓ are just points, this means solving the system of equations $y = f(x, a)$, where we denoted $y \overset{\text{def}}{=} (y_1, \ldots, y_m)$, $x \overset{\text{def}}{=} (x_1, \ldots, x_n)$, and $a \overset{\text{def}}{=} (a_1, \ldots, a_k)$. From this viewpoint, the above problem can be viewed as an interval generalization of the problem of solving a system of equations, or, informally, as a problem of solving the corresponding interval system of equations.

The set X of all appropriate values $x = (x_1, \ldots, x_n)$ can be formally described as

$$X = \{x : \text{ for some } a_\ell \in [\underline{a}_\ell, \overline{a}_\ell], f_i(x_1, \ldots, x_n, a_1, \ldots, a_k) \in [\underline{y}_i, \overline{y}_i] \text{ for all } i\}.$$

This set is known as the *control solution* to the corresponding interval system of equations [3,14].

Situation when we need to find the parameters from the data. Sometimes, we know that the values a_i are the same for all the cases, but we do not know these values. These values must then be determined based on measurements: we measure x_j and y_i several times, and we find the values of the parameters a_ℓ that match all the measurement results.

Let us number all membership cycles by values $c = 1, \ldots, C$. After each cycle of measurements, we conclude that:

– the actual (unknown) value of $x_j^{(c)}$ is in the interval $[\underline{x}_j^{(c)}, \overline{x}_j^{(c)}]$ and
– the actual value of $y_i^{(c)}$ is in the interval $[\underline{y}_i^{(c)}, \overline{y}_i^{(c)}]$.

We want to find the set A of all the values a for which $y^{(c)} = f(x^{(c)}, a)$ for some $x^{(c)}$ and $y^{(c)}$:

$$A = \{a : \forall c\, \exists x_j^{(c)} \in [\underline{x}_j^{(c)}, \overline{x}_j^{(c)}]\, \exists y_i^{(c)} \in [\underline{y}_i^{(c)}, \overline{y}_i^{(c)}]\, (f(x^{(c)}, a) = y^{(c)})\}.$$

This set A is known as the *united solution* to the interval system of equations [3,14].

Comment. To avoid confusion, it is worth mentioning that our notations are somewhat different from the notations used in [3,14].

The main reason for this difference is that the main focus of this paper is on the *motivations* for different types of solutions. As a result, we use the notations related to the meaning of the corresponding variables. In general, in our description, y denotes the desired (difficult-to-measure) quantities, x denote easier-to-measure quantities, and a denote parameters of the dependence between these quantities.

Within this general situation, we can have different problems.

– In some cases, we have some information about the parameters a, and we need to know the values x – this is the case of the control solution.
– In other practical situations, we have some information about the quantities x, and we need to know the values a – this is the case, e.g., for the united solution.

As a result, when we use our meaning-of-variables notations, sometimes x's are the unknowns, and sometimes a's are the unknowns.

Alternatively, if we were interested in actually solving the corresponding problems, it would be more appropriate to use different notations, in which, e.g., the unknown is always denoted by x and the known values are denoted by a – irrespective of the physical meaning of the corresponding variables. In these notations, the united solution would take a different form

$$X = \{x : \forall c\, \exists a_j^{(c)} \in [\underline{a}_j^{(c)}, \overline{a}_j^{(c)}]\, \exists y_i^{(c)} \in [\underline{y}_i^{(c)}, \overline{y}_i^{(c)}]\, (f(x, a^{(c)}) = y^{(c)})\}.$$

What can we do once we have found the range of possible values of a. Once we have found the set A of possible values of a, we can first find the range of possible values of y_i based on the measurement results, i.e., find the range

$$\{f_i(x_1, \ldots, x_n, a) : x_j \in [\underline{x}_j, \overline{x}_j] \text{ and } a \in A\}.$$

This is a particular case of the main problem of interval computations.

If we want to make sure that each value y_i lies within the given bounds $[\underline{y}_i, \overline{y}_i]$, then we must find the set X of possible values of x for which $f_i(x, a)$ is within these bounds for all possible values $a \in A$, i.e., the set

$$X = \{x : \forall a \in A\, \forall i\, (f_i(x, a) \in [\underline{y}_i, \overline{y}_i])\}.$$

This set is known as the *tolerance solution* to the interval system of equations [3,14].

Sometimes, we know that the values a may change. In the previous text, we consider the situations when the values a_ℓ are either fixed forever, or can be changed by us. In practice, these values may change in an unpredictable way – e.g., if these parameters represent some physical processes that influence y_i's. We therefore do not know the exact values of these parameters, but what we do know is some a priori bounds on these values.

We may know bounds $[\underline{a}_\ell, \overline{a}_\ell]$ on each parameter, in which case the set A of all possible combinations $a = (a_1, \ldots, a_k)$ is simply a box:

$$A = [\underline{a}_1, \overline{a}_1] \times \ldots \times [\underline{a}_k, \overline{a}_k].$$

We may also have more general sets A – e.g., ellipsoids.

In this case, we can still solve the same two problems whose solutions we described above; namely:

- we can solve the main problem of interval computations – the problem of computing the range – and find the set Y of possible values of y;
- we can also solve the corresponding tolerance problem and find the set of values x that guarantee that each y_i is within the desired interval.

Is this all there is? There are also more complex problems (see, e.g., [14]), but, in a nutshell, most practical problems are either range estimation, or finding control, united, or tolerance solution. These are the problems solved by most interval computation packages [3,6].

Is there anything else? In this paper, we show that there is an important class of practical problems that does not fit into one of the above categories. To solve these practical problems, we need to use a different notion of a solution to interval systems of equations: the notion of an *algebraic* (equality-type) solution, the notion that has been previously proposed and theoretically analyzed [1,2,5, 7,9,12–14] but is not usually included in interval computations packages.

2 Remaining Problem of How to Find the Set A Naturally Leads to Algebraic (Equality-Type) Solutions to Interval System of Equations

Finding the set A: formulation of the problem. In the previous text, we assumed that when the values of the parameter a can change, we know the set A of possible values of the corresponding parameter vector. But how do we find this set?

What information we can use to find the set A. All the information about the real world comes from measurements – either directly from measurements, or by processing measurement results. The only relation between the parameters a and measurable quantities is the formula $y = f(x, a)$. Thus, to find the set A of possible values of a, we need to use measurements of x and y.

We can measure both x and y many times. As a result, we get:

- the set X of possible values of the vector x and
- the set Y of possible values of the vector y.

Both sets can be boxes, or they can be more general sets.

Based on these two sets X and Y, we need to find the set A.

In this problem, it is reasonable to assume that x and a are *independent* in some reasonable sense. Let us formulate this requirement in precise terms.

Independence: towards a formal definition. The notion of independence is well known in the probabilistic case, where it means that probability of getting a value $x \in X$ does not depend on the value $a \in A$: $P(x \mid a) = P(x \mid a')$ for all $a, a' \in A$. An interesting corollary of this definition is that, in spite of being formulated in a way that is asymmetric with respect to x and a, this definition is actually symmetric: one can prove that a is independent of x if and only if x is independent of a.

In the interval case, we do not know the probabilities, we only know which pairs (x, a) are possible and which are not. In other words, we have a *set* $S \subseteq X \times A$ of possible pairs (x, a). It is natural to say that the values x and a are independent if the set of possible values of x does not depend on a. Thus, we arrive at the following definition.

Definition 1. *Let $S \subseteq X \times A$ be a set.*

- *We say that a pair (x, a) is* possible *if $(x, a) \in S$.*
- *Let $x \in X$ and $a \in A$. We say that a value x is* possible under a *if $(x, a) \in S$. The set of possible-under-a values will be denoted by S_a.*
- *We say that the variables x and a are* independent *if $S_a = S_{a'}$ for all a, a' from the set A.*

Proposition 1. *Variable x and a are independent if and only if S is a Cartesian product, i.e., $S = s_x \times s_a$ for some $s_x \subseteq X$ and $s_a \subseteq A$.*

Proof. If $S = s_x \times s_a$, then $S_a = s_x$ for each a and thus, $S_a = S_{a'}$ for all $a, a' \in A$.

Vice versa, let us assume that x and a are independent. Let us denote the common set $S_a = S_{a'}$ by s_x. Let us denote by s_a, the set of all possible values $a \in A$, i.e., the set of all $a \in A$ for which $(x, a) \in S$ for some $x \in X$. Let us prove that in this case, $S = s_x \times s_a$.

Indeed, if $(x, a) \in S$, then, by definition of the set s_x, have $x \in S_a = s_x$, and, by definition of the set s_a, we have $a \in s_a$. Thus, by the definition of the Cartesian product $B \times C$ as the set of all pairs (b, c) of all pairs of elements $b \in B$ and $c \in C$, we have $(x, a) \in s_x \times s_a$.

Vice versa, let $(x, a) \in s_x \times s_a$, i.e., let $x \in s_x$ and $a \in s_a$. By definition of the set s_x, we have $S_a = s_x$, thus $x \in S_a$. By definition of the set S_a, this means that $(x, a) \in S$. The proposition is proven.

As a corollary, we can conclude that the independence relation is symmetric – similarly to the probabilistic case.

Corollary. *Variables x and a are independent if and only if a and x are independent.*

Proof. Indeed, both case are equivalent to the condition that the set S is a Cartesian product.

What can we now conclude about the dependence between A, X, and Y. Since we assumed that a and x are independent, we can conclude that the set of possible values of the pair (x, a) is the Cartesian product $X \times A$. For each such pair, the value of y is equal to $y = f(x, a)$. Thus, the set Y is equal to the range of $f(x, a)$ when $x \in X$ and $a \in A$.

The resulting solutions to interval systems of equations. So, we look for sets A for which

$$Y = f(X, A) \overset{\text{def}}{=} \{f(x, a) : x \in X \text{ and } a \in A\}.$$

It is reasonable call the set A satisfying this property an *equality-type solution* to the interval system of equations.

Such solutions for the interval system of equations $y = f(x, a)$, in which we want the interval versions Y and $f(X, A)$ of both sides of the equation to be exactly equal, are known as *algebraic* or, alternatively, *formal* solutions; see, e.g., [1,2,5,7,9,12–14].

3 What If the Interval System of Equations Does Not Have an Algebraic (Equality-Type) Solution: A Justification for Enhanced-Zero Solutions

But what if an equality-type solution is impossible: analysis of the problem. The description in the previous section seems to make sense, but sometimes, the corresponding problem has no solutions. For example, in the simplest case when $m = n = k = 1$ and $f(x, a) = x + a$, if we have $Y = [-1, 1]$ and $X = [-2, 2]$, then clearly the corresponding equation $Y = X + A$ does not have a solution: no matter what set A we take the width of the resulting interval $X + A$ is always larger than or equal to the width $w(X) = 4$ of the interval X and thus, cannot be equal to $w(Y) = 2$. What shall we do in this case? How can we then find the desired set A?

Of course, this would not happen if we had the *actual* ranges X and Y, but in reality, we only have estimates for these ranges. So, the fact that we cannot find A means something is wrong with these estimates.

How are ranges X and Y estimated in the first place? To find out what can be wrong, let us recall how the ranges can be obtained from the experiments. For example, in the 1-D case, we perform several measurements of the quantity x_1 in different situations. Based on the corresponding measurement results $x_1^{(c)}$, we conclude that the interval of possible values must include the set $[\underline{x}_1^{\approx}, \overline{x}_1^{\approx}]$, where $\underline{x}_1^{\approx} \overset{\text{def}}{=} \min\limits_c x_1^{(c)}$ and $\overline{x}_1^{\approx} \overset{\text{def}}{=} \max\limits_c x_1^{(c)}$. Of course, we can also have some values outside this interval – e.g., for a uniform distribution on an interval $[0, 1]$, the interval formed by the smallest and the largest of the C random numbers is slightly narrower than $[0, 1]$; the fewer measurement we take, the narrower this interval.

So, to estimate the actual range, we *inflate* the interval $[\underline{x}_1^{\approx}, \overline{x}_1^{\approx}]$. In these terms, the fact that we have a mismatch between X and Y means that one of these intervals was not inflated enough.

The values x correspond to easier-to-measure quantities, for which we can make a large number of measurements and thus, even without inflation, get pretty accurate estimates of the actual range X. On the other hand, the values y are difficult to measure; for these values, we do not have as many measurement results and thus, there is a need for inflation.

From this viewpoint, we can safely assume that the range for X is reasonably accurate, but the range of Y needs inflation.

So how do we find A? In view of the above analysis, if there is no set A for which $Y = f(X, A)$, the proper solution is to inflate each components of the set Y so that the system becomes solvable.

To make this idea precise, let us formalize what is an inflation.

What is an inflation: analysis of the problem. We want to define a mapping I that transforms each non-degenerate interval $\mathbf{x} = [\underline{x}, \overline{x}]$ into a wider interval

$$I(\mathbf{x}) \supset \mathbf{x}.$$

What are the natural properties of this transformation? The numerical value x of the corresponding quantity depends on the choice of the measuring unit, on the choice of the starting point, and – sometimes – on the choice of direction.

- For example, we can measure temperature t_C in Celsius, but we can also use a different measuring unit and a different starting point and get temperatures in Fahrenheit $t_F = 1.8 \cdot t_C + 32$.
- We can use the usual convention and consider the usual signs of the electric charge, but we could also use the opposite signs – then an electron would be a positive electric charge.

It is reasonable to require that the result of the inflation transformation does not change if we simply change the measuring units or change the starting point or change the sign:

- Changing the starting point leads to a new interval $[\underline{x}, \overline{x}] + x_0 = [\underline{x} + x_0, \overline{x} + x_0]$ for some x_0.
- Changing the measuring unit leads to $\lambda \cdot [\underline{x}, \overline{x}] = [\lambda \cdot \underline{x}, \overline{x}]$ for some $\lambda > 0$.
- Changing the sign leads to $-[\underline{x}, \overline{x}] = [-\overline{x}, -\underline{x}]$.

Thus, we arrive at the following definition.

Definition 2. *By an* inflation*, we mean a mapping that maps each non-degenerate interval $\mathbf{x} = [\underline{x}, \overline{x}]$ into a wider interval $I(\mathbf{x})$ so that:*

- *for every x_0, we have $I(\mathbf{x} + x_0) = I(\mathbf{x}) + x_0$;*
- *for every $\lambda > 0$, we have $I(\lambda \cdot \mathbf{x}) = \lambda \cdot I(\mathbf{x})$; and*
- *we have $I(-\mathbf{x}) = -I(\mathbf{x})$.*

Proposition 2. *Every inflation operation has the form*

$$[\widetilde{x} - \Delta, \widetilde{x} + \Delta] \to [\widetilde{x} - \alpha \cdot \Delta, \widetilde{x} + \alpha \cdot \Delta]$$

for some $\alpha > 1$.

Comment. A similar result was proven in [4].

Proof. It is easy to see that the above operation satisfies all the properties of an inflation. Let us prove that, vice versa, every inflation has this form.

Indeed, for intervals $\mathbf{x}$ of type $[-\Delta, \Delta]$, we have $-\mathbf{x} = \mathbf{x}$, thus $I(\mathbf{x}) = I(-\mathbf{x})$. On the other hand, due to the third property of an inflation, we should have $I(-\mathbf{x}) = -I(\mathbf{x})$. Thus, for the interval $[\underline{v}, \overline{v}] \stackrel{\text{def}}{=} I(\mathbf{x})$, we should have $-[\underline{v}, \overline{v}] = [-\overline{v}, -\underline{v}] = [\underline{v}, \overline{v}]$ and thus, $\underline{v} = -\overline{v}$. So, we have $I([-\Delta, \Delta]) = [-\Delta'(\Delta), \Delta'(\Delta)]$ for some Δ'. Since we should have $[-\Delta, \Delta] \subset I([-\Delta, \Delta])$, we must have

$$\Delta'(\Delta) > \Delta.$$

Let us denote $\Delta'(1)$ by α. Then, $\alpha > 1$ and $I([-1, 1]) = [-\alpha, \alpha]$. By applying the second property of the inflation, with $\lambda = \Delta$, we can then conclude that $I([-\Delta, \Delta]) = [-\alpha \cdot \Delta, \alpha \cdot \Delta]$. By applying the first property of the inflation operation, with $x_0 = \widetilde{x}$, we get the desired equality

$$I([\widetilde{x} - \Delta, \widetilde{x} + \Delta]) = [\widetilde{x} - \alpha \cdot \Delta, \widetilde{x} + \alpha \cdot \Delta].$$

The proposition is proven.

So how do we find A**?** We want to make sure that $f(X, A)$ is equal to the result of a proper inflation of Y.

How can we tell that an interval Y' is the result of a proper inflation of Y? One can check that this is equivalent to the fact that the difference $Y' - Y$ is a symmetric interval containing 0; such intervals are known as *extended zeros* [10,11].

Thus, if we cannot find the set A for which $Y = f(X, A)$, we should look for the set A for which the difference $f(X, A) - Y$ is an extended zero.

Historical comment. This idea was first described in [10,11]; in this paper, we provide a new theoretical justification of this idea.

Multi-D case. What if we have several variables, i.e., $m > 1$? In this case, we may have different inflations for different components Y_i of the set Y, so we should look for the set A for which, for all i, the corresponding difference $f_i(X, A) - Y_i$ is an extended zero.

Acknowledgments. This work was supported in part by the grant DEC-2013/11/B/ST6/00960 from the National Science Center (Poland), by the US National Science Foundation grants HRD-0734825 and HRD-1242122 (Cyber-ShARE Center of Excellence) and DUE-0926721, and by an award "UTEP and Prudential Actuarial Science Academy and Pipeline Initiative" from Prudential Foundation.

The authors are thankful to the anonymous referees for valuable suggestions.

References

1. Chakraverty, S., Hladík, M., Behera, D.: Formal solution of an interval system of linear equations with an application in static responses of structures with interval forces. Appl. Math. Model. **50**, 105–117 (2017)
2. Chakraverty, S., Hladík, M., Mahato, N.R.: A sign function approach to solve algebraically interval system of linear equations for nonnegative solutions. Fundam. Inform. **152**, 13–31 (2017)
3. Jaulin, L., Kiefer, M., Dicrit, O., Walter, E.: Applied Interval Analysis. Springer, London (2001). https://doi.org/10.1007/978-1-4471-0249-6
4. Kreinovich, V., Starks, S.A., Mayer, G.: On a theoretical justification of the choice of epsilon-inflation in PASCAL-XSC. Reliab. Comput. **3**(4), 437–452 (1997)
5. Lakeyev, A.: On the computational complexity of the solution of linear systems with moduli. Reliab. Comput. **2**(2), 125–131 (1996)
6. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
7. Nickel, K.: Die Auflösbarkeit linearer Kreisscheineb- und Intervall-Gleichingssyteme. Linear Algebra Appl. **44**, 19–40 (1982)
8. Rabinovich, S.G.: Measurement Errors and Uncertainty: Theory and Practice. Springer, Berlin (2005). https://doi.org/10.1007/0-387-29143-1
9. Ratschek, K., Sauer, W.: Linear interval equations. Computing **25**, 105–115 (1982)
10. Sevastjanov, P., Dymova, L.: Fuzzy solution of interval linear equations. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 1392–1399. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68111-3_147
11. Sevastjanov, P., Dymova, L.: A new method for solving interval and fuzzy equations: linear case. Inf. Sci. **17**, 925–937 (2009)
12. Shary, S.P.: Algebraic approach to the interval linear static identification, tolerance, and control problems, or one more application of Kaucher arithmetic. Reliab. Comput. **2**(1), 3–33 (1996)
13. Shary, S.P.: Algebraic approach in the 'outer problem' for interval linear equations. Reliab. Comput. **3**(2), 103–135 (1997)
14. Shary, S.P.: A new technique in systems analysis under interval uncertainty and ambiguity. Reliab. Comput. **8**, 321–418 (2002)
15. Sheskin, D.J.: Handbook of Parametric and Nonparametric Statistical Procedures. Chapman and Hall/CRC, Boca Raton (2011)

Workshop on Complex Collective Systems

Application of Local Search with Perturbation Inspired by Cellular Automata for Heuristic Optimization of Sensor Network Coverage Problem

Krzysztof Trojanowski, Artur Mikitiuk(✉), and Krzysztof J. M. Napiorkowski

Cardinal Stefan Wyszyński University in Warsaw, Warsaw, Poland
{k.trojanowski,a.mikitiuk}@uksw.edu.pl

Abstract. A cellular automata inspired approach to the problem of effective energy management in a sensor network is presented. The network consisting of a set of sensors is disseminated over an area where a number of points of interest (POI) is localized. The aim is to maximize the time when a sufficient number of POIs is monitored by active sensors. A schedule of sensor activity over time is a solution of this problem. A new heuristic algorithm inspired by a cellular automata engine is proposed. It searches for such schedules maximizing the lifetime of the sensor network. We also present a set of test cases for experimental evaluation of our approach. The proposed algorithm is experimentally tested using these test cases and the obtained results are statistically verified to prove significant contribution of the algorithm components.

Keywords: Local search · Sensor networks
Maximum Lifetime Coverage Problem · Cellular automata

1 Introduction

Multiple sensor network applications stimulate research concerning optimization of their effectiveness and efficiency. One of the main goals of this research is the minimization of energy consumption or, in other words, maximization of the network lifetime. Maximization of a time when an area is monitored by a set of distributed sensors with a limited battery capacity is the subject of the presented research. We assume that a set of points of interest (POI) is distributed in the monitored area and a coverage constraint has to be satisfied by a network, that is, sensors have to provide uninterrupted monitoring of a sufficient number of POIs all the time. The number of sensors is high so their monitoring areas can overlap. Therefore, usually, there exists a group of POIs which are monitored by more than one sensor. This opens a possibility of the network lifetime optimization since we know that sleeping sensors save their energy and it is not necessary to have all the sensors in an active state all the time. Precisely, we search for a schedule which represents effective energy management in the sensor batteries, that is, the longest schedule defining sensor states over time satisfying the coverage constraint.

In the presented research an idea of a cellular automaton (CA) is employed in the initialization and perturbation procedures to generate slots of a network activity schedule. The main novelty lies in an adaptation of CA components, like a neighborhood structure and a transition function to perform this task. A new algorithm for generation of sensor activity schedules based on this idea is proposed. The schedule obtained by this algorithm is an input for a local search (LS) algorithm whose goal is to improve the result. The LS algorithm makes use of an original perturbation operator which generates neighbour schedules. We also present a set of test cases for experimental evaluation of our approach.

The paper consists of six sections. Section 2 gives the definition of the solved problem and all the necessary constraints. The proposed algorithms are presented in Sect. 3. A benchmark is defined in Sect. 4 and the results of experiments are discussed in Sect. 5. Section 6 concludes the paper.

2 Maximum Lifetime Coverage Problem (MLCP)

In the sensor coverage problem [2], $N_{\mathbf{S}}$ immobile sensors are randomly deployed over an area to control a set of points of interest (POI). Each sensor has a sensing range r_{sens} and its battery is at the beginning fully loaded. We assume that time is discrete and an active sensor consumes one unit of energy per time unit for sensing and communication. A sensor can be active during T_{batt} consecutive, or not, time steps. After deployment, the sensors schedule their activity.

Active sensors monitor all POIs located within their sensing range and one POI can be monitored by more than one active sensor at the same time. For effective monitoring of a given set of POIs, it is not necessary to control all of them all the time. We need to achieve a satisfying coverage level *cov* – some percentage of the number of POIs being monitored (usually 80–90%). On the other hand, we do not want to exceed this satisfying coverage level too much to save sensor batteries. Thus, our goal is to monitor all the time the percentage of POIs in the range $[cov, cov + \delta]$, where δ represents a tolerance factor.

During one time slice a sensor can be either active (in a *working* mode) or inactive (in a *sleeping* state). It is assumed that in a sleeping state the energy consumption is negligible. We ignore the problem of communication between sensors both in terms of energy consumption necessary for communication and in terms of connectivity in the communication graph. We assume that sensors are always able to communicate regardless of their localization even if some sensors are in a sleeping state.

A solution we look for is a schedule of sensor activity which gives the satisfying coverage level for the given set of POIs as long as possible. This is called the Maximum Lifetime Coverage Problem. The schedule is a matrix H of 0s and 1s representing states of sensors *off* and *on* in consecutive time slots. The value of a schedule is the length of the longest time period during which the coverage requirement is met.

There are many publications concerning sensor networks. However, literature about heuristic approaches to MLCP as defined above is not very rich. One can find, e.g., papers on sensor control methods [5] or schedule optimization based on evolutionary algorithms [6], simulated annealing [7], or graph cellular automata [8]. In [3] a method for the problem of multiple POI coverage is proposed but in this case, mobile sensors are considered.

3 Search Algorithm

A local search approach is used for sensor activity schedule optimization. The main novelty lies in the method for generation of a new schedule which is used both for generation of an initial one and for building a neighbour schedule. After the first schedule is created by Algorithm 1, the algorithm iteratively tries to improve it by the problem specific neighbour operator $generateNeighbour$ (see Algorithm 3). When a newly found schedule is longer than the current one, the new one takes place of the current and the process continues.

3.1 Generation of the Initial Schedule

A method of a new schedule generation presented in Algorithm 1 starts with a schedule which can be either empty or partially completed. The main idea is to add to the schedule new slots one by one even if they do not satisfy the coverage constraint. For such unfeasible slots, the battery levels are not updated. Eventually, when no more slots can be added, the unfeasible ones are deleted from the schedule.

General Description of the Method: Every slot in a schedule undergoes the same process: states of slot cells are computed with the use of a procedure inspired by Cellular Automata (PICA) first, and if the coverage constraint is satisfied the states are accepted. Otherwise, the proposed states are forgotten and another procedure sets cells states. PICA implements a sort of cellular automata procedure based on a graph universe (see, for example, [1,9] for details). Due to the fact that generation of a current state of cells requires states of cells for the previous time step, when the schedule is built from scratch the initial slot is filled entirely by zeros (line 2). Then, PICA is run with a copy of the slot from the previous time step as an input argument (lines 7–8). If the output of PICA does not satisfy the coverage constraint, another procedure for evaluation of slot cells is executed. First, all the sensors which have non-empty batteries are set to *on* (lines 9–10). Then, just to prevent lavish coverage settings, randomly selected sensors are switched off as long as the set of remaining active sensors is able to cover much more than a sufficient number of POIs, that is, $covPoi(H^t) > (cov + \delta)$ (lines 11–13). Eventually, whether the obtained slot is feasible or not, it is accepted in the schedule, however, only in the former case the battery levels and the set of active sensors are updated and the max effective coverage $cov_{S(max)}$ is reevaluated (lines 14–18). The loop stops when the set of remaining

Algorithm 1. CA Inspired Generation of a Schedule (CABG)

```
Require: $\mathbf{S}, cov, t, H$;
Ensure: $H$
 1: if $t = 0$ then                                   ▷ if a schedule is created from scratch..
 2:     for all $S \in \mathbf{S}$ do $H^t_S \leftarrow 0$
 3: $\mathbf{S}_{\text{work}} \leftarrow filter(\mathbf{S}, t)$        ▷ select subset of active sensors having POIs in range
 4: $cov_{S(max)} \leftarrow covPoi(\mathbf{S}_{\text{work}})$     ▷ evaluate coverage when all active sensors are on
 5: repeat
 6:     $t \leftarrow t + 1$
 7:     $H^t \leftarrow H^{t-1}$
 8:     $H^t \leftarrow \text{PICA}(\mathbf{S}, \mathbf{S}_{\text{work}}, H^t)$          ▷ build a slot with a cellular automaton
 9:     if $covPoi(H^t) < cov$ then         ▷ if the coverage constraint is not satisfied..
10:         for all $S \in \mathbf{S}_{\text{work}}$ do $H^t_S \leftarrow 1$          ▷ turn on all sensors from $\mathbf{S}_{work}$
11:     while $covPoi(H^t) > (cov + \delta)$ do              ▷ while the coverage is lavish ..
12:         $i \leftarrow rand(1, N_{\mathbf{S}})$                         ▷ .. select a sensor randomly, and ..
13:         $H^t_{S_i} \leftarrow 0$                                          ▷ .. set its state to off
14:     if $cov \leq covPoi(H^t) \leq (cov + \delta)$ then ▷ if the coverage constraint is satisfied..
15:         for all $S \in \mathbf{S} \mid H^t_S = 1$ do
16:             $batt(S) \leftarrow batt(S) - 1$                 ▷ .. sensor batteries level update
17:         $\mathbf{S}_{\text{work}} \leftarrow filter(\mathbf{S}, t)$                                   ▷ $\mathbf{S}_{\text{work}}$ update
18:         $cov_{S(max)} \leftarrow covPoi(\mathbf{S}_{\text{work}})$                         ▷ $cov_{S(max)}$ update
19: until $cov > cov_{S(max)}$
20: for $t \leftarrow 1, T_{\mathbf{max}}$ do     ▷ remove slots which do not satisfy the coverage constraint
21:     if $(covPoi(H^t) < cov) \vee (covPoi(H^t) > (cov + \delta))$ then $delete(H^t)$
22: return $H$
```

Algorithm 2. Slot Generation Inspired by Cellular Automata (PICA)

```
 1: procedure $\text{PICA}(\mathbf{S}, \mathbf{S}_{\text{work}}, H^t)$
 2:     for all $S \in \mathbf{S}$ do                              ▷ cellular automaton execution
 3:         $n \leftarrow 0$                ▷ initialize the number of active neighbour sensors for S
 4:         for all $R \in \mathbf{S} \mid R \neq S$ do
 5:             if $\mathbb{N}(S, R) \wedge H^t_R = 1$ then      ▷ if S and R are neighbours and R is on
 6:                 $n \leftarrow n + 1$   ▷ increase the number of active neighbour sensors for S
 7:         if $n = 0 \wedge S \in \mathbf{S}_{\text{work}}$ then
 8:             $H^t_S \leftarrow 1$                                       ▷ turn the sensor S on
 9:         else
10:             $H^t_S \leftarrow 0$                                       ▷ turn the sensor S off
11:     return $H^t$
```

active sensors is not able to cover a sufficient number of POIs even if all of them are on, that is, $cov > cov_{S(max)}$.

Details of Procedure Inspired by Cellular Automata (PICA): PICA implements steps typically executed in CA, however, it must be stressed that in contrast to CA its main aim is building a new slot from an existing one.

In Cellular Automata (CA) main components are (1) cells which can change their states, (2) a neighborhood relationship between cells and (3) a transition function. Cells are represented by sensors and can be in any of the two states: *on* or *off*. In a slot, the states are represented by 1 or 0, respectively. Two sensors are regarded as neighbors when their monitoring areas overlap. The number of neighbors for each of sensors can be different and depends only on their location in the monitored area. Thus, cells in our automata do not form a grid universe but rather a graph universe where nodes represent sensors and edges represent neighbour relationship. A transition function q_{tr} changes values in a slot, that is, the state of sensors in the network respectively to the state of their neighbour sensors. The function changes the state to *on* when all neighbors are *off* and the sensor battery is not empty. Otherwise, the sensor state is set to *off*. Formally, $q_{\mathrm{tr}}(S)^{t+1} = \prod_{i=1}^{n}(1 - q_{\mathrm{tr}}(R_i^t)) \cdot \mathrm{sgn}(batt(S^{t+1}))$ where S is the sensor to evaluate, R_i – a neighbour sensor of S, n – the number of neighbours of S, sgn – a signum function which returns one for values greater than zero, zero for zero, and minus one for values less than zero. The transition function works asynchronously, that is, the sensor state is changed instantly and may influence decisions concerning other sensors which have not changed their state yet. The order of sensors undergoing the state change is not deterministic and may be different in subsequent steps of CA. This way of cell update in this particular case is justified because it guarantees different results of perturbation for the same slot in subsequent trials. It is worth noting that sensors located in the boundary regions of the monitored area have fewer neighbors than the others, however, the transition rule is the same for all of them. Details of the cellular automata procedure (CA) are presented in Algorithm 2. A neighborhood function $\mathbb{N}(S, R)$ used in Algorithm 2 returns true in the case of the neighbour relationship between S and R, and false – otherwise.

3.2 Iterative Improvement of the Schedule

A method of a neighbour schedule generation is presented in Algorithm 3. In the first stage of this procedure (lines 2–4) five percent of cells in the input schedule is set to zero (it concerns also the cells already set to zero). Then, all slots are verified whether they still satisfy the coverage constraint. Slots which are unfeasible are deleted (lines 5–6). Next, sensor batteries are recovered respectively to the number of deleted slots and settings inside them. Finally, the CABG procedure is called for the modified schedule and the outcome of this procedure is returned as the neighbour schedule.

4 Benchmark SCP1

We prepared a set of test cases called SCP1 (Sensor Coverage Problem, Set No. 1) to evaluate the proposed algorithm. Every test case in this set can be described by a set of parameters. In all test cases, there are 2000 sensors with sensing range r_{sens} 1 unit. We require the coverage level $cov = 80\%$ with tolerance

Algorithm 3. A Neighbour Schedule Generation

```
1: procedure $generateNeighbour(\mathbf{S}, \mathbf{H}, \mathbf{T}_{\max}, cov, \delta)$
2:     for $t \leftarrow 1, T_{\mathbf{max}}$ do
3:         for all $S \in \mathbf{S}$ do
4:             if $rand(0,1) < 0.05$ then $H_S^t \leftarrow 0$
5:     for $t \leftarrow 1, T_{\mathbf{max}}$ do          ▷ check where the coverage constraint is not satisfied
6:         if $covPoi(H^t) < cov$ then $delete(H^t)$          ▷ delete unfeasible slot
7:     $T_{\mathbf{max}} \leftarrow length(H)$          ▷ update schedule length
8:     $batteryRecovery(\mathbf{S}, H)$          ▷ recover battery levels due to deleted slots in H
9:     return CABG$(\mathbf{S}, cov, T_{\mathbf{max}}, H)$          ▷ return the outcome of CABG
```

$\delta = 5\%$. The remaining parameters vary between test cases. There are two types of distribution of POIs – POIs can be located in nodes of a triangular grid or a rectangular grid. Coordinates of sensor localization can be obtained using either a random generator or a Halton generator [4]. The area under consideration is a square. Its side size can be 13, 16, 19, 22, 25, or 28 units.

The number of POIs is the same for different area sizes. Thus, the distances between POIs become larger as the square side grows. To avoid full regularity in the POIs distribution, 20% of nodes in the grid is not filled with POIs. These nodes are selected randomly for every instance of the test case. The number of POIs in subsequent test cases varies from 199 to 240 for the triangular grid and from 166 to 221 for the rectangular grid.

We selected for experiments 8 configurations of the test case which differ in the area size, the type of a grid for POIs, and the generator of sensor locations. It was our arbitrary decision to select these particular configurations – someone else could choose different values for some or all parameters or propose more than 8 configurations. A set of 40 instances was generated for every test case. Every instance consists of a file with coordinates of POI locations and a file with coordinates of sensor locations. Table 1 presents how many sensors can control on average given numbers of POIs for each of the eight test cases.

Table 1. Mean numbers of sensors covering 0, 1, 2, 3, 4, 5 and more that 5 POIs for the eight test cases of SCP1. △ means a triangular grid of POI locations, and ⋄ - a rectangular grid.

No.	Configuration	0	1	2	3	4	5	>5
1	13×13, △, rand	5.0	58.2	234.9	559.7	691.6	327.9	122.7
2	13×13, ⋄, Halton	18.3	126.6	369.4	691.2	693.5	95.4	5.7
3	16×16, △, Halton	24.6	211.1	679.2	902.7	182.4	0.0	0.0
4	19×19, ⋄, rand	135.1	763.0	951.2	128.8	21.8	0.0	0.0
5	19×19, △, rand	112.7	631.7	819.7	435.8	0.0	0.0	0.0
6	22×22, △, Halton	209.9	1012.6	665.9	111.6	0.0	0.0	0.0
7	25×25, △, rand	340.1	1303.4	350.9	5.6	0.0	0.0	0.0
8	28×28, △, Halton	450.1	1475.2	74.7	0.0	0.0	0.0	0.0

This table shows that in the first test case some sensors cover even 5 or more POIs. In the next cases, sensors can cover smaller and smaller numbers of POIs. In the last test case, almost 75% of sensors cover only one POI. Similarly, Fig. 1 shows, that in the first test case the intersections between neighbor sensor monitoring areas are greater and contain larger numbers of POIs. For the next test cases, these areas become smaller and smaller and the numbers of common POIs decrease as well.

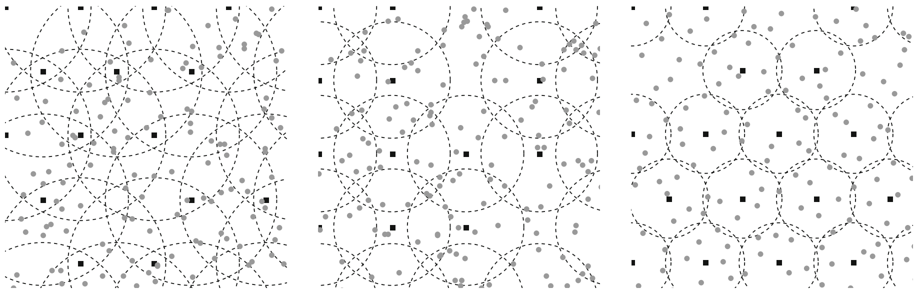

(a) No. 1: △, rand, 13 (b) No. 4: ⋄, rand, 19 (c) No. 8: △, Halton, 28

Fig. 1. Visualizations of a monitored area for selected instances of a three among the eight test cases in SCP1: squares represent POIs, dots — sensors, circles around POIs — which sensors have in its range the POI located in the circle center.

5 Results of Experiments

The SCP1 benchmark was used for experiments with the proposed algorithm. We generated 40 instances for every test case from SCP1 and the algorithm was executed once for each of these instances. The experiments were performed for five different values of T_{batt}: 10, 15, 20, 25, and 30. Results of these experiments are given in Table 2. Table rows show mean lengths of schedules obtained for respective 40 instances and min and max lengths among them.

The top half of Table 2 presents results returned by Algorithm 1 while the bottom half – results of the Local Search (LS) algorithm using the perturbation operator presented in Algorithm 3. The values in the series are paired because output schedules of Algorithm 1 were an input for the LS algorithm.

One can see that the results are better for the cases when the intersections between the sets of POIs controlled by neighboring sensors are greater. Moreover, mean lengths of schedules are proportional to the sensor lifetime T_{batt}.

We performed statistic t-tests for paired data to determine whether application of LS improves the length of the schedules. The null hypothesis is that any differences in schedule lengths before and after LS are due to chance. Table 3 shows obtained p-values. Since in every case the value is below 0.001, the null

Table 2. Mean, min and max lengths of schedules returned by Algorithm 1 – top part – and by the LS algorithm – bottom part – for each of the eight test cases in SCP1 and for five values of T_{batt} from 10 to 30

No.	T_{batt}	Mean	Min	Max	No.	T_{batt}	Mean	Min	Max
1	10	263.1	256	273	5	10	118.4	113	122
	15	394.0	383	406		15	178.0	169	186
	20	525.9	514	547		20	236.3	226	245
	25	657.7	638	683		25	295.8	280	305
	30	787.3	767	812		30	355.4	339	370
2	10	279.7	275	285	6	10	97.7	96	100
	15	420.7	412	433		15	147.0	142	151
	20	560.6	552	573		20	195.9	191	201
	25	701.9	687	718		25	245.3	238	252
	30	843.5	827	866		30	295.0	288	302
3	10	182.7	177	186	7	10	66.8	63	71
	15	274.5	268	283		15	100.0	94	104
	20	366.7	357	375		20	133.4	126	140
	25	459.2	448	474		25	167.0	158	173
	30	550.4	537	562		30	199.8	189	210
4	10	108.6	103	113	8	10	60.6	59	62
	15	163.4	155	170		15	91.0	88	93
	20	217.5	207	226		20	122.1	118	126
	25	271.9	261	284		25	152.7	149	156
	30	326.7	308	339		30	183.5	179	188
1	10	277.3	269	287	5	10	125.1	120	130
	15	415.4	403	429		15	188.3	179	197
	20	554.4	542	576		20	250.0	240	260
	25	693.0	672	718		25	313.1	297	323
	30	830.1	810	855		30	375.9	361	390
2	10	294.3	288	301	6	10	103.1	100	106
	15	442.6	435	456		15	155.3	149	160
	20	590.1	580	604		20	207.1	202	213
	25	738.8	724	755		25	259.2	252	266
	30	887.8	871	911		30	311.8	305	318
3	10	192.4	186	196	7	10	71.1	67	75
	15	289.4	282	298		15	106.5	100	111
	20	386.6	376	396		20	140.3	130	145
	25	484.1	473	498		25	177.5	168	184
	30	580.1	567	591		30	212.4	202	222
4	10	114.8	109	120	8	10	64.4	63	66
	15	172.6	164	180		15	96.8	94	99
	20	230.4	220	239		20	129.7	126	133
	25	287.6	275	300		25	158.4	153	162
	30	345.8	327	359		30	194.6	190	199

Table 3. Results of statistical tests for paired samples obtained from Algorithm 1 and LS: p-values, std.dev.#1 for the results from Algorithm 1 and std.dev.#2 for LS obtained for each of the eight test cases in SCP1 and for five values of T_{batt} from 10 to 30

No.	T_{batt}	p-value	s.d.#1	s.d.#2	No.	T_{batt}	p-value	s.d.#1	s.d.#2
1	10	$1.85E^{-53}$	4.22	4.32	5	10	$1.4E^{-42}$	2.65	2.82
	15	$2.4E^{-52}$	5.49	6.01		15	$7.4E^{-49}$	4.28	4.56
	20	$4.25E^{-59}$	8.01	8.18		20	$1.2E^{-50}$	5.01	5.06
	25	$7.46E^{-55}$	10.36	11.15		25	$5.72E^{-51}$	6.31	6.55
	30	$1.02E^{-55}$	12.18	12.55		30	$1.62E^{-53}$	7.79	8.01
2	10	$2.31E^{-51}$	2.90	3.19	6	10	$1.09E^{-39}$	1.23	1.40
	15	$1.14E^{-53}$	4.52	4.89		15	$1.52E^{-44}$	1.83	2.08
	20	$1.02E^{-56}$	5.82	6.31		20	$1.87E^{-47}$	2.19	2.48
	25	$3.73E^{-58}$	7.22	7.77		25	$1.71E^{-54}$	2.95	3.11
	30	$3.02E^{-57}$	8.42	9.03		30	$1E^{-53}$	3.37	3.56
3	10	$4.66E^{-50}$	2.15	2.23	7	10	$8.41E^{-38}$	1.92	2.01
	15	$7.23E^{-52}$	2.97	3.19		15	$7.79E^{-44}$	2.37	2.56
	20	$4.64E^{-57}$	4.05	4.33		20	$5.62E^{-46}$	3.48	3.49
	25	$4.97E^{-56}$	5.38	5.60		25	$7.99E^{-47}$	3.95	4.22
	30	$1.08E^{-61}$	5.55	5.77		30	$4.46E^{-47}$	5.36	5.37
4	10	$3.25E^{-42}$	2.59	2.47	8	10	$7.16E^{-38}$	0.74	0.74
	15	$4.48E^{-45}$	4.00	4.22		15	$8.75E^{-44}$	1.04	1.07
	20	$9.18E^{-49}$	5.16	5.22		20	$1.18E^{-45}$	1.58	1.52
	25	$7.19E^{-50}$	5.64	5.92		25	$7.75E^{-48}$	1.78	2.05
	30	$1.89E^{-54}$	7.63	7.92		30	$1.1E^{-52}$	2.40	2.37

hypothesis can be rejected. Hence, one can conclude with 99.9% confidence that the differences in schedule length before and after LS are not due solely to chance.

We also conducted another set of experiments in order to compare the performance of our algorithm with algorithms presented in [7,8]. In these experiments, means and standard dev. of lengths of schedules produced by LS were compared with means and standard dev. of schedules obtained by algorithms introduced in these two papers. We selected a set of four test cases. Three of them were taken from [8]. In these cases, 100 POIs are located in the form of a rectangular grid on an area of size 100×100, and the number of randomly deployed sensors is, respectively, 100, 200, and 300. The last problem was taken from [7]. In this problem, 400 POIs are located in the form of a rectangular grid on an area of size 100×100, and 100 sensors are randomly deployed. In all four cases $cov = 90\%$, a sensing range $r_{\text{sens}} = 20$ and $T_{\text{batt}} = 20$.

Table 4 shows that Algorithm 3 can give results much better than other methods (in cases #1-3 schedule length increased by almost 70%, in case #4 by almost

180%). However, according to Table 2 in the case of results of Algorithm 1, the LS algorithm gives improvement only by about 5%. This would indicate that Algorithm 1 does not give much room for improvement by local search.

Table 4. Mean and standard dev. of lengths of schedules returned by the LS algorithm using proposed perturbation operator (Algorithm 3) and presented in publications (here as the reference values)

No.	LS	Ref. value	No.	LS	Ref. value
case#1	139.43 ± 2.61	83.0 ± 2.23	case#3	419.1 ± 5.25	248.0 ± 2.82
case#2	278.87 ± 4.14	165.0 ± 2.44	case#4	136.4 ± 2.31	49

6 Conclusions

In this paper, we presented a new approach to solving the Maximum Lifetime Coverage Problem. Our method generates an initial schedule using for generation of new slots a procedure inspired by cellular automata based on a graph universe. If a new slot does not satisfy the required coverage constraint, the slot is modified by turning on all available sensors and turning off randomly selected sensors one sensor at a time. Next, the initial schedule is passed to a local search procedure for iterative improvement.

Moreover, a set of benchmarks for experimental evaluation of our algorithm has been proposed. Our experiments with these test cases show that the system is operational longer when many sensors can cover multiple POIs. Moreover, mean lengths of schedules are proportional to the sensor battery capacity. Our local search algorithm, when applied to initial schedules produced using a procedure inspired by cellular automata gives slightly longer schedules. However, this local search algorithm can give results much better than methods from [7,8].

References

1. Bozapalidis, S., Kalampakas, A.: Graph automata. Theor. Comput. Sci **393**(1–3), 147–165 (2008). https://doi.org/10.1016/j.tcs.2007.11.022
2. Cardei, I., Cardei, M.: Energy-efficient connected-coverage in wireless sensor networks. IJSNet **3**(3), 201–210 (2008). https://doi.org/10.1504/IJSNET.2008.018487
3. Erdelj, M., Loscrì, V., Natalizio, E., Razafindralambo, T.: Multiple point of interest discovery and coverage with mobile wireless sensors. Ad Hoc Netw. **11**(8), 2288–2300 (2013). https://doi.org/10.1016/j.adhoc.2013.04.017
4. Halton, J.H.: Algorithm 247: radical-inverse quasi-random point sequence. Commun. ACM **7**(12), 701–702 (1964). https://doi.org/10.1145/355588.365104
5. Tian, D., Georganas, N.D.: A coverage-preserving node scheduling scheme for large wireless sensor networks. In: Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA 2002), pp. 32–41. ACM Press (2002). https://doi.org/10.1145/570738.570744

6. Tretyakova, A., Seredynski, F.: Application of evolutionary algorithms to maximum lifetime coverage problem in wireless sensor networks. In: IPDPS Workshops, pp. 445–453. IEEE (2013). https://doi.org/10.1109/IPDPSW.2013.96
7. Tretyakova, A., Seredynski, F.: Simulated annealing application to maximum lifetime coverage problem in wireless sensor networks. In: Global Conference on Artificial Intelligence, GCAI, vol. 36, pp. 296–311. EasyChair (2015)
8. Tretyakova, A., Seredynski, F., Bouvry, P.: Graph cellular automata approach to the maximum lifetime coverage problem in wireless sensor networks. Simulation **92**(2), 153–164 (2016). https://doi.org/10.1177/0037549715612579
9. Wu, A.Y., Rosenfeld, A.: Cellular graph automata I and II. Inf. Control **42**, 305–353 (1979). https://doi.org/10.1016/S0019-9958(79)90288-2

A Fuzzy Logic Inspired Cellular Automata Based Model for Simulating Crowd Evacuation Processes

Prodromos Gavriilidis[1], Ioannis Gerakakis[1], Ioakeim G. Georgoudas[1](✉), Giuseppe A. Trunfio[2](✉), and Georgios Ch. Sirakoulis[1](✉)

[1] Department of Electrical and Computer Engineering, School of Engineering, Democritus University of Thrace, University Campus, Kimmeria, 67100 Xanthi, Greece
{pgavri,igerak,igeorg,gsirak}@ee.duth.gr

[2] DADU, University of Sassari, Piazza Duomo, 6, 07041 Alghero, SS, Italy
trunfio@uniss.it

Abstract. This work investigates the incorporation of fuzzy logic principles in a cellular automata (CA) based model that simulates crowd dynamics and crowd evacuation processes with the usage of a Mamdani type fuzzy inference system. Major attributes of the model that affect its response, such as orientation, have been deployed as linguistic variables whose values are words rather than numbers. Thus, a basic concept of fuzzy logic is realised. Moreover, fuzzy *if-then* rules constitute the mechanism that deals with fuzzy consequents and fuzzy antecedents. The proposed model also maintains its CA prominent features, thus exploiting parallel activation of transition rules for all cells and efficient use of computational resources. In case of evacuation, the selection of the appropriate path is primarily addressed using the criterion of distance. To further speed up the execution of the Fuzzy CA model the concept of the inherent parallelization was considered through the GPU programming principles. Finally, validation process of the proposed model incorporates comparison of the corresponding fundamental diagram with those from the literature for a building that has been selected for hosting the museum 'CONSTANTIN XENAKIS', in Serres, Greece.

Keywords: Crowd modelling · Cellular automata · Fuzzy logic
Evacuation · Flow-density diagram · Speed-density diagram

1 Introduction

Crowd evacuation is a research area that has been thoroughly investigated by the scientific community. Many researchers from different disciplines have applied various methodologies to approach realistically issues related to the movement of people when massively abandoning an area. The major challenge for all deployed mechanisms is to improve the safety standards of evacuation processes. In such studies, a large number of people are involved and the interactions between them can be hardly described by conventional equations because of the psychological factors that influence their behavior. Moreover, the layout of the facilities has significant impact on the evolution of the evacuation. Thus, evacuation dynamics incorporates nonlinear characteristics and is very complex [1].

It is of high interest that modeling approaches of crowd movement can achieve an acceptable level of realism, and can be efficiently validated with empirical data in order to provide robust results. According to a recent review about evacuation models [2], empirical research often focuses on the relationship between walking speed and density as well as the relationship between flow, people and density. These relationships are called the 'fundamental diagram', because of their importance in determining the optimal dimensions of pedestrian facilities [3]. Often, the main modeling issue is the ability to successfully handle congestion, in order to prevent unpleasant circumstances, such as stampede, trampling and casualties [4–6]. Concurrently, the applications of fuzzy logic have increased significantly. According to [7], fuzzy logic is a theory that tries to broaden the limits of a set of acceptable values by defining not crisp boundaries in which membership is a matter of degree. Furthermore, by introducing the notion of a *linguistic variable*, whose values are words rather than numbers, fuzzy logic could be considered as a methodology for computing with words rather than numbers that try to lower the cost of solution at the expense of decreased but acceptable precision.

Literature review shows that the combination of cellular automata (CA) models with fuzzy logic is quite effective. For instance, Bisgambiglia et al. [8] presented a method that incorporates fuzzy inference systems in activity-based CA simulations. Betel and Flocchini [9] investigated the relationship between fuzzy and Boolean CA, whereas Cattaneo et al. [10] and Adamatzky [11] developed CA models where the local transition rule is described by a fuzzy function. Moreover, Chaia et al. presented a safety evaluation of driver cognitive failures and driving errors on right-turn filtering movement based on fuzzy CA [12]. Finally, Al-Ahmadi et al. developed a fuzzy CA model of urban dynamics [13].

The approach proposed in this paper combines fuzzy logic and CA to build a reliable model that simulates crowd evacuation. The motivation for incorporating fuzzy logic stems from the fact that it enables computing with words, which 'are inherently less precise than numbers, but their use is closer to human intuition' [7]. For instance, it is the orientation of the direction (e.g. 'north', 'south-east', etc.) that plays a dominant role in the movement of an individual rather than an exact angle measured in degrees. The same can be adopted for the state of a cell, which can be adequately characterised by words, such as 'free', 'occupied' or 'obstacle' for the needs of a model that mainly targets to quick response and tries to lower the cost of solution. Specifically, a Mamdani type fuzzy inference system (FIS) has been developed using the MATLAB Fuzzy Logic Toolbox^{TM}. Major features of the model, such as state of a CA cell and direction have been represented as linguistic variables. Additionally, fuzzy *if-then* rules are properly constructed based on the descriptions of the input and output variables. The structure of the model follows the CA principles, that is, it focuses on optimized utilisation of computational resources and decreased complexity simultaneously maintaining accurate modeling of the evacuation process with microscopic and macroscopic characteristics. Furthermore, and to further speed up the execution of the introduced Fuzzy CA (FCA) model, the concept of the inherent parallelization was discussed in the view of the GPU programming principles in order to achieve a fast execution of the FCA. The simulation process is characterized by fundamental features of crowd evacuation, such as transition from uncoordinated to coordinated movement due to common purpose, arching in front of exits, herding behaviour [1]. Finally, the evaluation of the model process is addressed by the comparison of the flow-density and speed-density response of the model with corresponding representations from literature for a building that has been decided to host the museum 'CONSTANTIN XENAKIS', in Serres, Greece.

In the following, the theoretical principles of the proposed evacuation model are presented (Sect. 2). In Sect. 3, the GPU implementation of the proposed model is presented, whereas in Sect. 4, simulation scenarios and corresponding results are presented and discussed. Finally, conclusions are drawn in Sect. 5.

2 Model Description

The model is CA-based: the space is a two-dimensional grid of identical cells, which is homogeneous and isotropic. Each cell may be either free or occupied by an individual or an obstacle. The state of each cell, which is represented by $C^t_{i,j}$, with i and j being the coordinates of the cell and t the evolution time, is described as:

$$C^t_{i,j} = \{o, id\} \tag{1}$$

with o representing whether the cell is free or occupied and id representing the class of the cell, i.e. whether it is a person or an obstacle. The neighbourhood consists of the eight closest neighbour cells (Moore neighbourhood), thus allowing each person to move towards eight directions. To update the state of a given cell, it is required the knowledge of the target exit $C^t_{i_{exit},j_{exit}}$ as well as of the status of all neighbouring cells. Therefore, the evolution rule is defined as:

$$C^{t+1}_{i,j} = R\left(C^t_{i,j}, C^t_{i\pm1,j,}, C^t_{i,j\pm1}, C^t_{i-1,j+1}, C^t_{i+1,j-1}, C^t_{i+1,j+1}, C^t_{i_{exit},j_{exit}}\right) \tag{2}$$

The transition function R of the FCA that defines the position of each person during each time step is realized by a FIS, particularly a Mamdani type, that is built using the MATLAB Fuzzy Logic Toolbox^{TM} tool [14,15]. Both the state of each cell of the Moore neighbourhood and the orientation of the person relative to the closest exit are considered as inputs. The orientation is defined as:

$$\text{difference}_x = x_{exit} - x_{person}, \quad \text{difference}_y = y_{exit} - y_{person} \tag{3}$$

that is, as the position of the an individual relative to its closest exit. It should be noted that notation x represents the vertical axis, whereas y the horizontal one. The difference between the ordinate value of an exit (that corresponds to the outcome of a selection process described below) and that of the person designates the north-south orientation; when the difference is negative then the exit is located northern in regard to the person (denoted as ‘exit-x is North’ at the corresponding *if-then* rule), whereas when the difference is positive then the exit is located southern (‘exit-x is South’). Accordingly, the difference between the abscissa values of the exit and that of the person designates the east-west orientation; when it is negative then the exit lays western in regard to the person (‘exit-y is West’), whereas when it is positive then the exit lays eastern (‘exit-y is East’). The combination of the upper two parts defines the overall orientation. The implementation of the rule can be represented by a triangular membership function, as shown in Fig. 1. This is just the collection of three points that form a triangle. Thus, when the difference is negative, the value −5 is assigned to the membership function (‘North/West’ case), when the difference is positive the value 5 is assigned (‘South/East’ case), whereas value 0 is assigned when the individual shares the same ordinate (abscissa) with the exit (‘Non’ case).

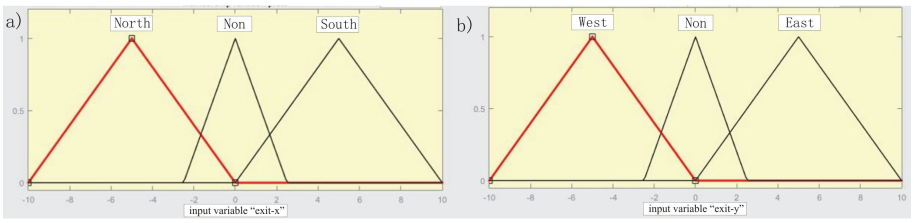


Fig. 1. The graphical representations of the membership functions that define (a) north-south orientation, and (b) east-west orientation regarding the direction of motion of an individual towards the chosen exit. ‘Non’ function represents the case of zero difference, i.e. when the individual shares the same ordinate (abscissa) with the exit.

In order all possible cell states, i.e. free, occupied and obstacle, to be represented for each part of all eight possible directions (N, NW, W, SW, S, SE, E, NE), a triangular membership function is utilized (Fig. 2). Value 0 is assigned when the corresponding state of the cell is free, whereas value 2 represents occupation by an individual and value 1 denotes the existence of an obstacle. Thus, the triangular function is adequate for representing the corresponding rule.

The list of rules that define the behaviour of the system is defined by a set of forty-eight (48) *if-then* rule statements that cover all possible cases of movement. These rules are used to define the conditional statements in terms of fuzzy logic. A sample of the whole set of rules is provided below:

1. If exit - x is North and exit - y is West and NW - State is Unoccupied then Output is NW
2. If exit - x is North and exit - y is West and NW - State is Occupied and W - State is Unoccupied then Output is W
3. If exit - x is North and exit - y is West and NW - State is Occupied and W - State is Occupied and N - State is Unoccupied then Output is N.

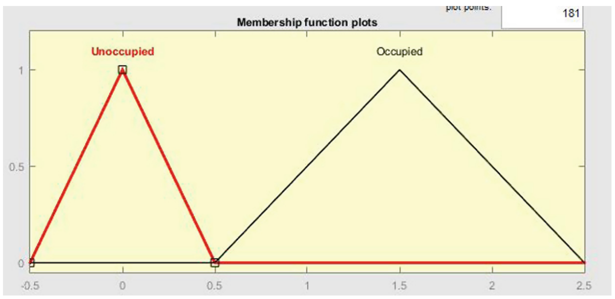


Fig. 2. The membership function that represents the state of each of the cells of the Moore neighbourhood.

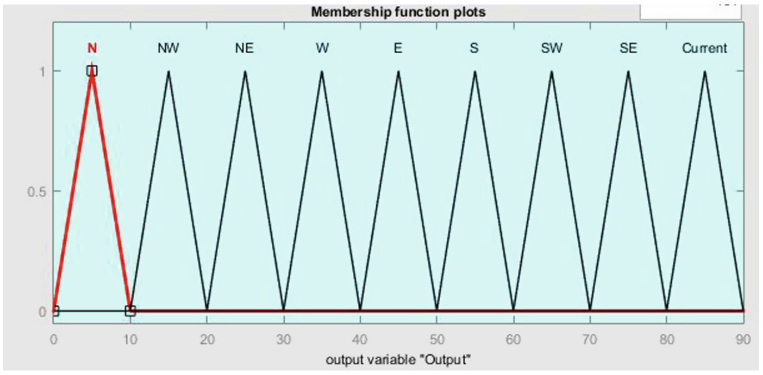


Fig. 3. The graphical representation of the response of the fuzzy inference system.

The corresponding response of the model is depicted in Fig. 3. The selection of the most appropriate exit for an individual to move during the upcoming time step is realized by utilizing the criterion of distance. In particular, the closest exit is computed according to the minimum Euclidean distance of the cell at (i, j) from all available exits:

$$R = \sqrt{(i - i_{exit})^2 - (j - j_{exit})^2} \tag{4}$$

Initially, the maximum distance from each exit is calculated according to the layout of the site. Maximum distance is algorithmically represented by a vector, named *max_dist*, the size of which is equal to the number of exits within the site. Then the distance of each individual is normalised by dividing the elements of *max_dist* by its greatest element. In case that an element of the normalized *max_dist* is greater than (or equal to) 2/3 and smaller (or equal to) than unity (1), then the corresponding distance is characterised 'long' and it is assigned value 2 in the corresponding element of a look-up table. Provided that the ratio is greater than (or equal to) 1/3 and smaller than 2/3 then the distance is characterised as moderate and is assigned value 1, whereas when the ratio is smaller than 1/3 then the distance is characterised as short and the corresponding value in the look-up table is equal to 0. The distance criterion is initially applied when the members of the crowd start to move aiming at the initialization of the look-up table and the characterization of each of the exits for every single person. Afterwards, the function *eval_pos* is activated, whose argument is a vector that carries the characterisations of the distances to all the exits (short, medium, long). Function *eval_pos* calculates a vector that includes all exits that share the similar characterisation of *short*, regarding to the minimum distance. In case that the output vector contains a single element, then that particular exit is indicated as the closest one, otherwise one of the exits that share the same minimum distance from the individual is chosen randomly.

Furthermore, the model has incorporated the auto-defined obstacle avoidance method, which is an automated process that enables people to overcome obstacles based on the effect of a virtual field generated near obstacles. The method is thoroughly described in [16]. In case that more than one person tries to reach the same cell then one of them is selected randomly. Finally, individuals are considered similar in terms of decision-making process, but they can be attributed different characteristics regarding speed. The model enables the assignment of dissimilar moving steps to groups of pedestrians, thus enhancing its reliability.

3 GPU Implementation

To enable fast simulations, we devised an implementation of the proposed model exploiting Graphics Processing Units (GPU) as parallel computing devices. In particular, we adopted a hybrid CPU-GPU approach based on the NVIDIA GPGPU platform with the well-known CUDA language. In the latter, the parallel computation is obtained by activating at each CA step a GPU thread associated to every CA cell occupied by a person (i.e., to every *active* cell). Such threads correspond to device (i.e. GPU) functions in C language, which are called *kernels*. When a kernel is issued by the CPU, a number of threads (i.e., one for each active cell, in our case) execute its code in parallel on different data. In the above CA simulation, the kernels operate on two distinct memory regions, representing the current and next states for the CA cells, respectively, where the state refer to the cell's occupancy by the simulated moving crowd, presence of exits and obstacles. The simulation begins by transferring from the host memory (i.e. that directly accessed by the CPU) to the GPU global memory the initial CA states, stored as arrays in order to favour faster coalesced accesses. Besides the state of the cells, we used some additional auxiliary arrays in the GPU global memory for storing the neighbourhood structure and the model parameters. During each CA step, the kernel execute the CA transition function described in Sect. 2, operating on the basis of the values from the current CA and exploiting some auxiliary functions (e.g. implementing the fuzzy membership functions); after, it writes the new state value into the appropriate elements of the next CA. At the end of each CA step, a *device-to-device* memory copy operation is used to re-initialise the current CA values with the next values. When the CA state is required by the CPU during the simulation (e.g. for depicting a graphical output), a *device-to-host* memory copy is carried out.

To implement the CA transition function, we developed a single kernel, which is executed on a dynamic grid of threads to improve the scalability with respect to both the number of involved pedestrians and the size of the CA. This was accomplished by keeping track of moving people in GPU global memory by means of an array B of integers. Each element of B encodes a pair $\langle id, c\rangle$, where id identifies a specific person contained in the cell c. Before the beginning of the simulation, B is initialized by the CPU through a *host-to-device* memory copy. Subsequently, *movePersonKernel* is executed on a dynamic one-dimensional grid of threads corresponding to the elements of the array B, which is updated by the kernel itself. More in details, the kernel operates on two global memory areas: B, which is the input container from where *movePersonKernel* takes the cells to process the current iteration; and B^*, which includes the active cells (i.e., those containing people) for the next iteration. At the end of each CA step, the pointers to B and B^* are simply exchanged. The first step of *movePersonKernel* consists of retrieving the pair $\langle id, c\rangle$ corresponding to the current thread. Then, a movement of the person id is considered only if a specific counter, initialized according to the person speed, is zero. Otherwise, the counter is simply decremented and B^* is updated with the insertion of $\langle id, c\rangle$ (i.e., the person id remains in the same cell). If the speed counter is zero and, according to the fuzzy rules outlined above, the person id can move to a neighbouring cell c^* different from an exit, then B^* is updated with the insertion of the pair $\langle id, c^*\rangle$. At the end of each iteration (i.e. when the kernel returns), the size of B^* is retrieved from global memory. If B^* is empty, the simulation ends because all people are outside the simulation area (i.e., they reached an exit). It is worth noting that to minimize expensive synchronizations, the new container B^* is cooperatively built by the involved threads by managing a hierarchy of smaller containers stored in shared memory, as suggested in [17,18]. In particular, thread-level arrays B_t are used, with the maximum size of a neighbourhood, in which the insertion can be executed by the owner thread without the need of synchronizations. Then, in each block, the thread-level arrays are copied into a single block-level array B_b by using a parallel prefix-scan approach. The latter can be implemented in a very efficient way in CUDA, only requiring two thread synchronizations and no atomic operations. Instead, a single *atomicAdd* is used by the first thread of each block for obtaining the offset required to copy its block-level array B_b into the final container B. A particular case that *movePersonKernel* takes into account is the conflict that may happen when more than one individuals are trying to reach the same cell. Also, this case is handled though a single *atomicMax* operation on the array containing the next states of the CA.

4 Simulation Results

In this study, the venue that is selected for all simulation purposes is the building that has been decided to host the museum 'CONSTANTIN XENAKIS'. It is located in Serres, a city of the administrative region of Central Macedonia, in Northern Greece. The building has not been redecorated yet, thus any useful conclusion regarding its standards of safety could be taken into consideration. The floor plan of the building can be found on the following site: http://serreonpoliteia.com/?page_id=10.

The following considerations were applied concerning the aforementioned place used for simulation purposed. As outputs of the under-study building, two exits as well as its main entrance are considered. It is also regarded that the building windows cannot be used for evacuation purposes. Both the length and the width of each cell within the CA grid is assumed to be equal with 0.3 m. Within the building, though, there are walls which are less than 0.3 m thick. Therefore, every part of the construction that is less than 0.3 m thick is supposed 0.3 m thick. Moreover, for constructions that are more than 0.3 m thick their corresponding dimension is calculated by dividing this dimension with 0.3 and applying rounding rules. Thus, for instance, an exit that is 4.75 m wide, it will be represented by $(4.75\,\text{m})/(0.3\,\text{m}) = 15.8333 \approx 16$ cells. Consequently, the maximum width of the building is calculated equal to 76 cells and its maximum length is equal to 148 cells. According to the simulation scenario 400 individuals are randomly assigned to positions within the museum (Fig. 4).

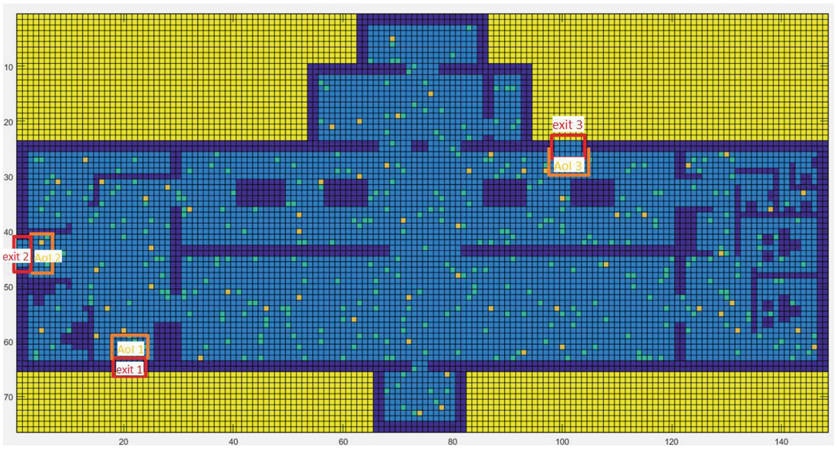


Fig. 4. Simulation scenario: random initialisation, main exits and corresponding areas of interest (AoI).

The model is evaluated by comparing the fundamental characteristics and graphical representations generated by the simulated three scenarios with the corresponding diagrams from literature. Particularly, the overall density of the crowd is measured by counting all individuals in the area of interest and then dividing by the area of this region. The total flow is calculated by dividing the total number of individuals who go through the exits per simulation step with the length of this intersection. The flow of people results from experimental relations of speed of people with the density of people. Taking as an assumption that people move smoothly, the flow per meter of width is given by $Q(\rho) = \rho V(\rho)$, where Q represents the flow per meter of width and ρ is the density of individuals. Each person covers an area equal to that of single cell, i.e. $0.4 \times 0.4 = 0.16\,\mathrm{m}^2$. For each step, the number of people within the area of interest (AoI) is counted and the value of the density is calculated from the relationship:

$$\rho = \frac{\text{num. of people in RoI}}{\text{area of AoI}} \tag{5}$$

There are three AoI, each corresponding to an exit (Fig. 4). The total AoI area consists of $5 \times 8 = 40$ cells, i.e. $6.4\,\mathrm{m}^2$. Regarding the simulation scenario and for the AoI of the first exit (AoI 1) the corresponding curve of flow vs. density is depicted in Fig. 5(a). In Fig. 5(b), the corresponding flow-density curves from literature are depicted [19]. As can be seen, the corresponding curves present common behavioural attributes, both qualitatively and quantitatively, thus enhancing the validity of the proposed approach.

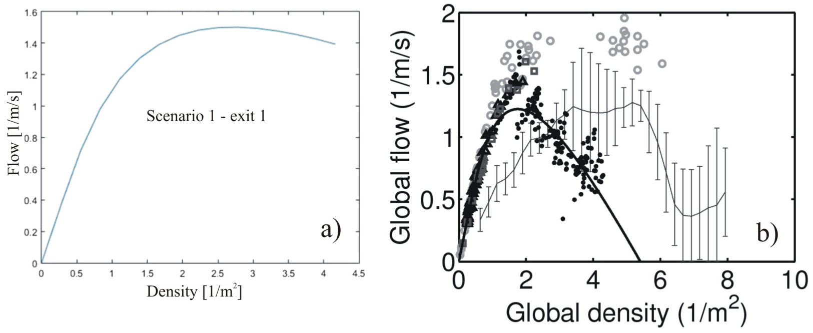


Fig. 5. (a) Simulation scenario. AoI of exit 1. Flow vs. density; (b) Flow-density results from literature.

5 Conclusions

Results prove that fuzzy type logic can find application in real-world evacuation conditions and in particular when describing crowd dynamics. More specifically, they respond to expected behaviours. According to the directional selection criterion, persons have full sense of orientation towards the desired output and perform movements as in real-world conditions. Furthermore, the implementation of the fuzzy CA rules for intuitive exit selection has been achieved, taking into account the distance of a person from the exit, so that they correspond to real conditions. The model need further to be validated with real data. Thus, it could be better calibrated, and it could be parameterised more efficiently.

References

1. Helbing, D., Johansson, A.: Pedestrian, crowd and evacuation dynamics. In: Meyers, R.A. (ed.) Encyclopedia of Complexity and System Science, vol. 16, pp. 6476–6495. Springer, New York (2010). https://doi.org/10.1007/978-0-387-30440-3_382
2. Vermuyten, H., Beliën, J., De Boeck, L., Reniers, G., Wauters, T.: A review of optimisation models for pedestrian evacuation and design problems. Saf. Sci. **87**, 167–178 (2016)
3. Schadschneider, A., Seyfried, A.: Empirical results for pedestrian dynamics and their implications for cellular automata models. In: Pedestrian Behavior - Models, Data Collection and Applications, pp. 27–44 (2009)
4. Georgoudas, I.G., Sirakoulis, G.C., Andreadis, I.T.: An anticipative crowd management system preventing clogging in exits during pedestrian evacuation processes. IEEE Syst. J. **5**(1), 129–141 (2010)
5. Vermuyten, H., Lemmens, S., Marques, I., Beliën, J.: Developing compact course timetables with optimized student flows. Eur. J. Oper. Res. **251**(2), 651–661 (2016)
6. Zarboutis, N., Marmaras, N.: Design of formative evacuation plans using agent-based simulation. Saf. Sci. **45**(9), 920–940 (2007)
7. https://www.mathworks.com/help/fuzzy/what-is-fuzzy-logic.html
8. Bisgambiglia, P.A., Innocenti, E., Gonsolin, P.R.: A new way to use fuzzy inference systems in activity-based cellular modeling simulations. In: IEEE International Conference on Fuzzy Systems (2017)
9. Betel, H., Flocchini, P.: On the relationship between fuzzy and Boolean cellular automata. Theor. Comput. Sci. **412**(8–10), 703–713 (2011)
10. Cattaneo, G., Flocchini, P., Mauri, G., Vogliotti, C.Q., Santoro, N.: Cellular automata in fuzzy backgrounds. Phys. D: Nonlinear Phenom. **105**(1–3), 105–120 (1997)
11. Adamatzky, A.I.: Hierarchy of fuzzy cellular automata. Fuzzy Sets Syst. **62**(2), 167–174 (1994)
12. Chaia, C., Wong, Y.D., Wang, X.: Safety evaluation of driver cognitive failures and driving errors on right-turn filtering movement at signalized road intersections based on Fuzzy Cellular Automata (FCA) model. Accid. Anal. Prev. **104**, 156–164 (2017)
13. Al-Ahmadi, K., See, L., Heppenstall, A., Hogg, J.: Calibration of a fuzzy cellular automata model of urban dynamics in Saudi Arabia. Ecol. Complex. **6**(2), 80–101 (2009)
14. Zadeh, L.A.: Fuzzy logic. Computer **1**(4), 83–93 (1988)
15. Mamdani, E.H.: Applications of fuzzy logic to approximate reasoning using linguistic synthesis. IEEE Trans. Comput. **26**(12), 1182–1191 (1977)
16. Georgoudas, I.G., Koltsidas, G., Sirakoulis, G.C., Andreadis, I.T.: A cellular automaton model for crowd evacuation and its auto-defined obstacle avoidance attribute. In: Bandini, S., Manzoni, S., Umeo, H., Vizzari, G. (eds.) ACRI 2010. LNCS, vol. 6350, pp. 455–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15979-4_48
17. Trunfio, G.A., Sirakoulis, G.C.: Computing multiple accumulated cost surfaces with graphics processing units. In: 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 694–701. IEEE (2016)
18. Teodoro, G., Pan, T., Kurc, T.M., Kong, J., Cooper, L.A.D., Saltz, J.H.: Efficient irregular wavefront propagation algorithms on hybrid CPU-GPU machines. Parallel Comput. **39**(4–5), 189–211 (2013)
19. Johansson, A., Helbing, D., A-Abideen, H.Z., Al-Bosta, S.: From crowd dynamics to crowd safety: a video-based analysis. Adv. Complex Syst. **11**(4), 497–527 (2008)

Nondeterministic Cellular Automaton for Modelling Urban Traffic with Self-organizing Control

Jacek Szklarski(✉)

Institute of Fundamental Technological Research, Polish Academy of Sciences, Warsaw, Poland
jszklar@ippt.pan.pl

Abstract. Controlling flow in networks by means of decentralized strategies have gained a lot of attention in recent years. Typical advantages of such approach – efficiency, scalability, versatility, fault tolerance – make it an interesting alternative to more traditional, global optimization. In the paper it is shown how the continuous, macroscopic, self-organizing control proposed by Lämmer and Helbing [10] can be implemented in the discrete, nondeterministic cellular automaton (CA) model of urban traffic. Using various examples, it is demonstrated that the decentralized approach outperforms the best nonresponsive solution based on fixed cycles. In order to analyse relatively large parameter space, an HPC cluster has been used to run multiple versions of a serial CA simulator. The presented model can serve as a test bed for testing other optimization methods and vehicle routing algorithms realized with the use of CA.

Keywords: Urban traffic · Nondeterministic cellular automaton
Self-organizing control · Decentralized control

1 Introduction

In communication networks, controlling strategies have a profound impact on the overall performance [1,2]. Particularly, optimization in traffic networks is especially important due to a tremendous affection it has on peoples everyday life. In order to address this issue, one has to apply some kind of traffic model and then propose optimization procedures.

There exists a large number of traffic models which generally fall in one of these classes: microscopic where vehicles are represented as particles (e.g., follow-the-leader models); cellular automata (CA) where a vehicle's state corresponds to a cell's state; based on some master equation (e.g., mean field models); macroscopic continuous models (e.g., kinetic waves), and more [3,4]. Obviously, a good traffic model has to reproduce all its properties which are observed in the real world.

Regarding optimization, one of the most common ways to do it is to choose some pre-calculated schemes, which are aimed at synchronizing green times along main arterials. In principle such methods force the traffic flow to comply with previously designed patterns in order to minimize travel times. However, since traffic demand varies, there is a need for some responsiveness to the current traffic state. In order to improve efficiency of control methods, it is necessary to implement on-line optimization techniques based on real time traffic intensity observations. This can be done in a centralized system, in which there exists a central unit possessing all information concerning current state of the network. Obviously all the measuring devices must be somehow connected to a central unit (which is expensive). Moreover, optimizing globally may be NP-hard [5,6] making it even more difficult to react in real-time. Consequently, there is a recent trend towards decentralized and self-organizing optimization techniques [7–12] which instantly and locally respond to the current traffic state (known, e.g., from vehicle detectors mounted at some distance before an intersection). Naturally, it is desired that such locally defined mechanisms will produce near-optimal global solution. One of the most efficient and versatile decentralized self-controlled strategies has been proposed by Lämmer and Helbing (LH, [10]). The authors have defined the scheme with the use of a model similar to kinematic waves approach [13].

In this paper it is shown in details how this LH controlling mechanism can be implemented in a network of cellular automata with the use of nondeterministic Nagel-Schreckenberg (NS) model [14] (i.e., with the randomization parameter $P > 0$). The efficiency of this solution is analysed by considering three scenarios in regular lattice networks. It is shown that the self-controlled intersections converge to the best possible cycles and phase-shifts for periodic networks, and that they outperform constant cycle (CC) solutions if vehicles are able to randomly change moving directions (e.g., they turn). Lastly, stochastic boundary conditions are applied and it is shown that the LH strategy clears the network significantly more efficient if additional perturbations are allowed.

CA traffic models can be relatively easily parallelized, making it a very useful tool for efficient prediction, analysis and optimization. Moreover, they can be implemented withe use of FPGA [15] or GPGPU [16] further increasing efficiency. The results presented here are calculated with a serial program designed to advance a network of CA's. However, since it was desired to obtain a full study of parameter space, these programs have been run in parallel in an HPC cluster for various initial conditions and control variables. Therefore, meaningful statistics could be calculated in a reasonable time (couple of hours).

2 The Model

The city traffic model is essentially similar to the work presented by Chowdhury and Schadschneider [17], and Brockfeld et al. [18]. There are N^2 nodes (intersections) $I_{i,j}$, $i = 1, \ldots, N$, $j = 1, \ldots, N$, which form a square lattice. Each node has two incoming links (one-lane and one-way streets): one from west-side and one from south-side, and two leaving links: one towards east-side and one towards north-side, Fig. 1. Nodes make a decision which traffic stream should be served. In addition to the previous work, [17,18], here a setup time $\tau = 2$ is specified (the amount of time for which the both streams have "red light" when switching from one stream to the other).

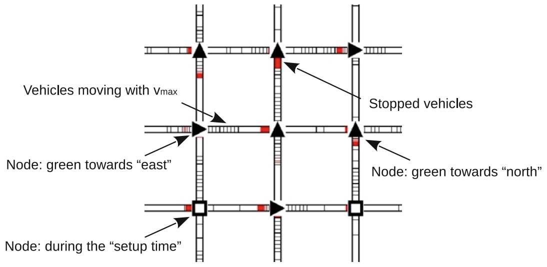

Vehicles moving with vmax
Stopped vehicles
Node: green towards "east"
Node: green towards "north"
Node: during the "setup time"

Fig. 1. A sample view of part of a grid-like network with link length $D = 100$.

The links in a network represent a single-lane street which is a one-dimensional cellular automaton with D cells ($D = 100$ is used throughout this paper). An occupied cell n symbolizes a single vehicle, and a discrete, integer variable v_n corresponds to its velocity. Let the maximum allowed velocity be $v_{\max}$ (here $v_{\max} = 5$) and the distance to the next vehicle is d_n, the distance to the next intersection is s_n. In the classical model by [14] with urban-like modifications [18], which take into account traffic light, the four consecutive steps for parallel updating at discrete time steps can be written as:

1. Acceleration: $v_n \leftarrow \min(v_n + 1, v_{\max})$,
2. Breaking:
   - Traffic light at the intersection to which the link is connected is "red" or the intersection is in setup time: $v_n \leftarrow \min(v_n, d_n - 1, s_n - 1)$
   - Traffic light is "green". If two cells behind the intersection are occupied: $v_n \leftarrow \min(v_n, d_n - 1, s_n - 1)$, otherwise $v_n \leftarrow \min(v_n, d_n - 1)$,
3. Randomization with the probability P: $v_n \leftarrow \max(v_n - 1,\ 0)$,
4. Vehicle movement: $x_n \leftarrow x_n + v_n$.

The initial density $\rho = m/D$ is the number of vehicles m divided by the total number of cells in the link, D. For given $v_{\max}$ there exists a maximum density for which all the vehicles can move freely with $v_{\max}$. In the deterministic limit $P = 0$, $\rho_{\max} = (v_{\max} + 1)^{-1}$ since for $\rho > \rho_{\max}$ there exists at least one vehicle which has less than $v_{\max}$ occupied cells in front of it, and therefore it is forced to slow down ($v_{\max} = 5$ give $\rho_{\max} = 0.16(6)$ for $P = 0$, and $\rho_{\max} \approx 0.15$ for $P = 0.1$). With each link there is associated the mean flux J' (number of vehicles leaving the link per unit time), for the entire network $\bar{J} = N^{-2} \sum_1^{N^2} J'_i$ is just the average of mean flux J'_i for each link i. Note that the assumed $v_{\max} = 5$ should be equivalent to about 50 km/h in a real city traffic flow, assuming that a single cell corresponds to a real size of 7.5 m (a vehicle length with safety distance in front and behind it), each step is about 2 s in real time.

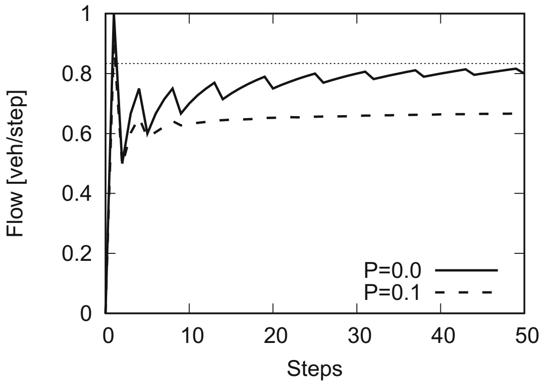


Fig. 2. Flux as a function of steps after opening an intersection for a CA fully filled with vehicles ($P = 0$ is exact, $P = 0.1$ is the average for 10^6 simulations). The horizontal line represents the exact limiting flux for $P = 0$, $J_{\max} = v_{\max}/(v_{\max} + 1) = 5/6$.

Boundary conditions can be either periodic or stochastic. If periodic boundary conditions are assumed, each vehicle leaving the network at east/north side will be placed at the beginning of corresponding links at west/south side. As the stochastic BCs, the so called expanded stochastic boundaries are applied [19]. These are formed by placing an additional CA with length equal to $v_{\max}$ as a source of vehicles. Vehicles appear at the beginning of such small CA with given probability P_{ins} and accelerate according to the CA rules. Such treatment is a proper insertion strategy which makes sure that all possible system states can be obtained. Here, in networks with stochastic sources, the right-most and the top-most nodes act as simple sinks, i.e. nothing prevents a vehicle from leaving the system.

2.1 Periodic Switching

The simplest possible strategy for control is to use cycle-based switching. For each node the cycle is: *(a)* "red light" for $(T - 2\tau)/2$ steps; *(b)* setup time for τ steps; *(c)* "green light" for $(T - 2\tau)/2$ steps; and *(d)* setup time for τ, giving T steps in total. Additionally, there can be phase shifts $T^{\phi}_{(i,j)}$ for different nodes in network. This means that the first step of the cycle for $N_{(i,j)}$ is realized at the time step $t + T^{\phi}_{(i,j)}$. It is easy to show that for unidirectional networks one can form "green waves" along a single direction by selecting the phase shifts as $T^{\phi}_{(i,j)} = (i + j - 2)T_{\text{delay}} \bmod (2T + 2\tau)$, $T_{\text{delay}} = D/v_{\max}$.

2.2 Self-controlling Strategy

As the responsive self-organizing controller a CA version of the LH strategy [10] is implemented. Below only brief summary of the most important principles is presented, see the original paper for detailed formulation and related proofs (the symbols used here are the same as in the cited work).

Let σ denote the stream which get "green light",

$$\sigma = \begin{cases} \text{head } \Omega & \text{if } \Omega \neq \emptyset \\ \arg\ \max_i \pi_i & \text{otherwise,} \end{cases} \tag{1}$$

where Ω is an ordered set containing stream indices π_i is a priority index for the corresponding stream i (the regular lattice networks have $i = 0$ or $i = 1$). The stabilization strategy assures that each stream i will be placed into the queue Ω at least once in $T_{\max}$ and, on average, once in T_{avg}. The priority index for stream i, provided that currently served stream is σ, is defined as

$$\pi_i = \frac{\hat{n}_i}{\tau^{\text{pen}}_{i,\sigma} + \tau + \hat{g}_i}, \tag{2}$$

where $\hat{n}_i$ is the number of vehicles expected to be served in time $\tau + \hat{g}_i$ for the stream i, τ is the remaining setup time, $\hat{g}_i$ is time required to clear existing queue at the intersection and all vehicles arriving just after clearing, provided that they arrive with the maximum flow rate (i.e., as a platoon traveling with $v_{\max}$), $\tau^{\text{pen}}_{i,\sigma}$ is the additional penalty term for switching from stream σ to i.

Originally, the authors have formulated the strategy using continuous equations based on kinematic waves approach [13]. Implementing it in a CA is not a straightforward task, especially if a nondeterministic NS model is considered, $P > 0$. It has been done in previous work [12], however, here calculating predictive variables is improved and the more realistic $P > 0$ is implemented. Note that non-zero randomization, $P > 0$, is of fundamental importance for the NS model. It makes it possible to reproduce such phenomena as spontaneous jam formation and destroys any artificial metastable states.

The difficulty for implementing $P > 0$ comes from the fact, that in order to calculate the priority index (2), one has to find variables characterizing the state of a crossing node at the current time step and also in the future. For each node, it is necessary to calculate the anticipated amount of the green time $\hat{g}_i$ which is the largest possible solution of

$$N^{\text{dep}}_i(t) + \hat{g}_i(t)Q^{\max}_i = N^{\exp}_i(t + \tau_i(t) + \hat{g}_i(t)), \tag{3}$$

where $N^{\text{dep}}_i(t)$ denotes the number of vehicles which have departed from the crossing, $N^{\exp}_i(t)$ is the number of vehicles which are expected to arrive at the node by the time t, $\tau_i(t)$ i the remaining setup time, $Q^{\max}_i$ is the saturation flow rate. The number of vehicles expected to leave the intersection is $\hat{n}_i(t) = \hat{g}_i(t)Q^{\max}_i$.

In the discussed CA model, it is trivial to keep track of N^{dep}_i: for each intersection one has to count the number of vehicles which have left the node. In order to find the number of vehicles which will approach the node in the following steps $t + \Delta t$ (t being the current step), $N^{\exp}_i(t + \Delta t)$, a temporary CA is created, which consists of a link connecting to the node and a link which leaves this node. Then this temporary automata is advanced for Δt steps according to the NS rules. Joining the two links is necessary in order to take into account any spill-back effects arising when there is some congestion immediately after the intersection. This procedure may be a bit time consuming but it can be efficiently implemented using appropriate caching mechanisms.

Note that this method of calculating N_i^{exp} will inevitably lead to inefficiency of the controlling method if $P > 0$. The reason for this is that advancing the temporary CA may give different value of N_i^{exp} then the "real" value obtained when advancing the entire CA system. This is desirable since in any realistic traffic model, there will be some velocity fluctuations making it impossible to exactly predict the value of $N_i^{\text{exp}}(t + \Delta t)$.

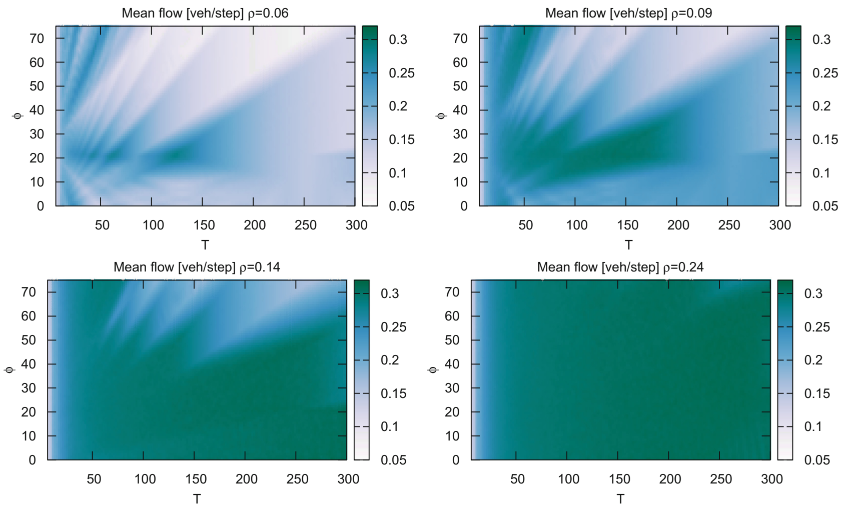


Fig. 3. A map of the mean flow $\bar{J}$ in the regular periodic network as a function of periods T and phase-shifts ϕ for four different densities for the fixed cycle controlling. $N = 6, D = 100, P = 0.1$.

Additionally, there is an important difference in defining the maximum flow rate Q_i^{max} in the continuous approach and the one using a cellular automata. In the former it can be assumed as a constant value, whereas in the latter it depends on time. Consider an infinitely long CA fully filled with vehicles and connected to an intersection with "red light". Assuming that at the moment $t = 0$, the light will turn green, vehicles will leave the intersection at the flow rate J which is presented in Fig. 2. It can be shown that in the deterministic limit $P = 0$, the limiting maximum flux is $J_{\text{max}} = v_{\text{max}}/(v_{\text{max}}+1)$ (p. 240 in [3]). For non-deterministic models, $0 < P < 1$, there is no analytic solution for the limiting J_{max}. However, in order to properly implement the LH mechanism, one has to use $Q_i^{\text{max}}(t_g)$ which depends on the time t_g which denotes for how many steps the considering link has been granted "green light". In any case considered here, $Q_i^{\text{max}}(t_g)$ has been precalculated: averaged over 10^6 stop-and-go CA simulations and tabularized in order to be useful for finding $\hat{g}_i$.

Finally, if there is more than one CA which belong to the same stream i (multiple lanes, bidirectional networks), the corresponding values of N_i^{exp}, Q_i^{max}, etc., are simply summed up for all the CA and a single value of π_i is calculated.

3 The Results

The correctness of implementation end efficiency of the LH strategy has been validated using three various scenarios: periodic network; periodic network with the possibility of vehicle turning; a bidirectional network with stochastic BCs and random intersection blocking.

3.1 Periodic Network $N = 6$

The dynamics of fixed cycle based switching for periodic networks with $P = 0$, has been discussed in detail in [18]. Here it is shown how the mean flow $J(\rho, T)$ depends on ρ, T and T^{ϕ} for wide range of relevant parameters for $N = 6$ and the non-deterministic $P = 0.1$. All the results are averaged by performing 10^5 steps for 10 different initial conditions.

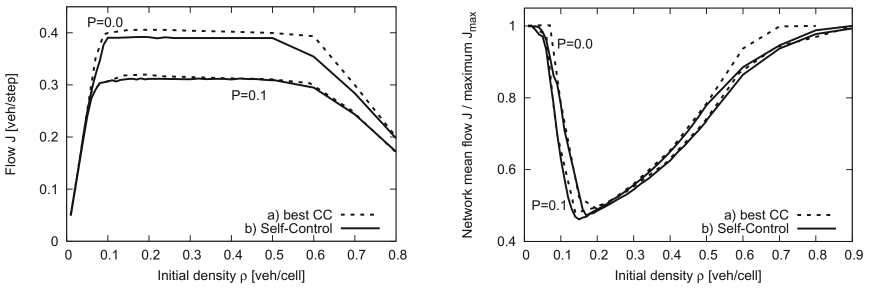


Fig. 4. Left: mean flow $J_{\mathrm{best}}^{\mathrm{CC}}$ in the $N = 6$ network for CC and J^{SC} for SC. Right: the same $\bar{J}$ but normalized with the maximum flux J_{max} taken from the fundamental diagram for $P = 0$ and $P = 0.1$ (precalculated and interpolated).

Figure 3 displays how the mean flow $\bar{J}(T, \phi)$ depends on the fixed cycle length T and phase-shifts ϕ for four different densities. Naturally, this CC strategy imposes a certain dynamical situation rather than being responsive to the current traffic state. If T and ϕ are properly adjusted, vehicle platoons which are formed get "green wave" giving maximum possible flow rate $\bar{J}$. If density is small enough, i.e., platoon length $\rho v_{\mathrm{max}} D$ per link is shorter than $D/2 - \tau v_{\mathrm{max}}$, that is $\rho < (2v_{\mathrm{max}})^{-1} - \tau/D$, then there exists cycles and for which vehicles can move without stopping and the resulting mean flow $\bar{J} = J_{\mathrm{max}}$. On the other hand, for some values of T platoons are always stopped when arriving to the intersection. Consequently one can observe significant variations (by $\approx 100\%$) in $\bar{J}$ especially for smaller densities, $\rho < \rho_{\mathrm{max}}$, where clearly there is the largest potential for optimization. If density is too large, nothing can be done in terms of adjusting T and ϕ and there is no optimization which can significantly improve situation.

Comparison between CC and SC strategies for various densities is shown in Fig. 4. In these plots $J_{\text{best}}^{\text{CC}}$ for CC represent the maximum possible value, i.e., is calculated for given ρ by simulating flows for all $1 \leq T \leq 300$ and $0 \leq \phi \leq 300$ and choosing the largest $\bar{J}$ (the same procedure is done in the next section). The decentralized SC converges to the optimum in the region where optimization is possible (the stabilization parameters are $T_{\text{avg}} = 150$ and $T_{\text{max}} = 300$).

3.2 Periodic Network with Non-deterministic Turning

Introducing the possibility of vehicle turning (with the probability P_{turn}) makes an important difference when comparing to the previous case. Regular vehicle platoons can not be formed anymore, since they are separated with empty spaces resulting from changing a vehicle's direction (which in turn can form other platoons). A constant cycle controller can not adjust to such situation.

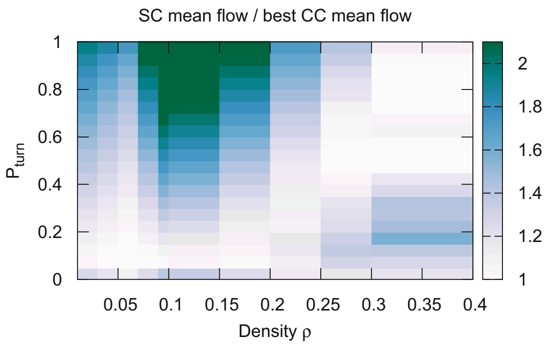


Fig. 5. The ratio $J^{\text{SC}}/J_{\text{best}}^{\text{CC}}$ as a function of mean density $\bar{\rho}$ and vehicle turning probability P_{turn}.

Figure 5 depicts the ratio of mean flows for the self-controlled and the best CC outcome. It is clear that in the region where optimization is possible (sufficiently small density), the SC outperforms the best possible CC by a factor of 2.

3.3 Network with Stochastic Input

As a final example we use a network with more realistic, stochastic boundaries (as described earlier) at the east and the south side, and open BCs at the west and the north side. For a single lane, $P_{\text{ins}} = 1.0$ will produce a flow with the maximum J_{max}. Obviously for concurring streams, J_{max} can not be reached for $\rho > \rho_{\text{max}}$, hence there must be a maximal P_{ins} above which one the mean flow will not increase. Figure 6(a) shows how $\bar{J}$ depends on the vehicle insertion probability. Also in this case, the decentralized strategy is able to form vehicle

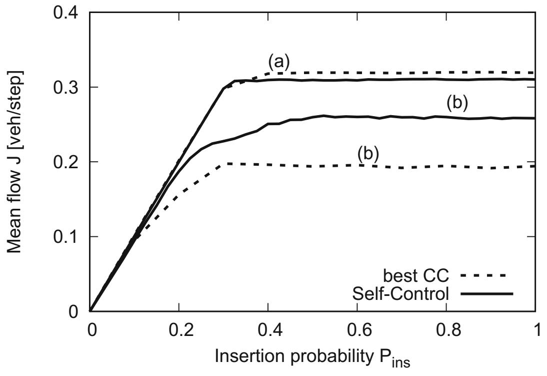


Fig. 6. Mean network flow as a function of vehicle insertion probability. (a) no turning, (b) turning into south-north direction with the probability $P_{\text{turn}} = 1/4$ completely breaks down the CC control.

platoons and green waves so the optimal J is reached. Moreover, if heterogeneous turning is introduced – vehicle can turn from east-west towards south-north lanes with $P_{\text{turn}} = 0.25$ – all the coordination in CC controlled network is lost. On the other hand, SC is able to recover quite well.

4 Conclusions

It has been shown how the self-controlled strategy proposed in [10] can be implemented in the classical cellular automata model of traffic [14] in the context of urban road networks [18]. Since the original formulation of the SC control is with continuous model based on kinematic waves, it is not straightforward to apply it in a CA model. In particular, the problems arise if the nondeterministic breaking in the CA is applied, $P > 0$. This is solved by using appropriate precalculated time-depended maximum fluxes $Q_i^{\max}$.

The presented simulations demonstrate that SC, by means of self organization, converges to the best possible fixed cycles in the case of regular networks with periodic and stochastic BCs. Additionally, if randomized scenarios are considered (e.g., vehicle turning), CC can not control flow in an optimal way since some responsiveness is required. In these cases SC significantly outperforms the best fixed cycle networks. In the future work, the presented model will serve as a test bed for other optimization methods for more complex network topologies and vehicle routing algorithms.

References

1. Danila, B., Yu, Y., Marsh, J., Bassler, K.: Optimal transport on complex networks. Phys. Rev. E **74**(4), 046106 (2006)
2. Cascone, A., Manzo, R., Piccoli, B., Rarità, L.: Optimization versus randomness for car traffic regulation. Phys. Rev. E **78**(2), 026113 (2008)

3. Chowdhury, D., Santen, L., Schadschneider, A.: Statistical physics of vehicular traffic and some related systems. Phys. Rep. **329**(4–6), 199–329 (2000)
4. Helbing, D.: Traffic and related self-driven many-particle systems. Rev. Mod. Phys. **73**(4), 1067 (2001)
5. Papadimitriou, C.H., Tsitsiklis, J.N.: The complexity of optimal queuing network control. Math. Oper. Res. **24**, 293 (1999)
6. Danila, B., Sun, Y., Bassler, K.E.: Collectively optimal routing for congested traffic limited by link capacity. Phys. Rev. E **80**(6), 066116 (2009)
7. Helbing, D., Lämmer, S., Lebacque, J.P.: Self-organized control of irregular or perturbed network traffic. In: Deissenberg, C., Hartl, R.F. (eds.) Optimal Control and Dynamic Games, vol. 7, pp. 239–274. AICM. Springer, Boston (2005)
8. Gershenson, C.: Self-organizing traffic lights. Complex Syst. **16**, 29 (2004)
9. Lämmer, S., Donner, R., Helbing, D.: Anticipative control of switched queueing systems. Eur. Phys. J. B **63**(3), 341 (2007)
10. Lämmer, S., Helbing, D.: Self-control of traffic lights and vehicle flows in urban road networks. J. Stat. Mech. Theory Exp. **2008**(4), P04019 (2008)
11. Tang, M., Liu, Z., Liang, X., Hui, P.: Self-adjusting routing schemes for time-varying traffic in scale-free networks. Phys. Rev. E **80**(2), 026114 (2009)
12. Szklarski, J.: Cellular automata model of self-organizing traffic control in urban networks. Bull. Pol. Acad. Sci.: Tech. Sci. **58**(3), 435–441 (2010)
13. Lighthill, M.J., Whitham, G.B.: On kinematic waves II. A theory of traffic flow on long crowded roads. Proc. R. Soc. Lond. A: Math. Phys. Eng. Sci. **229**(1178), 317–345 (1955)
14. Nagel, K., Schreckenberg, M.: A cellular automaton model for freeway traffic. J. Phys. I **2**(12), 2221–2229 (1992)
15. Kalogeropoulos, G., Sirakoulis, G.C., Karafyllidis, I.: Cellular automata on FPGA for real-time urban traffic signals control. J. Supercomput. **65**(2), 664–681 (2013)
16. Shen, Z., Wang, K., Zhu, F.: Agent-based traffic simulation and traffic signal timing optimization with GPU. In: 2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC), pp. 145–150, October 2011
17. Chowdhury, D., Schadschneider, A.: Self-organization of traffic jams in cities: effects of stochastic dynamics and signal periods. Phys. Rev. E **59**(2), R1311–R1314 (1999)
18. Brockfeld, E., Barlovic, R., Schadschneider, A., Schreckenberg, M.: Optimizing traffic lights in a cellular automaton model for city traffic. Phys. Rev. E **64**(5), 056132 (2001)
19. Barlovic, R., Huisinga, T., Schadschneider, A., Schreckenberg, M.: Open boundaries in a cellular automaton model for traffic flow with metastable states. Phys. Rev. E **66**, 046113 (2002)

Towards Multi-Agent Simulations Accelerated by GPU

Kamil Piętak and Paweł Topa(✉)

Department of Computer Science, AGH University of Science and Technology, Kraków, Poland
{kpietak,topa}@agh.edu.pl

Abstract. At present, GPUs (Graphics Processing Units) are commonly used to speedup any kind of computations. In this paper we present how GPUs and Nvidia CUDA can be used to accelerate the updating of and agent state in Multi-Agent Simulations. We use the AgE (Agent Evolution) software framework written in Java, which supports agent-based computations. In our simulations agents represent living organisms that interact with the virtual habitat and with each other. At each step of the simulation thousands of agents update their state according to a defined set of rules. We use Java bindings for CUDA (JCUDA) to move massive computations to GPU.

Keywords: Multi-Agent Systems · Evolutionary computations
General purpose computations on GPU

1 Introduction

Graphic Processing Units (GPUs) are currently widely used to speed up any type of computation. Their architecture assumes that a device is equipped with hundreds of processing units, which are designed to maximize computation throughput. The control units in GPUs are not as sophisticated as those in general purpose CPU processors. As a result, the programmer must organize data structures and algorithms to be efficiently processed by a GPU.

Computation with GPUs shows its power when a large number of identical sequences of instructions (threads) are applied to regularly organized data, i.e. arrays. In such a case the whole block of data is processed in a perfectly parallel manner. Any conditional instruction that may change the path of execution in some threads spoils this perfection.

Agent-Based Computing is a paradigm that at first glance may benefit from using GPU. Usually, the computation involves hundreds or more of agents, which usually have homogeneous sets of parameters and rules of behaviour. Their interactions are usually local. Their states are updated at each step of simulation. In the case of Cellular Automata, which may be partially treated as a kind of Multi-Agents System with strictly defined network of agents neighbourhood, the application of GPU results in an enormous speedup of computation — see examples in [9,17,20].

However, the classic definition of an agent does not impose such strict assumptions on its position and relation with other agents. Moreover, unlike in Cellular Automata, the agents are usually processed asynchronously. Thus, the implementation of a typical Multi-Agent System requires an additional procedure that prepares the population of agents to be processed on a GPU.

In this paper we present how GPUs can be used to improve the performance of an agent-based simulator designed and implemented for modelling the population of Foraminifera, a marine microorganism [6]. The simulator, named eVolutus [18,19], is built over AgE, a software framework [5] that supports the Evolutionary Multi-Agent System (EMAS) computation paradigm [3,4]. EMAS is an approach that combines the notion of an agent with evolutionary algorithms. It assumes that agents carry a some kind of genome that can be passed to their offspring. During reproduction the genome is subjected to the genetic operators: mutation and crossing over. Selection, which is a crucial procedure in any evolutionary algorithm, are indirectly realised by agents dying childless. EMAS was invented to solve some engineering problems, but it can be adapted naturally to model the population dynamics and evolution of living organisms.

The rest of the paper is organized as follows. Section 2 discusses existing applications of GPU computing in agent-based systems. Next, we briefly presents the architecture of AgE and eVolutus, and then the modifications that was necessary to employ a GPU for processing agents are explained. Section 5 contains the results of performance evaluation. At the end we add some concluding remarks.

2 Agent-Based System with Support for GPU Computing

As was mentioned above, in order to achieve high efficiency in computing on GPUs, the data structures should have regular structure, and the access pattern to this data also should be regular. Such requirements are often fulfilled by models based on the Cellular Automata (CA) paradigm. The CA approach, especially when the rules of local interaction are more complex and sophisticated, is considered to be very similar to the Multi-Agent Systems paradigm. Such an approach is very attractive for large scale modelling of crowd dynamics. In this application, simulations involve a huge number of pedestrians (up to 10^6) and the rules that govern their behaviour must take into account not only interactions with closest neighbours but also with the environment, e.g. the locations of obstacles and exits. These models are also required to be able to provide results “faster than real time” when they are used to verify various evacuation scenarios. Thus, many interesting solutions can be found in this area of research.

Wąs et al. [9,10] introduced a very efficient implementation of their Social Distance Model of pedestrian dynamics. His works is focused on optimizing algorithms for massive parallel computation. The algorithms are modified to ensure that all the GPU cores will be filled with the same stream of instructions. In [9] there are also discussed and tested various scenarios of using configurations with multiple GPUs.

Aaby et al. [1,12] present more general analysis of using GPU for Agent-Based Modelling. They investigate how a single GPU speeds up computation using well known ABM benchmarks. Next they investigated more complex hybrid configurations which integrates GPU computing with message passing parallel programming.

In [11] the authors address the most important problem, which confronts ABM when a GPU is used: the irregular and dynamic structure of data in ABM, as well as the randomized memory access during computation. Their solution introduces two important mechanisms to cope with these problems. Agent manager supervises the GPU memory when agents are created or removed to optimize this process. Interaction manager supports neighbourhood calculation which is a major issue in models with continuous space. The authors compare the proposed solution with existing ABM computing platforms FLAME and Mason, with positive results.

Out of the many Agent-Based modelling environments, at this moment only FLAME [8] supports GPU computation [16]. In this environment, agents are declaratively specified using the templates. The specification is used to generate the code of the simulator. The framework employs various GPU optimized algorithms which address main issues in agent-based computation, e.g., calculating the neighbourhood. FLAME GPU demonstrates its efficiency in many applications such as crowd dynamics [7], biology [15], sociology [14].

3 eVolutus — EMAS Simulator of Evolution and Population Dynamics

In eVolutus, Foraminifera individuals are represented by agents. The behaviour of agents is controlled by several rules and a set of parameters. These parameters are treated as a virtual genome and are passed to offspring in the process of reproduction. Depending on the Foraminifera species reproduction is either asexual (only a mutation operator is employed) or sexual (a crossing over operator is used).

The marine habitat is modelled using an approach similar to Cellular Automata. The space is partitioned using a regular grid of cells. Each cell has some parameters that correspond to real physical properties, e.g., depth, temperature, insolation, salinity. During the simulation these parameters may change governed by defined rules.

Cells can be occupied by agents. In order to save computational resources the exact position of agent inside the cell is not tracked. Agent interacts only with other agents in the same cell and it is affected only by conditions in the cell in which is located.

eVolutus is implemented over the AgE framework, developed at AGH University of Science and Technology [5,13]. In Fig. 1 the mapping between AgE objects and eVolutus components is presented. Each single cell of habitat is implemented as an AgE aggregate agent. Each has strict and unchangeable position in the grid of cells. The AgE aggregate works as a container for foram agents located inside this cell at this moment. Altogether, the aggregates form a workplace responsible for running the computation and being a proxy for communication with aggregates located in other workplaces. Simulation may involve several workplaces when the habitat is partitioned over the distributed computing architecture or when the habitat forms a network of independently processed islands. Foram agents are able to migrate between different aggregates in the whole environment.

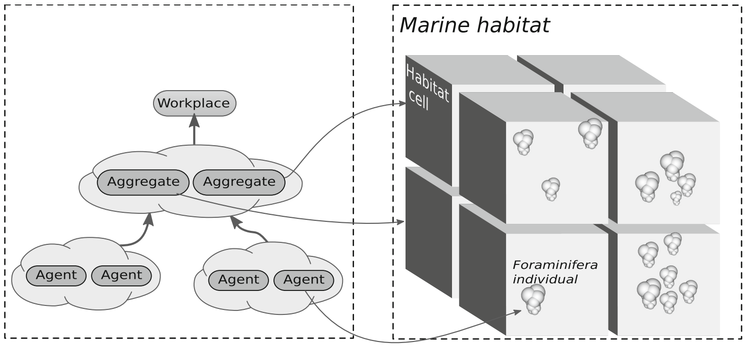


Fig. 1. The eVolutus simulator implemented over the AgE framework

The aggregate and Foraminifera agents are processed in a pseudo-parallel manner by calling `step()` method. It means that however the `step()` methods are invoked sequentially, the operations that change the state of the environment or the state of other agents (called actions) are delayed and executed and the end. The `step` method of foram agents is called by their parent, which is an aggregate agent. The aggregate iterates over its children in an undetermined order, calls synchronously the `step` method and then executes the actions registered by children agents in their step. Thus, when the agents make decisions, they have exactly the same information about the environment.

In eVolutus, the `step()` method of Foraminifera agents has a very clear form:

```
public void step() {
    consumeStepEnergy();
    if (shouldDie(...))
        die();
    eat();
    if (canReproduce(...))
        reproduce();
    if (canCreateChamber(...))
        createChamber();
    if (canMigrate(...))
        tryMigrate();
    age += stepDurationInHours;
}
```

At each step of simulation, an agent always consumes some amount of energy to maintain its life and gather some food from the environment. The rest of the actions are optional, when the necessary conditions are fulfilled, the agent may die, reproduce, grow or migrate.

The eVolutus provides high level of configurability due to using the Oracle Nashorn technology. This framework allows to execute the Javascript code in Java Virtual Machine with the same performance as native Java code. User defines agent's action without touching Java code by writing short functions in Javascript, i.e.,:

```
function shouldDie(envState, foramState, time) {
    var energyLow = foramState.energy < foramState.minEnergy;
    return energyLow;
}
```

This functionality of the eVolutus is explained with details in [18].

The organization of agent processing implemented in AgE is very clear and convenient from the developer point of view. Unfortunately, it cannot be directly applied when GPU processing is used. In fact, one needs to collect necessary information from foram agents, form a big regular block of data, send to GPU and execute in parallel hundreds of threads with the same sequence of instructions.

4 Implementation Using GPGPU

When computations using evolutionary techniques with GPGPU are considered, the global parallelisation model [2] in the form of master-slave architecture is a natural choice. The architecture assumes that the CPU performs most of the evolutionary process and the unit delegates some of most expensive computations to the GPU. Here, CPU processes a model sequentially but it is also possible to parallelize these computations.

To allow effective computing using GPGPU, some crucial modifications in the algorithm presented in Sect. 3 were required. The decisions made by particular foram agents (such as *should die, reproduce, create chamber or migrate?*), are a good choice to delegate to slaves. We are sure that these functions have to be executed by all agents at each step of simulation. The other procedures may be invoked occasionally and only by a small group of agents at the same time. These decisions are extended with required conversions between CPU and GPU representations of forams attributes (such as genotype, energy, age) required to make these decisions. To minimize the communication overhead and use the parallel nature of GPU, these operations are performed for a whole population of foram agents (located at a particular aggregate) at once. The GPU interface receives the variable number of individuals that belong to an ocean fragment and next, as a result, returns a set of decisions for each foram. Based on the returned set, each foram performs sequentially appropriate operations such as eating, reproduction, dying or migration.

All of this requires some changes in foram agents processing and their `step` method:

– all decisions made by foram agents are extracted into a new method called `processForamsDecisions`,

- an aggregate executes firstly for all child agents called `processForams Decisions` method,
- then, the aggregate for each child agent executes its `step` method based on the decisions made in the previous phase.

Additionally, each agent at each step performs two obligatory actions: gather food from the environment and use stored energy to maintain vital functions of agents. These two actions refer to the inner state of the agent (although the gathering food results in updating the environment resources at the end of the step) and their processing can be easily moved to the GPU.

In eVolutus each agent has several parameters (including "virtual genes") which are used during computation. Some of them are changed more or less frequently, e.g., the level of stored energy is updated at each step, the size of Foraminifera body usually changes when the agent performs growth action. On the other hand, the "virtual genes" remain constant through the whole life of the agent. At this moment at each step of simulation the relatively large numbers of parameters which are implemented as a field of agents' class have to be copied into arrays and sent to GPU's global memory. In the same way, the results are transferred from GPU to agent. This is inconvenient from the programmers point of view and also may impact the performance. Thus, our further investigations will be connected to find optimal, from the GPU point of view, organization of agents' data.

The following listing presents a simplified Java code snippet that illustrates how an aggregate processes the population of Foraminifera agents:

```java
Map<Foram, Decision> decisions =
    processForamDecisions(population, environment);

for (Foram foram : decisions.keySet()) {
    decision = foram.get(foram);

    foram.eat(environment);

    if (decision.doGrowth)
        foram.createChamber(foramState, environment);

    if (decision.doMigration)
        foram.tryMigrate(foramState, environment);

    if (decision.doReproduction)
        reproduce(foram, foramState, environment);

    if (decision.doDie)
        foram.die(environment);

    foram.makeOlder();
}
```

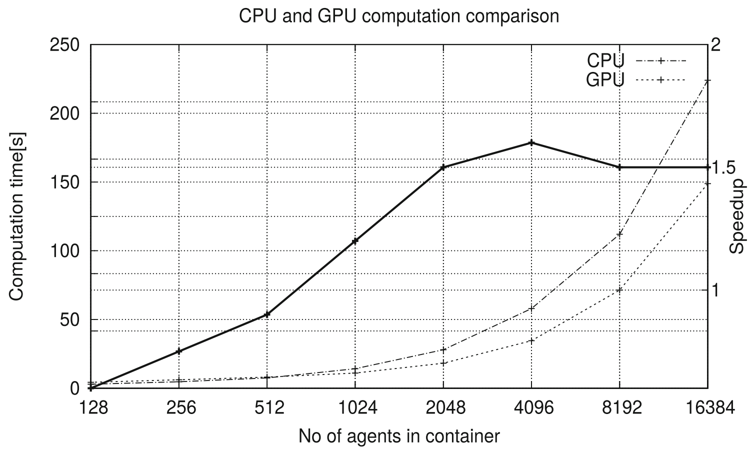
The `processForamDecisions` method can be implemented in various ways. In this case, two realizations have been introduced:

- sequential CPU implementation, in which the decision for every foram agent is performed in a loop,
- parallel GPGPU implementation with required conversion between data structures.

These two versions allow for comparing results between pure CPU+GPGPU and CPU implementations of the algorithm.

5 Performance Evaluation

For testing purposes we use Nvidia GeForce GTX750i graphic card. This graphics card has a Maxwell processor with 640 of CUDA core and 2 GB of GDDR5 memory. The processor supports CUDA Compute Capability 5.0. We compared the GPU version with pure sequential CPU-only implementation. The CPU is Intel Core i7 4700MQ 2.40 GHz with 6 MB cache.



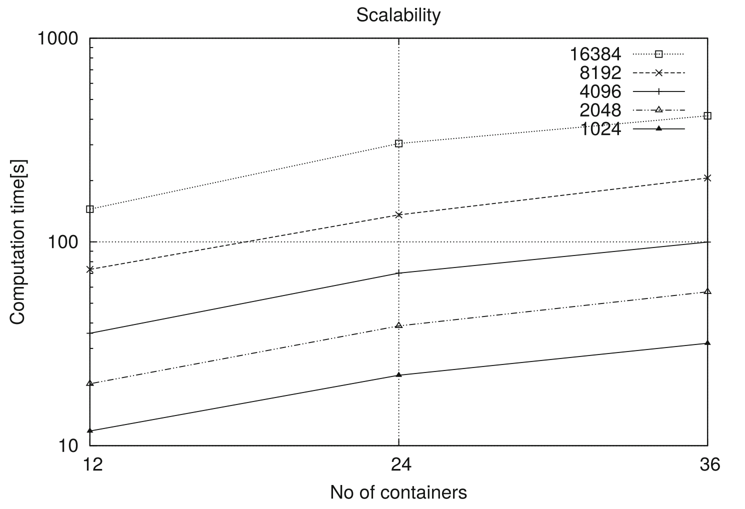


Fig. 2. Comparions of CPU and GPU version. The first chart shows speedup achieved by GPU version over the CPU one. The second chart presents the results of scalability tests.

We prepared a testing scenario that represents stable Foraminifera habitat with various densities of agents. The original Javascript functions were translated into the CUDA kernels. In both simulators, a similar dynamics of populations were observed.

The tests were performed for various numbers of agents inside the containers: 128, 256, 1024, 2048, 4096, 8192 and 16384. The results show (see Fig. 2, first chart) that the GPU version of the simulator is up to 1.5 faster than version for CPU. The performance is poor when the number of agents is low (below 1024). It is the expected behaviour — the amount of calculations have to be large enough to balance the effort related to invoking GPU computation.

We also tested the performance of the GPU implementation for various sizes of habitat. The tests were performed for the habitat consisting of 12, 24 and 36 containers (we assumed that containers always has the same size). The results demonstrate that the implementation keeps good level of scalability for each of the tested sizes of population (see Fig. 2, second chart). In the most demanding configuration, the simulation included over 5×10^5 agents.

6 Summary

The eVolutus simulator, implemented over the AgE platform has a very clear and readable structure that has benefits when the model is tested and calibrated — any modifications and new features can be added to the agents' code in a safe way. Our goal is to save this feature when a GPU is used.

The results of the investigations presented, although very preliminary, seem to be very promising. When the number of agents inside the container is larger the 10^3 the performance of the GPU implementation exceeds the results achieved by using the CPU-only version. The results show that the main problem is to provide enough tasks to compute. The mandatory calculations made the agents still very "light" and in order to exploit the power of the GPU we have to provide huge number agents to process.

The part of the model that is executed on CPU is simply processed sequentially. However, the multicore architecture of today's CPUs encourages the use of multithreaded processing. The comparison of multithreading implementation with GPU implementation might be interesting from practical point of view. Although, here we limit our investigation to the most basic configuration what allows to identify the potential directions of further works.

Here, we use only a global memory. Recent GPU processors are equipped with cache memory which supports transfers from and to global memory. Any optimizations that will use a shared memory in this case require a more detailed analysis.

At this moment, we use a relatively simple division of agent's actions into two groups performed on host and on device (GPU). In order to increase the amount of computation performed by GPU, we try to introduce new schema of agent's actions and new method of allocating the agents's actions to the GPU. We believe that in such a solution most of the computations will be processed by the GPU. Only actions that require communication with containers of agents will be processed by host.

Acknowledgements. The work presented in this paper received support from the Polish National Science Centre (DEC-2013/09/B/ST10/01734).

References

1. Aaby, B.G., Perumalla, K.S., Seal, S.K.: Efficient simulation of agent-based models on multi-GPU and multi-core clusters. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, p. 29. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2010)
2. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Trans. Evol. Comput. **6**(5), 443–462 (2002)
3. Byrski, A., Kisiel-Dorohinicki, M.: Evolutionary Multi-Agent Systems: From Inspirations to Applications. SCI, vol. 680. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51388-1
4. Cetnarowicz, K., Kisiel-Dorohinicki, M., Nawarecki, E.: The application of evolution process in multi-agent world to the prediction system. In: Proceedings of the Second International Conference on Multi-Agent Systems, ICMAS, vol. 96, pp. 26–32 (1996)
5. Faber, Ł., Piętak, K., Byrski, A., Kisiel-Dorohinicki, M.: Agent-based simulation in AgE framework. In: Byrski, A., Oplatková, Z., Carvalho, M., Kisiel-Dorohinicki, M. (eds.) Advances in Intelligent Modelling and Simulation. SCI, vol. 416, pp. 55–83. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28888-3_3
6. Goldstein, S.: Foraminifera: A Biological Overview. Kluwer Academic Publishers, Dordrecht (1999)
7. Karmakharm, T., Richmond, P., Romano, D.M.: Agent-based large scale simulation of pedestrians with adaptive realistic navigation vector fields. TPCG **10**, 67–74 (2010)
8. Kiran, M., Richmond, P., Holcombe, M., Chin, L.S., Worth, D., Greenough, C.: Flame: simulating large populations of agents on parallel hardware architectures. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2010, pp. 1633–1636 (2010)
9. Kłusek, A., Topa, P., Wąs, J., Lubas, R.: An implementation of the social distances model using multi-GPU systems. Int. J. High Perform. Comput. Appl. 1094342016679492 (2016)
10. Kłusek, A., Topa, P., Wąs, J.: Towards effective GPU implementation of social distances model for mass evacuation. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 550–559. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_51
11. Li, X., Cai, W., Turner, S.J.: Supporting efficient execution of continuous space agent-based simulation on GPU. Concurr. Comput.: Pract. Exp. **28**(12), 3313–3332 (2016). https://doi.org/10.1002/cpe.3808
12. Perumalla, K.S., Aaby, B.G.: Data parallel execution challenges and runtime performance of agent simulations on GPUs. In: Proceedings of the 2008 Spring simulation multiconference, pp. 116–123. Society for Computer Simulation International (2008)

13. Piętak, K., Kisiel-Dorohinicki, M.: Agent-based framework facilitating component-based implementation of distributed computational intelligence systems. In: Nguyen, N.-T., Kołodziej, J., Burczyński, T., Biba, M. (eds.) Transactions on Computational Collective Intelligence X. LNCS, vol. 7776, pp. 31–44. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38496-7_3
14. Richmond, P., Buesing, L., Giugliano, M., Vasilaki, E.: Democratic population decisions result in robust policy-gradient learning: a parametric study with GPU simulations. PLoS ONE **6**(5), e18539 (2011)
15. Richmond, P., Coakley, S., Romano, D.: Cellular level agent based modelling on the graphics processing unit. In: 2009 International Workshop on High Performance Computational Systems Biology, HIBI 2009, pp. 43–50. IEEE (2009)
16. Richmond, P., Walker, D., Coakley, S., Romano, D.: High performance cellular level agent-based simulation with flame for the GPU. Brief. Bioinform. **11**(3), 334–347 (2010)
17. Topa, P.: Cellular automata model tuned for efficient computation on GPU with global memory cache. In: 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, 12–14 February, 2014, Torino, Italy, pp. 380–383 (2014)
18. Topa, P., Komosinski, M., Tyszka, J., Mensfelt, A., Rokitta, S., Byrski, A., Bassara, M.: eVolutus: a new platform for evolutionary experiments. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K., Kitowski, J., Wiatr, K. (eds.) PPAM 2015. LNCS, vol. 9574, pp. 570–580. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32152-3_53
19. Topa, P., Faber, Ł., Tyszka, J., Komosinski, M.: Modelling ecology and evolution of foraminifera in the agent-oriented distributed platform. J. Comput. Sci. **18**, 69–84 (2017)
20. Topa, P., Młocek, P.: Using shared memory as a cache in high performance cellular automata water flow simulations. Comput. Sci. **14**(3), 385 (2013)

Tournament-Based Convection Selection in Evolutionary Algorithms

Maciej Komosinski(✉) and Konrad Miazga

Institute of Computing Science, Poznan University of Technology, Piotrowo 2, 60-965 Poznan, Poland
maciej.komosinski@cs.put.poznan.pl

Abstract. One of the problems that single-threaded (non-parallel) evolutionary algorithms encounter is premature convergence and the lack of diversity in the population. To counteract this problem and improve the performance of evolutionary algorithms in terms of the quality of optimized solutions, a new subpopulation-based selection scheme – the *convection selection* – is introduced and analyzed in this work. This new selection scheme is compared against traditional selection of individuals in a single-population evolutionary processes. The experimental results indicate that the use of subpopulations with fitness-based assignment of individuals yields better results than both random assignment and a traditional, non-parallel evolutionary architecture.

Keywords: Evolutionary algorithms · Selection scheme
Convection selection · Diversity · Exploration

1 Introduction

A selection scheme is one of the most important elements of evolutionary algorithms [2,6,14]. Not only it determines the selective pressure in the population, but it also controls the distribution of this pressure among all individuals. Over the years many selection schemes were proposed, some of the most popular ones being tournament selection [3,13], ranking selection [4], proportional selection [7] and sigma scaling [1]. A common element for all of them is the monotonicity of the probability of selection with respect to fitness – a sensible property in optimization, since better individuals deserve a higher chance of propagating their genes. In this paper we show that a more complex, non-monotonic selection scheme can improve the performance of evolutionary algorithms.

In a recent paper [11], Komosinski proposed two methods of dividing the population into subpopulations based only on fitness values of individuals, which does not require computation of any additional, potentially complex and time-consuming, similarity measures. The performance gain obtained by these methods has been verified experimentally in a parallel setting (hence it was a *distribution* technique). The paper discussed the logic behind this way of splitting of the population and provided some explanations on why it was beneficial. This population-splitting scheme was called the *convection distribution* because it facilitates continuous evolutionary progress just like a convection current or a conveyor belt: each subpopulation always tries to independently improve genotypes of a specific fitness range which overall ensures more fitness diversity and avoids the domination of (and the convergence towards) the current globally best genotypes [5]. Occasional, short ascending trends (convections) are visible in the entire range of fitness values. As mentioned in [11], this idea can be directly implemented in a standard, single-threaded (i.e., non-parallel) evolutionary algorithm, where it becomes the *convection selection* scheme.

It is known that given the same computational cost, parallel evolutionary algorithms [12,16,18] can sometimes yield better results in optimization tasks than standard sequential evolutionary algorithms, mostly because of the local exchange of individuals between independent subpopulations. Local exchange of individuals leads to increased exploration of the search space, which is often desirable [16]. Increased exploration can also be achieved in sequential evolutionary optimization using methods such as sharing or restricted mating [15]. Such methods require however calculating of additional measures of similarity between individuals, which may be time consuming, especially in applications where individuals are complex [11], such as evolutionary design or artificial life.

Convection selection techniques may be perceived as super-selection techniques in that they determine which individual should be assigned to which subpopulation, yet within these subpopulations traditional selection schemes are still employed. Thus convection selection can be combined with any traditional selection method, constituting convection tournament selection, convection roulette selection, etc. Moreover, while in this work we will discuss one-level convection selection (i.e., a population divided into sub-populations), this technique can act on multiple levels with subpopulations recursively embedded in each other.

The experiments reported in [11] proved that convection distribution methods yielded significantly better results than random distribution of genotypes among subpopulations. In this work, we investigate when the *convection selection* (assigning individuals to subpopulations based on fitness values) can yield better results compared to standard, single-population positive selection schemes such as tournament selection. We also analyze the underlying mechanisms responsible for the success of this new approach.

Apart from implementing three subpopulation-based selection techniques in a single-threaded (non-parallel) evolutionary algorithm and comparing their performance, we also compare these three approaches against a standard, single-population evolutionary algorithm. In all comparisons we ensure that the overall computational cost is the same – in each evolutionary run, we keep the number of evaluations of individuals equal, and the computational cost of managing subpopulations and migrations between subpopulations is negligible. Moreover, we test each of the four mentioned approaches (Fig. 1) using various selective pressures and populations sizes, and for each approach we choose the best performance among its various parametrizations to ensure a fair comparison.

2 Methods

All the experiments described in this paper were performed using Framsticks software [9,10]. Framsticks allows to evolve bodies and brains of 3D designs (agents) towards a goal specified by some fitness function. This area of application of evolutionary algorithms benefits the most from selection schemes that improve the performance yet are still computationally inexpensive. This is because optimization tasks in evolutionary design are extremely difficult and solutions are very complex due to sophisticated genotype-to-phenotype mappings, so calculating sophisticated properties of such solutions or estimating their similarity is usually very costly and should be avoided if possible.

We have used two fitness functions that differ in the difficulty of optimization: *velocity* and *height*. The *velocity* criterion is used to evolve individuals that move fast on land (so body and brain are coevolved and must be coordinated), whereas *height* is used to evolve static tall structures (their neural network is disabled) with the center of mass as elevated as possible.

The "f1" genetic encoding was employed [8,9]. This encoding is a direct mapping between symbols and parts of a 3D structure: 'X' represents a rod (a stick), parentheses encode branches in the structure, and additional characters influence properties like length or rotation. Neurons are described in square brackets and index numbers in their connections are relative, so the information about connections is local and persists when a part of a genotype is cut out. The encoding is able to represent tree-like 3D body structures and neural networks of arbitrary topology. Mutations modify individual aspects of the agent by adding or removing parentheses in random locations in the genotype, by adding and removing random symbols that affect the structure, by adding and removing neurons and connections, and by adding random Gaussian-distributed values to neural weights.

For both fitness functions, evolution was started from the simplest individual (i.e., 'X' in the $f1$ encoding). The steady-state (also known as "incremental") evolutionary algorithm [17] was used. To limit the number of factors that might influence the performance of convection selection schemes, no crossover was employed in the experiments reported here. The crossover was however used in the experiments discussed in [11], where convection selection schemes provided superior results. The absence of the crossing over operator in this work and the fact that convection selection schemes still yielded superior results means that the crossover operator is not the only mechanism responsible for the efficiency of these selection techniques.

In the convection selection schemes, individuals are first sorted according to their fitness. Then each subpopulation receives a subset of individuals that fall within a range of fitness values. In our experiments, two methods of determining fitness ranges are considered. In the first method denoted $EqualWidth$ (Fig. 1c), the entire fitness range has been divided into equal intervals (as many as there are subpopulations); if there are no individuals in some fitness range, the corresponding subpopulation receives individuals from the nearest lower non-empty fitness interval. In the second method denoted $EqualNumber$ (Fig. 1d), once the

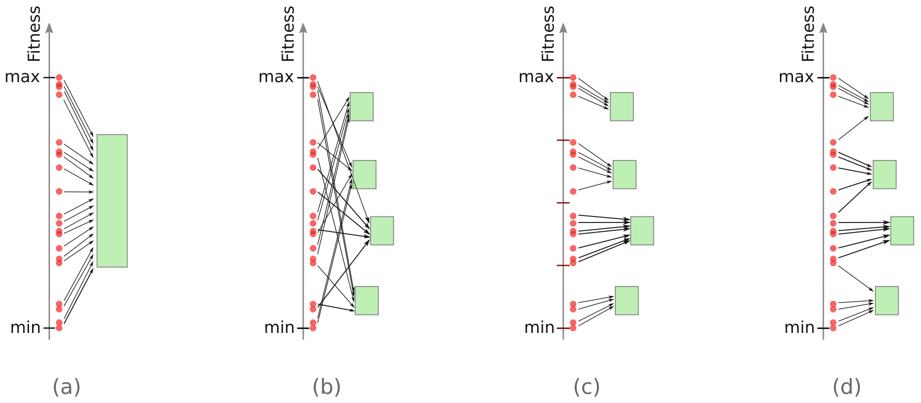


Fig. 1. An illustration of four compared selection schemes. The fitness of 20 individuals is shown as red circles, and 4 subpopulations are depicted as green boxes. (a) Standard evolutionary algorithm with a single population. (b) Random assignment of individuals to subpopulations. (c) Convection selection with fitness intervals of equal width. (d) Convection selection with fitness intervals yielding equal number of individuals. (Color figure online)

individuals are sorted according to their fitness, they are divided into as many sets as there are subpopulations so that each subpopulation receives the same number of individuals.

We compare here four approaches to selection (three of which use subpopulations), and in each of them the underlying traditional selection mechanism is the tournament selection. The logic of the three evolutionary processes that use selection to assign individuals to subpopulations (i.e., $Random$, $EqualWidth$, or $EqualNumber$) is implemented as follows. Every $R \cdot \frac{N}{M}$ evaluations (where R is the migration period scaling factor which defines how frequently subpopulations should merge, N is the size of the entire population, and M is the number of subpopulations), M subpopulations are merged and then all individuals from the complete (merged) population are split again into M subpopulations according to the applied selection scheme ($Random$, $EqualWidth$, or $EqualNumber$). After that, the algorithm cycles through all subpopulations in sequence so that each subpopulation becomes "current" in turn. The steady-state evolutionary algorithm selects one individual from the current subpopulation (using tournament selection with the tournament of size t), mutates it and adds the newly mutated offspring to the current subpopulation. Once this new individual has been evaluated, the negative selection process removes randomly one individual from a random subpopulation, so the size of the complete population remains constant. Then, the next subpopulation in sequence becomes current. After all subpopulations have been processed, the cycle starts again unless it is time to merge all subpopulations and redistribute individuals to newly constructed subpopulations.

In this paper we perform two kinds of analyses. The first kind compares the quality of solutions obtained from the standard single-population evolutionary algorithms with the results yielded by the three proposed subpopulation-based selection schemes. The proper comparison between the single-population algorithm and the three subpopulation-based approaches is not simple, as each of these two concepts uses a slightly different set of parameters. Moreover, even the parameters that are shared between the four approaches can have different optimal values for each approach. If one wants to properly compare the quality of solutions achieved with each of the considered selection schemes, one should compare the best results obtained across a series of many different parametrizations for each selection scheme. Therefore, within each parametrization, the representative result for that parametrization is considered to be the average value of the best fitness values obtained across many independent runs (repetitions).

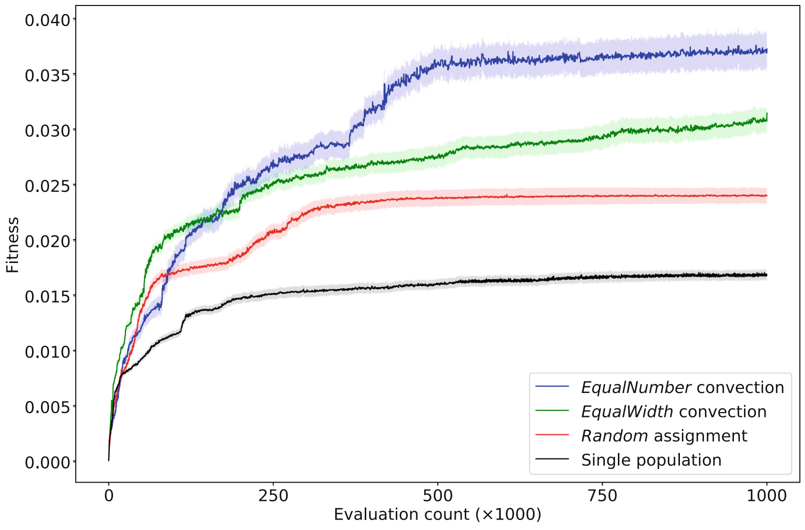
The second kind of the analysis takes a more detailed look into the results obtained by the three population-based selection schemes, two of which are convection selection schemes. We compare the average results achieved for each set of parameter values in order to understand which combinations of parameter values work well together, which combinations work poorly, and what are the potential reasons for such behavior.

The data required for both of the analyses discussed above were obtained from the following experiments. In each of the evolutionary runs, 10^6 individuals were evaluated, so that even though the selection schemes were different, they did not differ significantly in the overall computational cost. Two fitness functions were considered: *velocity* and *height*. For the single-population evolution and tournament selection, we have tested all the combinations of two parameters: population size $N \in \{100, 200, 500, 1000\}$ and tournament size $t \in \{2, 3, 5\}$. For three subpopulation-based selection schemes, all the combinations of the following sets of parameter values were tested: population size $N = 1000$, tournament size $t \in \{2, 3, 5\}$, number of subpopulations $M \in \{4, 10, 25, 50\}$, and the number of individual evaluations between merging the subpopulations (given as the multiple of the size of subpopulations) $R \in \{2, 10, 50\}$. Such a setup means that to obtain one result (i.e., best fitness value from one evolutionary run) for each combination of fitness functions and parameter values, we needed to perform $2 \times ((4 \times 3) + 3 \times (3 \times 4 \times 3)) = 240$ independent evolutionary runs. Since the evolutionary process is non-deterministic, to obtain averages and standard deviations for each parametrization, these runs were repeated 10 times which yielded 2400 independent evolutionary runs.

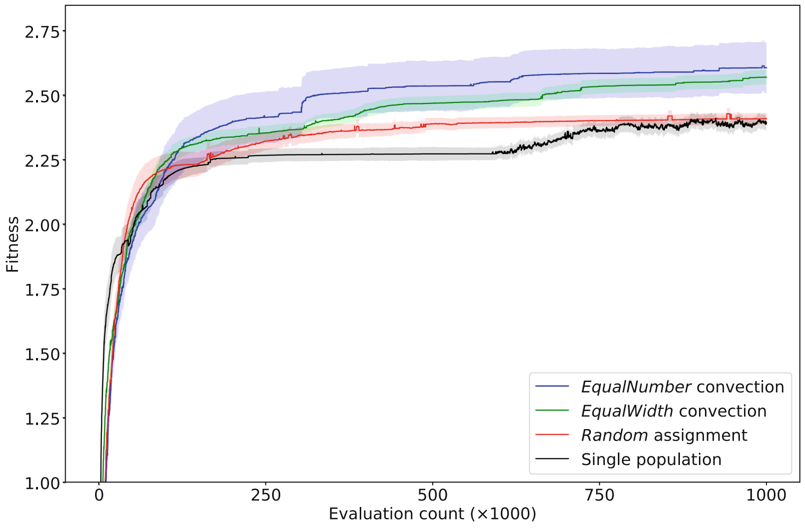
3 Results

3.1 The Performance of Different Selection Schemes

Figure 2 shows the performance of the evolutionary algorithms in time (measured as the number of individual evaluations) for four selection schemes – one single-population tournament selection, and three subpopulation-based algorithms with super-selection schemes. Since the influence of parameter values for



(a) *velocity* fitness function.



(b) *height* fitness function.

Fig. 2. Comparison of the performance of single-population tournament selection and three proposed meta-selections. Each series consists of the high bound (i.e., best) of the average fitness value obtainable for a given selection scheme, for any of the tested sets of parameter values. The band around each series represents 25% of the standard deviation for that series (25% is used instead of 100% to avoid overlapping bands and improve the readability of the plots).

the single-population approach and the three subpopulation approaches is not directly comparable (even for the same parameters), in order to provide a fair comparison we show the best average fitness value achieved by any parametrization for each approach, computed separately for each point in time. This means that the chart is a high-level comparison of the best performance of the four selection schemes that can be achieved over all parametrizations.

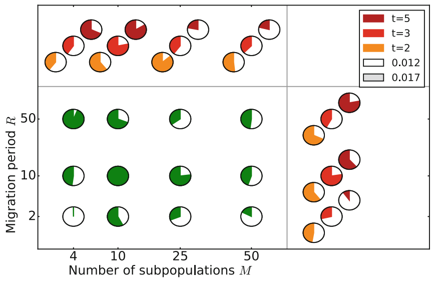
For the *velocity* fitness function, the performance of subpopulation-based selection schemes is clearly superior to the single-population tournament selection. While the fitness values for single-population evolution stabilize near the value of 0.017, the convection-based schemes manage to overtake it by a significant margin. The *Random* assignment selection scheme stabilizes only around the value of 0.024, whereas the convection schemes continue to improve in time (see Fig. 9 in [11] for the distributions of fitness values that illustrate the convection effect), ultimately reaching 0.031 for the *EqualWidth* method and 0.037 for the *EqualNumber* method.

The plot for the *height* fitness function presents similar, although less pronounced relationships. Once again the subpopulation-based schemes yield better results than the single-population selection, with convection selection schemes outperforming the *Random* assignment of individuals to subpopulations. It is worth noting however that for the first few thousand evaluations, single-population selection leads to better individuals than the subpopulation-based schemes – in this phase the optimization is relatively easy, and so population diversity (exploration) is not as beneficial as intensive, fast exploitation. Once the solutions reach the fitness values above 2 it is much harder to produce better individuals, at which point the subpopulation-based schemes overtake the single-population selection.

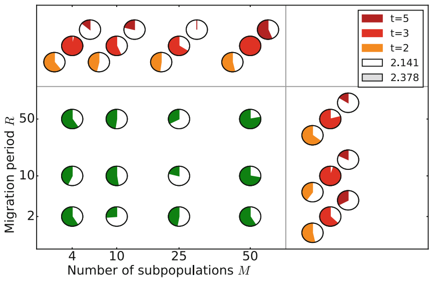
3.2 The Influence of Parameters of the Convection Selection

Figure 3 presents the effect that parameter values of convection selection have on the quality of solutions that were found by the evolutionary algorithm. Depending on the selection scheme, various trends can be seen. For *Random* assignment of individuals to subpopulations (Fig. 3a and b) no clear patterns emerged – parameter values do not demonstrate any direct influence on fitness, which may suggest that without any specific logic like fitness-based selection, the algorithm cannot fully exploit the advantages of working with multiple subpopulations.

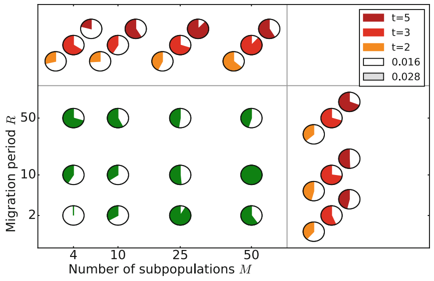
The opposite is however visible for the *EqualNumber* convection selection scheme (Fig. 3e and f). For the *height* fitness function (Fig. 3f), high selective pressure yields better results, as represented by darker circles being more filled up than the light ones. For both fitness functions, increasing the migration period scaling factor R (the vertical axis) leads to better results. The increase in the value of R allows each of the subpopulations to significantly increase the quality of its solutions before the subpopulations are merged; for longer migration periods, the contents of each subpopulation can change considerably between migrations which facilitates diversity, and this is a desired property for hard optimization problems.



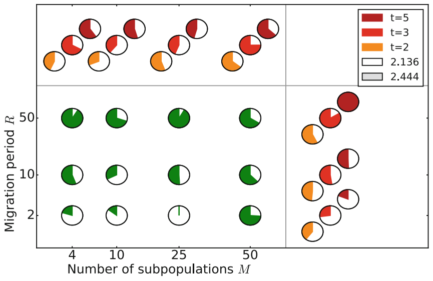
(a) Fitness: *velocity*
Selection: *Random* assignment



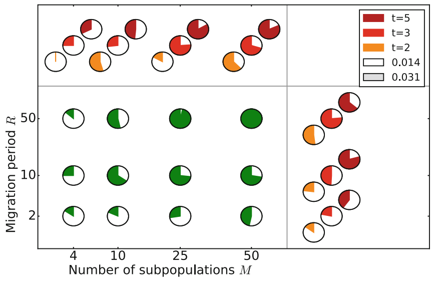
(b) Fitness: *height*
Selection: *Random* assignment



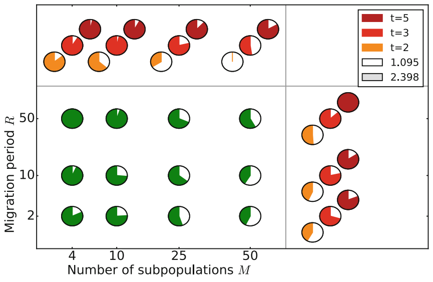
(c) Fitness: *velocity*
Selection: *EqualWidth* convection



(d) Fitness: *height*
Selection: *EqualWidth* convection



(e) Fitness: *velocity*
Selection: *EqualNumber* convection



(f) Fitness: *height*
Selection: *EqualNumber* convection

Fig. 3. Average best fitness values for each combination of parameter values after 10^6 evaluations, additionally averaged along each of the three dimensions (parameters). Empty circles represent the minimal fitness value present in each chart, and full circles represent the maximal fitness value in each chart. The minimal and maximal fitness values are shown in the legend.

The number of subpopulations M has a different effect on fitness values for each fitness function. For *velocity* (Fig. 3e), increasing the number of subpopulations (and hence reducing their size) has a positive effect on the quality of results, which is indicated by the circles filling up along the horizontal axis, while for *height* (Fig. 3f) and for early evolution of *velocity* (around the first 50k evaluations) it has a negative effect. While it is not clear what causes this difference, one possible explanation is related to different properties of these fitness functions, as demonstrated in Fig. 2. While the *velocity* criterion allows the algorithm to continuously improve the quality of solutions by exploring new ideas of "how to be fast" (i.e., more possibilities for exploration), the evolution of *height* quickly leads to a plateau, where the improvement can be achieved mostly by fine-tuning of existing solutions ("local optima") that are easy to break.

Although no obvious trends are visible for the $EqualWidth$ selection scheme (Fig. 3c and d), it is worth noting that the combination of a small number of big subpopulations and frequent migrations is unfavorable for both fitness functions, as indicated by primarily empty circles in the bottom-left part of these plots. The most likely explanation of this is the low level of exploration that results from such parametrization.

4 Conclusions

In this article, we investigated the concept of convection distribution and convection selection [11] in single-threaded (non-parallel) evolutionary algorithms and demonstrated that dividing the population into subpopulations based on fitness values of individuals can significantly improve the quality of optimized solutions. We have discussed potential mechanisms responsible for superior results of the convection-based methods, the most important ones being the diversification of the population and the ability to constantly explore diverse paths in fitness landscape [11]. If many subpopulations are allowed to evolve independently for longer periods of time, we can expect that each of them will produce unique, fit solutions which can then compete and cooperate every time the subpopulations are merged.

There are a number of issues that should still be examined. Even though the experiments reported in this paper were computationally highly expensive due to a large number of combinations of parameter values and very complex evolutionary goals, it would be worthwhile to extend the ranges of parameters to test the space of possible parameter combinations more comprehensively. It would be advantageous to test the proposed approaches on more fitness functions, including well-known benchmark optimization problems. Apart from convection selection and random assignment of individuals to subpopulations, we would like to additionally test the policy that ensures the best individual is placed in each subpopulation. For larger populations, the convection selection may have multiple levels so that it is applied recursively and subpopulations are nested in each other – this concept is worth testing too, along with dynamic, adaptive strategies of splitting and merging subpopulations and recursion levels. It would be useful to devise a formal statistical model behind convection selection to understand its mechanisms and causes of its success. Finally, it will be interesting to investigate to what extent can crossover benefit from convection-based schemes where fitness of parents is in most cases similar.

Acknowledgments. The research presented in the paper received support from Polish National Science Center (DEC-2013/09/B/ST10/01734).

References

1. Back, T.: Selective pressure in evolutionary algorithms: a characterization of selection mechanisms. In: Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, pp. 57–62. IEEE (1994)
2. Back, T.: Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms. Oxford University Press, Oxford (1996)
3. Blickle, T., Thiele, L.: A mathematical analysis of tournament selection. In: ICGA, pp. 9–16. Citeseer (1995)
4. Blickle, T., Thiele, L.: A comparison of selection schemes used in evolutionary algorithms. Evol. Comput. **4**(4), 361–394 (1996)
5. Črepinšek, M., Liu, S.H., Mernik, M.: Exploration and exploitation in evolutionary algorithms: a survey. ACM Comput. Surv. (CSUR) **45**(3), 35 (2013)
6. Dasgupta, D., Michalewicz, Z.: Evolutionary Algorithms in Engineering Applications. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-662-03423-1
7. Goldberg, D.E., Deb, K.: A comparative analysis of selection schemes used in genetic algorithms. In: Foundations of Genetic Algorithms, vol. 1, pp. 69–93 (1991)
8. Komosinski, M., Rotaru-Varga, A.: Comparison of different genotype encodings for simulated 3D agents. Artif. Life J. **7**(4), 395–418 (2001)
9. Komosinski, M., Ulatowski, S.: Framsticks: creating and understanding complexity of life, chap. 5. In: Komosinski, M., Adamatzky, A. (eds.) Artificial Life Models in Software, 2nd edn, pp. 107–148. Springer, London (2009). https://doi.org/10.1007/978-1-84882-285-6_5
10. Komosinski, M., Ulatowski, S.: Framsticks web site (2016). http://www.framsticks.com
11. Komosinski, M., Ulatowski, S.: Multithreaded computing in evolutionary design and in artificial life simulations. J. Supercomput. **73**(5), 2214–2228 (2017). http://www.framsticks.com/files/common/MultithreadedEvolutionaryDesign.pdf
12. Luque, G., Alba, E., Dorronsoro, B.: Parallel genetic algorithms. In: Parallel Metaheuristics: A New Class of Algorithms, pp. 107–126 (2005)
13. Miller, B.L., Goldberg, D.E.: Genetic algorithms, tournament selection, and the effects of noise. Complex Syst. **9**(3), 193–212 (1995)
14. Sastry, K., Goldberg, D.E., Kendall, G.: Genetic algorithms. In: Burke, E., Kendall, G. (eds.) Search Methodologies, pp. 93–117. Springer, Heidelberg (2014). https://doi.org/10.1007/978-1-4614-6940-7_4
15. Spears, W.M.: Simple subpopulation schemes. In: Proceedings of the Evolutionary Programming Conference, vol. 3, pp. 296–307. World Scientific River Edge, NJ (1994)
16. Sudholt, D.: Parallel evolutionary algorithms. In: Kacprzyk, J., Pedrycz, W. (eds.) Springer Handbook of Computational Intelligence, pp. 929–959. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-43505-2_46
17. Syswerda, G.: A study of reproduction in generational and steady state genetic algorithms. In: Foundations of Genetic Algorithms, vol. 2, pp. 94–101 (1991)
18. Tomassini, M.: Parallel and distributed evolutionary algorithms: a review (1999)

Multi-agent Systems Programmed Visually with Google Blockly

Szymon Górowski, Robert Maguda, and Paweł Topa(✉)

Department of Computer Science, AGH University of Science and Technology, Kraków, Poland
topa@agh.edu.pl

Abstract. In this paper we propose a very user-friendly method for programming multi-agent systems. We use the well known visual programming library Blockly from Google. With this library the behaviour of agents can by programmed intuitively even by those not skilled in programming. We demonstrate this idea using an agent-based simulator named eVolutus, designed and implemented for conducting large scale ecological and evolutionary experiments.

Keywords: Agent-based programming · Multi-agent systems
Visual programming

1 Introduction

Computer science provides methods and tools that are used by other scientific disciplines: physics, chemistry, geology, biology and so on. In order to make designing and implementing such tools efficient and fast, many sophisticated programming languages and programming libraries have been invented and developed. From the programmer's point of view they are more reliable and convenient than old-fashioned languages like Fortran or C. Unfortunately, they still require high levels of skills and experience from programmers.

Users usually expect that computer programs have interfaces optimized for the tasks they want to perform. In some cases the functionality of programs can be enclosed within menus, dialogues and other components of a typical GUI. In many applications, such an approach is insufficient, and various scripting languages optimized for specific tasks are provided.

One of the main problems for inexperienced users of scripting languages are syntax errors that may appear in the script due to poor knowledge of instructions or as a result of typos. In such the cases, the visual programming technique can be useful. Visual Programming Languages (VPLs) [5] work like Lego bricks—programs are constructed using components that can be connected only in particular, permitted way.

VPLs have long history, and they have been applied with success in software for data processing and visualisation (see [4]). In such programs the data flow model of programming is especially useful: data sets are processed by a chain of "black boxes".

Currently, VPLs are often used for educational purposes. Scratch [8] is one of the most popular platforms that use visual programming for teaching basic programming concepts. It is freely available and used in many projects and applications, e.g., it is used to program the famous robotics suite Lego Wedo. Another example is Blockly—an open source library developed by Google. It is used in the application presented in this paper and we introduce it in more detail below.

There are also several examples of using VPLs in Agent-Based Modelling (ABM). The SeSAm [7] platform allows for specifying the complete model of a given phenomena. The behaviour of agents is defined by activity diagrams similar to UML diagrams. Analogously, the user defines the environment, its state and evolution. Diagrams are also used to define interactions between agents and their reactions to the environmental conditions. Plugins can be used to extend SeSAM's functionality. Before starting the simulation, SeSAm generates the code of the simulator to achieve higher efficiency.

VPL are also used by DeltaTick [13], which is an extension to NetLogo [12], one of the most popular ABM computational environments. In order to a build model, the user creates actors and assigns them actions (behaviour) selected from an existing collection. Such an approach is very convenient and makes it possible to build a complete model very fast. On the other hand, it is not as flexible as SeSAm, where a user can define their own low-level procedures.

The project demonstrated in this paper was created as an extension of eVolutus—an agent based simulator of ecology and evolution [9]. eVolutus was designed and developed for microbiologists that investigate Foraminifera [6], single-cellular marine microogranisms. This simulator makes it possible to program the behaviour of a single agent using short functions named kernels. We believe that Blockly can support scientists that do not have programming skills in conducting sophisticated experiments with eVolutus.

The paper is organized as follows: In the next section the eVolutus simulator and its crucial functionality, behaviour and environmental kernels, are briefly introduced. Next, we describe the Oracle Nashorn framework, which allows for implementing the kernels' functionality. The next section discusses the Blockly library. Sect. 3 contains detailed information on using Blockly to program eVolutus agents. Next, we present some examples of kernels generated using this software. At the end, we summarize the presented results.

2 eVolutus: The Simulator of Multiscale Evolutionary Processes Tested on Foraminifera

eVolutus [9] is the simulator of population dynamics and evolutionary processes designed using the Individual Based Modelling (IBM) paradigm [11]. This method assumes that a population of individuals (animals, pedestrians) is represented in a computer model as a set of entities. Each entity has its own state and its behaviour is governed by a set of rules. Global behaviour of population emerges as a result of interactions between entities and interactions with environmental conditions. A model based on the IBM paradigm can be implemented in natural way using the Agent-Based Modelling (ABM) approach [12].

One of the main advantages of IBM/ABM methodology is a bottom-up approach to building the model. The modeller focuses on the behaviour of a single agent, its survival strategy, as well as its reactions to its environment and to other agents. These rules can be easily encoded as a set of procedures that the agent calls when the related events are occurring.

When a model based on the IBM/ABM approach is developed, tested, verified and calibrated, most modifications are related to the agents' behaviour. Thus, it is reasonable to construct model with clearly distinguished section of code that is executed by agent in each step of simulation. Later, when the model is tuned, only these parts of the code are changed. Moreover, when the simulator is used by an experimenter who tries to encode their own knowledge related to the modelled phenomena, any modifications can be limited to only this part of code. In case of any problems with execution it is easier to detect the location of errors.

Currently, the eVolutus is able to model populations of Foraminifera [6], a large group of marine single cellular microorganisms, however its structure allows for a relatively simple change the modelled object.

In eVolutus each individual is represented by an agent that has its own parameters and rules of behaviour. The Foraminiferal habitat is represented in a manner similar to the Cellular Automata approach. The whole space is partitioned using a regular grid. Each cell represents a portion of the habitat of a given size. In the computer model each cell is described by several numerical values that correspond to various environmental factors,e.g., insolation, depth, temperature, salinity. During the simulation these parameters may change according to defined rules.

2.1 The eVolutus Kernels

Both the behaviour of agents and the evolution of the habitat are governed by functions which are invoked at each step of simulation for each agent. These short functions have been named "kernels" by analogy with the CUDA kernels. i.e., relatively short procedures processed by the cores of the GPU (Graphics Processing Unit) in a massive parallel manner (hundreds of threads execute the same stream of instructions at the same time). Here, the kernels also contain, usually short, sequence of instructions that have to be processed for thousands of agents, potentially in a parallel manner.

eVolutus is created for users that are not skilled programmers. Thus, the main goal of this project is to provide an interface which is simple but does not limit the experimenter. The functionality supplied by the Nashorn framework [3] partially fulfils these requirements. Users do not have to use Java programming language, which is relatively difficult to use. Instead, the Javascript scripting language is used which has a much more simple syntax. Nashorn framework is able to execute Javascript code on JVM with the same performance as native Java code. However, this approach is still prone to syntax errors. Finding and correcting these errors may be a problem for inexperienced users.

One solution could be a visual programming approach. Using this method, user constructs programs or algorithms using graphical blocks that represent various instructions (see Fig. 1). We assume that the implementer of the simulator has to create a template which contains all the information necessary to properly generate and handle blocks. Such templates can be used to generate the stubs of kernels which will be filled with instructions by the experimenter.

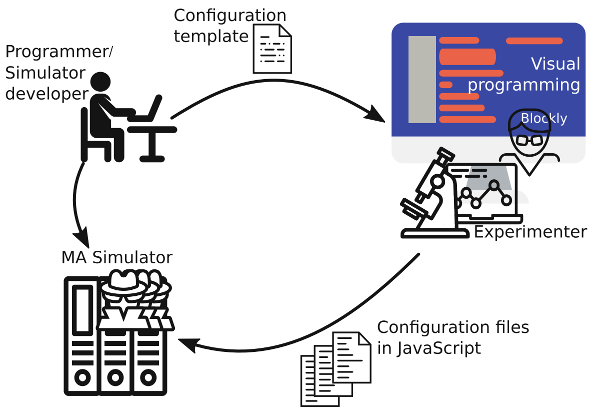


Fig. 1. Visual programming for multi-agent modelling using Google Blockly

The eVolutus kernels are the functions that determine the behaviour of agents in various situations:

– gathering food,
– moving,
– reproduction,
– shortage of food/energy etc.

The sample kernel presented below evaluates a decision about growth (more on the model of Foraminifera morphogenesis in [10]):

```
function canCreateChamber(envState, foramState, time) {
    var volumeOfCytoplasm    =
        foramState.energy/foramState.genotype.metabolicEffectiveness[0];
    var needMoreSpace        =
        volumeOfCytoplasm > 0.95 * foramState.shell.volumeShell;
    var energyEnough         =
        foramState.energy > energyNeededForGrowth(envState,foramState,time);
    return  needMoreSpace && energyEnough
        && !isInHibernationState(envState,foramState,time);
}
```

In this function, the current volume of cytoplasm is calculated using the current level of energy and a parameter that describes metabolic effectiveness of the individual (a very detailed explanations of the model of Foraminifera physiology applied in eVolutus can be found in [9]). Next instruction check whether new Foraminifera should growth and calculate necessary amount of energy. Function returns logical value which is used to turn on the procedure of growing.

Additionally, a separate group of kernels is used to govern the evolution of the habitat (these are called environmental kernels). Each cell of habitat changes its state according to the calculations made in the appropriate kernels, e.g., insolation inside a box of water may be controlled using the following code:

```
function insolation(time, envStates) {
    var surfaceInsolation = 1.0;
    var light = surfaceInsolation − 10 ∗ envStates[0].position.z;
    return Math.max(0.0, light);
}
```

Here, it is assumed that at the water's surface the level is at its maximum and that it decreases with depth.

2.2 Google Blockly

Blockly [2] is a library written using JavaScript that allows visual programming. At this moment it is supported by most web browsers. Since 2011 it is developed by Google, but its source code is publicly available as an open-source project.

Blockly has a defined set of graphic components (blocks) with basic programming instructions. Those blocks can be connected (in a specified manner) to compound more complex instructions or sequences of instructions. Blockly is able to generate code in JavaScript, Python, PHP, Lua and Dart.

Blockly is widely used in many projects, especially those that are targeted to support students who are learning programming. It can be also used by users that are not skilled programmers. *Blockly Games*[1] is a publicly available portal devoted to teaching children the basic ideas of programming, e.g. turtle graphics. Another example of using Blockly is App Inventor for Android [1] which is a web-based platform for creating programs for mobile devices running on Android.

One of the main advantages of Blockly is the licence that allows it to be freely modified and extended. Thus, we are able to tune the project to provide its users the set of blocks that they need for this particular application.

3 Implementation

The software has been intentionally designed not to be strictly dependent on the architecture of the eVolutus simulator. Thus, it was assumed that the function stubs must be generated using templates. The developer of the simulator has to provide files that describe all the functions and data structures that are interpreted by Nashorn during computation. Kernels are defined using the following format:

[1] https://blockly-games.appspot.com/.

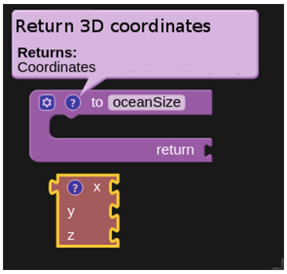
```
// @description description of function
// @param argName1 type description
// @param argName2 type description
// @param ...
// @return type description
func name argName1 argName2 [...]
```

Field `description` is used to include short information about defined function. Fields `param` describe the name and type of the function arguments, they can be also supplied with short explanation. Finally, the returned value is specified in an analogous way. In a similar manner we can define new data structures:

```
// @description struct description
// @outputField x Number field description
// @outputField y Number field description
// @outputField z Number field description
struct output coordinates x y z
```

The code below contains a real template of a function which returns direction and speed of ocean current in a single habitat cell. This definition assumes that functions have to return an array with three numbers (coordinates). Thus we have to define a block named `Coordinates` which obviously represents coordinates in 3D.

```
// @description Return 3D coordinates
// @param time Number Timestep
// @param envStates Environment state
// @return Coordinates
func currentDirection time envStates


// @description Points or vectors in 3D
// @outputField x Number x-coordinate
// @outputField y Number y-coordinate
// @outputField z Number z-coordinate
struct output coordinates x y z
```

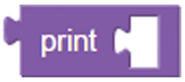


As already mentioned, the software generates dedicated blocks for each data structure defined in the template. For this purpose, a Blockly mechanism was used to create custom blocks. Blockly blocks consist of 3 components:

- Block Definition Object - an object that defines the appearance and behaviour of the block, including its fields and connections
- Toolbox Reference - a reference to the block type in XML describing the content of the toolbox (to allow the user to add it to the workspace)
- Generator Function - a function that generates block code in a given programming language

Creating a new type of block requires implementing these components and adding them to the script files from which Blockly loads the blocks (resulting block is included):

```
Blockly.Blocks['print'] = {
  init: function() {
    this.appendValueInput("text").setCheck(null).
        appendField("print");
    this.setInputsInline(true);
    this.setOutput(true, null);
    this.setColour(270);
    this.setTooltip('');
    this.setHelpUrl('');
  }
};
```



```
Blockly.JavaScript['print'] = function(block) {
  var value_text =
      Blockly.JavaScript.valueToCode(block, 'text',
          Blockly.JavaScript.ORDER_ATOMIC);
  var code = 'window.alert(' + value_text + ')';
  return [code, Blockly.JavaScript.ORDER_NONE];
};
```

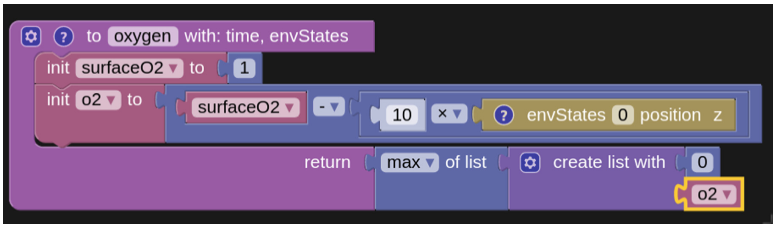
In the application, new types of blocks are created dynamically each time a template is loaded, because each template can define different data structures for which different types of blocks are needed. *Blockly Developer Tools*[2] make creating new block types significantly easier. They are web-based tools that allow for building one's own blocks using the graphical interface and generate Block Definition Objects for them in Blockly format.

4 Sample Scripts

The platform is available for users in the most convenient way, i.e., as a web service[3]. The user has their own account with a database of templates. New templates can be uploaded to the server. After choosing the template, a set of empty functions and corresponding blocks is generated and visualised. Now, the user can fill the functions with instructions and data structures. In addition to the standard blocks, the available set has additional ones generated using the templates. At any moment, the current state of work can be saved to continue development later.

In Figs. 2 and 3 we present sample diagrams created using the platform as well as the resulting code.

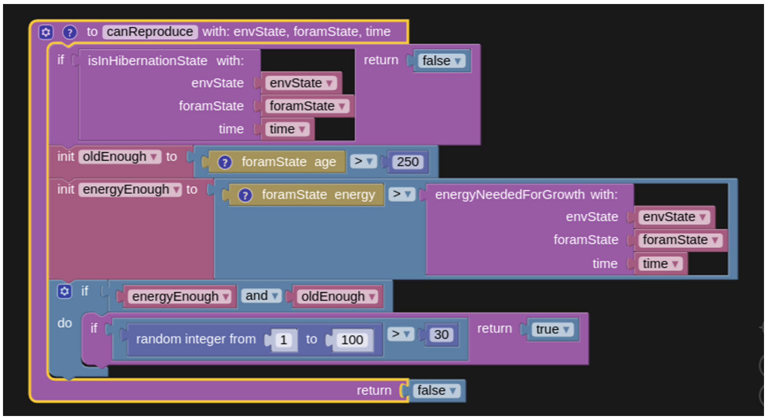
[2] https://blockly-demo.appspot.com/static/demos/blockfactory/index.html.
[3] https://mongo.icsr.agh.edu.pl/ace.


to oxygen with: time, envStates
init surfaceO2 to 1
init o2 to surfaceO2 - 10 × envStates 0 position z
return max of list create list with 0
o2

```
function oxygen(time, envStates) {
  var surfaceO2 = 1;
  var o2 = surfaceO2 − 10 ∗ envStates[0].position.z;
  return Math.max.apply(null, [0, o2]);
}
```

```
def oxygen(time, envStates):
  surfaceO2 = 1
  o2 = surfaceO2 − 10 ∗ envStates[0].position.z
  return max([0, o2])
```

Fig. 2. Visually programmed environmental kernel for oxygene distribution inside the modelled habitat. Below, the Javascript code accepted by eVolutus simulator as well as the the equivalent code in Python.


to canReproduce with: envState, foramState, time
if isInHibernationState with: envState envState foramState foramState time time
return false
init oldEnough to foramState age > 250
init energyEnough to foramState energy > energyNeededForGrowth with: envState envState foramState foramState time time
if energyEnough and oldEnough
do if random integer from 1 to 100 > 30
return true
return false

```
function canReproduce(envState, foramState, time) {
  if (isInHibernationState(envState, foramState, time))
    return false;
  var oldEnough = foramState.age > 250;
  var energyEnough =
      foramState.energy > energyNeededForGrowth(envState, foramState,
          time);
  if (energyEnough && oldEnough) {
    if (mathRandomInt(1, 100) > 30)
      return true;
  }
  return false;
}
```

Fig. 3. Behavioural kernel which controls the moment of reproduction. Agent cannot be in hibernation state, it must be mature enough and have the necessary amount of energy. Reproduction is triggered with some probability.

5 Conclusions

Simulators based on ABM paradigm are designed for users that are focused on solving some scientific and engineering problems. The architecture of computation and its implementation should be a black-box for them. The agent-based systems in such an application have a big advantage—the code that control agents behaviour can be isolated from the rest of the software and exposed to direct modification by the user. The ABM paradigm assumes that the solution to the problem can be found by mutual interactions of many agents, as well as by their interactions with the environment. Thus, the code that governs agents' behaviour is crucial for solving the defined problems.

Our platform provides a convenient method for programming agents in ABM-based software. The Blockly-based interface supports the user in creating code that is free of syntax errors. Moreover, the blocks that represent data structures and instructions can be provided with an extensive manual which leads the user through the process of programming the agents.

Acknowledgements. The work presented in this paper received support from the Polish National Science Centre (DEC-2013/09/B/ST10/01734).

References

1. App inventor for android. https://ai2.appinventor.mit.edu/
2. Google blockly. https://developers.google.com/blockly/
3. Nashorn official website. http://blogs.oracle.com/nashorn/
4. Boshernitsan, M., Downes, M.S.: Visual programming languages: a survey. Technical report, University of California, Berkeley, USA (2004)
5. Erwig, M., Smeltzer, K., Wang, X.: What is a visual language? J. Vis. Lang. Comput. **38**, 9–17 (2017). SI:In honor of Prof. S.K. Chang
6. Goldstein, S.T.: Foraminifera: A Biological Overview. Kluwer Academic Publishers, Dordrecht (1999)
7. Klügl, F.: Sesam: visual programming and participatory simulation for agent-based models. In: Multi-agent Systems: Simulation and Applications, pp. 477–508 (2009)
8. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al.: Scratch: programming for all. Commun. ACM **52**(11), 60–67 (2009)
9. Topa, P., Faber, Ł., Tyszka, J., Komosinski, M.: Modelling ecology and evolution of Foraminifera in the agent-oriented distributed platform. J. Comput. Sci. **18**, 69–84 (2017)
10. Tyszka, J., Topa, P.: A new approach to modeling of Foraminiferal shells. Paleobiology **31**(30), 526–541 (2005)
11. Uchmański, J., Grimm, V.: Individual-based modelling in ecology: what makes the difference? Trends Ecol. Evol. **11**(10), 437–441 (1996)
12. Wilensky, U., Rand, W.: An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo. MIT Press, Cambridge (2015)
13. Wilkerson-Jerde, M.H., Wilensky, U.: Restructuring change, interpreting changes: the deltatick modeling and analysis toolkit. In: Proceedings of Constructionism (2010)

Author Index