Kristján Jónasson (Ed.)

LNCS 7133

Applied Parallel and Scientific Computing

10th International Conference, PARA 2010
Reykjavík, Iceland, June 2010
Revised Selected Papers, Part I

1 Part I

Springer

Kristján Jónasson (Ed.)

Applied Parallel and Scientific Computing

10th International Conference, PARA 2010
Reykjavík, Iceland, June 6-9, 2010
Revised Selected Papers, Part I

Springer

Volume Editor

Kristján Jónasson
University of Iceland
School of Engineering and Natural Sciences
Department of Computer Science
Hjardarhagi 4, 107 Reykjavík, Iceland
E-mail: jonasson@hi.is

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-28150-1 e-ISBN 978-3-642-28151-8
DOI 10.1007/978-3-642-28151-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012930011

CR Subject Classification (1998): G.1-4, F.1-2, D.1-3, J.1, C.2

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

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

Printed on acid-free paper

Preface

The tenth Nordic conference on applied parallel computing, Para 2010: State of the Art in Scientific and Parallel Computing, was held in Reykjavík, Iceland during June 6–9, 2010. The topics of the conference were announced to include software, hardware, algorithms, tools, environments, as well as applications of scientific and high-performance computing. The conference was hosted by the School of Engineering and Natural Sciences of the University of Iceland, and the conference venue was in the School of Education of the University of Iceland. Three companies in Reykjavík supported the conference financially: the video game developer CCP, Microsoft Íslandi, and Opin kerfi (Hewlett Packard distributor for Iceland).

The series of Para meetings began in 1994. The Danish Computing Centre for Research and Education (UNI-C) and the Department of Informatics and Mathematical Modelling of the Technical University of Denmark (IMM/DTU) in Lyngby, Denmark, organized a series of workshops on Applied Parallel Computing, named Para94, Para95 and Para96. Jerzy Waśniewski, senior researcher at DTU, initiated these workshops and Jack Dongarra, professor at the University of Tennessee, became involved during an extended visit to Lyngby. He played a key part in promoting the meetings internationally. Since 1998, the workshops have become a Nordic effort, but both Jerzy and Jack have continued to be an integral part of the meetings. In fact Jerzy has been a keen advocate of holding a Para conference in Iceland. The themes and locations of the Para meetings have been:

PARA94 Parallel Scientific Computing, Lyngby, Denmark
PARA95 Physics, Chemistry and Engineering Science, Lyngby, Denmark
PARA96 Industrial Problems and Optimization, Lyngby, Denmark
PARA 1998 Large Scale Scientific and Industrial Problems, Umeå, Sweden
PARA 2000 New Paradigms for HPC in Industry and Academia, Bergen, Norway
PARA 2002 Advanced Scientific Computing, Helsinki, Finland
PARA 2004 State of the Art in Scientific Computing, Copenhagen, Denmark
PARA 2006 State of the Art in Scientific and Parallel Computing, Umeå, Sweden
PARA 2008 State of the Art in Scientific and Parallel Computing, Trondheim, Norway
PARA 2010 State of the Art in Scientific and Parallel Computing, Reykjavík, Iceland

The Para 2010 conference included five keynote lectures, one tutorial, 11 minisymposia consisting of a total of 90 presentations, 39 other contributed presentations organized under 10 separate topics, four poster presentations, and eight presentations from industry. Except for the keynote lectures, that were 45 minutes long each, the presentations were organized in five tracks or parallel streams, with 25-minute slots for each presentation, including discussion. The total number of presentations was thus 147. There were altogether 187 participants from 20 countries:

Denmark 9	Canada 1	Poland 16
Finland 4	Czech Republic 3	Russia 2
Iceland 38	France 12	Spain 7
Norway 13	Germany 32	Switzerland 1
Sweden 17	Italy 1	Turkey 1
Australia 2	Japan 4	USA 20
Austria 2	Netherlands 2	

There were volcanic eruptions in Eyjafjallajökull in southern Iceland from March until June 2010 disrupting international flights, and these may have had an adverse effect on participation.

Extended abstracts (in most cases four pages long) of all the minisymposium and contributed presentations were made available on the conference website, http://vefir.hi.is/para10, and in addition a book of short abstracts (also available on the website) was handed out at the conference.

After the conference the presentation authors were invited to submit manuscripts for publication in these peer-reviewed conference proceedings. The reviewing process for the articles appearing here was therefore performed in two stages. In the first stage the extended abstracts were reviewed to select contributions to be presented at the conference, and in the second stage the full papers submitted after the conference were reviewed. As a general rule three referee reports per paper were aimed for, and in most cases these were successfully obtained. However, in cases where it proved difficult to find three willing referees, acquiring only two reports was deemed acceptable.

Fred G. Gustavson, emeritus scientist at IBM Research, New York, and professor at Umeå University, and Jerzy Waśniewski gave a tutorial on matrix algorithms in the new many core era. Fred celebrated his 75th birthday on May 29, 2010, and the Linear Algebra Minisymposium was held in his honor. The material of the tutorial is covered in Fred Gustavson's article in these proceedings.

A conference of this size requires considerable organization and many helping hands. The role of the minisymposium organizers was very important. They reviewed and/or organized reviewing of contributions to their respective minisymposia, both the original extended abstracts and the articles for these proceedings, and in addition they managed the minisymposium sessions at the conference. Several members of the local Organizing Committee helped with the reviewing of other contributed extended abstracts: Elínborg I. Ólafsdóttir, Hjálmtýr Hafsteinsson, Klaus Marius Hansen, Ólafur Rögnvaldsson, Snorri Agnarsson and Sven Þ. Sigurðsson. Other colleagues who helped with this task were Halldór Björnsson, Kristín Vogfjörð and Viðar Guðmundsson.

The editor of these proceedings organized the reviewing of manuscripts falling outside minisymposia, as well as manuscripts authored by the minisymposium organizers themselves. There were 56 such submissions. The following people played a key role in helping him with this task: Sven Þ. Sigurðsson, Julien

Langou, Bo Kågström, Sverker Holmgren, Michael Bader, Jerzy Waśniewski, Klaus Marius Hansen, Kimmo Koski and Halldór Björnsson. Many thanks are also due to all the anonymous referees, whose extremely valueable work must not be forgotten.

The conference bureau Your Host in Iceland managed by Inga Sólnes did an excellent job of organizing and helping with many tasks, including conference registration, hotel bookings, social program, financial management, and maintaining the conference website. Apart from Inga, Kristjana Magnúsdóttir of Your Host was a key person and Einar Samúelsson oversaw the website design. Ólafía Lárusdóttir took photographs for the conference website. The baroque group Custos and the Tibia Trio, both led by recorder player Helga A. Jónsdóttir, and Helgi Kristjánsson (piano) provided music for the social program. Ólafur Rögnvaldsson helped to secure financial support from industry. Jón Blöndal and Stefán Ingi Valdimarsson provided valuable TeX help for the editing of the proceedings.

Finally, I wish to devote a separate paragraph to acknowledge the help of my colleague Sven Þ. Sigurðsson, who played a key role in helping with the conference organization and editing of the proceedings through all stages.

October 2011 Kristján Jónasson

Organization

PARA 2010 was organized by the School of Engineering and Natural Sciences of the University of Iceland.

Steering Committee

Jerzy Waśniewski, Chair, Denmark
Kaj Madsen, Denmark
Anne C. Elster, Norway
Petter Bjørstad, Norway
Hjálmtýr Hafsteinsson, Iceland
Kristján Jónasson, Iceland
Juha Haatja, Finland
Kimmo Koski, Finland
Björn Engquist, Sweden
Bo Kågström, Sweden
Jack Dongarra, Honorary Chair, USA

Local Organizing Committee

Kristján Jónasson, Chair
Sven Þ. Sigurðsson, Vice Chair
Ólafur Rögnvaldsson, Treasurer
Ari Kr. Jónsson
Ebba Þóra Hvannberg
Elínborg Ingunn Ólafsdóttir
Hannes Jónsson
Helmut Neukirchen
Hjálmtýr Hafsteinsson
Jan Valdman
Klaus Marius Hansen
Sigurjón Sindrason
Snorri Agnarsson
Tómas Philip Rúnarsson

Sponsoring Companies

CCP, Reykjavík – video game developer
Microsoft Íslandi, Reykjavík
Opin kerfi, Reykjavík – Hewlett Packard in Iceland

PARA 2010 Scientific Program

Keynote Presentations

Impact of Architecture and Technology for Extreme Scale on Software and Algorithm Design
Jack Dongarra, University of Tennessee and Oak Ridge National Laboratory

Towards Petascale for Atmospheric Simulation
John Michalakes, National Center for Atmospheric Research (NCAR), Boulder, Colorado

Algorithmic Challenges for Electronic-Structure Calculations
Risto M. Nieminen, Aalto University School of Science and Technology, Helsinki

Computational Limits to Nonlinear Inversion
Klaus Mosegaard, Technical University of Denmark

Efficient and Reliable Algorithms for Challenging Matrix Computations Targeting Multicore Architectures and Massive Parallelism
Bo Kågström, Umeå University

Tutorial

New Algorithms and Data Structures for Matrices in the Multi/Many Core Era
Fred G. Gustavson, Umeå University and Emeritus Scientist at IBM Research, New York, and *Jerzy Waśniewski*, Danish Technical University

General Topics

Cloud Computing (1 presentation)
HPC Algorithms (7 presentations and 1 poster)
HPC Programming Tools (4 presentations)
HPC in Meteorology (3 presentations)
Parallel Numerical Algorithms (8 presentations and 1 poster)
Parallel Computing in Physics (2 presentations and 1 poster)
Scientific Computing Tools (10 presentations)
HPC Software Engineering (2 presentations and 1 poster)
Hardware (1 presentation)
Presentations from Industry (8 presentations)

Minisymposia

Simulations of Atomic Scale Systems (15 presentations)
Organized by *Hannes Jónsson*, University of Iceland

Tools and Environments for Accelerator-Based Computational Biomedicine (6 presentations)
Organized by *Scott B. Baden*, University of California, San Diego

GPU Computing (9 presentations)
Organized by *Anne C. Elster*, NTNU, Trondheim

High-Performance Computing Interval Methods (6 presentations)
Organized by *Bartlomiej Kubica*, Warsaw University of Technology

Real-Time Access and Processing of Large Data Sets (6 presentations)
Organized by *Helmut Neukirchen*, University of Iceland and *Michael Schmelling*, Max Planck Institute for Nuclear Physics, Heidelberg

Linear Algebra Algorithms and Software for Multicore and Hybrid Architectures, in honor of Fred Gustavson on his 75th birthday (10 presentations)
Organized by *Jack Dongarra*, University of Tennessee and *Bo Kågström*, Umeå University

Memory and Multicore Issues in Scientific Computing – Theory and Practice (6 presentations)
Organized by *Michael Bader*, Universität Stuttgart and *Riko Jacob*, Technische Universität München

Multicore Algorithms and Implementations for Application Problems (9 presentations)
Organized by *Sverker Holmgren*, Uppsala University

Fast PDE Solvers and A Posteriori Error Estimates (8 presentations)
Organized by *Jan Valdman*, University of Iceland and *Talal Rahman*, University College Bergen

Scalable Tools for High-Performance Computing (12 presentations)
Organized by *Luiz DeRose*, Cray Inc. and *Felix Wolf*, German Research School for Simulation Sciences

Distributed Computing Infrastructure Interoperability (4 presentations)
Organized by *Morris Riedel*, Forschungszentrum Jülich

Speakers and Presentations

For a full list of authors and extended abstracts, see http://vefir.hi.is/para10.

Abrahamowicz, Michal: Alternating conditional estimation of complex constrained models for survival analysis
Abramson, David: Scalable parallel debugging: Challenges and solutions
Agnarsson, Snorri: Parallel programming in Morpho
Agullo, Emmanuel: Towards a complexity analysis of sparse hybrid linear solvers
Aliaga, José I.: Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors
Anzt, Hartwig: Mixed precision error correction methods for linear systems – Convergence analysis based on Krylov subspace methods
Aqrawi, Ahmed Adnan: Accelerating disk access using compression for large seismic datasets on modern GPU and CPU
Arbenz, Peter: A fast parallel poisson solver on irregular domains
Bader, Michael: Memory-efficient Sierpinski-order traversals on dynamically adaptive, recursively structured triangular grids
Bartels, Soeren: A posteriori error estimation for phase field models
Belsø, Rene: Structural changes within the high-performance computing (HPC) landscape
Bientinesi, Paolo: The algorithm of multiple relatively robust representations for multicore processors
Bjarnason, Jón: Fighting real time – The challenge of simulating large-scale space battles within the Eve architecture
Blaszczyk, Jacek Piotr: Aggregated pumping station operation planning problem (APSOP) for large-scale water transmission system
Bohlender, Gerd: Fast and exact accumulation of products
Borkowski, Janusz: Global asynchronous parallel program control for multicore processors
Bozejko, Wojciech: Parallelization of the tabu search algorithm for the hybrid flow shop problem
Breitbart, Jens: Semiautomatic cache optimizations using OpenMP
Brian J. N. Wylie: Performance engineering of GemsFDTD computational electromagnetics solver
Britsch, Markward: The computing framework for physics analysis at LHCb
Brodtkorb, André R.: State of the art in heterogeneous computing
Buttari, Alfredo: Fine granularity sparse QR factorization for multicore-based systems
Cai, Xiao-Chuan: A parallel domain decomposition algorithm for an inverse problem in elastic materials
Cai, Xing: Detailed numerical analyses of the Aliev-Panfilov model on GPGPU
Cambruzzi, Sandro: The new features of Windows HPC Server 2008 V3 and Microsoft's HPC strategy
Cankur, Reydan: Parallel experiments on PostgreSQL (poster)
Casas, Marc: Multiplexing hardware counters by spectral analysis

Cheverda, Vladimir A.: Simulation of seismic waves propagation in multiscale media: Impact of cavernous/fractured reservoirs

Cheverda, Vladimir A.: Parallel algorithm for finite difference simulation of acoustic logging

Contassot-Vivier, Sylvain: Impact of asynchronism on GPU accelerated parallel iterative computations

Cytowski, Maciej: Analysis of gravitational wave signals on heterogeneous architecture

Danek, Tomasz: GPU accelerated wave form inversion through Monte Carlo sampling

Davidson, Andrew: Toward techniques for auto-tuning GPU algorithms

DeRose, Luiz: Automatic detection of load imbalance

Doll, Jimmie D.: Recent developments in rare-event Monte Carlo methods

Dongarra, Jack: Impact of architecture and technology for extreme scale on software and algorithm design (keynote lecture)

Dongarra, Jack: LINPACK on future manycore and GPU-based systems

Dubcova, Lenka: Automatic hp-adaptivity for inductively heated incompressible flow of liquid metal

Einarsdóttir, Dóróthea M.: Calculation of tunneling paths and rates in systems with many degrees of freedom

Ekström, Ulf Egil: Automatic differentiation in quantum chemistry

Elster, Anne C.: Current and future trends in GPU computing

Elster, Anne C.: Visualization and large data processing – State of the art and challenges

Fjukstad, Bård: Interactive weather simulation and visualization on a display wall with manycore compute nodes

Fujino, Seiji: Performance evaluation of IDR(s)-based Jacobi method

Gagunashvili, Nikolai: Intellectual data processing for rare event selection using a RAVEN network

Gepner, Pawel: Performance evaluation of Intel® Xeon® 7500 family processors for HPC

Gerndt, Michael: Performance analysis tool complexity

Gjermundsen, Aleksander: LBM vs. SOR solvers on GPU for real-time fluid simulations

Goerling, Andreas: Novel density-functional methods for ground and excited states of molecules and first steps towards their efficient implementation

Greiner, Gero: Evaluating non-square sparse bilinear forms on multiple vector pairs in the I/O-model

Gross, Lutz: Algebraic upwinding with flux correction in 3D numerical simulations in geosciences

Guðjónsson, Halldór Fannar: HPC and the Eve cluster game architecture

Gustafsson, Magnus: Communication-efficient Krylov methods for exponential integration in quantum dynamics

Gustavson, Fred G.: New Algorithms and data structures for matrices in the multi/manycore era, parts 1, 2, 4 (tutorial)

Gustavson, Fred G.: Enduring linear algebra
Henkelman, Graeme: Accelerating molecular dynamics with parallel computing resources
Hess, Berk: Molecular simulation on multicore clusters and GPUs
Holm, Marcus: Implementing Monte Carlo electrostatics simulations on heterogeneous multicore architectures
Jacobson, Emily R.: A lightweight library for building scalable tools
Jakl, Ondrej: Solution of identification problems in computational mechanics – Parallel processing aspects
Jenz, Domenic: The computational steering framework steereo
Jiang, Steve: GPU-based computational tools for online adaptive cancer radiotherapy
Jónsson, Kristján Valur: Using stackless python for high-performance MMO architecture
Kågström, Bo: Efficient and reliable algorithms for challenging matrix computations targeting multicore architectures and massive parallelism (keynote lecture)
Kamola, Mariusz: Software environment for market balancing mechanisms development, and its application to solving more general problems in parallel way
Karlsson, Lars: Fast reduction to Hessenberg form on multicore architectures
Khan, Malek Olof: Molecular simulations on distributed heterogeneous computing nodes
Kimpe, Dries: Grids and HPC: Not as different as you might think?
Kirschenmann, Wilfried: Multi-target vectorization with MTPS C++ generic library
Kjelgaard Mikkelsen, Carl Christian: Parallel solution of banded and block bidiagonal linear systems
Klüpfel, Peter: Minimization of orbital-density-dependent energy functionals
Knüpfer, Andreas: Rank-specific event tracing for MPI – Extending event tracing to cope with extreme scalability
Kraemer, Walter: High-performance verified computing using C-XSC
Kreutz, Jochen: Black-Scholes and Monte Carlo simulation on accelerator architectures
Krog, Øystein E.: Fast GPU-based fluid simulations using SPH
Kubica, Bartlomiej: Using the second-order information in Pareto-set computations of a multi-criteria problem
Kubica, Bartlomiej: Cache-oblivious matrix formats for computations on interval matrices
Köstler, Harald: Optimized fast wavelet transform utilizing a multicore-aware framework for stencil computations
Lacoursiere, Claude: Direct sparse factorization of blocked saddle point matrices
Langlois, Philippe: Performance evaluation of core numerical algorithms: A tool to measure instruction level parallelism

Langou, Julien: Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures

Langou, Julien: Choosing a reduction tree for communication avoiding QR

Lee, Joo Hong: A hybrid parallel programming model for biological systems simulation

Lieber, Matthias: Highly scalable dynamic load balancing in the atmospheric modeling system COSMO-SPECS+FD4

Luque, Emilio: Scalability and efficiency for SPMD applications on multicore clusters

Luque, Emilio: PAS2P tool, parallel application signature for performance prediction

Lysgaard, Steen: Computational analysis of the interaction of materials for energy storage with water (poster)

Malinen, Mika: The development of fully coupled simulation software by reusing segregated solvers

Mallett, Jacky: Challenges in MMO scaling

Marciniak, Andrzej: An interval version of the Crank-Nicolson method – The first approach

Maronsson, Jón Bergmann: Elastic band of dimers for locating potential energy ridges and second order saddle points

McCulloch, Andrew D.: GPU-accelerated cardiac electrophysiology

Meinel, Michael: The FlowSimulator framework for massively parallel CFD applications

Michalakes, John: Towards petascale for atmospheric simulation (keynote lecture)

Mikkelsen, Kurt V.: Electromagnetic properties of large systems

Missirlis, Nikolaos: The diffusion method using neighbors of path length two in a torus graph

Mitra, Dhrubaditya: Direct numerical simulations of MHD using PENCIL-CODE

Moore, Shirley V.: Scalability study of a quantum simulation code

Morajko, Anna: MATE: Toward scalable automated and dynamic performance tuning environment

Mosegaard, Klaus: Computational limits to nonlinear inversion (keynote lecture)

Mueller, Frank: ScalaTrace: Tracing, analysis and modeling of HPC codes at scale

Mukunoki, Daichi: Implementation and evaluation of quadruple precision BLAS on GPU

Mýrdal, Jón Steinar G.: Computational screening of double cation metal borhydrides for solid Li electrolytes

Nehmeier, Marco: Parallel detection of interval overlapping

Neic, Aurel: Algebraic multigrid solver on clusters of CPUs and GPUs

Neukirchen, Helmut: Testing distributed and parallel systems with TTCN-3

Nguyen, Hong Diep: Efficient implementation for interval matrix multiplication

Nieminen, Risto M.: Algorithmic challenges for electronic-structure calculations (keynote lecture)

Niewiadomska-Szynkiewicz, Ewa: Software environment for parallel optimization of complex systems

Niewiadomska-Szynkiewicz, Ewa: A software tool for federated simulation of wireless sensor networks and mobile ad hoc networks

Pedersen, Andreas: Atomistic dynamics using distributed and grid computing

Pizzagalli, Laurent: Computation of transition states for extended defects in materials science: Issues and challenges from selected exemples

Rahman, Talal: A fast algorithm for a constrained total variation minimization with application to image processing

Remón, Alfredo: Accelerating model reduction of large linear systems with graphics processors

Riedel, Morris: The European middleware initiative: Delivering key technologies to distributed computing infrastructures

Riedel, Morris: Grid infrastructure interoperability in EU FP7th Euforia project

Roman Wyrzykowski: Towards efficient execution of erasure codes on multicore architectures

Ruud, Kenneth: Parallelization and grid adaptation of the Dalton quantum chemistry program

Rögnvaldsson, Ólafur: On-demand high-resolution weather forecast for search-and-rescue (SAR)

Saga, Kazushige: Grid interoperation in the RENKEI grid middleware

Schifano, Sebastiano F.: Monte Carlo simulations of spin systems on multicore processors

Schmelling, Michael: Boosting data analysis for the LHC experiments

Schnupp, Michael: Experimental performance of I/O-optimal sparse matrix dense vector multiplication algorithms within main memory

Schwanke, Christoph: Parallel particle-in-cell Monte Carlo algorithm for simulation of gas discharges under PVM and MPI (poster)

Shende, Sameer Suresh: Improving the scalability of performance evaluation tools

Signell, Artur: An efficient approximative method for generating spatially correlated multivariate random normals in parallel

Skovhede, Kenneth: CSP channels for CELL-BE

Skúlason, Egill: Simulations of atomic scale transitions at charged interfaces

Strey, Alfred: Implementation of clustering algorithms on manycore architectures

Strzelczyk, Jacek: Parallel kriging algorithm for unevenly spaced data

Stussak, Christian: Parallel computation of bivariate polynomial resultants on graphics processing units

Tillenius, Martin: An efficient task-based approach for solving the n-body problem on multicore architectures

Tudruj, Marek: Distributed Java programs initial mapping based on extremal optimization

Tudruj, Marek: Scheduling parallel programs with architecturally supported regions

Tudruj, Marek: Streaming model computation of the FDTD problem (poster)

Ujaldón, Manuel: The GPU on the 2D wavelet transform survey and contributions

Ujaldón, Manuel: CUDA 2D stencil computations for the Jacobi method

Unat, Didem: Optimizing Aliev-Panfilov model of cardiac excitation on heterogeneous systems

Urbah, Etienne: EDGI brings desktop grids to distributed computing interoperability

Valdman, Jan: Fast MATLAB assembly of FEM stiffness- and mass matrices in 2D and 3D: Nodal elements

Vazquez-Poletti, Jose Luis: A model for efficient onboard actualization of an instrumental cyclogram for the Mars MetNet mission on a public cloud infrastructure

Vialle, Stéphane: InterCell: A software suite for rapid prototyping and parallel execution of fine-grained applications

Waśniewski, Jerzy: New algorithms and data structures for matrices in the multi/manycore era, part 3 (tutorial)

Widener, Patrick: High-performance computing tools for data-intensive research patterns

Wikfeldt, Kjartan Thor: Thermodynamical and x-ray spectroscopic insights on the structure of liquid water from large-scale molecular dynamics simulations

Wolf, Felix: Further improving the scalability of the Scalasca toolset

Wylie, Brian J.N.: Performance engineering of GemsFDTD computational electromagnetics solver

Wyrzykowski, Roman: Towards efficient execution of erasure codes on multicore architectures

Table of Contents – Part I

Table of Contents – Part II

On Aggressive Early Deflation in Parallel Variants of the QR Algorithm

Bo Kågström[1], Daniel Kressner[2], and Meiyue Shao[1]

[1] Department of Computing Science and HPC2N
Umeå University, S-901 87 Umeå, Sweden
{bokg,myshao}@cs.umu.se
[2] Seminar for Applied Mathematics, ETH Zürich, Switzerland
kressner@math.ethz.ch

Abstract. The QR algorithm computes the Schur form of a matrix and is by far the most popular approach for solving dense nonsymmetric eigenvalue problems. Multishift and aggressive early deflation (AED) techniques have led to significantly more efficient sequential implementations of the QR algorithm during the last decade. More recently, these techniques have been incorporated in a novel parallel QR algorithm on hybrid distributed memory HPC systems. While leading to significant performance improvements, it has turned out that AED may become a computational bottleneck as the number of processors increases. In this paper, we discuss a two-level approach for performing AED in a parallel environment, where the lower level consists of a novel combination of AED with the pipelined QR algorithm implemented in the ScaLAPACK routine `PDLAHQR`. Numerical experiments demonstrate that this new implementation further improves the performance of the parallel QR algorithm.

1 Introduction

The solution of matrix eigenvalue problems is a classical topic in numerical linear algebra, with applications in various areas of science and engineering. The QR algorithm developed by Francis and Kublanovskaya, see [9,19] for recent historic accounts, has become the de facto standard for solving nonsymmetric and dense eigenvalue problems. Parallelizing the QR algorithm has turned out to be highly nontrivial matter [13]. To our knowledge, the ScaLAPACK [5] routine `PDLAHQR` implemented nearly 10 years ago based on work by Henry, Watkins, and Dongarra [14], is the only publicly available parallel implementation of the QR algorithm. Recently, a novel parallel QR algorithm [10] has been developed, which turns out to be more than a magnitude faster compared to `PDLAHQR` for sufficiently large problems. These improvements are attained by parallelizing the multishift and aggressive early deflation (AED) techniques developed by Braman, Byers, and Mathias [6,7] for the sequential QR algorithm.

Performed after each QR iteration, AED requires the computation of the Schur form for a trailing principle submatrix (the so called AED window) that is relatively small compared to the size of the whole matrix. In [10], a slightly modified version of the ScaLAPACK routine `PDLAHQR` is used for this purpose. Due to the small size of the AED window, the execution time spent on AED remains negligible for one or only a few processors but quickly becomes a dominating factor as the number of processors increases. In fact, for a $100\,000 \times 100\,000$ matrix and 1024 processor cores, it was observed in [10] that 80% of the execution time of the QR algorithm was spent on AED. This provides a strong motivation to reconsider the way AED is performed in parallel. In this work, we propose to perform AED by a modification of the ScaLAPACK routine `PDLAHQR`, which also incorporates AED at this lower level, resulting in a two-level recursive approach for performing AED. The numerical experiments in Section 4 reveal that our new approach reduces the overall execution time of the parallel QR algorithm from [10] by up to 40%.

2 Overview of the QR Algorithm with AED

In the following, we assume some familiarity with modern variants of the QR algorithm and refer to [15,18] for introductions. It is assumed that the matrix under consideration has already been reduced to (upper) Hessenberg form by, e.g., calling the ScaLAPACK routine `PDGEHRD`. Algorithm 1 provides a high-level description of the sequential and parallel QR algorithm for Hessenberg matrices, using multiple shifts and AED. Since this paper is mainly concerned with AED, we will only mention that the way the shifts are incorporated in the multishift QR sweep (Step 4) plays a crucial role in attaining good performance, see [6,10,17] for details.

Algorithm 1 . *Multishift Hessenberg QR Algorithm with AED*

```
    WHILE not converged
1.      Perform AED on the $n_{\mathsf{win}} \times n_{\mathsf{win}}$ trailing principle submatrix.
2.      Apply the accumulated orthogonal transformation to the
        corresponding off-diagonal blocks.
3.      IF enough eigenvalues have been deflated in Step 1
            GOTO Step 1.
        END IF
4.      Perform a multishift QR sweep with undeflatable
        eigenvalues from Step 1 as shifts.
5.      Check for negligible subdiagonal elements.
    END WHILE
```

In the following, we summarize the AED technique proposed by Braman, Byers, and Mathias [7]. Given an $n \times n$ upper Hessenberg matrix H, we partition

$$H = \begin{array}{c} \\ n-n_{\mathsf{win}}-1 \\ 1 \\ n_{\mathsf{win}} \end{array} \begin{array}{c} \begin{array}{ccc} n-n_{\mathsf{win}}-1 & 1 & n_{\mathsf{win}} \end{array} \\ \begin{pmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ 0 & H_{32} & H_{33} \end{pmatrix} \end{array},$$

where n_{win} denotes the size of the AED window. Then a (real) Schur decomposition $H_{33} = VTV^T$ is performed, where V is orthogonal and T in upper quasi-triangular form. Setting

$$U = \begin{array}{c} \\ n{-}n_{\mathsf{win}}{-}1 \\ 1 \\ n_{\mathsf{win}} \end{array} \begin{array}{c} \begin{array}{ccc} n{-}n_{\mathsf{win}}{-}1 & 1 & n_{\mathsf{win}} \end{array} \\ \begin{pmatrix} I & & \\ & 1 & \\ & & V \end{pmatrix} \end{array},$$

we obtain

$$U^T H U = \begin{pmatrix} H_{11} & H_{12} & H_{13}V \\ H_{21} & H_{22} & H_{23}V \\ 0 & s & T \end{pmatrix},$$

where $s \in \mathbb{R}^{n_{\mathsf{win}}}$ is the so called spike, created from the subdiagonal entry contained in H_{32}. The eigenvalues of T are checked subsequently for convergence and possibly deflated. The eigenvalue (or 2×2 block) in the bottom right corner of T can be deflated if the magnitude of the last component (or the last two components) of the spike is negligibly small. Undeflatable eigenvalues are moved to the top left corner of T by a swapping algorithm [4,11]. After this transformation is completed, the next eigenvalue in the bottom right corner of T is treated in the same way. The orthogonal transformations for swapping eigenvalues are accumulated in an orthogonal matrix $\tilde{V} \in \mathbb{R}^{n_{\mathsf{win}} \times n_{\mathsf{win}}}$. After all eigenvalues of T have been processed, the entire matrix is reduced back to Hessenberg form and the off-diagonal blocks H_{13} and H_{23} are multiplied with the product of all involved orthogonal transformations. It is recommended to choose n_{win} somewhat larger, e.g., by 50%, than the number of shifts in the multishift QR iterations [6].

Dramatic performance gains from AED have been observed both for sequential and parallel variants of the QR algorithm. These gains can be achieved essentially no matter how the rest of the QR algorithm is implemented, in particular how many shifts are used in the multishift QR sweep [7]. In effect, any implementation of the QR algorithm may benefit from AED; a fact that we will use below to improve the ScaLAPACK routine `PDLAHQR`. A convergence analysis, partially explaining the success of AED, can be found in [16].

3 Parallel Implementation of AED

Since the main aim of this paper is to improve the parallel QR algorithm and implementation described in [10], we first recall the structure of the main routines from this implementation, see Figure 1. The entry routine is `PDHSEQR`, which branches into `PDLAQR1` for small to medium-sized matrices and `PDLAQR0` for larger ones. The cut-off point for what is considered medium-sized will be explained in the numerical experiments, see Section 4. The main purpose of `PDLAQR0` is to call `PDLAQR3` for performing AED and `PDLAQR5` for performing multishift QR iterations. The former routine invokes `PDLAQR1` for performing the Schur decomposition of the AED window. In [10], `PDLAQR1` amounts to the

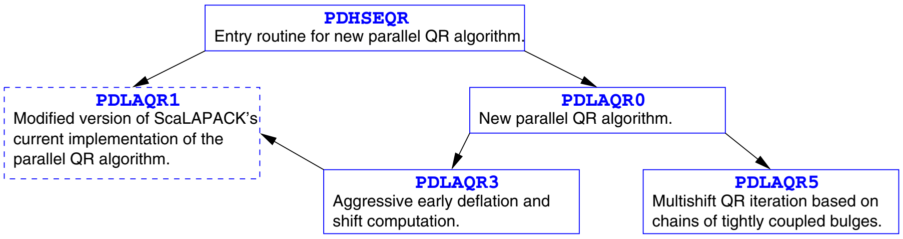

PDHSEQR
Entry routine for new parallel QR algorithm.
PDLAQR1
Modified version of ScaLAPACK's current implementation of the parallel QR algorithm.
PDLAQR0
New parallel QR algorithm.
PDLAQR3
Aggressive early deflation and shift computation.
PDLAQR5
Multishift QR iteration based on chains of tightly coupled bulges.

Fig. 1. Partial software structure for the parallel QR algorithm from [10]

ScaLAPACK routine PDLAHQR with minor modifications concerning the processing of 2×2 blocks in the real Schur form and the multithreaded application of small Householder reflectors. In the following, we will reconsider this choice for PDLAQR1.

3.1 Choice of Algorithm for Performing AED

A number of alternative choices are available for performing the Schur decomposition of the relatively small AED window:

- A recursive call to PDHSEQR or PDLAQR0, implementing the parallel QR algorithm with multishifts and AED.
- A call to PDLAQR1, a minor modification of ScaLAPACK's PDLAHQR.
- Assembling the AED window in local memory and a call to the sequential LAPACK [2] routine DLAHQR (or DLAQR4).

According to the numerical experiments in [10], a recursive call of PDLAQR0 may not be the optimal choice, mainly because of the fact that the way multishift QR iterations are implemented in PDLAQR0 suffers from poor scalability for relatively small matrices. ScaLAPACK's PDLAHQR achieves better scalability but does not incorporate modern developments, such as AED, and therefore suffers from poor performance. The third alternative, calling a sequential algorithm, should be used for submatrices that are too small to justify the overhead incurred by parallelization. In our experimental setup this was the case for submatrices of size 384 or smaller.

In this work, we propose to modify PDLAQR1 further and add AED to the parallel pipelined QR algorithm implemented in ScaLAPACK's PDLAHQR. Since the main purpose of PDLAQR1 is to handle small to medium-sized submatrices, a parallel implementation of AED, as in [10], will not be efficient on this level, since the size of the AED window is even smaller and does not allow for reasonable parallel performance in the Schur decomposition or the swapping of diagonal blocks. We have therefore chosen the third alternative for performing AED on the lowest level and invoke the sequential LAPACK routine DLAQR3 [8]. The accumulated orthogonal transformations returned by DLAQR3 are applied to the off-diagonal blocks in parallel. Therefore, $O(\sqrt{p})$ processors are used for updating the off-diagonal blocks. A high-level description of the resulting procedure is given in Algorithm 2.

Algorithm 2 . *Parallel pipelined QR algorithm with AED (new* `PDLAQR1`*)*

```
    WHILE not converged
1.      Copy the $(n_{\mathsf{win}}+1) \times (n_{\mathsf{win}}+1)$ trailing submatrix to local memory
        and perform sequential AED on an $n_{\mathsf{win}} \times n_{\mathsf{win}}$ window.
2.      Apply the accumulated orthogonal transformations to the
        corresponding off-diagonal blocks in parallel.
3.      IF enough eigenvalues have been deflated in Step 1
            GOTO Step 1.
        END IF
4.      Compute the eigenvalues of a trailing submatrix.
5.      Perform a pipelined QR sweep with the eigenvalues computed
        in Step 4 as shifts.
6.      Check for negligible subdiagonal elements.
    END WHILE
```

3.2 Implementation Details

In the following we discuss some implementation issues of Algorithm 2. The basis for our modification is `PDLAQR1` from [10], referred to as the *old* `PDLAQR1` in the following discussion. Following the notation established in the (Sca)LAPACK implementations of the QR algorithm, we let `NH=IHI-ILO+1` denote the dimension of the active `NH` $\times$ `NH` diagonal block and `NS` the number of shifts in the multishift QR sweep.

– In the special case when the active diagonal block is small enough, say `NH` $\leq$ 384, we copy this block to local memory and call `DLAHQR/DLAQR4` directly. The off-diagonal blocks are updated in parallel. This reduces communication while the required extra memory is negligible. We have observed that this modification reduces the total execution time by a non-negligible amount, especially during the final stages of the QR algorithm.
– The size of the deflation window, n_{win}, is determined by the return value of the LAPACK routine `IPARMQ`, see [8] for more details. In `PDLAHQR/PDLAQR1`, `NS` is mainly determined by the process grid and does not exceed 32. This is usually smaller than the number of shifts suggested by `IPARMQ`. Also, typical values of n_{win} returned by `IPARMQ` are 96, 192 and 384, which is much larger than if we chose `NS*3/2`. Based on the observation that the optimal AED window size does not depend strongly on the number of shifts used in the QR sweeps, we prefer to stick to large n_{win} rather than using `NS*3/2`. This increases the time spent on AED, but the overhead is compensated by fewer pipelined QR sweeps.
– The criterion for restarting another AED process rightaway, without an intermediate QR iteration, is the same as in LAPACK [8]:

1. The number of undeflatable eigenvalues is smaller than NS; or
2. the number of deflated eigenvalues is larger than $n_{\mathsf{win}} \times 14\%$.

Note that we choose the criterion in accordance with the window size suggested by IPARMQ.

– In contrast to Algorithm 1, undeflatable eigenvalues are not used as shifts in subsequent multishift QR sweep. This choice is based on numerical experiments with the following three shift strategies:
   1. Use undeflatable eigenvalues obtained from AED as shifts.
   2. Compute and use the eigenvalues of the NS $\times$ NS trailing submatrix after AED as shifts (by calling DLAHQR/DLAQR4).
   3. Compute and use some of the eigenvalues of the $(n_{\mathsf{win}} + 1) \times (n_{\mathsf{win}} + 1)$ trailing submatrix after AED as shifts (by calling DLAHQR/DLAQR4).

  An illustration of these strategies is given in Figure 2. Based on the experiments, we prefer the third strategy despite the fact that it is the computationally most expensive one. However, it provides shifts of better quality, mainly because of the larger window size, which was found to reduce the number of pipelined QR sweeps and to outweigh the increased cost for shift computation.

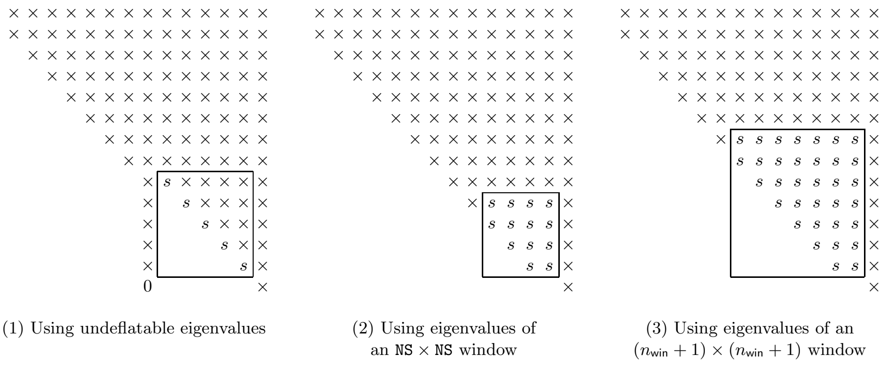

(1) Using undeflatable eigenvalues
(2) Using eigenvalues of an NS × NS window
(3) Using eigenvalues of an $(n_{\mathsf{win}}+1) \times (n_{\mathsf{win}}+1)$ window

Fig. 2. Three shift strategies ($n_{\mathsf{win}} = 6$, NS=4)

– When performing AED within the new PDLAQR1, each processor receives a local copy of the trailing submatrix and calls DLAQR3 to execute the same computations concurrently. This implies redundant work performed in parallel but it reduces communication since the orthogonal transformation matrix, to be applied in parallel in subsequent updates, is readily available on each processor. A similar approach is suggested in the parallel QZ algorithm by Adlerborn et al. [1]. If the trailing submatrix is not laid out across a border of the processor mesh, we call DGEMM to perform the updates. If the trailing submatrix is located on a 2×2 processor mesh, we organize the computation and communication manually for the update. Otherwise, PDGEMM is used for updating the off-diagonal blocks.

4 Numerical Experiments

All the experiments in this section were run on the 64-bit low power Intel Xeon Linux cluster *Akka* hosted by the High Performance Computing Center North (HPC2N). Akka consists of 672 dual socket quadcore L5420 2.5GHz nodes, with 16GB RAM per node, connected in a Cisco Infiniband network. The code is compiled by the PathScale compiler version 3.2 with the flags `-O2 -fPIC -TENV:frame_pointer=ON -OPT:Olimit=0`. The software libraries OpenMPI 1.4.2, BLACS 1.1 patch3, ScaLAPACK/PBLAS 1.8.0, LAPACK 3.2.1 and GOTOBLAS2 1.13 [12] are linked with the code. No multithreaded features, in particular no mixture of OpenMP and MPI, were used. We chose `NB = 50` as the block size in the block cyclic distribution of ScaLAPACK. The test matrices are dense square matrices with entries randomly generated from a uniform distribution in [0,1]. The ScaLAPACK routine `PDGEHRD` is used to reduce these matrices initially to Hessenberg form. We only measure the time for the Hessenberg QR algorithm, i.e., the reduction from Hessenberg to real Schur form.

4.1 Improvement for `PDLAQR1`

We first consider the isolated performance of the new `PDLAQR1` compared to the old `PDLAQR1` from [10]. The sizes of the test matrices were chosen to fit the typical sizes of the AED windows suggested in [10]. Table 1 displays the measured execution time on various processor meshes.

For the sequential case (1×1 mesh), `PDLAQR1` calls the LAPACK routine `DLAQR4` directly for small matrices (see the first remark in Section 3.2). `PDLAQR0` also implements a blocked QR algorithm almost identical to the new LAPACK algorithm [8], but some algorithmic parameters (e.g., number of shifts) can be different. Since the parameters in `PDLAQR0` largely depend on the block size in the block cyclic matrix data distribution of ScaLAPACK, `PDLAQR0` can be a bit slower than LAPACK.

For determining the cross-over point for switching from `PDLAQR0` to `PDLAQR1` in the main routine `PDHSEQR`, we also measured the execution time of `PDLAQR0`.

The new implementation of `PDLAQR1` turns out to require much less execution time than the old one, with a few, practically nearly irrelevant exceptions. Also, the new `PDLAQR1` scales slightly better than `PDLAQR0`, especially when the size of matrix is not large. It is worth emphasizing that the scaling of all implementations eventually deteriorates as the number of processor increases, simply because the involved matrices are not sufficiently large to create enough potential for parallelization.

Quite naturally, `PDLAQR0` becomes faster than the new `PDLAQR1` as the matrix size increases. The dashed line in Table 1 indicates the crossover point between both implementations. A rough model of this crossover point result is given by $n = 220\sqrt{p}$, which fits the observations reasonably well and has been incorporated in our implementation.

Table 1. Execution time in seconds for old `PDLAQR1` (1st line for each n), new `PDLAQR1` (2nd line) and `PDLAQR0` (3rd line). The dashed line is the crossover point between the new `PDLAQR1` and `PDLAQR0`.

Matrix size	Processor mesh						
(n)	1×1	2×2	3×3	4×4	6×6	8×8	10×10
96	0.01	0.05	0.11	0.18	0.15	0.25	0.27
	0.08	0.08	0.02	0.05	0.03	0.08	0.07
	0.14	0.40	0.96	1.11	2.52	3.16	2.95
192	0.09	0.17	0.18	0.22	0.32	0.47	0.64
	0.09	0.07	0.07	0.13	0.16	0.12	0.26
	0.15	0.30	0.61	1.05	3.73	4.34	3.64
384	0.60	0.73	0.61	0.63	0.78	1.09	1.24
	0.27	0.29	0.28	0.36	0.40	0.48	0.48
	0.47	0.55	0.72	0.89	2.08	3.23	3.76
768	7.38	3.53	2.53	2.35	2.61	2.80	3.52
	3.77	2.24	1.73	1.57	1.73	2.17	2.25
	1.83	1.51	1.61	1.68	2.70	3.03	3.31
1536	133.31	20.68	13.23	11.12	9.79	10.48	13.05
	35.94	9.27	6.54	5.52	5.11	5.31	6.33
	12.34	6.61	5.63	4.86	6.26	6.76	6.84
3072	2313.61	139.05	96.73	66.06	50.64	41.82	63.22
	522.81	45.72	33.13	22.60	19.08	18.12	22.23
	80.71	30.67	21.34	15.82	15.56	15.09	14.98
6144		1049.56	623.63	351.44	231.70	199.75	227.45
		144.96	167.71	103.15	78.75	66.90	70.48
		198.54	129.58	87.07	55.40	47.61	44.07

4.2 Overall Improvement

As the main motivation for the development of the new `PDLAQR1` is its application to AED within the parallel QR algorithm, we have also measured the resulting reduction of the overall execution time of `PDHSEQR`. From the results presented in Table 2, it is clear that `PDHSEQR` with the new `PDLAQR1` is almost always better than the old implementation. The improvement varies between 5% and 40%. We remark that the measured execution time for the 4000×4000 problem using 64 processors is less than running the same problem on 100 processors. However, situations may occur when we prefer to solve a 4000×4000 problem using 100 processors. For example, if this is a subproblem in a large-scale computation, it would be too costly to redistribute the matrix and use only 64 of the available processors. Among the measured configurations, there is one notable exception: $n = 32000$ on a 6×6 processor grid. This is actually the only case for which `PDLAQR0` is called within the AED phase, which seems to indicate that the choice of the crossover point requires some additional fine tuning.

Note that the largest AED window in all these experiments is of size 1536. According to Table 1, we expect even more significant improvements for larger matrices, which have larger AED windows.

Table 2. Execution time in seconds for old `PDHSEQR` (1st line for each n), new `PDHSEQR` (2nd line). The third lines show the relative improvement.

Processor mesh	Matrix size (n) 4000	8000	16000	32000
1×1	162.43			
	161.28			
	0.71%			
2×2	71.34	501.83		
	68.02	452.70		
	4.65%	9.79%		
4×4	39.18	170.75	1232.40	
	30.68	158.66	1037.93	
	22.69%	7.08%	15.78%	
6×6	35.96	123.46	617.97	3442.08
	24.62	96.23	509.38	3584.74
	31.54%	22.06%	17.57%	-4.14%
8×8	33.09	97.20	435.52	2639.32
	20.59	67.42	366.31	2016.93
	37.78%	31.64%	15.89%	24.58%
10×10	36.05	101.75	355.38	2053.16
	21.39	62.29	291.06	1646.30
	41.67%	39.58%	18.10%	19.82%

5 Summary

We have reconsidered the way AED is performed in the parallel QR algorithm [10]. A recursive approach is suggested, in which the ScaLAPACK routine `PDLAHQR` is combined with AED to address medium-sized problems. The focus of this work has been on minimizing the total execution time instead of how to use all the processors or how well the algorithm scales. Computational experiments demonstrate the efficiency of our approach, but also reveal potential for further improvements by a more careful fine tuning of the crossover point for switching between different implementations of the parallel QR algorithm.

Acknowledgments. We would like to thank Robert Granat, Lars Karlsson, and Åke Sandgren for their help and support.

This work was conducted using the resources of the High Performance Computing Center North (HPC2N), `http://www.hpc2n.umu.se`, and was supported by the Swedish Research Council under grant VR7062571, the Swedish Foundation for Strategic Research under grant A3 02:128, and UMIT Research Lab (EU Mål 2 grant).

In addition, support has been provided by eSSENCE, a collaborative e-Science programme funded by the Swedish Research Council within the framework of the strategic research areas designated by the Swedish Government.

References

1. Adlerborn, B., Kågström, B., Kressner, D.: Parallel Variants of the Multishift QZ Algorithm with Advanced Deflation Techniques. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 117–126. Springer, Heidelberg (2007)
2. Anderson, E., Bai, Z., Bischof, C.H., Blackford, S., Demmel, J.W., Dongarra, J.J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.C.: LAPACK User's Guide, 3rd edn. SIAM, Philadelphia (1999)
3. Bai, Z., Demmel, J.W.: On a Block Implementation of Hessenberg Multishift QR Iteration. Intl. J. of High Speed Comput. 1, 97–112 (1989)
4. Bai, Z., Demmel, J.W.: On Swapping Diagonal Blocks in Real Schur Form. Linear Algebra Appl. 186, 73–95 (1993)
5. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J.W., Dhillon, I., Dongarra, J.J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. SIAM, Philadelphia (1997)
6. Braman, K., Byers, R., Mathias, R.: The Multishift QR Algorithm. Part I: Maintaining Well-focused Shifts and Level 3 Performance. SIAM J. Matrix Anal. Appl. 23(4), 929–947 (2002)
7. Braman, K., Byers, R., Mathias, R.: The Multishift QR Algorithm. Part II: Aggressive Early Deflation. SIAM J. Matrix Anal. Appl. 23(4), 948–973 (2002)
8. Byers, R.: LAPACK 3.1 xHSEQR: Tuning and Implementation Notes on the Small Bulge Multi-shift QR Algorithm with Aggressive Early Deflation. LAPACK Working Note 187 (2007)
9. Golub, G., Uhlig, F.: The QR Algorithm: 50 Years Later Its Genesis by John Francis and Vera Kublanovskaya and Subsequent Developments. IMA J. Numer. Anal. 29(3), 467–485 (2009)
10. Granat, R., Kågström, B., Kressner, D.: A Novel Parallel QR Algorithm for Hybrid Distributed Memory HPC Systems. SIAM J. Sci. Comput. 32(4), 2345–2378 (2010) (An earlier version appeared as LAPACK Working Note 216)
11. Granat, R., Kågström, B., Kressner, D.: Parallel Eigenvalue Reordering in Real Schur Forms. Concurrency and Computat.: Pract. Exper. 21(9), 1225–1250 (2009)
12. GOTO-BLAS – High-performance BLAS by Kazushige Goto, http://www.tacc.utexas.edu/tacc-projects/#blas
13. Henry, G., van de Geijn, R.: Parallelizing the QR Algorithm for the Nonsymmetric Algebraic Eigenvalue Problem: Myths and Reality. SIAM J. Sci. Comput. 17, 870–883 (1997)
14. Henry, G., Watkins, D.S., Dongarra, J.J.: A Parallel Implementation of the Nonsymmetric QR Algorithm for Distributed Memory Architectures. SIAM J. Sci. Comput. 24(1), 284–311 (2002)
15. Kressner, D.: Numerical Methods for General and Structured Eigenvalue Problems. LNCSE, vol. 46. Springer, Heidelberg (2005)
16. Kressner, D.: The Effect of Aggressive Early Deflation on the Convergence of the QR Algorithm. SIAM J. Matrix Anal. Appl. 30(2), 805–821 (2008)
17. Lang, B.: Effiziente Orthogonaltransformationen bei der Eigen- und Singulärwertzerlegung. Habilitationsschrift (1997)
18. Watkins, D.S.: The Matrix Eigenvalue Problem: GR and Krylov Subspace Methods. SIAM, Philadelphia (2007)
19. Watkins, D.S.: Francis's Algorithm. Amer. Math. Monthly (2010) (to appear)

Limits to Nonlinear Inversion

Klaus Mosegaard

Department of Informatics and Mathematical Modeling
and Center for Energy Resources Engineering
Technical University of Denmark
Richard Petersens Plads, 2800 Lyngby, Denmark
http://www.imm.dtu.dk/~kmos

Abstract. For non-linear inverse problems, the mathematical structure of the mapping from model parameters to data is usually unknown or partly unknown. Absence of information about the mathematical structure of this function prevents us from presenting an analytical solution, so our solution depends on our ability to produce efficient search algorithms. Such algorithms may be completely problem-independent (which is the case for the so-called 'meta-heuristics' or 'blind-search' algorithms), or they may be designed with the structure of the concrete problem in mind.

We show that pure meta-heuristics are inefficient for large-scale, nonlinear inverse problems, and that the 'no-free-lunch' theorem holds. We discuss typical objections to the relevance of this theorem.

A consequence of the no-free-lunch theorem is that algorithms adapted to the mathematical structure of the problem perform more efficiently than pure meta-heuristics. We study problem-adapted inversion algorithms that exploit the knowledge of the smoothness of the misfit function of the problem. Optimal sampling strategies exist for such problems, but many of these problems remain hard.

1 Introduction

Nonlinear inverse problems occur frequently in analysis of physical data, and a variety of algorithms are used to produce acceptable solutions and to analyze their properties. Some problems are only weakly nonlinear and can be locally approximated by linear problems, but others are strongly nonlinear and require special treatment. Modern digital computers have greatly improved our ability to perform nonlinear data inversion, but still the limitations of current techniques are strongly felt.

In this paper we intend to review and analyze some fundamental computational limitations to the solution of nonlinear inverse problems. We will put special emphasis on the interplay between the solution algorithm and the structure of the problem to be solved. Our exposition will, in principle, be relevant for the solution of inverse problems in general, but given the fact that nonlinear inverse theory relies much more on the theory of search- and sampling algorithms than linear theory does, our considerations will be most relevant for the nonlinear case.

2 The Blind Inversion Problem

It is often reported in the literature that solutions to nonlinear inverse problems were obtained by problem-independent algorithms, the so-called *meta-heuristics*. This type of algorithms is claimed to work efficiently because of some general, external (problem independent) principle. For example, simulated annealing [5] inherits its efficiency from thermodynamic principles, genetic algorithms [6,7] exploit evolutionary principles, and taboo-search [8] uses some 'common sense' strategy. In the following, we will call these algorithms *blind inversion algorithms*, and we will investigate their efficiency in some detail.

Blind inversion schemes may be very different in character. They include deterministic as well as Monte Carlo algorithms, and they all share the basic property that they operate in a way that is independent of the particular inverse problem. A blind inversion scheme operates - sequentially or in parallel - by evaluating a misfit function (or fit function) in points in the parameter space, and a particular scheme is solely characterized by the strategy by which it selects new evaluation points from earlier selected points and their misfits. Some algorithms estimate gradients of the misfit (e.g., steepest descent), some use a proposal distribution combined with an acceptance probability (e.g., simulated annealing), others compare misfits at several points and use a selection strategy (genetic algorithms and the neighborhood algorithm [4,9,10].

In the following we shall analyze the performance of blind algorithms, but first we will briefly review some general topological properties of general inverse problems that impede algorithm efficiency when searching high-dimensional parameter spaces.

3 Basic Limitations in Blind Inversion Arising from the Structure of the Problem

Consider a discrete, nonlinear inverse problem

$$\mathbf{d} = \mathbf{g}(\mathbf{m}) \tag{1}$$

where $\mathbf{d}$ is a vector with $N + P$ components, the first N equations relate data $(d_1, \ldots, d_N)$ to M model parameters $\mathbf{m}$, and the remaining P equations express our prior information through $P \leq M - N$ equality constraints. If we assume that $\mathbf{g}$ is a $\mathcal{C}^1$-function, and that we have a solution $\hat{\mathbf{m}}$ to equation (1) for which

$$\mathbf{g}'(\hat{\mathbf{m}}) = \left[\frac{\partial g_i}{\partial m_j}\right]_{\mathbf{m}=\hat{\mathbf{m}}} \tag{2}$$

has full row rank, the implicit function theorem [11] implies that eq. (1) defines a solution-submanifold of dimension $M-(N+P)$ in the neighborhood around $\hat{\mathbf{m}}$ in the parameter space. If data uncertainties and 'softness' of the prior constraints are described by a neighborhood around $\mathbf{d}$, acceptable solutions in the parameter space can be found in a neighborhood around the solution manifold intersecting $\hat{\mathbf{m}}$.

In inversion aimed at locating only one acceptable solution, one will generally try to supply a sufficient number of tight a priori constraints to ensure that $M = N + P$. In this case, the solution space consists of neighborhoods around isolated points in the parameter space.

Of course, the above description is based on a number of simplifications. For instance: (1) We assumed that the matrix in equation (2) has full row rank. If this is not the case, the mapping g is not surjective, meaning that the dimension of the solution space is larger than in the full-rank case. (2) We have only considered prior information defined as (possibly softened) equality constraints. Introducing constraints that allow the solution to exist only inside a bounded, possibly non-convex, region may render the inverse problem even harder. (3) Strictly speaking, in our theory M should not be the number of model parameters, but the number of degrees of freedom in the model space. The arguments above can easily be modified to take this into account.

4 Basic Algorithmic Limitations in Blind Inversion

4.1 The Performance of Blind Inversion Algorithms

We now turn to the question of assessing the relative merits of blind inversion algorithms. The usual two ways of evaluating relative performances have been (1) to argue for or against algorithms using common-sense, physical or other arguments to discuss their ability to, e.g., locate acceptable misfit regions, or (2) to select suitable test problems and arrange numerical contests between selected algorithms. The first approach does not provide quantitative measures of relative performance, and the latter method is so sensitive to the selected problems and the 'tuning' of each of the considered algorithms that general conclusions are tentative.

Here, we will follow a different path leading to a quantitative comparison. Our reasoning will be similar to the one behind the so-called *No-Free-Lunch Theorem* [14]. It is based on a discretization where a finite number of parameters and data are only allowed to attain a finite set of values. This *double discretization* is actively used in genetic algorithms where parameters are often assumed to be binary numbers with only a few bits. All other numerical methods are, of course, also doubly discretized, because they run on digital computers with a finite precision.

The doubly discretized inverse problem and the information collected by an arbitrary, blind inversion algorithm will be described through the following notation. We consider:

- A *finite set* $\mathcal{M}$ of models. This set consists of all combinations of a finite number of values attained by a finite set of parameters.
- A *finite set* set $\mathcal{S}$ of real numbers. These numbers are the possible fit or misfit values that can be generated by models in $\mathcal{M}$.

- The set $\mathcal{F}_{\mathcal{M}}$ of all fit functions $f : \mathcal{M} \to \mathcal{S}$.
- A sample of size $m < |\mathcal{M}|$ generated by an algorithm having sampled $f \in \mathcal{F}$ in m distinct points:

$$\{(\mathbf{m}_1, s_1), \dots, (\mathbf{m}_m, s_m)\}. \tag{3}$$

  Note, that points resampled by an algorithm will only count once. On the other hand, auxiliary sample points used by the algorithm, e.g. points sampled with the sole purpose of calculating an approximate 'gradient', count on equal footing with other sample points.
- The (time) ordered set of sample points (arguments indicate time-ordering):

$$C = \{\mathbf{m}_1, \dots, \mathbf{m}_m\}. \tag{4}$$

- The (time) ordered set of corresponding values of f:

$$s_1, \dots, s_m. \tag{5}$$

- The set $\mathcal{F}_{\mathcal{M}|C}$ of all fit functions/probability distributions defined on $\mathcal{M}$, but with fixed values in C.

Consider a blind inversion problem where we have no knowledge of the actual fit function, and we search for at least one acceptable solution to the problem. From the outset, the total number of possible fit functions is equal to

$$|\mathcal{F}_{\mathcal{M}}| = |\mathcal{S}|^{|\mathcal{M}|}. \tag{6}$$

We can now ask: What is the probability that an algorithm, when sampling $\mathcal{M}$ in m distinct points, sees the function values $s_1, \dots, s_m$? To compute this we observe that, when fixing the function values at the m points of subset C (see Figure 1), the number of remaining, possible fit functions is narrowed in and is equal to

$$|\mathcal{F}_{\mathcal{M}|C}| = |\mathcal{S}|^{|\mathcal{M}|-m}. \tag{7}$$

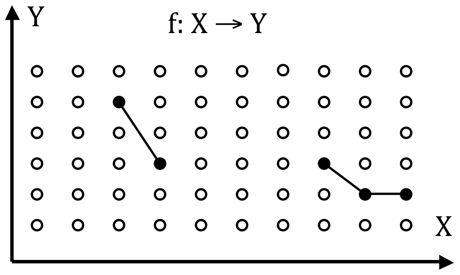


Fig. 1. Knowledge of an unknown fit function f after evaluation of the function in 5 points

This means that the probability that an algorithm in m function evaluations sees the particular fit function values $s_1, \ldots, s_m$ is

$$P(s_1, \ldots, s_m | a, m) = |\mathcal{F}_{\mathcal{M}|C}| |\mathcal{F}_{\mathcal{M}}|^{-1} = |\mathcal{S}|^{-m} \tag{8}$$

This number is independent of the location of the sample points and hence the algorithm used. Any algorithm a performing m iterations where it visits (algorithm dependent) points $\mathbf{m}_1^{(a)}, \ldots, \mathbf{m}_m^{(a)}$ can obtain functional values $s_1, \ldots, s_m$ with exactly $|\mathcal{F}_{\mathcal{M}|C}|$ different fit functions $f \in \mathcal{F}_{\mathcal{M}}$, and $|\mathcal{F}_{\mathcal{M}|C}|$ is independent of a, so for any pair of algorithms a_1 and a_2 we have

$$P(s_1, \ldots, s_m | \, m, a_1) = P(s_1, \ldots, s_m | \, m, a_2) \tag{9}$$

where $P(\cdot \, | \, \cdot)$ denotes conditional probability.

Any performance measure for inversion algorithms searching for near-optimal data fits is of the form $\Phi : \mathcal{S}^m \rightarrow \mathbf{R}$, for instance:

$$\Phi(s_1, \ldots, s_m) = \max\{s_1, \ldots, s_m\}, \tag{10}$$

which must be large for good performance. Even Monte Carlo algorithms, aimed at importance-sampling of probability distributions over the parameter space, operate as near-optimization algorithms in the computer-intensitive, initial phase (the burn-in phase) where the first acceptable solution is sought.

The probability distribution of $\Phi(s_1, \ldots, s_m)$ depends only on $P(s_1, \ldots, s_m | \, m, a)$, and as a consequence of equation (9) it is therefore independent of the algorithm a. We have now shown

Theorem 1. *(Similar to the* **No-Free-Lunch Theorem** *by Wolpert and Macready [14]). The distribution of any performance measure for inversion, when all fit functions are equally probable (blind inversion), is exactly the same for all inversion algorithms.*

4.2 Critique of the No-Free-Lunch Theorem

One obvious critique of the usefulness of the No-Free-Lunch Theorem is that, in typical fields of application, the fit functions belong to a *narrow subfamily* of functions (e.g., smooth functions), and some algorithms work better than others on such families. This objection is based on the observation that for each function sub-family $\mathcal{G}_{\mathcal{M}} \subseteq \mathcal{F}_{\mathcal{M}}$, the total number of ways a particular set of fit values $s_1, \ldots, s_m$ can be obtained in a set of m sample points $\mathbf{m}_1, \ldots, \mathbf{m}_m$ from fit functions $f \in \mathcal{G}_{\mathcal{M}}$ will in general depend on $\mathbf{m}_1, \ldots, \mathbf{m}_m$ (and hence on the algorithm a that is choosing the sample points). However, this objection is, as we shall see now, invalid in the blind inversion case.

Consider all subfamilies $\mathcal{G}_{\mathcal{M}}$ of functions in $\mathcal{F}_{\mathcal{M}}$. Functions in subfamily $\mathcal{G}_{\mathcal{M}}$ with fixed values on the subset C form the set $\mathcal{G}_{\mathcal{M}} \cap \mathcal{F}_{\mathcal{M}|C}$. The average number of functions in such a subfamily is proportional to

$$\sum_{\forall \mathcal{G}_{\mathcal{M}} \subseteq \mathcal{F}_{\mathcal{M}}} |\mathcal{G}_{\mathcal{M}} \cap \mathcal{F}_{\mathcal{M}|C}| = \sum_{\forall \mathcal{G}_{\mathcal{M} \setminus C}} |\mathcal{G}_{\mathcal{M} \setminus C}| \tag{11}$$

The right-hand side of (11) depends only on $|\mathcal{M}| - |C|$ and hence only on the number m. Hence, the actual elements of C, which are chosen as a result of a's search strategy, does not influence the average number of functions in $\mathcal{G}_{\mathcal{M}} \cap \mathcal{F}_{\mathcal{M}|C}$. We have demonstrated

Theorem 2. *The efficiency of blind inversion algorithms, averaged over all subfamilies of fit functions, is the same for all inversion algorithms.*

We have found that the efficiency of all blind inversion schemes, e.g., Steepest ascent/descent, Simulated Annealing, Markov-Chain Monte Carlo, Random Search, Genetic Algorithm, Neighborhood Algorithm, Taboo Search, Line Search and Exhaustive Search are exactly the same, when averaged over alle possible inverse problems or subsets of problems.

How can we accept this surprising result when all experience shows that, for instance, Random Search is much less efficient than Genetic Algorithms? The only explanation for the apparent paradox is that the above mentioned blind algorithms are, in practice, supplemented with procedures that inform the algorithm about the structure of the problem. As we shall discuss in the next section, this is indeed the case.

5 The Informed Inversion Problem

5.1 The Necessity of Algorithm Tuning

We have demonstrated above that the performance of all blind inversion schemes are exactly the same, when averaged over all conceivable fit functions, or subsets hereof. This means that if, for example, a genetic algorithm performs better than a crude random search on certain problems, it will perform worse on other problems. Apparently, this result is contradicted by the experience of a vast number of researchers who have seen popular algorithms outperforming crude random search by several orders of magnitude.

The only way we can resolve this paradox is to point at the *tuning* of inversion algorithms. Most expositions explaining the functioning of inversion algorithms emphasize the external ideas behind their design, and attribute the algorithm's efficiency to these ideas. Simulated Annealing is, for example, relying on an idea taken from natural minimization of the internal energy seen in thermodynamical systems under slow cooling, Genetic algorithms use ideas from natural biological selection to generate near-optimal solutions, etc. The consequence of Theorem 1 above is, however, that none of these external design ideas can provide any degree of success. Instead, we must turn to the tuning of algorithms to obtain the desired efficiency.

Unfortunately, there is a vast number of very different ways of tuning inversion algorithms. For this reason we shall only discuss one of the most important ideas, namely tuning to a known smoothness of the fit function through the choice of distance between sample points, for some algorithms termed the 'step length'. An algorithm that is tuned to the problem is no longer a blind algorithm (a meta-heuristic). It has become a so-called *heuristic* – an informed inversion algorithm.

5.2 A Lower Bound on the Required Number of Iterations by a Smoothness-Tuned Algorithm

Assume that the parameter space $\mathcal{M} \in \mathbf{R}^M$ is an M-dimensional cube of edge length L, and consider a class $\mathcal{F}$ of fit functions defined over $\mathcal{M}$. Assume that fit functions $f \in \mathcal{F}$ can be approximated to any precision by

$$f(\mathbf{m}) \approx \sum_{j=1}^{J} u_j \; \phi_j(\mathbf{m}) \tag{12}$$

where $\phi_1, \phi_2, \ldots, \phi_J$ is a set of linearly independent basis functions, and where J will depend on the required precision. For instance, according to a theorem by Brown [15], any *continuous* fit function $f \in \mathcal{F}$ can be approximated to any precision by (12) if $\phi_1, \phi_2, \ldots, \phi_J$ are *radial basis functions* of the form

$$\phi_j(\mathbf{m}) = g(\|\mathbf{m} - \mathbf{m}_j\|^2), \quad \mathbf{m}_j \in Q \tag{13}$$

where g is a non-constant, completely monotone[1] function defined on $[0, \infty[$, and Q is a compact subset of $\mathbf{R}^M$ containing more than one point. The radial basis functions $\phi_k(\mathbf{m})$ have 'spherical symmetry', and they are translates of each other (have different $\mathbf{m}_j$). The class of radial basis functions is very wide, containing members as, for instance, $\exp(-\|\mathbf{m}-\mathbf{m}_j\|^2)$, $\|\mathbf{m}-\mathbf{m}_j\|$ and $\|\mathbf{m}-\mathbf{m}_j\|^2 \ln(\|\mathbf{m}-\mathbf{m}_j\|)$.

Assume first that the only information we have about a fit function f is that it is 'band limited' in the sense that it is given exactly by the finite linear combination

$$f(\mathbf{m}) = \sum_{j=1}^{J} u_j \; \phi_j(\mathbf{m}), \tag{14}$$

and that we search for acceptable solutions to the inverse problem. Given that an algorithm has sampled f in k distinct points $\mathbf{m}_1, \ldots, \mathbf{m}_K$ and recorded the corresponding function values $s_1, \ldots, s_K$, what is the chance that we have located the neighborhood of the global maximum/minimum for f? To answer this question, let us express our knowledge after K iterations through the equations:

$$s_k = \sum_{j=1}^{J} u_j \; g(\|\mathbf{m}_k - \mathbf{m}_j\|^2) \qquad k = 1, \ldots, K \tag{15}$$

If we define the vectors $\mathbf{s} = (s_1, \ldots, s_K)^T$, $\mathbf{u} = (u_1, \ldots, u_J)^T$, and the matrix $\{\mathbf{G}\}_{kj} = g(\|\mathbf{m}_k - \mathbf{m}_j\|^2)$, equation (15) can be written

$$\mathbf{s} = \mathbf{G}\mathbf{u} \; . \tag{16}$$

[1] A function g defined on $[0, \infty[$ is completely monotone if g is continuous on $[0, \infty[$, infinitely often differentiable on $]0, \infty[)$, and $(-1)^k g^{(k)}(t) \geq 0$ for $t > 0$ and $k = 1, 2, 3, \ldots$.

If the solution $\mathbf{u}$ to this equation is non-unique, at least one of its components $u_1, \ldots, u_J$ is unbounded (that is, can take any value without violating equation (16)). If u_l is such an unbounded component, f is (at best) only known modulo an unbounded additional term $u_l \; g(\|\mathbf{m} - \mathbf{m}_l\|^2)$. It is clear that the unknown u_l renders f's value in $\mathbf{m}_l$, and hence f's global maximum/minimum, undetermined. In other words, unless we can uniquely determine $\mathbf{u}$, we will be unable to locate the global maximum for f. $\mathbf{u}$ is uniquely determined only if $\mathbf{G}^T\mathbf{G}$ is non-singular, and a necessary condition for this to be satisfied is that $K \geq J$. We have shown

Theorem 3. *An inverse problem, whose fit function is known to be a linear combination of a linear independent set of J basis functions, cannot be solved through less than J distinct function evaluations.*

In the general case, we cannot expect to express f exactly with a finite number of basis functions. Then f is only expressed with a certain accuracy $\epsilon > 0$, in the sense that the discrepancy

$$n(\mathbf{m}) = f(\mathbf{m}) - \sum_{j=1}^{J} u_j \; \phi_j(\mathbf{m}) \tag{17}$$

is constrained by

$$max_{\mathbf{m}} \, |n(\mathbf{m})| \leq \epsilon \quad . \tag{18}$$

In this case the problem is that equation (16) does not reliably determine the coefficient vector $\mathbf{u}$ when $\mathbf{G}^T\mathbf{G}$ is ill-conditioned. The discrepancy propagates into the coefficient vector, creating large errors, and the location of the global maximum of f remains unknown.

The above considerations set a fundamental, unavoidable lower limit to the number of iterations required by any inversion algorithm working with a 'band-limited' fit function. We have not described how an ideal (maximum efficiency) algorithm should work, but it is clear from the discussion that, contrary to the 'blind inversion' case, not all algorithms are equally good.

To see this, consider again equation (16). An algorithm choosing its first J distinct sampling points such that $\mathbf{G}^T\mathbf{G}$ is non-singular and well-conditioned has collected sufficient information to locate the global maximum for f (although we have not shown how to do this). On the other hand, an algorithm choosing its first J distinct sampling points such that $\mathbf{G}^T\mathbf{G}$ is singular or ill-conditioned is still missing information about the location of the global maximum of f. For instance, a sub-optimal algorithm may, after J distinct function evaluations, have failed to sample all J basis functions in points sufficiently near their maxima. Such an algorithm will need more than J distinct function evaluations to render equation (16) solvable.

6 The Complexity of an Inverse Problem with Known Smoothness

Consider an inverse problem with a fit function f defined in an interval $\mathcal{M}$ of edge length L in $\mathbf{R}^M$ (an M-dimensional 'box'). Assume that f is smooth in the sense that it can be expanded in a linear independent, finite set of *radial* basis functions, centered in a regular grid in $\mathcal{M}$ with grid spacing l. The required, total number of basis functions is then $(L/l+1)^M$, and according to Theorem (3) this is the smallest number of distinct function evaluations needed to solve the inverse problem. To reach this minimum number of evaluations, the safest strategy for an algorithm is to sample close to the maxima of the basis functions, and this calls for a sampling distance close to a multiple of l.

It should be noted, however, that in this important case the smallest number of distinct function evaluations needed by any algorithm to solve the inverse problem grows at least exponentially in M. This growth is severe, and shows that the inverse problem is *hard* in the sense that the solution time grows faster than any polynomial function [16]. In practice, this means that even significant improvements in computer speed will only allow the inverse problem to be solved with a few more model parameters. Let us summarize this important observation in the following:

Theorem 4. *Consider an inverse problem for which our only knowledge is that its fit function can be expanded in a set of linearly independent, radial basis functions, and assume that the basis functions are centered in a regular grid covering the model space. Then the computation time for any algorithm aimed at solving the inverse problem will grow at least exponentially with the number of unknown model parameters.*

7 Discussion and Conclusion

Wolpert and Macready [14] showed that, contrary to the belief of many practicians, there is no difference between the performance of the many existing, and popular, meta-heuristics, unless they are tuned to the problem at hand (and therefore no longer problem-independent). This means that all attempts to improve on inversion algorithms must focus on the tuning. One of the most common tuning parameters is the 'step length' (as in, e.g., simulated annealing) or 'sampling density' (as in, e.g., the neighborhood algorithm). This kind of tuning applies to cases where smoothness is the only known property of the fit function.

Usually, the smoothness is determined empirically through experimentation with a range of sampling densities. An example is steepest descent algorithms where step lengths are adjusted in order to avoid 'instability' of the algorithm. Another example is the adjustment of the step length in Markov-chain Monte Carlo methods, until the rate of accepted moves is reasonable [17]. A third example is the neighborhood algorithm where the density of resampling can be adjusted.

In many inverse problems arising in physical sciences, the data, and hence the fit (or misfit) functions are smooth. An example is seismic waveform inversion, where the band-limitation of the seismic source function is inherited by the fit function. If one attempts to solve this problem with an algorithm that is only informed about the smoothness of the fit function (through numerical experimentation or through some spectral information), the solution time will grow exponentially with the number of unknown parameters. This means that large-scale, seismic waveform inversion problems are essentially unsolvable in this way. On the other hand, it is well known that solution of such problems may be feasible with 'well-informed' algorithms, based on the theory of seismic wave propagation.

We should mention a couple of objections that could be raised against our exposition. First, does the above theory account for the situation that an algorithm may sometimes, by accident, start its search/sampling close to a solution? The answer to this question is that in large-scale inverse problems with many unknown parameters, there is a negligible probability that an algorithm, only informed about the smoothness of f, would start near an acceptable solution. We have therefore ignored this situation.

A second objection concerns the fact that we have treated deterministic methods (searching only for one feasible solution), and Monte Carlo sampling methods (aiming at finding many feasible solutions) in a unified theory. Clearly, sampling methods start with a 'burn-in phase' which is comparable to deterministic methods in its aim at locating one acceptable model, but this phase is followed by a 'sampling phase' which is apparently the real production phase of the sampling algorithm. To what extent is the sampling phase considered in our theory? The answer is, that our theory only considers the burn-in phase of a Monte Carlo sampling. In this connection it should be remembered that the burn-in process not only concerns the initial search for acceptable solutions. If the fit function for the problem has many isolated islands of acceptable solutions, the burn-in time is also a measure of the time it takes for the algorithm to move from one solution island to the next.

As a final remark we should note that, although we have demonstrated that all blind inversion schemes are equally (in)efficient, and that efficient algorithms can only be obtained through problem-dependent tuning, it is certainly possible that some algorithm designs are more easily tuned than others. This may be responsible for some of the differences that practitioners observe between popular algorithms.

Acknowledgments. The ideas behind this paper grew out of numerous inspiring discussions with Albert Tarantola who died tragically 6. december 2009. The author would also like to thank Thomas Mejer Hansen for constructive discussions and criticism. The work was supported by The Danish Council for Independent Research - Natural Sciences (FNU) under contract no. 272-08-0568, and DONG Energy, Denmark.

References

1. Nolet, G., van Trier, J., Huisman, R.: A formalism for nonlinear inversion of seismic surface waves. Geoph. Res. Lett. 13, 26–29 (1986)
2. Nolet, G.: Partitioned wave-form inversion and 2D structure under the NARS array. J. Geophys. Res. 95, 8499–8512 (1990)
3. Snieder, R.: The role of nonlinearity in inverse problems. Inverse Problems 14, 387–404 (1998)
4. Sambridge, M.: Exploring multi-dimensional landscapes without a map. Inverse Problems 14(3), 427–440 (1998)
5. Kirkpatrick, S.C., Gelatt, D., Vecchi, M.P.: Optimization by simulated annealing. Science 220, 671–680 (1983)
6. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
7. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Kluwer Academic Publishers, Boston (1989)
8. Glover, F.: Future Paths for Integer Programming and Links to Artificial Intelligence. Comput. & Ops. Res. 13(5), 533–549 (1986)
9. Sambridge, M.: Geophysical inversion with a Neighbourhood algorithm -I. Searching a parameter space. Geoph. Jour. Int. 138, 479–494 (1999a)
10. Sambridge, M.: Geophysical inversion with a Neighbourhood algorithm -II. Appraising the ensemble. Geoph. Jour. Int. 138, 727–746 (1999b)
11. Cauchy, A.L.: First Turin Memoir (1831)
12. Mosegaard, K., Tarantola, A.: Monte Carlo sampling of solutions to inverse problems. J. Geophys. Res. 100(B7), 12,431–12,447 (1995)
13. Khachiyan, L.G.: The problem of computing the volume of polytopes is np-hard. Uspekhi Mat. Nauk. 44, 199–200 (1989)
14. Wolpert, D.H., Macready, W.G.: No Free Lunch Theorems for Optimization. IEEE Transactions on Evolutionary Computation 1, 67–82 (1997)
15. Brown, A.L.: Uniform approximation by radial basis functions. In: Light, W.A. (ed.) Advances in Numerical Analysis, vol. 2, pp. 203–206. Oxford University Press, Oxford (1992)
16. Papadimitriou, C.H., Steiglitz: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, Inc., Mineola (1998)
17. Hastings, W.K.: Monte Carlo sampling methods using Markov Chain and their applications. Biometrika 57, 97–109 (1970)

Cache Blocking

Fred G. Gustavson[1,2]

[1] IBM T.J. Watson Research Center, Emeritus
[2] Umeå University
fg2935@hotmail.com

Abstract. Over the past five years almost all computer manufacturers have dramatically changed their computer architectures to Multicore (MC) processors. We briefly describe Cache Blocking as it relates to computer architectures since about 1985 by covering the where, when, how and why of Cache Blocking as it relates to dense linear algebra. It will be seen that the arrangement in memory of the submatrices A_{ij} of A that are being processed is very important.

1 Introduction

PARA 2010 coincided with my 75th birthday. I want to thank the organizers for allowing me to address the attendees about the subject of Cache Blocking. I devoted a lot of time during my last 25 years at IBM research on library development; this library was called ESSL for IBM Engineering Scientific Subroutine Library. In 2011, ESSL celebrated its 25th birthday. In the area of Dense Linear Algebra, DLA, ESSL is compatible with LAPACK; ESSL has a parallel library called PESSL and it is compatible with ScaLAPACK. DLA is a subject that benefits greatly from the use of cache blocking, and DLA researchers have contributed heavily to the development and understanding of cache blocking. This paper describes cache blocking as it relates to DLA.

We cover the where, when, how and why of cache blocking. The where is everywhere. By this we mean that almost all processors use a design that incorporates cache blocking; i.e., their memory hierarchies are designed in a tiered fashion called caches. Processing of data only occurs in the lowest level caches; today these data processing areas are called cores.

The when occurred in the mid 1980's when caches were first introduced. Cache blocking was first invented by my group at IBM in 1984 [21] and the Cedar project at the University of Illinois [12]. As processor speeds increased, the law of physics that governed the speed of light started to affect all processor design. Previously, processors had a uniform or single memory hierarchy and thus all data could be ready for processing in a single CPU operation. Uniform memory processors of the 1980's were the Cray 1 machines. However, later on, the Cray 2 machines became cache based machines. DLA researchers then introduced the Level-3 BLAS [10] for improving new DLA libraries that were later introduced. Two examples were the LAPACK and ScaLAPACK libraries. This was a time period when "Moore's law" started to come into play; this law accurately predicted processor speed increases for the next twenty years. Those speed increases ended around 2005 with the introduction of multi-core, MC.

The how or what can be described as an application of the Algorithms and Architecture Approach [15]. Linear Algebra has a “fundamental principle” called the “Principle of Linear Superposition”. Using it, one can describe the factorization algorithms of DLA of a matrix A in terms of its sub matrices A_{ij} instead of its elements a_{ij}. This leads to automatic cache blocking! The LAPACK and ScaLAPACK libraries are based on this fundamental principle.

Now we describe the why of Cache Blocking. Simply put “why” has to do with the speed of data processing. For peak performance of DLA factorization, all matrix operands must be used multiple times when they enter an L1 cache. This ensures that the initial cost of bringing an operand into cache is amortized by the ratio of $O(n^3)$ arithmetic to $O(n^2)$ elements. Multiple reuse of all operands can *only* occur if all matrix operands map well into the L1 cache. For MC processors, an L1 cache holds data for the cores. For MC it is critical to get the matrix to the cores as fast as possible. The standard programming interface, called API, of matrices for the BLAS and DLA libraries is the 2-D array of the Fortran and C programming languages. For this API, submatrices A_{ij} are held in 2-D arrays of Fortran and C. They *cannot* be moved to and from the memory hierarchy to various cores in a fast or optimal manner! Using New Data Structures, acronym NDS, to hold these submatrices A_{ij} corrects this problem. We shall “prove” why this is true using dimension theory [28].

Multicore/Manycore (MC) is considered a revolution in Computing. Actually, MC is a radical change in Architectures. We have talked about the fundamental triangle of Algorithms, Architectures and Compilers in [3,15,11]. The fundamental triangle concept says that all three areas are inter-related. This means Compilers and Algorithms must change in significant ways. Over the last five years the LAPACK library has been carefully examined and it is now directed toward basic structural changes to gain better performance on MC. One major change has been to adopt NDS.

For nearly 15 years, Bo Kågström’s Group at Umeå, Sweden, J. Waśniewski’s Team at Danish Technical University in Lyngby, Denmark, and I at IBM Research in Yorktown Heights have been applying recursion and NDS to increase the performance of DLA factorization algorithms, DLAFA. Our results apply equally well to MC processors; e.g., the introduction of NDS [14,15,11,5,16]. In this paper NDS will mean matrix data structures that can be used directly by BLAS-3 kernel routines [15,16]. The essence of MC is many cores on a single chip. The Cell BE (Broadband Engine) is an example. Cell is a heterogeneous chip consisting of a single traditional PPE (Power Processing Element) and 8 SPEs (Synergistic Processing Element) and a novel memory system interconnect. Each SPE core can be thought of as a processor and a “cache memory”. Because of this, “cache blocking” is still very important.

For MC the disproportion between multiple CPU processing and memory speed is much higher. However, the API for BLAS-3 hurts performance as it requires repeated matrix data reformatting from its API to NDS. A new “BLAS-3” concept is to use NDS in concert with “BLAS-3” kernels [15,25,24,8]. For MC, the broad idea of “cache blocking” is mandatory as matrix elements must be fed to the SPE's as fast as possible. The arrangement in memory of the submatrices A_{ij} of A that are being processed is equally important. So, this is what we will call "cache blocking" for MC.

We describe algorithms presented at PARA08, called `VIPX` and `BIPX`, that require little or no extra storage to transpose a M by N rectangular (non-square) matrix A in-place. They are much faster versions of the scalar in-place transpose algorithms presented at PARA06 in [17]. Also of interest and very relevant is my paper with Lars Karlsson and Bo Kågström [20] and Lars' paper [22]. These PARA08 algorithms and those of [22,20] are quite illuminating as they show in a fundamental way why NDS are superior to the current standard matrix formats of DLA. They demonstrate a novel form of cache blocking! It is only when one uses NDS in concert with these algorithms that it is possible to achieve cache blocking. We only use two matrix layouts in this paper. First, we assume that the matrices are stored in Rectangular Block (RB) format. RB format stores a M by N matrix A as contiguous rectangular submatrices A_{ij} of size `MB` by `NB`. Square Block (SB) format is a special case of RB format when the rectangle is a square. It was first introduced in 1997, see [14], and has been described in five recent mini-symposiums at PARA06, PPAM07, PARA08, PPAM09, and PARA10. It turns out that our results on NDS [14,15,11,5,16] are very relevant to MC: Of all 2-D data layouts for common matrix operations SB format minimizes L1 and L2 cache misses as well as TLB misses. The essential reason for this is that a SB of order `NB` is also a contiguous 1-D array of size $\mathtt{NB}^2$ and for almost all cache designs a contiguous array whose size is less than the cache size is mapped from its place in memory into the cache by the *identity* mapping. SB format is the same as block data layout. Block data layout is described in [27] and the authors show that this format leads to minimal L1, L2, TLB misses for matrix operations that treat rows and columns equally.

RB format has a number of other advantages. A major one is that it naturally partitions a matrix to be a matrix of sub-matrices. This allows one to view matrix transposition of a M by N matrix A where $M = m\mathtt{MB}$ and $N = n\mathtt{NB}$ as a **block transposition** of a much smaller m by n block matrix A. However, usually M and N are *not* multiples of `MB` and `NB`. So, RB format as we define it here, would pad the rows and columns of A so that M and N become multiples of some blocking factors `MB` and `NB`. We add that padding appears to be an **essential condition** for this type of "cache blocking". The second format for storing matrices is the standard 2-D array format of the Fortran and C programming languages. For the in-place algorithms we consider it appears that by only using these standard formats it then becomes impossible to achieve highly performing algorithms. In other words, "cache blocking" for DLAFA is *not* possible when one uses the standard API of 2-D arrays to hold a global matrix A.

Cache blocking using NDS will be described in Section 2. We show how it can be automatically incorporated into DLAFA by using NDS and "BLAS-3" kernel routines instead of using the current Level 3 BLAS. Section 2 closes with a discussion of Dimension Theory. It shows why Fortran and C arrays *cannot* be truly multi-dimensional. In Section 3, we describe the features of In-Place

Transformations between standard full layouts of matrices and the new rectangular block (RB) or square block (SB) formats of NDS. These algorithms demonstrate a novel form of cache blocking which is made possible by transforming to and then using the NDS. Serial performance results for the algorithms of Section 3 are given in [19]. Some early history of my involvement with Cache Blocking is given in Section 4. A short Summary and Conclusion is given in Section 5.

2 Cache Blocking

We address "cache blocking" as it relates to DLAFA; see [15]. We will sketch a proof that DLAFA can be viewed as just doing matrix multiplication (MM) by adopting the linear transformation approach of applying equivalence transformations to a set of linear equations $Ax = b$ to produce an equivalent (simpler) form of these equations $Cx = d$. Examples of the simpler form are $LU = PA$, for Gaussian elimination, $LL^T = A$, for Cholesky Factorization, and $QR = A$, for Householder's factorization. We adopt this view to show a general way to produce a whole collection of DLAFA as opposed to the commonly accepted way of describing the same collection as a set of distinct algorithms [13]. A second reason is to indicate that for each linear transformation we perform we are invoking the definition of MM. Here is the gist of the proof as it applies to $LU = PA$.

1. Perform $n = \lceil N/\texttt{NB} \rceil$ rank NB linear transformations on A to get U.
2. Each of these n composed NB linear transformations is MM by definition.
3. By the principle of equivalence we have $Ax = b$ if and only if $Ux = L^{-1}Pb$.

MM clearly involves "cache blocking". Around the mid 1990's we noticed (see page 739 of [14]) that the API for Level 3 BLAS GEMM could hurt performance. In fact, this 1-D API is also the API for 2-D arrays in Fortran and C. An explanation of dimension is given in Section 2.1. One can prove that it is impossible to lay out a matrix in 1-D fashion and maintain closeness of its elements. LAPACK and ScaLAPACK also use this API for full arrays. On the other hand, high performance implementations of GEMM do *not* use this API as doing so leads to sub-optimal performance. In fact, some amount of data copy is usually done by most high performance GEMM implementations. Now, Level 3 BLAS are called multiple times by DLAFA. This means that multiple data copies will usually occur in DLAFA that use standard Level 3 BLAS. The NDS for full matrices are good for GEMM. DLAFA algorithms can be expressed in terms of scalar elements $a_{i,j}$ which are one by one block matrices. Alternatively, they can be expressed in terms of partitioned submatrices, $A(I : I + \texttt{NB} - 1, J : J + \texttt{NB} - 1)$ of order NB. See [13] for a definition of colon notation. The algorithms are almost identical. However, the latter description *automatically incorporates cache blocking* into a DLAFA. Take the scalar statement $c_{i,j} = c_{i,j} - a_{i,k}b_{k,j}$ representing scalar MM as a fused multiply-add. The corresponding statement for partitioned submatrices becomes a kernel routine for Level 3 BLAS GEMM. However, it is imperative to store the order NB SB's as contiguous blocks of matrix data, as this is what many Level 3 BLAS GEMM implementations do internally. This is *not* possible using the standard Fortran and C API. This fact emphasizes the importance of storing the submatrices of DLAFA as contiguous blocks of storage. An essence of NDS for full matrices is to store their submatrices as contiguous blocks of storage. The simple format of full NDS has each RB or SB stored in standard Column Major (CM) or standard Row Major (RM) format; see [15] for details.

Early results of the PLASMA project as it related to the Linpack benchmark $LU = PA$ when running on the Cell processor emphasize their use of SB format [25]. According to Dongarra's Team it was crucial that NDS be used as their matrix format. In particular, using the standard API of Fortran and C did not yield them good performance results. Also, earlier results obtained by considering the IBM new Blue Gene/L computers [9] emphasized the same thing. However, the simple format of full NDS needs to be rearranged internally to take into account "cache blocking" for the L0 cache. The L0 cache is a new term defined in [16] and it refers to the register file of the FPU or core that is attached to the L1 cache. Some ideas about this are given in [16].

2.1 Dimension Theory and Its Relation to Standard CM and RM Arrays of Fortran and C

All multi-dimensional arrays in Fortran and C are actually 1-D layouts. This means that the API for 2-D "arrays in Fortran and C" is really one dimensional. A finite version of the Fundamental Theorem of Dimension Theory implies that it is *impossible* to preserve a *neighborhood principle* of closeness of all points p of a D dimensional object when one uses a d dimensional coordinate system to describe the object when $D > d$; see pages 106 to 120 of [28]. We use the phrase "preserve data locality" and we note that when data is contiguous in computer memory then its mapping into cache is the identity mapping; clearly, this is the fastest way to move data and also to preserve it in cache. This result says that it is impossible to lay out a matrix in 1-D fashion and maintain closeness of all of its elements.

3 In-place Transposition between Standard Full Layouts and RB Format

The in-place transpose algorithms in [17], although fast, or very fast compared to the existing algorithms for the same problem, are necessarily very slow on today's processors. We explain why: Suppose that we have an $M \times N$ matrix A stored in CM format. The element a_{ij} is stored at offset $k = i + jM$ or $A[k]$. These algorithms implement an in-place permutation P of the entries of A so that $a_{ij} = A[k]$ ends up at offset $\bar{k} = iN + j$ or $A[\bar{k}]$. Thus, the algorithms overwrite A stored in CM format with its transpose A^T also stored in CM format. This mapping, $\bar{k} = P(k) = kN \bmod q$ where $q = MN - 1$, is a permutation of the integers $0 : MN - 1$. The mapping P, with different parameters N and q, also defines different pseudo-random number generators [23, Section 3.2.1.3]. Therefore, an algorithm that transposes a matrix in-place using P directly must exhibit a random memory access pattern and thus have very poor performance: Each memory access will likely miss in each level of the cache hierarchy. A cache miss penalty is huge, in the hundreds of cycles, for multi-core processors.

Today's processors lay out memories in chunks of size LS called lines and when an element is accessed the entire line containing the element is brought into the L1 cache. To obtain high performance it is therefore imperative to utilize or process all elements in a line once the element enters the L1 and L0 caches. We speculate that the reason in-place transposition has *not* been used for DLA algorithms is because one can prove that in-place transposition is *impossible* for sub-matrices of a matrix A stored in standard format. Also, these algorithms are *very slow* relative to out-of-place transposition algorithms which are almost universally used instead.

Here we use the RB format, which is a generalization of the more commonly used SB format. It will be evident that our results also hold for SB format. In the RB format version of our new Block In-Place Xpose[1] (BIPX) algorithm our M by N matrix A, usually padded, can be considered a m by n block matrix where each submatrix has MB rows and NB columns. Padding should occur when either $M < m\mathtt{MB}$ or $N < n\mathtt{NB}$. Now the governing permutation P has length $q = mn - 1$. So, each "element" moved is a RB of size MB by NB whose elements are contiguous and hence consist of $\lceil \mathtt{MB} \cdot \mathtt{NB}/\mathtt{LS} \rceil$ contiguous lines. Hence, the performance problems of the two previous paragraphs *disappear* and our BIPX algorithm will perform at about the same speed as current out-of-place algorithms; see [19] for performance results. In this Case 1, the BIPX algorithm has one stage and hence is more efficient than the other two cases which we now describe for transposing in-place matrices A stored in standard Fortran and C 2-D arrays. The second case has three stages and the third case has five stages.

3.1 The Three Stage Case 2 Algorithm

The main idea here is to use the BIPX algorithm and hence we need to transform A, in a standard CM or RM format, to be in RB format. This is done by using a vector version of the IPT or MIPT algorithms of [17]; it is called the VIPX algorithm and it has similar features to our BIPX algorithm. The VIPX algorithm maps in-place a $M = m\mathtt{MB}$ by NB submatrix of A, in standard CM format with $\mathtt{LDA} = m\mathtt{MB}$, to become m size MB by NB RB concatenated together. We call this submatrix of A a column swath of A. Repeating algorithm VIPX n times on the n concatenated column swaths that make up CM A converts CM A to become A in RB format.

Now we can describe the Case 2 algorithm: It assumes CM A has a certain layout space in terms of standard 2-D layout terminology: CM A and RB format A will occupy $M \le m\mathtt{MB}$ by $N \le n\mathtt{NB}$ arrays with $\mathtt{LDA} = m\mathtt{MB}$ where $m = \lceil M/\mathtt{MB} \rceil$. So, the array A holding CM A and RB format A will have space for $m\mathtt{MB}n\mathtt{NB}$ elements where $n = \lceil N/\mathtt{NB} \rceil$. Now we give the three stage algorithm. First algorithm VIPX, applied n times on the n column swaths of A produces

[1] Xpose stands for Transpose.

RB A. Second algorithm BIPX computes RB A^T. Third the inverse of the VIPX algorithm, applied m times on the m column swaths of A^T, computes CM A^T. Hence, it will be less efficient than the Case 1 algorithm by approximately a factor of three. However, it will be *very efficient* compared to a one stage algorithm MIPT or IPT of [17] applied directly to A stored a standard CM or RM format; see the start of Section 3 where the reason why was given.

3.2 The Five Stage Case 3 Algorithm

Clearly, matrix A rarely has its M a multiple of MB and its N a multiple of NB or that A is contained in an array A with size $|\mathtt{A}| \geq m\mathtt{MB}n\mathtt{NB}$ elements. Case 3 is where A is in standard CM format, A has size M by N with $\mathtt{LDA} \geq M$, and the conditions for Case 2 do *not* hold. In Case 3, we set $m_1 = \lfloor M/\mathtt{MB} \rfloor$ and $n_1 = \lfloor N/\mathtt{NB} \rfloor$ to define the space for a $M_1 = m_1\mathtt{MB}$ by $N_1 = n_1\mathtt{NB}$ smaller A_1 submatrix of A inside the original array space of A. This requires that we save the leftover $M - M_1$ rows and $N - N_1$ columns of A in a buffer. We fill this buffer using out-of-place transpose operations on these leftover rows and columns of A. Then we move the M_1 by N_1 matrix A_1 to be in standard CM order. A_1 is now declared in Fortran as $\mathtt{A}(0 : \mathtt{M}_1 - 1, 0 : \mathtt{N}_1 - 1)$ in the array space A. This is easy to do as a series of DCOPY type calls of length M_1. Note that matrix A_1 has its $\mathtt{LDA} = M_1$ and it is now a Case 2 matrix. We apply the Case 2 algorithm to CM A_1 to get CM A_1^T. Next, we expand A_1^T in the array space of A using DCOPY type calls of length N_1 thereby making "holes" in array A for the submatrices of A in the buffer. Finally, we transfer the buffer with the saved rows and columns to the "holes" in A using standard out-of-place transpose and copy algorithms to get the final CM A^T matrix. The Case 3 algorithm contains four additional steps of save $A - A_1$, contract A_1, expand A_1 and restore $A - A_1$ over the Case 2 algorithm and hence is the least efficient of the three algorithms. Since it passes over A about five times it will perform about five times slower than the Case 1 algorithm when A is large; see [18].

We can only describe the BIPX, VIPX and Case 2 algorithms due to space considerations. There is some literature on this subject in the form of a patent disclosure [26] which we discovered after we finished this work. This disclosure is incomplete, and furthermore its algorithms are not really in-place.

3.3 The VIPX(MB,m,NB,A) Column Swath Algorithm

We briefly describe how one gets from standard CM format to RB format. Let $A1$ have $M = m\mathtt{MB}$ rows and $N = n\mathtt{NB}$ columns with its $\mathtt{LDA} = M$. Thus, $A1$ consists of n column swaths that are concatenated together. Denote any such swath as a submatrix $A3$ of $A1$ and note that $A3$ consists of NB contiguous columns of CM matrix $A1$. So, $A3$ has M rows and $s = \mathtt{NB}$ columns. Think of $A3$ as an m by s matrix whose elements are column vectors of length $r = \mathtt{MB}$. Now apply algorithm MIPT or IPT of [17] to this m by s matrix $A3$ of vectors of length r. Now $A3$ has been replaced (over-written) by $A3^T$ which is a size s by m matrix of vectors of length r. It turns out, as a little reflection will indicate, that $A3^T$ can also be viewed as consisting of m RB matrices of size r by s that are concatenated together. For matrix $A1$ we do n parallel $A3 \rightarrow A3^T$ operations for each of the $n = N/s$ concatenated submatrices $A3$ that make up matrix $A1$. After completion of these n parallel computation steps we have transformed CM $A1$ in-place to become matrix $A2$. $A2$ is a RB matrix consisting of m block rows by n block columns stored in standard CM block order. Of course, $A1$ and $A2$ are different representations of the same matrix. However, we have "cache blocked" matrix $A2$! The `VIPX` algorithm just described is algorithm `MIPT` or `IPT` of [17] modified to move contiguous vectors of length r instead of scalars of length one.

3.4 The `BIPX(MB,NB,m,n,A)` Block Transpose Algorithm

We briefly describe how one gets from RB format to the transpose of RB format in-place. Let $A2$ have m block rows and n block columns where each block element of $A2$ is a standard CM matrix having `MB` rows and `NB` columns with `LDA=MB`. These m by n block matrices are laid out in standard CM block order; see Section 3.3. Now apply algorithm `MIPT` or `IPT` of [17] to this m by n matrix $A2$ of RB matrices. $A2$ will be replaced (over-written) by $A2^T$. Each block element of $A2^T$ is a CM matrix having `NB` rows and `MB` columns with `LDA=NB`; ie, each new RB matrix is the transpose of an old RB matrix. The `BIPX` algorithm uses either algorithm `MIPT` or `IPT` of [17] modified to transpose out-of-place RB matrices of size `MB` by `NB` according to a permutation cycle of the `MIPT` or `IPT` algorithm applied to the m by n RB matrix $A2$. Note that any permutation of disjoint block cycles is easy to parallelize. One can clearly see how using $A2$ produces a parallel form of cache blocking!

3.5 The Case 2 In-place Transpose Algorithm

The Case 2 Algorithm was described in the previous Sections 3.1, 3.3, and 3.4. When $M = N$ one calls a standard in-place transpose algorithm. We now present the $M \neq N$ case where `MB = NB`:

```
m1=m/nb ! A is a Fortran m by n matrix declared as A(0:m-1,0:n-1)
n1=n/nb ! A will become a SB matrix of size m1 by n1
nb2=nb*nb ! each SB holds nb^2 matrix elements
if(m1.gt.1)then ! Stage 1 of CM to SB
  call VIPX1(nb,m1,nb,A,temp,L,nL)
  do k=1,n1-1
    call VIPX2(nb,m1,nb,A(k*m*nb),temp,L,nL)
  enddo
endif
if(m1.eq.1.or.n1.eq.1)then ! A is a block vector matrix
  ! A is a m1 by 1 block vector or a 1 by n1 block vector
  do i=0,max(m1,n1)-1 ! max(m1,n1,1) is either m1 or n1
    call DGETMI(A(i*nb2),nb,nb) ! transpose m1 or n1 blocks
  enddo
else ! A is a SB matrix of size m1 by n1; min(m1,n1) > 1
```

```
      call BIPX(m,n,A,m,nb,temp) ! Stage 2 of SB to SB^T
    endif
    if(n1.gt.1)then ! Stage 3 of SB^T to CM
      call VIPX1(nb,nb,n1,A,temp,L,nL)
      do k=1,m1-1
        call VIPX2(nb,nb,n1,A(k*n*nb),temp,L,nL)
      enddo
    endif
```

We have broken stages one and three into an initial call to VIPX1 and their remaining calls to VIPX2. VIPX1 is VIPX of Section 3.1 where we save the leaders [17] of VIPX in vector L of length NL. Hence, further calls to VIPX can be handled by the more efficient VIPX2 which receives its leaders in L as input. Routine DGETMI is the ESSL in-place transpose routine [21]. Array temp will hold a vector of length NB or NB^2.

4 Some Early IBM History on Cache Blocking

In the early 1980's I became manager of a small research group and project called Algorithms and Architectures. IBM was to introduce a Vector Processor into its new cache based 3080 series mainframe line. My group initially had researchers Ramesh Argawal, James Cooley and Bryant Tuckerman. We first produced novel scalar and vector elementary functions that were nearly perfectly rounded and very fast. This work became state-of-the-art [1]; today this design still is. Next came the formation of the IBM product ESSL. This latter work was a joint venture with IBM Development in Poughkeepsie and Kingston, NY headed by Stanley Schmidt and Joan McComb. ESSL was conceived during 1982. For linear algebra, we decide to make ESSL subroutines compatible with Linpack. In May to June of 1984 we produced a successful design of matrix multiply, _GEFA and _POFA. Our internal report said "a conceptual design has been identified in which data is brought (and completely used) into cache only once. This approach allows full use of the multipy add instruction". Thus, this is when "cache blocking" was born in my group. ESSL was initially released in February 1986 [21]; it will celebrate its 25th anniversary in 2011. In 1988, my group showed how "algorithmic lookahead" could be used to obtain perfect parallel speed-up for Linpack benchmark [2]. This key idea is used to get high performance on multi-core processors.

In the late 1980's ESSL and my group was presented a new challenge as IBM decided to introduce RISC computers called the POWER (RS6000) line of workstations. ESSL had grown substantially and had put out four mainframe releases. A huge programming effort began and 497000 lines of Fortran code was produced by my small group of four regular people. We called our effort EFL standing for ESSL Fortran Library; the whole library was written in Fortran! Sometime later Jim Demmel and his graduate students at UC Berkeley started a project with a grant from IBM to try to automatically produce code to get better performance than EFL code. They produced PHIPAC; later Jack Dongarra's group followed with ATLAS. After POWER1, came a remarkable machine called POWER2 [3]. It possessed very high bandwidth. In 1992 my group published a report [4] on how to use overlapped communication to produce peak performing matrix multiplication on distributed memory computers. Today, this algorithm is still the algorithm of choice.

5 Conclusions and Summary

We indicated that DLAFA are mainly MM algorithms. The standard API for matrices use arrays. All array layouts are *one* dimensional. It is *impossible* to maintain locality of reference in a matrix or any higher than 1-D object using a 1-D layout; see [28]. MM requires row and column operations and thus requires matrix transposition (MT). Our results on in-place MT show that performance suffers greatly if one uses a 1-D layout. Using NDS for matrices "approximates" a 2-D layout; thus, one can dramatically improve in-place MT performance. Our message is that DLAFA are mostly MM. MM requires MT and both require NDS. Thus, DLAFA can and do perform well on multicore if one uses NDS.

References

1. Agarwal, R.C., Cooley, J.W., Gustavson, F.G., Shearer, J.B., Slishman, G., Tuckerman, B.: New scalar and vector elementary functions for the IBM System/370. IBM Journal of Research and Development 30(2), 126–144 (1986)
2. Agarwal, R.C., Gustavson, F.G.: A Parallel Implementation of Matrix Multiplication and LU factorization on the IBM 3090. In: Wright, M. (ed.) Proceedings of the IFIP WG 2.5 on Aspects of Computation on Asynchronous Parallel Processors, pp. 217–221. North Holland, Stanford (1988)
3. Agarwal, R.C., Gustavson, F.G., Zubair, M.: Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. IBM Journal of Research and Development 38(5), 563–576 (1994)
4. Agarwal, R.C., Gustavson, F.G., Zubair, M.: A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. IBM Journal of Research and Development 38(6), 673–681 (1994)
5. Andersen, B.S., Gunnels, J.A., Gustavson, F.G., Reid, J.K., Waśniewski, J.: A Fully Portable High Performance Minimal Storage Hybrid Cholesky Algorithm. ACM TOMS 31(2), 201–227 (2005)
6. Anderson, E., et al.: LAPACK Users' Guide Release 3.0. SIAM, Philadelphia (1999)
7. Blackford, L.S., et al.: ScaLAPACK Users' Guide. SIAM, Philadelphia (1997)
8. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algorithms for multicore architectures. Parallel Comput. 35(1), 38–53 (2009)
9. Chatterjee, S., et al.: Design and Exploitation of a High-performance SIMD Floating-point Unit for Blue Gene/L. IBM Journal of Research and Development 49(2-3), 377–391 (2005)
10. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: A Set of Level 3 Basic Linear Algebra Subprograms. TOMS 16(1), 1–17 (1990)

11. Elmroth, E., Gustavson, F.G., Jonsson, I., Kågström, B.: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM Review 46(1), 3–45 (2004)
12. Gallivan, K., Jalby, W., Meier, U., Sameh, A.: The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design. International Journal of Supercomputer Applications 2(1), 12–48 (1988)
13. Golub, G., VanLoan, C.: Matrix Computations, 3rd edn. John Hopkins Press, Baltimore (1996)
14. Gustavson, F.G.: Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. IBM Journal of Research and Development 41(6), 737–755 (1997)
15. Gustavson, F.G.: High Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices. IBM Journal of Research and Development 47(1), 31–55 (2003)
16. Gustavson, F.G., Gunnels, J.A., Sexton, J.C.: Minimal Data Copy for Dense Linear Algebra Factorization. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 540–549. Springer, Heidelberg (2007)
17. Gustavson, F.G., Swirszcz, T.: In-Place Transposition of Rectangular Matrices. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 560–569. Springer, Heidelberg (2007)
18. Gustavson, F.G., Gunnels, J., Sexton, J.: Method and Structure for Fast In-Place Transformation of Standard Full and Packed Matrix Data Formats. United State Patent Office Submission YOR920070021US1 and Submission YOR920070021US1(YOR.699CIP) US Patent Office, 35 pages (September 1, 2007); 58 pages (March 2008)
19. Gustavson, F.G.: The Relevance of New Data Structure Approaches for Dense Linear Algebra in the New Multicore/Manycore Environments, IBM Research report RC24599; also, to appear in PARA 2008 proceeding, 10 pages (2008)
20. Gustavson, F.G., Karlsson, L., Kågström, B.: Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion. ACM TOMS, 34 pages (to appear 2011)
21. IBM. IBM Engineering and Scientific Subroutine Library for AIX Version 3, Release 3. IBM Pub. No. SA22-7272-00 (February 1986)
22. Karlsson, L.: Blocked in-place transposition with application to storage format conversion. Tech. Rep. UMINF 09.01. Department of Computing Science, Umeå University, Umeå, Sweden (January 2009) ISSN 0348-0542
23. Knuth, D.: The Art of Computer Programming, 3rd edn., vol. 1 & 2. Addison-Wesley (1998)
24. Kurzak, J., Buttari, A., Dongarra, J.: Solving systems of Linear Equations on the Cell Processor using Cholesky Factorization. IEEE Trans. Parallel Distrib. Syst. 19(9), 1175–1186 (2008)
25. Kurzak, J., Dongarra, J.: Implementation of mixed precision in solving mixed precision of linear equations on the Cell processor: Research Articles. Concurr. Comput.: Pract. Exper. 19(10), 1371–1385 (2007)
26. Lao, S., Lewis, B.R., Boucher, M.L.: In-place Transpose United State Patent No. US 7,031,994 B2. US Patent Office (April 18, 2006)
27. Park, N., Hong, B., Prasanna, V.: Tiling, Block Data Layout, and Memory Hierarchy Performance. IEEE Trans. Parallel and Distributed Systems 14(7), 640–654 (2003)
28. Tietze, H.: Three Dimensions–Higher Dimensions. In: Famous Problems of Mathematics, pp. 106–120. Graylock Press (1965)

A Model for Efficient Onboard Actualization of an Instrumental Cyclogram for the Mars MetNet Mission on a Public Cloud Infrastructure[*]

Jose Luis Vázquez-Poletti[1], Gonzalo Barderas[2], Ignacio M. Llorente[1], and Pilar Romero[2]

[1] Departamento de Arquitectura de Computadores y Automática
Facultad de Informática, Universidad Complutense de Madrid, Spain
28040 Madrid, Spain

[2] Sección Departamental de Astronomía y Geodesia
Facultad de Matemáticas, Universidad Complutense de Madrid
28040 Madrid, Spain

Abstract. Until now, several heuristics for scheduling parameter sweep applications in environments such as cluster and grid have been introduced. Cloud computing has revolutionized the way applications are executed in distributed environments, as now it is the infrastructure which is adapted to the application and not vice versa. In the present contribution an astronomy application from the next mission to Planet Mars with Finnish-Russian-Spanish flag is ported on to a cloud environment, resulting in a parameter sweep profile. The number of needed executions and the deadline provided required a big quantity of computing resources in a short term and punctual situations. For this reason, we introduce and validate a model for an optimal execution on a public cloud infrastructure by means of time, cost and a metric involving both.

1 Introduction

Cloud computing allows access to an on-demand and flexible computing infrastructure. As soon as production infrastructures have been available to the scientific community, the first applications have started to run on the cloud [1,2]. In many Research areas, the leap from cluster and grid computing to this new paradigm has been mandatory, as the needed applications evolve in their computational needs [3].

In cloud computing a research institution does not need to care about its own cluster machines, neither the availability of remote computing resources nor the software installed on them. When needed, an user may start a virtual machine (VM), or even a group of identical VMs booted from the same image. Another remarkable fact is that in cloud computing is the application which defines its

[*] This research was supported by the following projects: MEDIANET (Comunidad de Madrid S2009/TIC-1468), HPCcloud (MICINN TIN2009-07146) and MEIGA-METNET-PRECURSOR (AYA2009-14212-C05-05/ESP).

level of parallelism and not the hardware, unlike in cluster or grid computing. A cloud user may choose from two main infrastructure types: public or private. In a private cloud (also named internal or corporate cloud) the *bare-metal* machines, those hosting the running VM's, are maintained by his own Institution. These private clouds can be built with virtualization technologies such as Nimbus [4], OpenNebula [5] or Eucalyptus [6]. On the other hand, users may choose to externalize the cloud service and pay per deployed VM and unit of time, like in ElasticHosts[1] and Amazon's Elastic Compute Cloud[2].

In the present contribution, an astronomy application used in the context of the Finnish-Russian-Spanish Mission to Mars that will be launched in 2011 (Section 2) is studied for its optimal execution on Amazon's cloud infrastructure (Sections 3 and 4). The reasons for choosing a public cloud infrastructure are that executions are meant to be sporadic, intensive and the existing computing infrastructure is very limited. Upgrading this computing infrastructure was discarded, so the *pay-as-you-go* philosophy suits perfectly the application. With this in mind, the objective then was a valid model that allows to choose the best setup by means of total execution time, cost and a metric involving both (Section 5) and validate it through experimental results given an infrastructure setup (Section 6). This model provided in this contribution is ready to be used not only in this mission to Planet Mars but in the next ones.

2 Phobos Eclipses on Mars for the MetNet Precursor Lander localization

The MetNet Mars Precursor Mission (MMPM) is a new type of atmospheric science mission to Mars. The project is being fulfilled in collaboration between the Finnish Meteorological Institute (FMI), the Russian Lavoschkin Association (LA), the Russian Space Research Institute (IKI) and the Spanish National Institute for Aerospace Technology (INTA). The purpose of the MMPM is to confirm the concept of deployment of mini-meteorological stations onto the Martian surface to get atmospheric data during the descent phase and at the landing site with a life time design goal of several martian years. The probe is planned to be launched in 2011 as a secondary part of the Russian mission PhobosGrunt.

The determination of the landing site coordinates is fundamental to provide useful information for both scientific and mission engineering goals. The detection of the shadowing effect of Phobos on Mars is proposed to solve the localization problem as an alternative and to complement the use of radiometric signals. In order to implement an observational strategy to observe Phobos eclipses, an algorithm has been developed and coded to determine the eclipse conditions as well as the determination of the shadow motion in latitude and longitude. The implementation of the observational strategy is limited due to the ambiguity in the EDLS (Entry, Descent and Landing Site) concept. Thus, the cyclogram

[1] http://www.elastichosts.com/

[2] http://aws.amazon.com/ec2/

must be activated at the moment when EDSL conditions are known. Cloud will enable us to face with this punctual huge volume of computations in a fast and efficient way.

3 Phobos Eclipses on the Cloud

The original application, coded in Fortran 77, was not implemented with parallelization in mind so it needed an upper layer for distributing it. It's executable filesize is 754KB and it processes a single 4KB file where the tracing interval dates and times are read. The size of the resulting output file, containing Phobos' trajectories, varies depending the tracing interval and precision. The resulting parallelized application pertains to the parameter sweep profile, where independent tasks are executed having the executable in common but not the input [7,8,9]. This is the simplest distributed application profile and it is not new, as it has been used since the early stages of cluster [10] and grid computing [11,12].

Related work on parameter sweep applications on distributed environments was always based in finding the best scheduling heuristic. On a cloud environment scheduling is changed to provisioning, as resources can be adapted to the application and can be considered almost infinite. The present contribution does not aim a scheduling heuristic but a provisioning model for parameter sweep applications like the studied one, where homogeneous tasks are executed on a public cloud infrastructure.

Cost analysis for scientific applications on the cloud has already already been done. For instance, [13] shows a performance comparison on three different scientific workflows and analyses different aspects such as I/O, Memory and CPU usage. On the other hand, [14] focuses on one of these data intensive workflows and studies through a grid computing simulator [15] the cost performance tradeoffs for different executions and provision strategies. In both works, the use of cloud storage is highly recommended due to task data dependencies. The present contribution provides a valid model for an execution intensive application with few and little data transfers. For this reason, it has been studied more from a CPU usage point of view even if some assumptions regarding transfer are made, and data storage on the cloud was not considered as a solution. The application was brought to a basic level as it will be explained in the next Section, in order to avoid middleware overheads inherent to grid or cluster computing.

Before the landing of the Mars probe, tracing must be done for different possible locations, and each tracing with a time lapse of $1''$. Summing up all the tracing intervals for all candidate landing coordinates given a possible area, the result is approximately 800 years. The EDSL conditions will be only known $1h30'$ before the beginning of the landing procedure, becoming this a deadline for the computations. The obtained cyclogram is then sent to the probe for instrumental actualization and no more executions would be needed for the rest of the mission. As stated before, the Cloud Computing approach on a public infrastructure suits

Table 1. Characteristics of the different machine types offered by Amazon EC2 in the USA-East infrastructure. C.U. corresponds to EC2 compute units per core, the equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

Machine Type	Cores	C.U.	Memory	Platform	Price/hour
Small (Default)	1	1	1.7GB	32bit	$0.085
Large	2	2	7.5GB	64bit	$0.34
Extra Large	4	2	15GB	64bit	$0.68
High CPU Medium	2	2.5	1.7GB	32bit	$0.17
High CPU Extra Large	8	2.5	7GB	64bit	$0.68

perfectly this punctual need of HPC power. Experiments done for this work are performed in a fixed location but starting in the year 1609, when astronomer Galileo Galilei crafted his first telescope. This way, results from 400 years ago can be compared with historical data, and new data can be gathered for the next 400 years.

4 Cloud Infrastructure

The Amazon EC2 cloud, based in two USA locations and in one European, provides its users a wide range of machine images that can be instantiated in several modalities. These modalities or instance types depend on the memory and number of cores per virtual machine as detailed in Table 1. The user is given a single or a set of virtual machines with the requested specifications and the exclusivity of operation at all levels. When there is no need of the computing resources anymore, the user only has to terminate them. However, accessing to an almost infinite computing infrastructure is not free. Having chosen the cloud services provided by Amazon has its price, as can be seen in Table 1, this depends on the type of VM instantiated per hour.

The system implemented for tracing Phobos' trajectories consisted in a physical machine located at Universidad Complutense of Madrid (Spain), that coordinates the distributed execution of the application at the virtual machines through a Perl script. These VM's were instantiated from images provided by Alestic.com with Ubuntu 9.10 Karmic Koala Server installed, following a collection of best practices collected from the EC2 and Ubuntu communities. The duration of the tracing interval processed by each task was presumed to be the same, as performance does not vary with different tracing dates but with different tracing intervals in small local tests. However, in order to anticipate to irregular times in large scale executions, the system performs dynamic scheduling where a continuous polling of free cores guarantees a constant resource harnessing.

5 Execution Model

The process for the execution model formulation is divided in three stages. Firstly, the application tracing Phobos is executed for a reduced number of years in three different Amazon EC2 instances. After that, individual task behaviour is extrapolated to the rest of instance types with different tracing intervals. Finally the best cases are identified so the complete execution model is formulated.

5.1 Individual Task Characterization

Both the application and input filesize are constant, but this is not the case of the output file when the tracing interval varies. Resulting filesizes from the single experiments conducted are shown with their linear regression in Figure 1a. Moving to execution times, these clearly depend on the selected instance and in particular, its EC2 Compute Units. As explained in Table 1, this metric applied to instance cores is a way to establish a speed relationship between them. Because of instances with the same EC2 Compute Units value, experiments were conducted only with the Small, Large and HighCPU-Medium types, as the difference with the rest reside in the number of cores. Only one core was used on each instance by experiment as shared memory latency is not considerable in this particular execution profile.

The results of these initial experiments are represented along with a linear regression for each instance in Figure 1b. The differences between slopes match clearly each instance type core speed: 1 for the Small, 2 for the Large and 2.5 for the HighCPU-Medium type. Even if there is a big difference between Large and Small instance times, this is not the case of HighCPU-Medium and Large. This difference would surely turn more evident if increasing the tracing intervals range. Additionally, data transfers were not taken into account during these experiments because they take place during the following task execution, as it will be explained in Section 5.3.

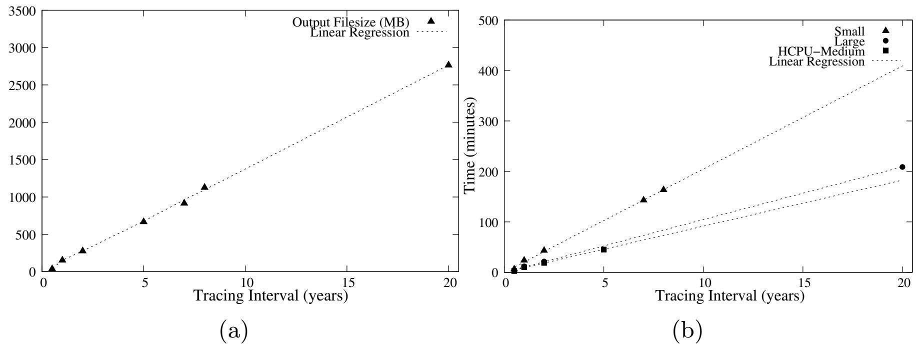


Fig. 1. Output filesizes corresponding to different tracing intervals *(a)* and execution times corresponding to different tracing intervals and Amazon EC2 instances *(b)*

5.2 Full Experiment Characterization

Having the formulas for execution times of individual tasks as a starting point, it is possible to provide the expressions for every desired experiment. This way, the total execution time can be obtained with the following formula:

$$T = \frac{T_{exe} I}{i N_{vm}}, \tag{1}$$

where task execution time ($Texe$) is calculated using the linear regression obtained in the previous Section, depending on the selected instance. Because of equal CPU characteristics, task execution time for Extra-Large instances is calculated with the Large Formula, and the same happens with HighCPU-ExtraLarge and HighCPU-Medium instances. The values of I and i correspond to the whole tracing interval and the tracing interval per task. Finally, N_{vm} is the number of Virtual Machines instantiated in the experiment.

But as explained before, time is not the only aspect to be considered when executing an application on a public Cloud infrastructure. Cost must be considered and it can be calculated with:

$$C = \frac{C_h N_{vm}}{N_c} \lceil T \rceil \tag{2}$$

where C_h is the machine's usage price per hour as shown in Table 1, and T is obtained from the previous formula and obviously expressed in hours. Variable T is rounded here because prices correspond to each usage hour of the requested instance. Existing instances with more than a core (Table 1) are considered by using the N_c variable. Nevertheless, there is a need of finding a compromise between execution time and cost in order to evaluate the best setup. This is accomplished by the Cost/Performance (C/P) metric, which establishes relationship between both of them and can be obtained by multiplying Cost (C) by Time (T), being the best setup that with the lowest metric value.

5.3 Model Formulation

Focusing in a general model that would establish the best conditions for executing this application on a public cloud infrastructure given an instance type, the following expression can be formulated:

$$C/P_{best} = min(C/P) = min(CT) = min(\frac{C_h T_{exe} I}{i N_c^2} \lceil \frac{T_{exe} I}{i N_{vm} N_c} \rceil) \tag{3}$$

where the used variables correspond to those used in Formulas 1 and 2 and their values depend on the selected instance type. At this point, a procedure for using the proposed model can be established: *(i)* obtain the best C/P values for each instance type, *(ii)* evaluate which is the best instance type by means of walltime or cost, and *(iii)* obtain the number of needed virtual machines from the expression. These steps were applied for each instance shown in Table 1. In

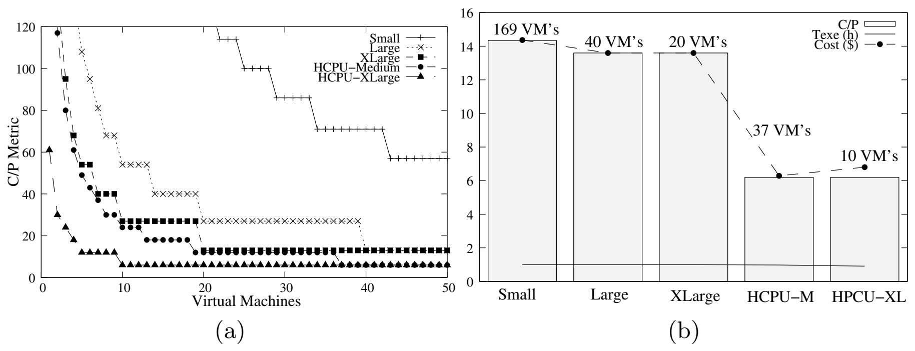


Fig. 2. Comparison of the C/P Metric values for all the considered Amazon EC2 instances with a tracing interval of 0.5 years *(a)*, and execution time ($Texe$), number of virtual machines and cost associated to the best values for the C/P Metric *(b)*

every estimation, 800 years of Phobos' orbit tracing were considered varying the interval per task from 6 months to 12 years, and the number of virtual machines instantiated.

The lowest values for the C/P metric in all the estimated experiments correspond to the 0.5 years tracing interval. As seen on Figure 2a, the minimum value is reached with a different number of virtual machines depending the considered instance type. Figure 2b identifies these values for each case and its associated cost and execution time. Analysing the Large and XLarge cases, equal C/P values and cost are obtained but the number of virtual machines varies in a $2x$ factor. This relationship is the same for the number of cores in the chosen instance types as shown in Table 1. This behaviour is not strictly repeated in the HighCPU instances. The HighCPU-Medium case uses 37 instances which is near to 4 times the number of instances used by the HighCPU-XLarge case, as the number of cores of this instance type is 4 times those from the first one (Table 1). The reason of this difference even with the same C/P values, is that the execution time for HighCPU-Medium's best case is 0.98 and that of HighCPU-XLarge is 0.91. Additionally, this difference in needed virtual machines results in a drop of the total cost.

Choosing the instance type is translated to choosing to pay more for the infrastructure or decrease the level of parallelism. Returning to Figure 2a, the Small instances approach is the most expensive solution but with the highest level of parallelism (169 simultaneous cores). The rest of instance types provide around 80 simultaneous cores for a lower price finding a great difference when moving from Large and XLarge to HighCPU instances, but not from Small to Large and XLarge. Nevertheless, having considered only execution times at the simulations, an overhead in the walltime must be assumed. These overheads are due to data transfers between the execution nodes and the local machine. At the very beginning of the experiment, the executable must be copied to each machine and then, before each single execution, the corresponding input file has to be placed in the working directory. This preprocess results in copying 758 KB the

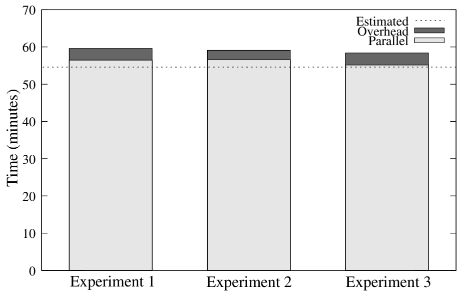


Fig. 3. Parallel time (that computed with task mean execution times) and overhead for the three most representative experiments. All of them are compared with the value expected by the model.

first time and 4 KB the rest of times. However, with the 0.5 years tracing interval, a 37 MB output file is generated by task. Data transfers can be done during the execution of following tasks, but after the last one, there will be as many transfers as parallel cores in the infrastructure. Considering these overheads, it's necesary to count on a setup that does not execute everything in strictly 1 hour, as any delay will result in paying the price for another hour. For this reason, both the HighCPU instance types are interesting, as they offer a margin of $1'02''$ in the HighCPU-Medium case and $5'04''$ in the HighCPU-XLarge case.

6 Experimental Results

In order to validate the proposed model, the HighCPU-XLarge case was chosen where 10 virtual machines are instantiated. The reason for choosing this instance type and not the HighCPU-Medium was that despite the slight cost increase (Figure 2b), the time margin for the last data transfers and overheads is bigger, as it was explained in the previous Section. Measurements did not consider the creation of local directories with the input files, initial executable upload (one for each instance) and the output files retrieval. On the other hand, input file transfer was considered, as this was done each time a task was assigned to a free core. From the different experiments performed, the results from the 3 most representative ones are shown in Figure 3. These results correspond to the longest, an average valued and the shortest experiment. In the Figure, the bars represent the walltime for each experiment and the dotted line, the estimated value. The walltime is decomposed in two values: the parallel time and the overhead. The parallel time is that considering that all tasks are executed in the same time, which is the mean. Consequently, the parallel time is the result of multiplying the task mean time by 20, which is the number of executions harnessing all the available cores, for processing the 1600 tasks from the experiment. The overhead is then the difference between the walltime and the parallel time.

From Figure 3 it can be deduced that the parallel time value is nearer to the expected value than the walltime. This difference can be understood if analysing

Table 2. Mean times for each experiment with their standard deviation and the number of outliers (tasks with $T_{exe} > 5'$)

Experiment	Mean	Std. Deviation	Outliers
1 (long)	2' 49.37"	3.41"	5
2 (avg.)	2' 49.65"	3.89"	8
3 (short)	2' 45.44"	6.91"	5

the task times, which mean values and standard deviation are shown in Table 2. There were some tasks with an execution time over 5 minutes which is almost the double of the expected and the mean time, but no $4'$ values were obtained. In fact, values leap from about $3'30''$ to nearly $5'50''$. This behaviour is sporadic, not subject to a specific task and has its origin in Amazon EC2 itself. Related work on Amazon EC2 benchmarking claim that it is not always providing what it is paid for in terms of available CPU [16]. The effect of these outliers, which represent the 0.5% of the tasks, can be seen more in detail in the first two experiments. Being the first one the longest, its task mean execution time is less than that from the second, but the number of outliers is greater as also shown in Table 2.

7 Conclusions

Cloud computing is a paradigm that aids scientific areas with a high demand of flexible computational resources. The computational challenge has been moved from task scheduling to resource provisioning, as this time the distributed application does not adapt itself to the infrastructure but vice versa. Additionally, using a public cloud infrastructure results in a price to be paid. In the present contribution, a parameter sweep application pertaining to the astronomy domain is studied for its execution on Amazon's public cloud. A Model for obtaining the best infrastructure setup by means of walltime, cost and a metric involving both has been introduced and validated. Even if there was a small percentage of execution times with outlier values due to Amazon's service, the outcome was that expected, reaching the deadline requirements.

This has been the first use of a public cloud infrastructure made by applications pertaining to a Space mission (NASA started to use Amazon EC2 services on November 2010). Due to the great results, more MetNet Project applications are being considered for their porting to a cloud infrastructure (public and private) as more missions are expected. Additionally, the Cluster instances offered since July and November are to be considered for this and future applications.

References

1. Sterling, T., Stark, D.: A High-Performance Computing Forecast: Partly Cloudy. Computing in Science and Engineering 11, 42–49 (2009)

2. Vouk, M.A.: Cloud computing – Issues, research and implementations. In: Proc. 30th International Conference on Information Technology Interfaces, pp. 31–40. IEEE Press (2008)
3. Foster, I., Zhao, Y., Raicu, I., Lu, S.: Cloud Computing and Grid Computing 360-Degree Compared. In: Proc. Grid Computing Environments Workshop, pp. 1–10. IEEE Computer Society (2008)
4. Foster, I., Freeman, T., Keahey, K., Scheftner, D., Sotomayor, B., Zhang, X.: Virtual Clusters for Grid Communities. In: Proc. 6th IEEE Int. Symp. on Cluster Computing and the Grid (CCGrid 2006), pp. 513–520. IEEE Computer Society (2006)
5. Moreno, R., Montero, R.S., Llorente, I.M.: Elastic Management of Cluster-based Services in the Cloud. In: Proc. 1st Workshop on Automated Control for Datacenters and Clouds (ACDC 2009) on the 6th International Conference on Autonomic Computing and Communications, pp. 19–24. ACM Digital Library (2009)
6. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus Open-Source Cloud-Computing System. In: Proc. 9th IEEE Int. Symp. on Cluster Computing and the Grid (CCGrid 2009), pp. 124–131. IEEE Computer Society (2009)
7. Pinedo, M.: Scheduling: Theory, Algorithms, and Systems, 2nd edn. Prentice Hall (2002)
8. Chrétienne, P., Coffman Jr., E.G., Lenstra, J.K., Liu, Z. (eds.): Scheduling Theory and its Applications. John Wiley and Sons (1995)
9. El-Rewini, H., Ali, H.H., Lewis, T.G.: Task Scheduling in Multiprocessing Systems. Computer 28, 27–37 (1995)
10. Hsu, T.S., Lee, J.C., Lopez, D.R., Royce, W.A.: Task Allocation on a Network of Processors. IEEE Trans. Computers 49(12), 1339–1353 (2000)
11. Casanova, H., Legrand, A., Zagorodnov, D., Berman, F.: Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In: Ninth Heterogeneous Computing Workshop, pp. 349–363. IEEE Computer Society Press (2000)
12. Vázquez-Poletti, J.L., Huedo, E., Montero, R.S., Llorente, I.M.: A Comparison Between two Grid Scheduling Philosophies: EGEE WMS and GridWay. Multiagent and Grid Systems 3, 1339–1353 (2007)
13. Deelman, E., Singh, G., Livny, M., Berriman, B., Good, J.: The cost of doing science on the cloud: the Montage example. In: Procs. 2008 ACM/IEEE Conference on Supercomputing, SC 2008. IEEE Press (2008)
14. Juve, G., Deelman, E., Vahi, K., Mehta, G., Berriman, B., Berman, B.P., Maechling, P.: Scientific workflow applications on Amazon EC2. In: Proc. 5th IEEE International Conference on E-Science (Workshops), pp. 14–18. IEEE Computer Society Press (2009)
15. Buyya, R., Murshed, M.: GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. Concurrency and Computation: Practice and Experience 14, 1175–1220 (2002)
16. Walker, E.: Benchmarking Amazon EC2 for HP Scientific Computing. Login 33, 18–23 (2008)

Impact of Asynchronism on GPU Accelerated Parallel Iterative Computations

Sylvain Contassot-Vivier[1,2], Thomas Jost[2], and Stéphane Vialle[2,3]

[1] Loria, University Henri Poincaré, Nancy, France
Sylvain.Contassotvivier@loria.fr
[2] AlGorille INRIA Project Team, France
Thomas.Jost@loria.fr
[3] SUPELEC - UMI 2598, France
Stephane.Vialle@supelec.fr

Abstract. We study the impact of asynchronism on parallel iterative algorithms in the particular context of local clusters of workstations including GPUs. The application test is a classical PDE problem of advection-diffusion-reaction in 3D. We propose an asynchronous version of a previously developed PDE solver using GPUs for the inner computations. The algorithm is tested with two kinds of clusters, a homogeneous one and a heterogeneous one (with different CPUs and GPUs).

Keywords: Parallelism, GPGPU, Asynchronism, Scientific computing.

1 Introduction

Scientific computing generally involves a huge amount of computations to obtain accurate results on representative data sets in reasonable time. This is why it is important to take as much advantage as possible of any new device which can be used in the parallel systems and bring a significant gain in performances. In that context, one of our previous works was focused on the use of clusters of GPUs for solving PDEs [19]. The underlying scheme is a two-stage iterative algorithm in which the inner linear computations are performed on the GPUs [18]. Important gains were obtained both in performance and energy consumption. Since the beginning of parallelism, several works related to asynchronism in parallel iterative algorithms (see for example [7,10,2]) have shown that this algorithmic scheme could be a very interesting alternative to classical synchronous schemes in some parallel contexts. Although a bit more restrictive conditions apply on *asynchronous parallel algorithms* [5], a wide family of scientific problems support them. Moreover, contexts in which this algorithmic scheme is advantageous compared to the synchronous one have also been identified. As asynchronism allows an efficient and implicit overlapping of communications by computations, it is especially well suited to contexts where there is a significant ratio of communication time relatively to the computation time. This is for example the case in large local clusters or grids where communications through the system are expensive compared to local accesses.

Our motivation for conducting the study presented in this paper comes from the fact that a local cluster of GPUs represents a similar context of costly communications according to computations. Indeed, the cost of data transfers between the GPU memory and the CPU memory inside each machine is added to the classical cost of local communications between the machines. So, we propose in this work to study the interest of using asynchronism in our PDE solver in that specific context.

In fact, our long term objective is to develop auto-adaptive multi-algorithms and multi-kernels applications in order to achieve optimal executions according to a user defined criterion such as the execution time, the energy consumption, or the energy-delay product [15]. We aim at being able to dynamically choose between CPU or GPU kernels and between *synchronous* or *asynchronous* distributed algorithms, according to the nodes used in a cluster with heterogeneous CPUs and GPUs.

The test application used for our experiments in this study is the classical advection-diffusion-reaction problem in a 3D environment and with two chemical species (see for example [17]). Two series of experiments have been performed, one with a homogeneous cluster and another one with a heterogeneous cluster with two couples of CPU-GPU. Both computing performances and energy consumption have been measured and analyzed in function of the cluster size and the cluster heterogeneity.

The following section presents the algorithmic scheme of our iterative PDE solver together with the implementation sketch of the asynchronous version. Then, the experiments are presented and the results are discussed in Section 3.

2 Asynchronous PDE Solver

It is quite obvious that over the last few years, the classical algorithmic schemes used to exploit parallel systems have shown their limit. As the most recent systems are more and more complex and often include multiple levels of parallelism with very heterogeneous communication links between those levels, one of the major drawbacks of the previous schemes has become their synchronous nature. Indeed, synchronizations may noticeably degrade performances in large or hierarchical systems, even for local systems (*i.e.* physically close nodes connected through a fast local network).

Since the very first works on asynchronous iterations [9,20,4], the interest of those schemes has increased in the last few decades [5,8,1,13,14]. Although they cannot be used for all problems, they are efficiently usable for a large part of them. In scientific computing, asynchronism can be expressed only in iterative algorithms. We recall that iterative methods perform successive approximations toward the solution of a problem (notion of convergence) whereas direct methods give the exact solution within a fixed number of operations. Although iterative methods are generally slower than direct ones, they are often the only known way to solve some problems. Moreover, they generally present the advantage of being less memory consuming.

The asynchronous feature consists in suppressing any idle time induced by the waiting for the dependency data to be exchanged between the computing units of the parallel system. Hence, each unit performs the successive iterations on its local data with the dependency data versions it owns at the current time. The main advantage of this scheme is to allow an efficient and implicit overlapping of communications by computations. On the other hand, the major drawbacks of asynchronous iterations are: a more complex behavior which requires a specific convergence study, and a larger number of iterations to reach convergence. However, the convergence conditions in asynchronous iterations are verified for numerous problems and, in many computing contexts, the time overhead induced by the additional iterations is largely compensated by the gain in the communications [2]. In fact, as partly mentioned in the introduction, as soon as the frequency of communications relatively to computations is high enough and the communication costs are larger than local accesses, an asynchronous version may provide better performances than a synchronous version.

2.1 Multisplitting-Newton Algorithm

There are several methods to solve PDE problems, each of them including different degrees of synchronism/asynchronism. The method used in this study is the multisplitting-Newton [12] which allows for a rather important level of asynchronism [21]. In that context, we use a finite difference method to solve the PDE system. Hence, the system is linearized, a regular discretization of the spatial domain is used and the Jacobian matrix of the system is computed at the beginning of each simulation time step. The Euler equations are used to approximate the derivatives. Since the size of the simulation domain can be huge, the domain is split and homogeneously distributed among several nodes of a cluster. Each node solves a part of the resulting linear system and sends the relevant updated data to the nodes that need them. The algorithmic scheme of the method is as follows:

– Initialization:
  - Rewriting of the problem under a fixed point problem formulation: $x = T(x), x \in \mathbb{R}^n$ where $T(x) = x - F'(x)^{-1}F(x)$ and F' is the Jacobian
  - We get $F' \times \Delta x = -F$ with F' a sparse matrix (in most cases)
  - F' and F are homogeneously distributed over the computing units
– Iterative process, repeated for each time step of the simulation:
  - Each unit computes a different part of Δx using the quasi-Newton algorithm over its sub-domain as can be seen in Fig. 1
  - The local elements of x are directly updated with the local part of Δx
  - The non-local elements of x come from the other units using messages exchanges
  - F is updated by using the entire vector x

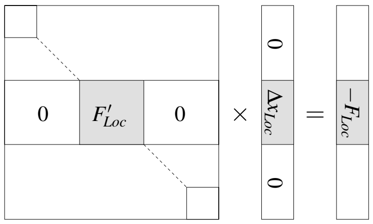


Fig. 1. Local computations associated with the sub-domain of one unit

2.2 Inner Linear Solver

The method described above is a two-stage algorithm in which a linear solver is needed in the inner stage. In fact, most of the time of the algorithm is spent in that linear solver. This is why we chose to use the most powerful elements of the parallel system on that part. Thus, the linear computations have been placed on the GPUs. Due to their regularity, those treatments are very well suited to the SIMD architecture of the GPU. Hence, on each computing unit, the linear computations required to solve the partial system are performed on the local GPU while all the algorithmic control, non-linear computations and data exchanges between the units are done on the CPU.

The linear solver has been implemented both on CPU and GPU, using the biconjugate gradient algorithm [11]. This linear solver was chosen because it performs well on non-symmetric matrices (on both convergence time and numerical accuracy), it has a low memory footprint, and it is relatively easy to implement. At very early stages of development, we also tried to use the Bi-CGSTAB algorithm [22] and local preconditioners (Jacobi and SSOR), but this provided very little or no gain in terms of computing time and numerical accuracy, so we decided to keep the first, simpler solution.

GPU Implementation. Several aspects are critical in a GPU: the regularity of the computations, the memory which is of limited amount and the way the data are accessed. In order to reduce the memory consumption of our sparse matrix, we have used a compact representation, depicted in Fig. 2, similar to the DIA (diagonal) format [16] in BLAS [6], but with several additional advantages. The first one is the regularity of the structure which allows us to do coalesced memory accesses most of the time. The second one is that it provides an efficient access to the transpose of the matrix as well as the matrix itself since the transpose is just a re-ordering of the diagonals. That last feature is essential as it is required in the biconjugate gradient method.

In order to be as efficient as possible, the shared memory has been used as a cache memory whenever it was possible in order to avoid the slower accesses to the global memory of the GPU. The different kernels used in the solver are divided to reuse as much data as possible at each call, hence minimizing transfers between the global memory and the registers. To get full details on those kernels, the reader should refer to [18].

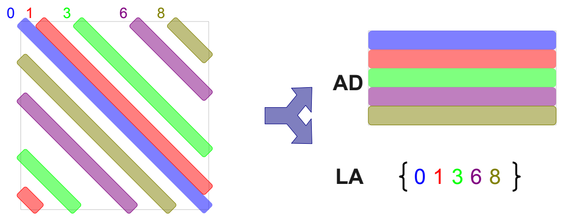


Fig. 2. Compact and regular sparse matrix representation

2.3 Asynchronous Aspects

In the asynchronous version, computing the state of the system (*i.e.* the concentration of the two chemical species across the space) at a given time of evolution (the EDP is time-dependent) is performed asynchronously. This typically involves solving several linear systems on each node, with some communications between each of these inner iterations. However, once this has been done, one synchronization is still required before beginning the next simulation time step, as illustrated in Fig. 3.

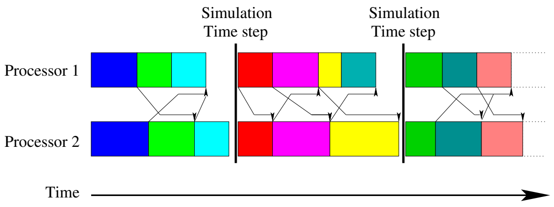


Fig. 3. Asynchronous iterations inside each time step of the computation

In practice, the main differences with the synchronous version lie in the suppression of some barriers and in the way the communications between the units are managed. Concerning the first aspect, all the barriers between the inner iterations inside each time step of the simulation are suppressed. The only remaining synchronization is the one between each time step as pointed out above.

The communications management is a bit more complex than in the synchronous version as it must enable sending and receiving operations at any time during the algorithm. Although the use of non-blocking communications seems appropriate, it is not sufficient, especially concerning receives. This is why a multi-threaded programming is required. The principle is to use separated threads to perform the communications, while the computations are continuously done in the main thread without any interruption, until convergence detection. In our version, we used non-blocking sends in the main thread and an additional thread to manage the receives. It must be noted that in order to be as reactive as possible, some communications related to the control of the algorithm (the global convergence detection) may be initiated directly by the receiving thread (for example to send back the local state of the unit) without requiring any process or response from the main thread.

Subsequently to the multi-threading, the use of mutex is necessary to protect the accesses to data and some variables in order to avoid concurrency and potentially incoherent modifications.

Another difficulty brought by the asynchronism comes from the convergence detection. To ensure the validity of the convergence detection, the simple global reduction of local states of the units must be replaced by some specific mechanisms. We have proposed a decentralized version of such a detection in [3]. The most general scheme may be too expensive in some simple contexts such as local clusters. So, when some information about the system are available (for example bounded communication delay), it is often more pertinent to use a simplified mechanism whose efficiency is better and whose validity is still ensured in that context. Although both general and simplified schemes of convergence detection have been developed for this study, the performances presented in the following section are related to the simplified scheme which gave the best performances.

3 Experimental Results

The platform used to conduct our experiments is a set of two clusters hosted by SUPELEC in Metz. The first one is composed of 15 machines with Intel Core2 Duo CPUs running at 2.66GHz, 4GB of RAM and one Nvidia GeForce 8800GT GPU with 512MB per machine. The operating system is Linux Fedora with CUDA 2.3. The second cluster is composed of 17 machines with Intel Nehalem CPUs (4 cores + hyperthreading) running at 2.67GHz, 6GB RAM and one Nvidia GeForce GTX 285 with 1GB per machine. The OS is the same as the previous cluster. In all the experiments, our program has been compiled with the `sm_11` flag to be compatible with both kinds of GPUs, and using OpenMPI 1.4.2 for message passing. Concerning the interconnection network, both clusters use a Gigabit Ethernet network. Moreover, they are connected to each other and can be used as a single heterogeneous cluster via the OAR management system.

In that hardware context, two initial series of experiments seemed particularly interesting to us. The first one consists in running our application for several problem sizes on one of the homogeneous clusters. We chose the most recent one, with the Nehalem CPUs and GTX 285 GPUs. The second series of experiments is similar to the first one except that instead of using only one cluster, we used the two clusters to obtain a heterogeneous system with 32 nodes.

The results are presented in Table 1 and Table 3. The problem size indicated in the left column corresponds to the number of spatial elements in the 3D domain. As we have two chemical species, for a volume of 50^3 elements, the global linear system is a square matrix with 2×50^3 lines and columns. Fortunately, the local nature of dependencies in the advection-diffusion-reaction problem implies that only 9 diagonals in that matrix are non-zero.

The results obtained in that context are interesting but not as good as could be expected. The decrease of the gain (last column in the tables) when the problem size increases is quite natural as the ratio of communications relatively to the computations decreases and the impact of synchronizations becomes less preponderant over the overall performances. However, the rather limited maximal gain is

Table 1. Execution times (in seconds) with the homogeneous cluster (17 machines)

Pb size	Sync	Async	Speed up Async/Sync	Gain (%)
50×50×50	16.52	14.85	1.11	10.10
100×100×100	144.52	106.09	1.36	26.59
150×150×150	392.79	347.40	1.13	11.55
200×200×200	901.18	866.31	1.04	3.87
250×250×250	1732.60	1674.30	1.03	3.36

a bit deceiving. In fact, it can be explained, at least partially, by the relatively fast network used in the cluster, the rather small amount of data exchanged between the nodes and the homogeneity of the nodes and loads. In such a context, it is clear that the synchronous communications through the Gigabit Ethernet network are not so expensive compared to the extra iterations required by the asynchronous version. Also, it can be deduced that although the GPU $\leftrightarrow$ CPU data transfers play a role in the overall performances, their impact on our PDE solver is less important than one could have thought at first glance.

Two additional experiments have been done with the same cluster but with less processors in order to observe the behavior of our PDE solver when the number of processors varies. The results are provided in Table 2.

Table 2. Execution times (in seconds) with 9 and 14 homogeneous machines

9 Machines of the newer cluster

Pb size	Sync	Async	Speed up Async/Sync	Gain (%)
50×50×50	39.68	25.81	1.54	34.95
100×100×100	249.63	200.25	1.25	19.78
150×150×150	714.85	635.78	1.12	11.06
200×200×200	1599.01	1617.28	0.99	-1.14

14 Machines of the newer cluster

Pb size	Sync	Async	Speed up Async/Sync	Gain (%)
50×50×50	20.95	17.83	1.17	14.89
100×100×100	182.85	132.35	1.38	27.62
150×150×150	486.69	442.16	1.10	9.15
200×200×200	1101.29	1029.61	1.07	6.51

Those results confirm the general trend of gain decrease when the problem size increases. It can also be observed that for smaller clusters, the limit of gain brought by asynchronism is reached sooner, which is not surprising according to the previous considerations.

Concerning the second context of use, the heterogeneous cluster, the results presented in Table 3 are quite unexpected.

Table 3. Execution times (in seconds) with the heterogeneous cluster (15 + 17 machines)

Pb size	Sync	Async	Speed up Async/Sync	Gain (%)
100×100×100	53.21	52.01	1.02	2.25
150×150×150	155.13	164.05	0.94	-5.75
200×200×200	322.11	395.11	0.81	-22.66

In fact, the heterogeneity of the machines should imply different computation speeds and the synchronizations should induce a global slow down imposed by the slowest machine. Nevertheless, the results tend to show that the difference in the powers of the machines is not large enough to induce a sufficiently perceptible unbalance between them. Moreover, it seems that the overhead of the asynchronism, due to the additional iterations, is rapidly more important than the gain in the communications, leading to a loss in performances.

Also, another point that may explain the degraded performances of the asynchronous version in the heterogeneous cluster is that the GPU cards used in the older cluster do not fully support double precision real numbers. Thus, as previously mentioned, the program is compiled to use only single precision numbers, which divides the data size by a factor two and then also the communications volumes, reducing even more the impact of the communications on the overall execution times.

As can be seen in the first two series of experiments, there are some fluctuations in the gains with the homogeneous cluster and rather deceiving results with the heterogeneous cluster, which denote a complex behavior of this kind of algorithm according to the context of use. Those observations imply additional experiments to identify the frontier of gain between synchronism and asynchronism in function of the number of processors and the problem size. Such experiments are presented below.

The first aspect addressed in our additional experiments is the evolution of the execution times according to the number of machines taken from the two available GPU clusters for a fixed problem size. As can be seen in Fig. 4, both surfaces are quite similar at first sight. However, there are some differences which are emphasized by the speedup distribution according to the sequential version, presented in Fig. 5. There clearly appears that the asynchronous version provides a more regular evolution of the speedup than the synchronous one. This comes from the fact that the asynchronous algorithm is more robust to the degradations of the communications performances. Such degradations appear when the number of processors increases, implying a larger number of messages transiting over the interconnection network and then a more important congestion. Thus, the asynchronism puts back the performance decrease due to slower communications in the context of a heterogeneous GPU cluster.

In order to precisely identify the contexts of use in which the asynchronism brings that robustness, we have plotted in Fig. 6 the speedup of the asynchronous GPU algorithm according to its synchronous counterpart.

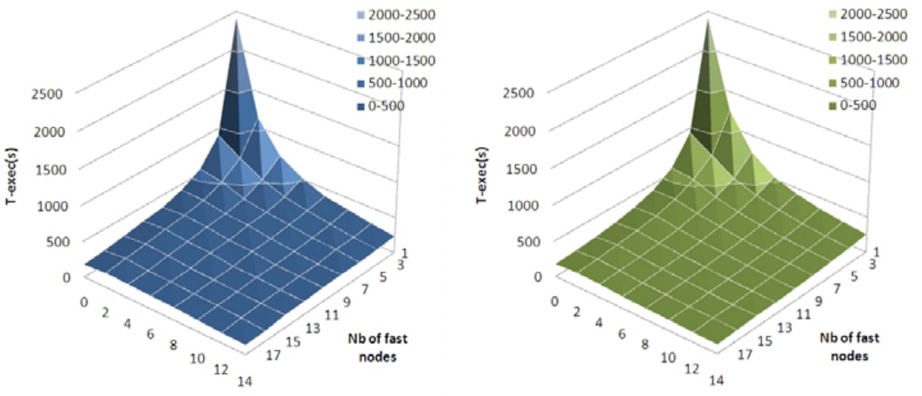


Fig. 4. Execution time of our PDE solver on a 100 × 100 × 100 problem, with the heterogeneous GPU cluster, with synchronous (left) and asynchronous (right) schemes

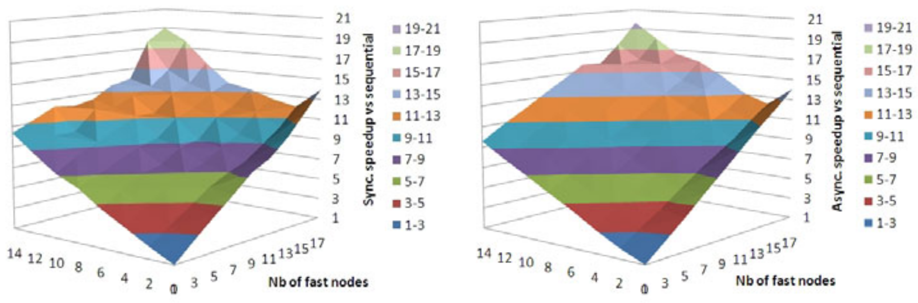


Fig. 5. Speedup of our PDE solver on a 100×100×100 problem, with the heterogeneous GPU cluster, with synchronous (left) and asynchronous (right) schemes, compared to the sequential version

First of all, we have the confirmation that asynchronism does not always bring a gain. As already mentioned, this comes from that fact that when the ratio of communications time over computations time is negligible, the impact of communications over the overall performances is small. So, on one hand the implicit overlapping of communications by computations performed in the asynchronous version provides a very small gain. On the other hand, the asynchronous version generally requires more iterations, and thus more computations, to reach the convergence of the system. Hence, in some contexts the computation time of the extra iterations done in the asynchronous version is larger than the gain obtained on the communications side. Such contexts are clearly visible on the left part of the speedup surface, corresponding to a large pool of slow processors and just a few fast processors.

As soon as the communication-times to computation-times ratio becomes significant, which is the case either when adding processors or taking faster ones, the gain provided by the asynchronism over the communications becomes more important than the iterations overhead, and the asynchronous version becomes faster. In those cases, the gains obtained are quite significant as they can exceed 20% of the total execution time (see Tables 1 and 2). Unfortunately, it can be observed in the example of Fig. 6 that the separation between those two contexts is not strictly regular and studying the relative speedup map will be necessary in order to achieve an automatic selection of the most efficient operating mode of this kind of PDE solver in every context of use.

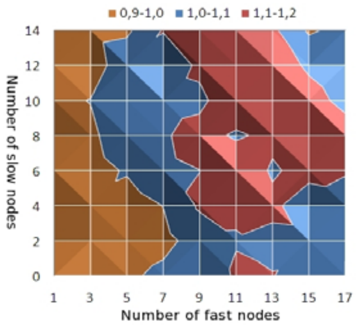


Fig. 6. Speedup of async. *vs* sync. version with the heterogeneous GPU cluster on a 100^3 problem.

4 Conclusion and Perspectives

Two versions of a PDE solver algorithm have been implemented and tested on two clusters of GPUs. The conclusion that can be drawn concerning the interest of asynchronism in such a context of parallel system for that kind of application is that gains are not systematic. Some interesting gains ($\geq$ 20%) can be observed in some contexts and our experiments have pointed out that asynchronism tends to bring a better scalability in such heterogeneous contexts of multi-level parallel systems. However, the frontier between the two algorithmic schemes is not simple, implying that the optimal choice of algorithmic scheme and hardware to use in combination requires a finer model of performance.

As far as we know, that study is among the very firsts of its kind and it shows that this subject requires further works. The obtained results are quite encouraging and motivate us to design a performance model of parallel iterative algorithms on GPU clusters. That model should be based on the different activities (CPU and/or GPU computing, communications,...) during the application execution. An obvious perspective is the auto-tuning by the precise identification of the areas in which one of the operating modes (synchronous or asynchronous) is better suited than the other one to a given context of number of processors and problem size. In addition, using load-balancing in that context should also improve performances of both versions.

Acknowledgments. Authors wish to thank Région Lorraine for its support, and Patrick Mercier for his continuous technical management of the GPU clusters.

References

1. Amitai, D., Averbuch, A., Israeli, M., Itzikowitz, S.: Implicit-explicit parallel asynchronous solver for PDEs. SIAM J. Sci. Comput. 19, 1366–1404 (1998)

2. Bahi, J., Contassot-Vivier, S., Couturier, R.: Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. Parallel Computing 31(5), 439–461 (2005)
3. Bahi, J., Contassot-Vivier, S., Couturier, R.: An Efficient and Robust Decentralized Algorithm for Detecting the Global Convergence in Asynchronous Iterative Algorithms. In: Palma, J.M.L.M., Amestoy, P.R., Daydé, M., Mattoso, M., Lopes, J.C. (eds.) VECPAR 2008. LNCS, vol. 5336, pp. 251–264. Springer, Heidelberg (2008)
4. Baudet, G.M.: Asynchronous iterative methods for multiprocessors. J. ACM 25, 226–244 (1978)
5. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods. Prentice Hall, Englewood Cliffs (1989)
6. Basic linear algebra subprograms, `http://www.netlib.org/blas/`
7. Bojanczyk, A.: Optimal asynchronous newton method for the solution of nonlinear equations. J. ACM 31, 792–803 (1984)
8. Bru, R., Migallon, V., Penadés, J., Szyld, D.B.: Parallel synchronous and asynchronous two-stage multisplitting methods. ETNA 3, 24–38 (1995)
9. Chazan, D., Miranker, W.: Chaotic relaxation. Linear Algebra Appl. 2, 199–222 (1969)
10. Cosnard, M., Fraignaud, P.: Analysis of asynchronous polynomial root finding methods on a distributed memory multicomputer. IEEE Trans. on Parallel and Distributed Systems 5(6) (June 1994)
11. Fletcher, R.: Conjugate gradient methods for indefinite systems. In: Watson, G. (ed.) Numerical Analysis. LNM, vol. 506, pp. 73–89. Springer, Heidelberg (1976), doi:10.1007/BFb0080116
12. Frommer, A., Mayer, G.: On the theory and practice of multisplitting mehods in parallel computation. Computing 49, 63–74 (1992)
13. Frommer, A., Szyld, D.B.: Asynchronous iterations with flexible communication for linear systems. Calculateurs Parallèles, Réseaux et Systèmes Répartis 10, 421–429 (1998)
14. Frommer, A., Szyld, D.B.: On asynchronous iterations. J. Comput. and Appl. Math. 123, 201–216 (2000)
15. Gonzalez, R., Horowitz, M.: Energy dissipation in general pupose microprocessors. IEEE Journal of Solid-State Circuits 31(9) (September 1996)
16. Heroux, M.A.: A proposal for a sparse blas toolkit. SPARKER working note #2. Cray research, Inc. (1992)
17. Hundsdorfer, W., Verwer, J.G.: Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations, 1st edn. Springer Series in Computational Mathematics, vol. 33. Springer, Heidelberg (2003)
18. Jost, T., Contassot-Vivier, S., Vialle, S.: An efficient multi-algorithm sparse linear solver for GPUs. In: Parallel Computing: From Multicores and GPU's to Petascale. Advances in Parallel Computing, vol. 19, pp. 546–553. IOS Press (2010)
19. Jost, T., Contassot-Vivier, S., Vialle, S.: On the interest of clusters of GPUs. In: Grid'5000 Spring School 2010, Lille, France (April 2010)
20. Miellou, J.-C.: Algorithmes de relaxation chaotique Ã retards. R.A.I.R.O. R 1, 55–82 (1975)
21. Szyld, D.B., Xu, J.: Convergence of partially asynchronous block quasi-newton methods for nonlinear systems of equations. J. Comp. and Appl. Math. 103, 307–321 (1999)
22. van der Vorst, H.A.: Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing 13(2), 631–644 (1992)

Simulation of Seismic Waves Propagation in Multiscale Media: Impact of Cavernous/Fractured Reservoirs

Victor Kostin[1], Vadim Lisitsa[2], Galina Reshetova[3], and Vladimir Tcheverda[2]

[1] ZAO Intel A/O
[2] Institute of Petroleum Geology and Geophysics SB RAS
3, Prosp. Koptyug, 630090, Novosibirsk, Russia
[3] Institute of Computational Mathematics and Mathematical Geophysics SB RAS

Abstract. In order to simulate the interaction of seismic waves with cavernous/fractured reservoirs, a finite-difference technique based on locally refined time-and-space grids is used. The need to use these grids is due primarily to the differing scale of heterogeneities in the reference medium and the reservoir. Domain Decomposition methods allow for the separation of the target area into subdomains containing the reference medium (coarse grid) and reservoir (fine grid). Computations for each subdomain can be carried out in parallel. The data exchange between each subdomain within a group is done using MPI through nonblocking iSend/iReceive commands. The data exchange between the two groups is done simultaneously by coupling the coarse and fine grids.

The results of a numerical simulation of a carbonate reservoir are presented and discussed.

Keywords: Finite-difference schemes, local grid refinement, domain decomposition, MPI, group of Processor Units, Master Processor Unit.

1 Introduction and Motivation

One of the key challenges in modern seismic processing is to use the surface and/or borehole data to restore the microstructure of the hydrocarbon reservoir. This microstructure can have a significant impact on oil and gas production. In particular, in many cases the carbonate reservoir's matrix porosity contains the oil but the permeability is mainly through the fracture corridors. In some carbonate reservoirs the in-place oil is contained in karstic caves. Because of this, the ability to locate these microstructures precisely and to characterize their properties is of a great importance. Recently various techniques have been developed to perform this analysis with the help of scattered seismic waves. Among them, the scattering index presented by Willis et al. ([9]) or a variety of the imaging techniques recently developed under the generic name of interferometry (see e.g. book of G.Schuster [7]).

The first step in the development of any inversion/imaging procedure is to simulate accurately the wave field scattered by fractures and caves. The numerical and computer constraints even on very large clusters place limitations on the resolution of the model described. Really, a reservoir beds typically at a depth of 2000 $\div$ 4000 meters, which is about 50$\div$70 dominant wavelength. The current practice for the finite-difference simulation of seismic waves propagation at such distances is to use grid cells of 0.05 - 0.1 of a dominant wavelength, usually between 5 - 10 meters. So, one needs to upscale heterogeneities associated with fracturing on a smaller scale (0.01 - 1 meter) and to transform them to an equivalent/effective medium. This effective medium will help reproduce variations in the travel-times and an average change of reflection coefficients but absolutely cancels the scattered waves that are a subject of the above mentioned methods for characterizing fracture distributions.

Thus, the main challenge with a full scale simulation of cavernous/fractured (carbonate) reservoirs in a realistic environment is that one should take into account both the macro- and microstructures. A straightforward implementation of finite difference techniques provides a highly detailed reference model. From the computational point of view, this means a huge amount of memory required for the simulation and, therefore, extremely high computer cost. In particular, a simulated model of dimension 10km $\times$ 10km $\times$ 10 km, which is common for seismic explorations, with a cell size of 0.5m claims 8×10^{12} cells and needs in $\approx 350Tb$ of RAM.

The popular approach to overcome these troubles is to refine a grid in space only and there are many publications dealing with its implementation (see [6] for a detailed review), but it has at least two drawbacks:

– To ensure stability of the finite-difference scheme the time step must be very small everywhere in the computational domain;
– Unreasonably small Courant ratio in the area with a coarse spatial grid leads to a noticeable increase in numerical dispersion.

Our solution to this issue is to use a mutually agreed local grid refinement in time and space: spatial and time steps are refined by the same factor.

2 Numerical and Parallel Implementation

In our considerations propagation of seismic waves is simulated with help of an explicit finite-difference scheme (FDS) on staggered grids approximating elastic wave equations (velocity-stress formulation):

$$\varrho\frac{\partial \boldsymbol{u}}{\partial t} - A\frac{\partial \boldsymbol{\sigma}}{\partial x} - B\frac{\partial \boldsymbol{\sigma}}{\partial y} - C\frac{\partial \boldsymbol{\sigma}}{\partial z} = 0;$$
$$D\frac{\partial \boldsymbol{\sigma}}{\partial t} - A^T\frac{\partial \boldsymbol{u}}{\partial x} - B^T\frac{\partial \boldsymbol{u}}{\partial y} - C^T\frac{\partial \boldsymbol{u}}{\partial z} = \boldsymbol{f};$$

written for vectors of the velocity $\boldsymbol{u} = (u_x, u_y, u_z)^T$ and the stress $\boldsymbol{\sigma} = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xz}, \sigma_{yz}, \sigma_{xy})$.

Staggered grid finite difference scheme updates values of unknown vectors in two steps:

1. from velocities at t to stresses at $t + \Delta t/2$;
2. from stresses at $t + \Delta t/2$ to velocities at $t + \Delta t$.

In view of the local spatial distribution of the stencil used in this finite difference scheme to update the vector at some point M and time $(t + \Delta t/2)$, the previous time level (t) corresponding values should be known in a neighborhood of this point.

Parallel implementation of this FDS is based on the decomposition of the computational domain to elementary subdomains, being assigned to its individual Processor Unit (PU) (Fig.1). Update unknown vectors while moving from a time layer to the next one requires two adjacent PU to exchange unknown vectors values in the grid nodes along the interface. Necessity of this exchange negatively impacts scalability of the method. However, the impact is less vivsble on 3D Domain Decomposition (DD) than in one- and two-dimensional ones (see theoretical estimates of acceleration for different versions of DD in Fig.2)). In our implementation we choose 3D Domain Decomposition, moreover, in order to reduce the idle time, the asynchronous computations based on nonblocking MPI procedures iSend/iReceive are used.

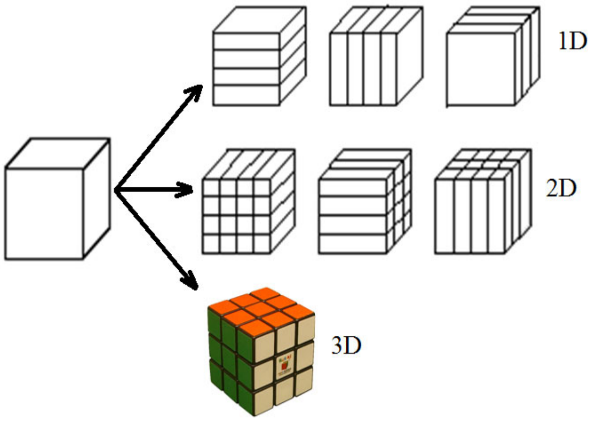


Fig. 1. Domain decomposition. From top to bottom: 1D, 2D, 3D.

In order to carry out the numerical simulation of seismic waves propagation through a multiscale medium we represent it as a superposition of the reference medium given on a coarse grid and the reservoir on a fine grid (see Fig.3). Each of these grids is again decomposed to elementary subdomains being assigned to individual PU. Now these PU are combined into two groups for coarse and fine grids, and special efforts should be applied in order to couple these groups.

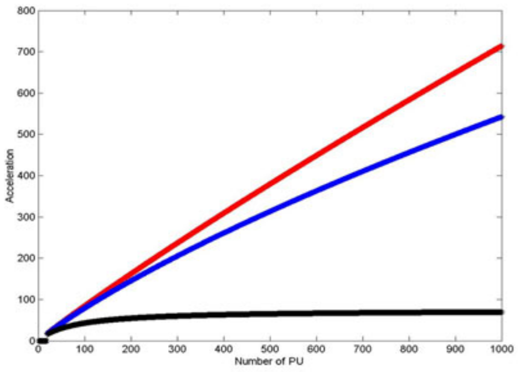


Fig. 2. Theoretical estimation of acceleration for different implementations of Domain Decomposition (top down): 3D, 2D and 1D (see Fig. 1)

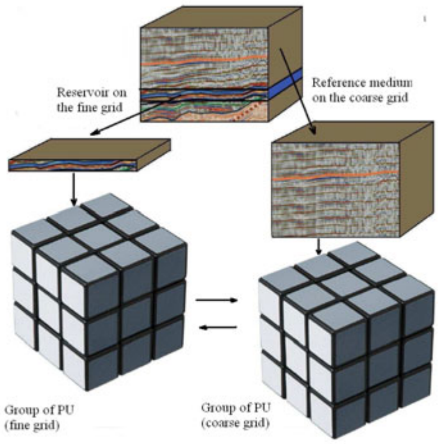

Reservoir on the fine grid
Reference medium on the coarse grid
Group of PU (fine grid)
Group of PU (coarse grid)

Fig. 3. Two groups of Processor Units

2.1 Coupling of Coarse and Fine Grids

First of all, let us explain how a coarse and a fine grids are coupled to each other. The necessary properties of the finite difference method based on a local grid refinement should be its stability and an acceptable level of artificial reflections. Scattered waves we are interested in have an amplitude of about 1% of the incident wave. Artifacts should be at least 10 times less, that is about 0.1% of the incident wave. If we refine the grid at once in time and space stability of the FDS on this way (see [1], [2] and [3]) can be provided via coupling coarse and fine grids on the base of energy conservation, which leads to an unacceptable level (more than 1%) of artificial reflections (see [4]). We modify the approach so that the grid is refined by turn in time and space on two different surfaces surrounding the target area with microstructure. This allows decoupling temporal and spatial grid refinement and to implement them independently and to provide the desired level of artifacts.

Refinement in Time. Refinement in time with a fixed 1D spatial discretization is clearly seen in Fig.4 and does not need any explanations. Its modification for 2D and 3D media is straightforward (see [4] for more detail).

Refinement in Space. In order to change spatial grids, the Fast Fourier Transform (FFT) based interpolation is used. Let us explain this procedure for a 2D problem. The mutual disposition of a coarse and a fine spatial grids is presented in Fig.4b, which corresponds to updating the stresses by velocities (updating stresses by velocities is implemented in the same manner). As can be seen, to update the stresses on a fine grid it is necessary to know the displacement at the points marked with small (red) triangles, which do not exist on the given coarse grid. Using the fact that all of them are on the same line (on the same plane for 3D statement), we seek the values of missing nodes by FFT based interpolation. Its main advantages are an extremely high performance and exponential accuracy. It is this accuracy allows us to provide the required low level of artifacts (about 0.001 with respect to the incident wave) generated on the interface of these two grids. For 3D statement we again perform the FFT based interpolation but this time 2D.

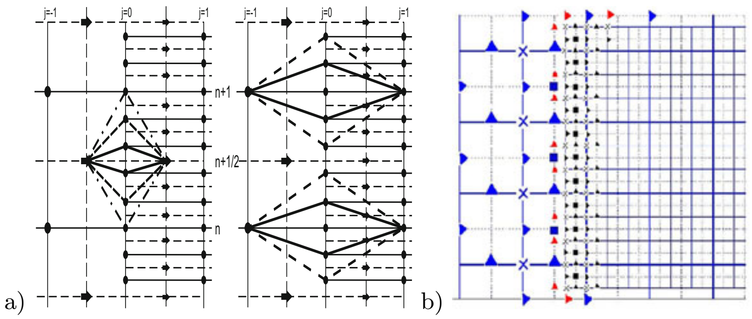


Fig. 4. From a coarse to a fine grid: a) refinement in time (left - displacement, right - stresses) b) refinement in space

2.2 Implementation of Parallel Computations

Our objective is to analyze the impact of cavernous-fractured reservoirs in the seismic waves for realistic 3D heterogeneous media. Therefore, parallel computations are necessary both in the reference medium, described by a coarse mesh, and in the reservoir itself, determined on a fine grid. The simultaneous use of a coarse and a fine grids and the need for interaction between them makes it difficult to ensure a uniform load of Processor Units under parallelization of computations based on Domain Decomposition. Besides, the user should be allowed to locate the reservoir anywhere in the background.

This problem is resolved through the implementation of parallel computations on two groups of Processor Units. One of them is fully placed 3D heterogeneous referent environment on a coarse grid, while the fine mesh describing the reservoir is distributed among the PU in the second group (Fig.3). Thus, there is a need for both exchanges between processors within each group and between the groups as well. The data exchange within a group is done via faces of the adjacent Processor Units by non-blocking iSend/iReceive MPI procedures. Interaction between the groups is much more complicated. It is carried out not so much for data sending/receiving only, but for coupling a coarse and a fine grids as well. Let us consider the data exchange from the first group (a coarse grid) of PU to the second (a fine grid) and backwards.

From Coarse to Fine. First are found Processor Units in the first group which cover the target area, and are grouped along each of the faces being in contact with the fine grid. At each of the faces there is allocated the Master Processor (MP), which gathers the computed current values of stresses/displacements and sends them to the relevant MP on a fine grid (see Fig.5). All the subsequent data processing providing the coupling of a coarse and a fine grids by the FFT based interpolation is performed by the relevant Master Processor in the second group (a fine grid). Later this MP sends interpolated data to each processor in its subgroup.

Interpolation performed by the MP of the second group essentially decreases the amount of sent/received data and, hence, the idle time of PU.

From Fine to Coarse. As in the previous case, primarily there are identified PU from the second group which perform computations on the faces covering the target area. Next, again for each face Master Processor is identified. This MP as its partner from the coarse grid collects data from the relevant face and performs their preprocessing before sending to the first group of PU (a coarse grid). Now

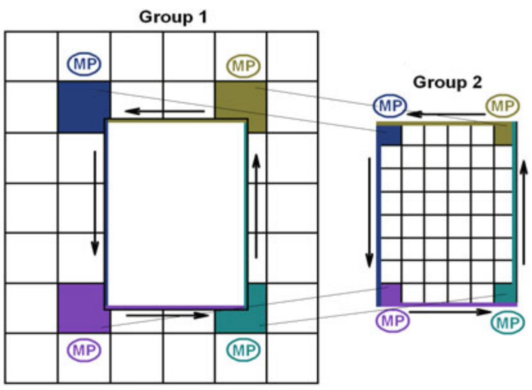


Fig. 5. Processor Units for a coarse (left) and a fine (right) grids. Relevant MP from different groups have the same color.

we do not need all data in order to move to the next time, but only those of them which fit the coarse grid. Formally, these data could be thinned out, but our experiments have proved that this way generates significant artifacts due to the loss of smoothness. Therefore for this direction (from fine to coarse) we also use the FFT based interpolation implemented by the relevant MP of the second group (a fine grid). The data obtained are sent to the first group.

3 Reservoir Simulation

3.1 2D Statement: Karstic Layer

In order to estimate the accuracy of the method, we first consider a 2D statement for a thin layered reservoir with karst intrusions presented in Fig. 6a). In order to describe the microstructure of karstic intrusions we should use a grid with $h_x = h_z = 0.5\,m$, while for the reference medium the dispersion analysis gives $h_x = h_z = 2.5\,m$. In Fig. 6a one can see an area with the fivefold grid refinement in time and space.

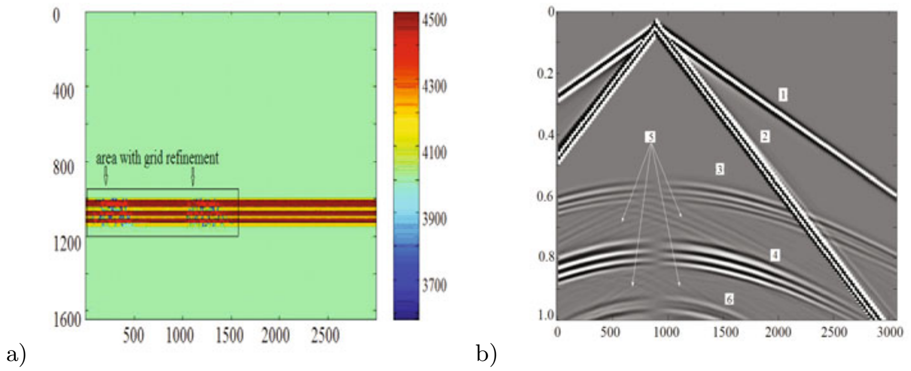
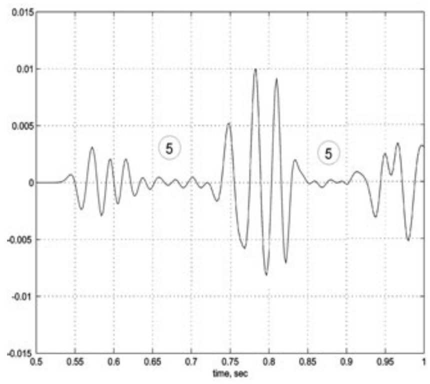


Fig. 6. a) Karstic layer b) Surface seismogram (horizontal component). 1 - direct P-wave, 2 - direct S-wave coupled with surface Rayleigh wave, 3 and 4 - reflected PP- and PS-waves, 5 - scattered PP- and PS-waves, 6 - reflected SP-wave.

Let us compare now the results of simulation for a uniform fine grid and a grid with the local refinement in time and space. In Fig. 6b, one can see a free surface seismogram (horizontal displacement) generated by the vertical point force with a Ricker pulse of a dominant frequency 25 Hz and simulated on the uniform fine grid. Fig. 7 represents a comparison of synthetic traces computed on a uniform fine grid and a grid with local refinement in time and space. As can be seen, there is an excellent coincidence of scattered PP-waves and rather good agreement of PS ones.

a) 

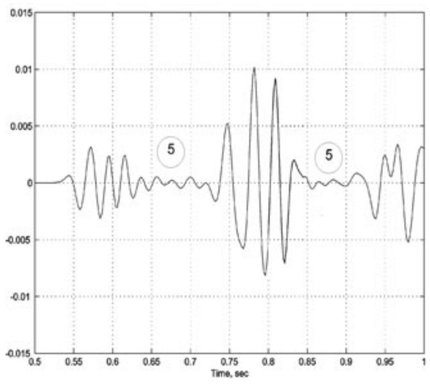
b) 

Fig. 7. Traces computed on a uniform fine grid (a) and on a grid with local refinement in time and space (b)

3.2 3D Statement: Fracture Corridors

Now we present the results of numerical simulation for some realistic model of a carbonate reservoir with fracture corridors. The reservoir is embedded into a homogeneous background with elastic properties equivalent to an average carbonate rock:

$$V_p = 4500m/s,\ V_s = 2500m/s,\ \text{density}\ \varrho = 2500kg/m^3$$

The reservoir is treated as a horizontal layer 200m thick and corresponds to a slightly softer rock with the elastic waves propagation velocities $V_p = 4400m/s$, $V_s = 2400m/s$ and the density $\varrho = 2200kg/m^3$ and contains two fractured layers 30m thick each. The fracturation is of a corridor type, that is, we have included into each layer a set of randomly distributed parallel fracture corridors. The fracture density varies from 0 in the non-fractured facies to 0.3 as a maximum. Finally, the fracture density was transformed to elastic parameters using the second order Hudson theory following [5]. Since fractures were filled with gas, the velocity diminishes down to 3600m/s the lowermost as compared to 4400m/s in the matrix. The fracture corridors were then randomly distributed into fractured layers until the desired fracture density was obtained. The final distribution of fracture corridors can be seen in Fig.8 (two side views).

3.3 Synthetic Seismograms

The developed parallel software was used for simulation of scattered waves for the reservoir model introduced in the previous section. The acquisition system can be seen in Fig.9. Three-component seismograms are presented in Fig.10. There is a visible difference between the seismograms along the parallel and the perpendicular lines with respect to fracture corridors.

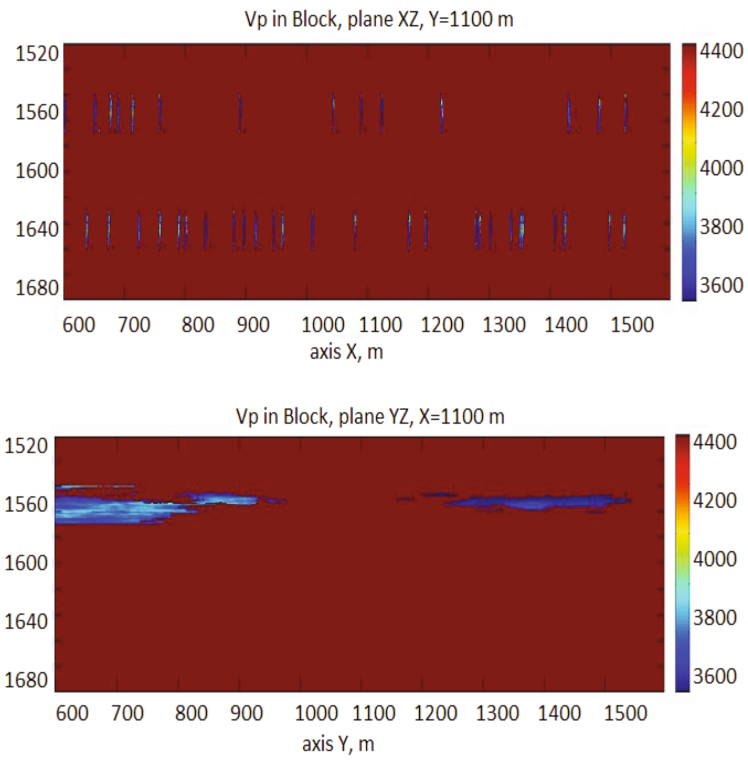


Fig. 8. Side view of fracture corridors within reservoir: orthogonal (top) and parallel (bottom) to the corridor direction

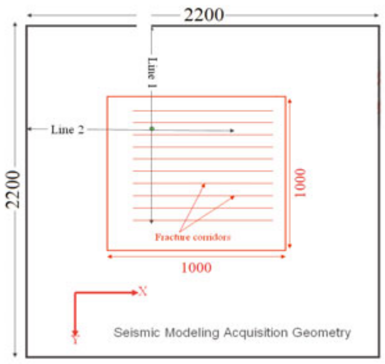


Fig. 9. Acquisition system. The source is at the intersection of Line 1 and Line 2.

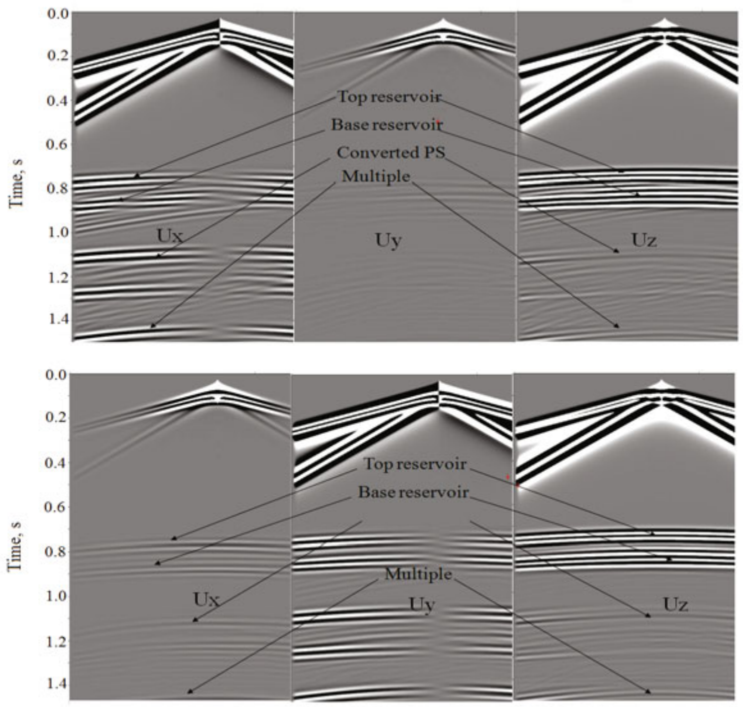


Fig. 10. 3C seismograms along Line 2 (top) and Line 1 (bottom). From left to right: X, Y and Z-displacements.

4 Conclusion

A finite difference method based on the use of grids with local space-time refinement is proposed, developed and verified. Implementing its parallel software opens up a fundamentally new opportunity to study the processes of formation and propagation of waves scattered by a microstructure of the cavernous/fractured reservoir for a realistic geological environment. The very first simulations carried out using this software, allow the following conclusions:

- Modeling techniques make possible to simulate the impact of fine-scale heterogeneities within a realistic 3D environment in an accurate manner;
- Scattered waves have a significant energy and can be acquired by the field observations, hence there should be a possibility not only to reveal cavities and fractures in the reservoir but to predict their orientation as well.

Acknowledgements. The Research is partially supported by Russian Foundation of Basic Research (grants 10-05-00233, 11-05-00947, 11-05-12022, 12-05-00943), grant no. MK-47.2011.5 of Russian Government and by Total SA under contract FR00001925.

Simulations were carried out on clusters of the Siberian Supercomputer Center of the Siberian Branch of RAS (Novosibirsk) and the Joint Supercomputer Center of the Russian Academy of Sciences (Moscow).

References

1. Collino, F., Fouquet, T., Joly, P.: A conservative space-time mesh refinement method for 1-D wave equation. Part I: Construction. Numerische Mathematik 95, 197–221 (2003)
2. Collino, F., Fouquet, T., Joly, P.: A conservative space-time mesh refinement method for 1-D wave equation. Part II: Analysis. Numerische Mathematik 95, 223–251 (2003)
3. Diaz, J., Grote, M.J.: Energy conserving explicit local time stepping for second-order wave equations. SIAM J. Sci. Comput. 31(3), 1985–2014 (2009)
4. Lisitsa, V., Reshetova, G., Tcheverda, V.: Local time-space mesh refinement for finite-difference simulation of waves. In: Kreiss, G., Lotstedt, P., Malqvist, A., Neytcheva, M. (eds.) Numerical Mathematics and Advanced Applications 2009, Proceedings of ENUMATH 2009, pp. 609–616. Springer, Heidelberg (2010)
5. Mavko, G., Mukerji, T., Dvorkin, J.: Rock Physics Handbook. Cambridge University Press, Cambridge (2009)
6. Kristek, J., Moczo, P., Galis, M.: Stable discontinuous staggered grid in the finite-difference modelling of seismic motion. Geophysical Journal International 183(3), 1401–1407 (2010)
7. Schuster, G.: Seismic interferometry. Cambridge University Press, Cambridge (2009)
8. Virieux, J.: P-SV wave propagation in heterogeneous media: Velocity - stress finite difference method. Geophysics 51(4), 889–901 (1986)
9. Willis, M., Burns, D., Rao, R., Minsley, B., Toksoz, N., Vetri, L.: Spatial orientation and distribution of reservoir fractures from scattered seismic energy. Geophysics 71, O43–O51

Improvements of a Fast Parallel Poisson Solver on Irregular Domains

Andreas Adelmann[1], Peter Arbenz[2,⋆], and Yves Ineichen[1,2]

[1] Paul Scherrer Institut, Villigen, Switzerland
[2] ETH Zürch, Chair of Computational Science, Zürich, Switzerland
arbenz@inf.ethz.ch

Abstract. We discuss the scalable parallel solution of the Poisson equation on irregularly shaped domains discretized by finite differences. The symmetric positive definite system is solved by the preconditioned conjugate gradient algorithm with smoothed aggregation (SA) based algebraic multigrid (AMG) preconditioning. We investigate variants of the implementation of SA-AMG that lead to considerable improvements in the execution times. The improvements are due to a better data partitioning and the iterative solution of the coarsest level system in AMG. We demonstrate good scalability of the solver on a distributed memory parallel computer with up to 2048 processors.

Keywords: Poisson equation, finite differences, preconditioned conjugate gradient algorithm, algebraic multigrid, data partitioning.

1 Introduction

The solver described in this paper is part of the general accelerator modeling tool Object Oriented Parallel Accelerator Library (OPAL) [3]. OPAL enables the solution of the most challenging problems in the field of high precision particle accelerator modeling. These include the simulation of high power hadron accelerators and of next generation light sources.

In these simulations the evolution of the charged particles is determined by the collisionless *Vlasov equation*. The most compute intense portion of the simulation is the determination of the electrostatic potential ϕ from the *Poisson equation*

$$-\varepsilon_0 \,\Delta\phi(\mathbf{x}) = \rho(\mathbf{x}), \tag{1}$$

in a coordinate system moving with the particles. Here, ρ denotes the spatial charge density and ε_0 is the dielectric constant. The electric field is obtained from

$$\mathbf{E} = -\nabla\phi. \tag{2}$$

An appropriate *Lorentz transformation* yields the electromagnetic fields in the static reference frame that are needed to move the particles. For details see [2].

⋆ Corresponding author.

The Poisson problem (1) discretized by finite differences can efficiently be solved on a rectangular grid by a Particle-In-Cell (PIC) approach [17]. The right hand side in (1) is discretized by sampling the particles at the grid points. In (2), ϕ is interpolated at the particle positions from its values at the grid points. We also note that the common FFT-based Poisson solvers and similar approaches [16,17] are restricted to box-shaped or open domains.

In section 2 we present our finite difference approach and how we treat the boundary at curved surfaces. In section 3 we review the iterative system solver. In section 4 we discuss numerical experiments on 512–2048 processors of a Cray XT-5. In particular, we are interested in the partitioning of the computational domain and in the coarse level solver of the multilevel preconditioner. Section 5 concludes the paper.

2 The Discretization

In this section we discuss the solution of the Poisson equation in a domain $\Omega \subset \mathbb{R}^3$ as indicated in Fig. 1. The boundary of the domain is composed of two parts, a curved, smooth surface Γ_1 and two planar portions at $z = -d$ and $z = +d$ that form together Γ_2. In physical terms Γ_1 forms the casing of the pipe, while Γ_2 is the open boundary at the inlet and outlet of the beam pipe, respectively. The centroid of the particle bunch is at the origin of the coordinate system. In practice the shape of Γ_1 can be quite complicated. Our code assumes that a ray that emanates perpendicularly from the z-axis crosses Γ_1 at most once.

The Poisson problem that we are going to solve is given by

$$\begin{gathered} -\epsilon_0\, \Delta\phi = \rho \text{ in } \Omega, \\ \phi = g \equiv 0 \text{ on } \Gamma_1, \quad \partial_n\phi + (1/d)\phi = 0 \text{ on } \Gamma_2. \end{gathered} \tag{3}$$

The parameter d in the Robin boundary condition is half the extent of Ω in z-direction [15].

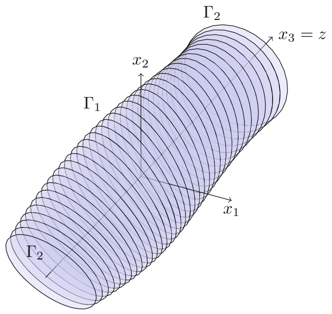


Fig. 1. Sketch of a typical domain

We discretize (3) by a standard second order finite difference scheme defined on a rectangular grid with grid spacing h_i in the i-th coordinate direction. It is natural to arrange the grid such that the two portions of Γ_2 lie in grid planes.

A lattice point is called an *interior* point if all its direct neighbors are in Ω. All other grid points are called *near-boundary* points. At interior points $\mathbf{x}$ we approximate $\Delta u(\mathbf{x})$ by the well-known 7-point difference star

$$-\Delta_h u(\mathbf{x}) = \sum_{i=1}^{3} \frac{-u(\mathbf{x}-h_i\mathbf{e}_i) + 2u(\mathbf{x}) - u(\mathbf{x}+h_i\mathbf{e}_i)}{h_i^2}. \tag{4}$$

At grid points near the boundary we have to take the boundary conditions in (3) into account by constant, linear, or quadratic extrapolation [2,7,14].

The finite difference discretization just described leads to a system of equations

$$A\mathbf{x} = \mathbf{b}, \tag{5}$$

where $\mathbf{x}$ is the vector of unknown values of the potential and $\mathbf{b}$ is the vector of the charge density interpolated at the grid points. The Poisson matrix A is an M-matrix irrespective of the boundary treatment [11]. Constant and linear extrapolation lead to a *symmetric* positive definite (spd) A while quadratic extrapolation yields a *nonsymmetric* but still positive definite Poisson matrix.

The boundary extrapolation can introduce large diagonal elements in A. In order to avoid numerical difficulties it is advisable to apply a symmetric scaling to the system (5).

3 The Solution Method

If A is spd the conjugate gradient (CG) algorithm [11,13] solves (5) in a fast and memory efficient way. In the case of quadratic boundary extrapolation A is nonsymmetric, however only ‘mildly’. There are some deviations from symmetry only at some of the boundary points. In this situation the CG algorithm is still applicable, i.e., it does not break down. However, it loses the finite termination property and may behave more like steepest descent [10]. In our experiments we observed a convergence behavior that did not deviate from the one for spd matrices.

To improve the convergence behavior of the CG methods we precondition (5) by smoothed aggregation-based algebraic multigrid (SA-AMG) preconditioners [8,19]. Aggregation-based AMG methods cluster the fine grid unknowns to aggregates as representation for the unknowns on the next coarser grid.

The multigrid preconditioner and iterative solver are implemented with the help of the Trilinos framework [12,18]. Trilinos provides state-of-the-art tools for numerical computation in various packages. The AztecOO package, e.g., provides iterative solvers and ML [8] provides multilevel preconditioners. By means of ML, we created our smoothed aggregation-based AMG preconditioner. We use ML’s “decoupled” aggregation strategy [19] which constructs aggregates consisting of cubes of 3×3×3 vertices. This strategy may entail non-optimal aggregates at the subdomain interfaces. The subdomains represent the portion of the problem dealt with by a processor or core. The partitioning in subdomains is done manually based on the encasing cubic grid.

As suggested in [1] we choose a Chebyshev polynomial smoother. The employed coarse level solver (Amesos-KLU) ships the coarse level problem to node 0 and solves it there by means of an LU factorization. An alternative is to apply a few steps of an iterative solver (e.g. Gauss–Seidel) at the coarsest level. A small number of iteration steps decreases the quality of the preconditioner and thus increases the PCG iteration count. A large number of iteration steps increases the time for applying the AMG preconditioner. In [2] we found three Gauss–Seidel iteration steps to be a good choice for our application. In this paper, we use Chebyshev iteration.

4 Numerical Experiments

In the recent paper [2] we conducted numerical experiments on the Cray XT-4 at the Swiss National Supercomputing Centre (CSCS) in order to assess the performance and scalability of our solver. In particular, we compared our solver with an FFT-based one. This is in fact not completely trivial, since the computational domains of the two solvers differ. The FFT-based solver requires a rectangular computational domain. Usually Neumann (free) boundary conditions are applied. Our finite difference solver approximates well the geometry of the device and uses homogeneous Dirichlet boundary conditions. In situations where the boundary is far away from the particle bunch the solutions of the two problems match quite well. However, in problems where the spatial extent of the beam is comparable with that of the beam pipe it is important to have an accurate representation of the field near the boundary. Then, the results of the computations can differ significantly, and the results of the FFT-based solver are questionable. Nevertheless, the FFT-based solver can be faster than our iterative solver by up to about a factor 5.

In this paper we discuss the issue of load balancing and communication overhead. We conduct the numerical experiments on the newest Cray XT-5 at CSCS [5]. This machine consists of 3688 AMD hexa-core Opteron processors clocked at 2.4 GHz. The system has 28.8 TB of DDR2 RAM, 290 TB disk space, and a high-speed interconnect with a bandwidth of 9.6 GB/s and a latency of 5 μs[1].

Our computational domains are embedded in a 3D rectangular domain, as illustrated in Fig. 1. The rectangular computational grid is generated inside the rectangular domain. Only grid points of the rectangular grid that are included in the computational domain Ω are used in the computation. In the computations in [2] the partitioning of the computational domain was based on the partitioning of the *whole* rectangular domain. (This underlying rectangular grid is induced by the particle code OPAL [3] that participates at the overall computation.)

[1] http://www.cscs.ch/455.0.html retrieved on July 13, 2010.

Therefore, some subdomains contained far fewer grid points than others, causing severe load imbalance. In fact, in many cases there were subdomains that contained no grid points at all. In our new approach we partition the computational domain to enhance load balancing. In this paper we compare old and new approach.

Table 1. Times in seconds and relative parallel efficiencies. The original data distribution is used, and the coarsest AMG level is solved with KLU.

cores	solution	construction	application	total ML	iterations
512	62.22 [1.00]	35.12 [1.00]	51.68 [1.00]	86.79 [1.00]	20
1024	32.95 [0.94]	19.95 [0.88]	27.47 [0.94]	47.41 [0.92]	20
2048	17.68 [0.87]	12.37 [0.71]	14.85 [0.87]	27.22 [0.80]	20

The computational domain we are dealing with in this paper is a circular cylinder embedded in a $1024 \times 1024 \times 1024$ grid. In this setup the problem size is still reasonably large when employing 2048 cores, i.e., a subcube contains (up to) $524'288$ grid points. We use linear extrapolation at the Dirichlet boundary Γ_1. The solver is the AMG-preconditioned conjugate gradient algorithm as implemented in Trilinos and discussed in section 3. Timings for three phases of the computation are given in Table 1. For this large problem with 840 million degrees of freedom we observe quite good efficiencies. The solver runs at 87% efficiency with 2048 cores relative to the 512-cores performance. Note that the solution phase contains the application of the preconditioner. Therefore, the difference of the two columns indicated by 'solution' and 'application' essentially gives the time for matrix-vector and inner products in the conjugate gradient algorithm. The column 'total ML' comprises the sum of columns 'construction' and 'application'. The construction phase is performing the worst with an efficiency of 71%. We found that much of the time in the construction of the preconditioner goes into the factorization of the coarsest level matrix. We therefore decided to replace the direct solver KLU by an iterative procedure. We apply one step of the Chebyshev semi-iterative method [9] with polynomial degree 10. The timings for this approach are given in Table 2. The times for the construction of the preconditioner have been reduced considerably, at the expense of a slightly more expensive solution phase. Now the construction phase scales perfectly. Notice that the iteration counts change only marginally.

Table 2. Times in seconds and relative parallel efficiencies. The original data distribution is used, and the coarsest AMG level is solved iteratively.

cores	solution	construction	application	total ML	iterations
512	63.12 [1.00]	32.09 [1.00]	52.73 [1.00]	84.80 [1.00]	20
1024	33.54 [0.94]	16.31 [0.98]	28.04 [0.94]	44.35 [0.96]	20
2048	18.56 [0.85]	8.10 [0.99]	15.66 [0.84]	23.76 [0.89]	21

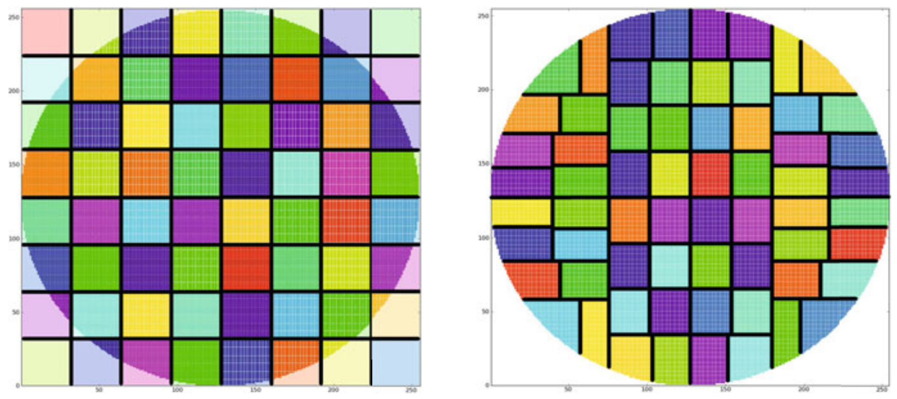


Fig. 2. (Left) Cross section of data distribution on 512 cores on a $8 \times 8 \times 8$ processor grid (colors indicate data owned by a processor) and (Right) the same data redistributed with recursive coordinate bisection (RCB)

In Fig. 2 on the left a cross section is shown of the case where the computational domain is embedded in a cube that has been subdivided in $8 \times 8 \times 8$ subcubes. It is easily seen that the four subcubes in the corners do not contain any points of the computational grid, while other subcubes contain at least a few grid points. In general, a fraction of only about $\pi/4 \approx 0.8$ of the grid points of the cube will be included in the computational domain. This number corresponds to the ratio of the volumes of the circular cylinder and its encasing cube. That is, about 20% of the nodes in the cube are not involved in the computation and, hence about 20% of the subcubes contain a reduced number of nodes. Since subcubes correspond to cores, empty or almost empty subcubes correspond to idle or underloaded cores.

Evidently, this partitioning entails a severe load imbalance. Nevertheless, the speedups shown in Table 1 look good! These good speedups are comprehensible if one considers the most loaded processes that correspond to the innermost cubes, i.e., those close to the z-axis. These subcubes contain $1024^3/p$ nodes. An increase of the processor number p leads to an optimal speedup, at least as long as the floating point operations dominate the work load. This is the case here, as even in the 2048 processor computation a core handles more than half a million nodes.

Although speedups are good with this crude distribution of work, a better balanced work will lead to improved execution times. After all, there are only about 80% of the 1.1 billion nodes inside the computational domain.

To better balance the load we partition data using the recursive coordinate bisection (RCB) algorithm as it is implemented in the Zoltan toolkit [6,4]. The Trilinos package Isorropia[2] provides a matrix-based interface to Zoltan. The recursive coordinate bisection (RCB) algorithm partitions the graph according

[2] See `http://trilinos.sandia.gov/packages/isorropia/`

to the coordinates of its vertices. The computational domain is first divided into two subdomains by a cutting plane orthogonal to one of the coordinate axes so that half the work load is in each of the subdomains. (Notice that other fractions than 50-50 are possible and in fact needed if the number of processors is not a power of 2.) That coordinate axis is chosen which is associated with the longest elongation of the domain. Clearly, this procedure can be applied recursively. RCB leads to nicely balanced data distributions. In rectangular grids, RCB is a particularly fast partitioner since the coordinates of the grid vertices are easily determined from their indices. In Fig. 2 on the right the cross section of the circular cylinder is shown with the partitioning into 64 subdomains. Now, all subdomains contain an almost equal number of nodes which leads to almost equal loads per processor. This subdivision evidently is more complicated than the one on the left of this figure. Except for subdomains in the center, i.e. close to the z axis, there are more than just six neighbors (in 3D). With the increased number of neighbors the number of messages that are to be sent in the communication steps increases. The communication volume does not change much. (Notice that Trilinos constructs the communication pattern transparent to the user.)

Table 3. Times in seconds and relative parallel efficiencies. Data is distributed by RCB. The coarsest AMG level is solved with KLU.

cores	solution	construction	application	total ML	iterations
512	50.32 [1.00]	27.37 [1.00]	44.00 [1.00]	71.37 [1.00]	20
1024	28.14 [0.89]	16.82 [0.81]	24.82 [0.89]	41.58 [0.86]	20
2048	15.26 [0.82]	15.81 [0.43]	13.47 [0.82]	29.28 [0.61]	19

Table 4. Times in seconds and relative parallel efficiencies. Data is distributed by RCB. The coarsest AMG level is solved iteratively.

cores	solution	construction	application	total ML	iterations
512	51.08 [1.00]	25.65 [1.00]	44.89 [1.00]	70.55 [1.00]	20
1024	27.38 [0.93]	12.96 [0.99]	24.51 [0.92]	37.07 [0.95]	20
2048	14.76 [0.87]	6.69 [0.96]	13.10 [0.86]	19.79 [0.89]	19

In Tables 3 and 4 the execution times of our code is given with the data redistributed by RCB. These numbers are to be compared with those in Tables 1 and 2, respectively, where the original data distribution was used. For 512 cores the execution times are significantly smaller with the RCB distribution, about 20%, as the previous discussion suggests. Tables 5 and 6 give more details. There, in brackets, the efficiencies of the runs with the original rectangular distribution are listed relative to the 512 processor run with the RCB distribution.

When using the iterative solver on the coarsest level, speedups and thus efficiencies are quite close, cf. Tables 2 and 4. However, there are significant differences when the coarsest level system is solved directly. In this case the efficiencies deteriorate more quickly with the RCB distribution than with the original distribution.

Table 5. Parallel efficiencies of the RCB partitioned runs and relative parallel efficiencies of the runs with original data distribution. The coarsest AMG level is solved with KLU.

cores	solution	construction	application	total ML
512	1.00 [0.81]	1.00 [0.78]	1.00 [0.85]	1.00 [0.82]
1024	0.89 [0.76]	0.81 [0.69]	0.89 [0.80]	0.86 [0.75]
2048	0.82 [0.71]	0.43 [0.55]	0.82 [0.74]	0.61 [0.66]

Table 6. Parallel efficiencies of the RCB partitioned runs and relative parallel efficiencies of the runs with original data distribution. The coarsest AMG level is solved iteratively.

cores	solution	construction	application	total ML
512	1.00 [0.81]	1.00 [0.80]	1.00 [0.85]	1.00 [0.83]
1024	0.93 [0.76]	0.99 [0.79]	0.92 [0.80]	0.95 [0.80]
2048	0.87 [0.69]	0.96 [0.79]	0.86 [0.72]	0.89 [0.74]

A more detailed analysis (not reproduced here) reveals that the significant difference between original and RCB partitioning is caused by the LU factorization of the matrix of the coarsest level. In fact, the factorization takes 3.64 sec on 1024 cores and 7.86 sec on 2048. We do not see a reason for this drastic increase of factorization time since the problem sizes are quite close (2048 vs. 1865). The fact that the partitions with RCB have more neighbors does not seem to be a strong enough reason. In any case, larger differences would have to be visible in Tables 2 *and* 4.

In this paper we restricted ourselves to problems posed on grids of size $1024 \times 1024 \times 1024$. In [2], investigating the original solver, we als included $512 \times 512 \times 512$ and $256 \times 256 \times 256$ grids. On the former grid the solver scaled similarly as on the 1024^3 grid. On the latter grid scalability was somewhat poorer. By replacing the direct coarse level solver by an iterative coarse level solver we expect similar improvements in the parallel performance of our solver also for these smaller problem sizes. As in the computations of this note, scalability will not be affected much by an improved partitioning, however load balance and execution time.

5 Conclusions

We have presented and discussed improvements of a scalable Poisson solver suitable to handle domains with irregular boundaries as they arise, for example, in beam dynamics simulations. The solver employs the conjugate gradient algorithm preconditioned by smoothed aggregation-based AMG. The code exhibits excellent scalability up to 2048 processors with cylindrical pipes embedded in meshes with up to 1024^3 grid points. We have reduced the execution time by about 20% by redistributing the data using the recursive coordinate bisection (RCB) algorithm. This removes the severe load imbalance of the original approach in [2]. Scalability was further improved by iteratively solving the linear system of equations on the coarsest level of the AMG preconditioner.

Acknowledgment. These computations have been performed in the framework of a Large User Project grant of the Swiss National Supercomputing Centre (CSCS). We acknowledge the help of the Cray support team at CSCS.

References

[1] Adams, M., Brezina, M., Hu, J., Tuminaro, R.: Parallel multigrid smoothing: polynomial versus Gauss–Seidel. J. Comput. Phys. 188(2), 593–610 (2003)
[2] Adelmann, A., Arbenz, P., Ineichen, Y.: A fast parallel Poisson solver on irregular domains applied to beam dynamics simulations. J. Comput. Phys. 229(12), 4554–4566 (2010)
[3] Adelmann, A., Kraus, C., Ineichen, Y., Yang, J.J.: The Object Oriented Parallel Accelerator Library Framework. Technical Report. PSI-PR-08-02, Paul Scherrer Institut (2008-2010), http://amas.web.psi.ch/docs/opal/opal_user_guide-1.1.6.pdf
[4] Boman, E., Devine, K., Fisk, L.A., Heaphy, R., Hendrickson, B., Vaughan, C., Çatalyürek, Ü., Bozdag, D., Mitchell, W., Teresco, J.: Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide. Sandia National Laboratories, Albuquerque, NM (2007), Tech. Report SAND2007-4748W, http://www.cs.sandia.gov/Zoltan/ug_html/ug.html
[5] Cray XT5 Brochure. Cray Inc., Seattle (2009), http://www.cray.com/Products/XT/Systems/XT5.aspx (retrieved on July 13, 2010)
[6] Devine, K., Boman, E., Heaphy, R., Hendrickson, B., Vaughan, C.: Zoltan data management services for parallel dynamic applications. Comput. Sci. Eng. 4(2), 90–97 (2002)
[7] Forsythe, G.E., Wasow, W.R.: Finite-difference methods for partial differential equations. Wiley, New York (1960)
[8] Gee, M.W., Siefert, C.M., Hu, J.J., Tuminaro, R.S., Sala, M.G.: ML 5.0 smoothed aggregation user's guide. Tech. Report SAND2006-2649, Sandia National Laboratories (May 2006)
[9] Golub, G.H., van Loan, C.F.: Matrix Computations, 3rd edn. The Johns Hopkins University Press, Baltimore (1996)
[10] Greenbaum, A.: Iterative Methods for Solving Linear Systems. SIAM, Philadelphia (1997)
[11] Hackbusch, W.: Iterative solution of large sparse systems of equations. Springer, Berlin (1994)
[12] Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An overview of the Trilinos project. ACM Trans. Math. Softw. 31(3), 397–423 (2005)
[13] Hestenes, M.R., Stiefel, E.: Methods of conjugent gradients for solving linear systems. J. Res. Nat. Bur. Standards 49, 409–436 (1952)
[14] McCorquodale, P., Colella, P., Grote, D.P., Vay, J.-L.: A node-centered local refinement algorithm for Poisson's equation in complex geometries. J. Comput. Phys. 201(1), 34–60 (2004)

[15] Pöplau, G., van Rienen, U.: A self-adaptive multigrid technique for 3-D space charge calculations. IEEE Trans. Magn. 44(6), 1242–1245 (2008)
[16] Qiang, J., Gluckstern, R.L.: Three-dimensional Poisson solver for a charged beam with large aspect ratio in a conducting pipe. Comput. Phys. Commun. 160(2), 120–128 (2004)
[17] Qiang, J., Ryne, R.D.: Parallel 3D Poisson solver for a charged beam in a conducting pipe. Comput. Phys. Commun. 138(1), 18–28 (2001)
[18] The Trilinos Project Home Page, http://trilinos.sandia.gov
[19] Tuminaro, R.S., Tong, C.: Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines. In: ACM/IEEE SC 2000 Conference, SC 2000, 21 pages (2000), doi:10.1109/SC.2000.10008

Distributed Java Programs Initial Mapping Based on Extremal Optimization

Eryk Laskowski[1], Marek Tudruj[1,3], Ivanoe De Falco[2], Umberto Scafuri[2], Ernesto Tarantino[2], and Richard Olejnik[4]

[1] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
{laskowsk,tudruj}@ipipan.waw.pl
[2] Institute of High Performance Computing and Networking, ICAR-CNR, Naples, Italy
{ivanoe.defalco,umberto.scafuri,ernesto.tarantino}@na.icar.cnr.it
[3] Polish-Japanese Institute of Information Technology, Warsaw, Poland
[4] Computer Science Laboratory, University of Science and Technology of Lille, France
richard.olejnik@lifl.fr

Abstract. An application of extremal optimization algorithm for mapping Java program components on clusters of Java Virtual Machines (JVMs) is presented. Java programs are represented as Directed Acyclic Graphs in which tasks correspond to methods of distributed active Java objects that communicate using the RMI mechanism. The presented probabilistic extremal optimization approach is based on the local fitness function composed of two sub-functions in which elimination of delays of task execution after reception of required data and the imbalance of tasks execution in processors are used as heuristics for improvements of extremal optimization solutions. The evolution of an extremal optimization solution is governed by task clustering supported by identification of the dominant path in the graph. The applied task mapping is based on dynamic measurements of current loads of JVMs and inter-JVM communication link bandwidth. The JVM loads are approximated by observation of the average idle time that threads report to the OS. The current link bandwidth is determined by observation of the performed average number of RMI calls per second.

Keywords: distributed systems, scheduling, evolutionary algorithms.

1 Introduction

Optimization of the execution time of Java programs on clusters of Java Virtual Machines (JVM) is a challenging task due to specific execution paradigm of object programs and particular architectural features of the JVM. This problem has attracted researchers attention in several papers on dynamic load balancing and scheduling of Java programs for clusters and Grids [8,17]. The proposed solutions assume that the optimization is done at program runtime with the use of centralized or distributed load monitoring agents. Java program task scheduling on Grids is reported in [5]. It requires adequate computational and communication load metrics covered in [16,13].

Load balancing algorithms for Java program should take into account improvements of initial mapping of Java distributed application. This problem has not been sufficiently covered in current literature, although some papers propose an initial optimization including Java objects static clustering [6,11] or distributed byte-code clustering [12] in a set of JVMs.

This paper is presents a new approach to finding optimal initial mappings of Java applications to resources in heterogeneous multiprocessor systems. It is based on an adaptation of Extremal Optimization (EO), which is a very fast co-evolutionary algorithm proposed by Boettcher and Percus [14]. EO works is based on developing a single solution comprising a number of components s_i, each of which is a variable of the problem. Two kinds of fitness function are used to evaluate the components and the global solution quality. In EO the worst component is randomly updated, so that the solution is transformed into a new acceptable solution. A probabilistic version of EO /τ–EO/ has been designed [4] which aims in avoiding the local minimum phenomenon.

The initial Java program placement algorithm proposed in this paper is based on the probabilistic EO approach. Environment monitoring (system observation) predicts CPU and network services availability based on current CPU load and network utilization (maximal number of RMI calls per second). Object behavior monitoring determines the intensity of communication between active objects. Its principle is based on measuring the number of method calls between ProActive active (global) objects and the volume of serialized data.

The proposed framework is intended for execution optimization of distributed Java applications, run in clusters of multicore computing nodes. Although it uses macro–dataflow graphs (DAGs) as an application program representation, it works at a higher level of granularity than modern multicore–optimized parallel libraries, like PLASMA [1]. The proposed EO-based optimization algorithms are enough general to be comparable to classic task graph scheduling methods (as presented in [15,10]) and scheduling algorithms for Grid computing [5].

In our experiments, we have used the ProActive Java framework for cluster and Grid computing [2] as distributed programs execution management support. It provides a Java API and a set of tools for program management in different environments such as desktop, SMP, LAN, clusters and Grid. The application model is based on mobile Active Objects, asynchronous execution with synchronization (*Futures* mechanism) and Web Services and Grid support.

The paper is composed of three parts. First part presents the program representation and the executive system features. Next, the extremal optimization foundations are outlined. Then, the extremal optimization for Java program scheduling is described. Finally, experiment results with programs executed on cluster of JVMs based on multicore workstations are presented.

2 Executive Environment

Distributed Java applications are usually run in cluster and Grid environments using an execution management support middleware. In the presented work, execution of distributed Java programs is done employing the ProActive framework for cluster and Grid computing [2].

2.1 ProActive Framework Overview

ProActive is a Java middleware library (available under GPL open source license) providing an API for parallel, distributed, and multi-threaded computing, also with support for mobility and security. It is based on the Active Objects design pattern and allows for simplified and uniform programming of Java applications distributed on Local Area Networks (LANs), Clusters, Internet Grids and Peer-to-Peer Intranets. Important ProActive features include: active objects mobility in the form of Remote Mobile Objects, Group Communication, OO SPMD parallel programming model, Web Services and Grid support, various communication and integration protocols: RMI, SSH, LSF, Globus, PBS.

A distributed ProActive application is composed of a set of active objects. An Active Object is implemented as a standard Java object with an attached thread of control. Incoming method calls are stored in a queue of pending requests in each active object, which decides in which order to serve them. Thus, method calls sent to active objects are asynchronous with transparent future objects and the synchronization handled by a *wait-by-necessity* mechanism.

The deployment of Active Objects on nodes of a parallel system is specified by external XML description and/or API calls. Invoking methods of remote objects does not require from the developer to use explicitly any communication or remote access mechanism – the placement of active objects is transparent to their clients.

The communication between active objects in ProActive is implemented using a Remote Method Invocation mechanism (Java RMI). The data to be communicated (Java objects) are serialized and passed as network messages. The communication semantics depends upon the signature of the method, with three possible cases: synchronous invocation, one-way asynchronous invocation, and asynchronous invocation with future result.

2.2 Program and System Representation

In the paper, we are interested in initial deployment optimization of distributed Java programs, which can be represented as directed acyclic graphs (DAGs). Thus, an application is described by a weighted directed acyclic graph $G_{dag} = \{P, E\}$, where P is a set of **communicating tasks**, and E is a set of **data transfers between tasks**. Each task $p_k, k \in \{1 \dots |P|\}$ has a weight γ_k which represents the number of instructions to be executed. These weights are determined either during the sample execution of the program for some representative data or provided by application developer. An edge weight ψ_{km} represents the amount of data to be sent from task k to another m-th task. Similarly, these weights can be sampled or provided explicitly by a developer. A program is executed according to the *macro data-flow* model. Tasks start their execution when all needed data arrived through task incoming edges. When task is finished, it sends produced data to the succeeding tasks. The graphs are static and deterministic. We assume that, to preserve the DAG constraints, all loops are unrolled or encircled inside the body of a single task.

The target executive system consists of N computing resources (nodes). Each node, identified by an integer value in the range $[0, N-1]$, is equipped with multicore processors. We assume that all cores in a single node i, which number is denoted as κ_i, are homogeneous. The current status of system resources is given by the **node power** α_i which is the number of instructions computed per time unit in a core of the node i and average load of each core $l_i^k(\Delta t)$ in a particular time span Δt: $l_i^k(\Delta t)$ ranges in $[0.0, 1.0]$, where 0.0 means a core with no load and 1.0 a core loaded at 100%. Thus $(1-l_i^k(\Delta t))\alpha_i$ represents the power of the core k of the node i available for the execution of the tasks scheduled by our algorithm. The **communication bandwidth** between any pair of nodes i and j is denoted as β_{ij}. The current status of the system is supposed to be contained in tables based either on statistical estimations in a particular time span or gathered by tracking periodically and by forecasting dynamically resource conditions.

The program representation corresponds to ProActive distributed application in which a task is a thread of an Active Object and an edge is a method call in another Active Object. The execution of an Active Object method, which constitutes the task, can start when this object collected all necessary data. During execution, an Active Object communicates only with local objects. At the end of execution, the method sends data to the successors.

3 Extremal Optimization Algorithm

In nature, highly specialized, complex structures often emerge when their most inefficient elements are selectively driven to extinction. Such a view is based on the principle that evolution progresses by selecting against the few most poorly adapted species, rather than by expressly breeding those species well adapted to the environment. This idea has been applied successfully in the Bak–Sneppen model [14] which shows the emergence of Self-Organized Criticality (SOC) in ecosystems. According to that model, each component of an ecosystem corresponds to a species, which is characterized by a fitness value. The evolution is driven by a process where the least fit species together with its closest dependent species are selected for adaptive changes. As the fitness of one species changes, those of its neighbours are affected. Thus, species coevolve and the resulting dynamics of this extremal process exhibits the characteristics of SOC, such as punctuated equilibrium [14].

3.1 Extremal Optimization Principle

Extremal Optimization was proposed by Boettcher and Percus [3] and draws upon the Bak–Sneppen mechanism, yielding a dynamic optimization procedure free of selection parameters. It represents a successful method for the study of

Algorithm 1. General EO algorithm

```
initialize configuration $S$ at will
$S_{\text{best}} \leftarrow S$
while maximum number of iterations $N_{\text{iter}}$ not reached do
   evaluate $\phi_i$ for each variable $s_i$ of the current solution $S$
   rank the variables $s_i$ based on their fitness $\phi_i$
   choose the rank $k$ according to $k^{-\tau}$ so that the variable $s_j$ with $j = \pi(k)$ is selected

   choose $S' \in Neigh(S)$ such that $s_j$ must change
   accept $S \leftarrow S'$ unconditionally
   if $\Phi(S) < \Phi(S_{\text{best}})$ then
      $S_{\text{best}} \leftarrow S$
   end if
end while
return $S_{\text{best}}$ and $\Phi(S_{\text{best}})$
```

NP–hard combinatorial and physical optimization problems [4,3] and a competitive alternative to other nature–inspired paradigms such as Simulated Annealing, Evolutionary Algorithms, Swarm Intelligence and so on, typically used for finding high–quality solutions to such NP–hard problems. Differently from the well–known paradigm of Evolutionary Computation (EC), which assigns a given fitness value to the whole set of the components of a solution based upon their collective evaluation against a cost function and operates with a population of candidate solutions, EO works with one single solution S made of a given number of components s_i, each of which is a variable of the problem and is thought to be a species of the ecosystem. Once a suitable representation is chosen, by assuming a predetermined interaction among these variables, a fitness value ϕ_i is assigned to each of them. Then, at each time step the overall fitness Φ of S is computed and this latter is evolved, by randomly updating only the worst variable, to a solution S' belonging to its neighbourhood $Neigh(S)$.

This last is the set of all the solutions that can be generated by randomly changing only one variable of S by means of a uniform mutation. However, EO is competitive with respect to other EC techniques if it can randomly choose among many $S' \in Neigh(S)$. When this is not the case, EO leads to a deterministic process, i.e., gets stuck in a local optimum. To avoid this behaviour, Boettcher and Percus introduced a probabilistic version of EO based on a parameter τ , i.e., τ–EO. According to it, for a minimization problem, the species are firstly ranked in increasing order of fitness values, i.e., a permutation π of the variable labels i is found such that: $\phi_\pi(1) \leq \phi_\pi(2) \leq \ldots \phi_\pi(n)$, where n is the number of species. The worst species s_j is of rank 1, i.e., $j = \pi(1)$, while the best one is of rank n. Then, a distribution probability over the ranks k is considered as follows: $p_k / k^{-\tau}$, $1 \leq k \leq n$ for a given value of the parameter τ. Finally, at each update a generic rank k is selected according to p_k so that the species s_i with $i = \pi(k)$ randomly changes its state and the solution moves to a neighbouring one $S' \in Neigh(S)$ unconditionally. The only parameters are the maximum number of iterations

N_{iter} and the probabilistic selection value τ. For minimization problems τ–EO proceeds as in the Algorithm 1.

3.2 Extremal Optimization Applied to Program Optimization

Using the provided system and program representation, the optimization of initial distribution of components of an application translates to the problem of mapping the application divided into P tasks on N nodes. Since tasks address non-dedicated resources, their own local computational and communication loads must be considered to evaluate the computation time of the tasks of the program to be scheduled. There exist several prediction methods to face the challenge of non–dedicated resources.

Solution Encoding. A scheduling solution S is represented by a vector $\mu = (\mu_1, \ldots, \mu_P)$ of P integers ranging in the interval $[0, N-1]$, where the value $\mu_i = j$ means that the solution S under consideration maps the i–th task p_i of the application onto processor node j. The number of processor cores is not represented inside the solution encoding, however, it is taken into account when estimating the global and local fitness functions while solving the scheduling problem. This will be explained below.

Global Fitness Function. The global fitness accounts for the time of execution of a scheduled program. The execution time of the scheduled program is provided by a program graph execution simulator (Algorithm 2). The simulator assigns time annotations to program graph nodes based on the processor computing power availability and communication link throughput available for a given program execution. Algorithm 2 determines also the data ready time DRT for each task.

Local Fitness Functions. We have designed and used three variants of local fitness function. All parameters necessary for computing the value of two variants of local fitness functions (**a, c**) are obtained during the execution of program graph simulation procedure (see Algorithm 2 and Fig. 1).

Local fitness function a. For the system of heterogeneous processors interconnected by a heterogeneous network in which our program is executed with sharing resources with other system load, the local fitness function **a** (LFFa) of a task is the delay of the execution start of a task comparing the data ready time DRT of a task. We call this delay the initial execution delay.

$$LFFa(t) = Availability\_time(t) - Ready\_time(t)$$

Local fitness function b. The second local fitness function (LFFb) is the extension of the LFFa function. The LFFb moves the tasks belonging to dynamic critical path of the graph and that are improperly placed on nodes, to other nodes. The dynamic critical path is the longest path in the scheduled graph. We determine the dynamic critical path by traversing the graph from the sink task

Algorithm 2. Program graph execution simulation procedure

```
Mark entry task of the graph as ready at the time 0
for each core $c$ of all processors: $Availability\_time(c) \leftarrow 0$
while not all tasks are visited do
    $t \leftarrow$ the ready task with the earliest starting time
    $n \leftarrow \mu_t$ {the node of task $t$}
    $c \leftarrow$ the core of $n$ which has the earliest $Availability\_time$
    Place task $t$ on core $c$ of node $n$
    $Starting\_time(t) \leftarrow \max(Availability\_time(c), Ready\_time(t))$
    $TaskCompletion\_time(t) \leftarrow Starting\_time(t) + Execution\_time(t)$
    $Availability\_time(c) \leftarrow TaskCompletion\_time(t)$
    Mark $t$ as visited
    for all succesor task $s_i^t$ of task $t$ do
        $DRT \leftarrow TaskCompletion\_time(t) + Communication\_time(t, s_i^t)$
        if $DRT > Ready\_time(s_i^t)$ then
            $Ready\_time(s_i^t) \leftarrow DRT$
        end if
        if $TaskCompletion\_time(t) > LastParent\_time(s_i^t)$ then
            $LastParent\_time(s_i^t) \leftarrow TaskCompletion\_time(t)$
        end if
        if all data of $s_i^t$ arrived then
            mark $s_i^t$ as ready
        end if
    end for
end while
return $\max(Availability\_time)$
```

to the entry task. Then we look for the tasks on critical path, whose parent task from the critical path is placed on different node than the task's node. We assign to all those tasks whose parents are assigned to different node, the maximal delay value found during the calculation of fitness function **a** (LFFa) or a arbitrary constant value, thus increasing the probability they would get selected during the ranking process.

$$LFFb(t) = \begin{cases} LFFa(t) & \text{when } t \text{ does not belong to DCP,} \\ Const & \begin{array}{l}\text{when } t \text{ belong to DCP and its parent} \\ \text{on DCP is not on the same node.}\end{array} \end{cases}$$

where $Const = \max(LFFa(t))$ if there exists t for which $LFFa(t) \neq 0$ otherwise $Const =$ arbitrary value > 0.

Local fitness function c. The local fitness function **c** (LFFc) of a task is the complete delay of task execution comparing the execution under optimal conditions, i.e. there is no communication overhead nor resource contention between tasks and the task is executed on the fastest processor. We call this delay the total execution delay.

$$LFFc(t) = TaskCompletion\_time(t) - LastParent\_time(t) - FastestExecution(t)$$

where $FastestExecution(t)$ – the execution time of task t on the fastest processor.

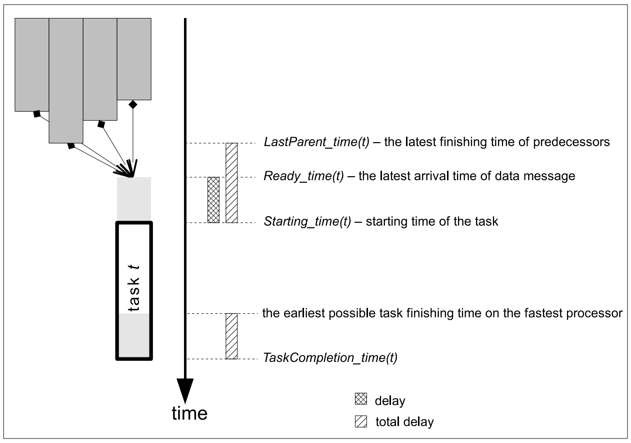

LastParent_time(t) – the latest finishing time of predecessors
Ready_time(t) – the latest arrival time of data message
Starting_time(t) – starting time of the task
task t
the earliest possible task finishing time on the fastest processor
TaskCompletion_time(t)
time
delay
total delay

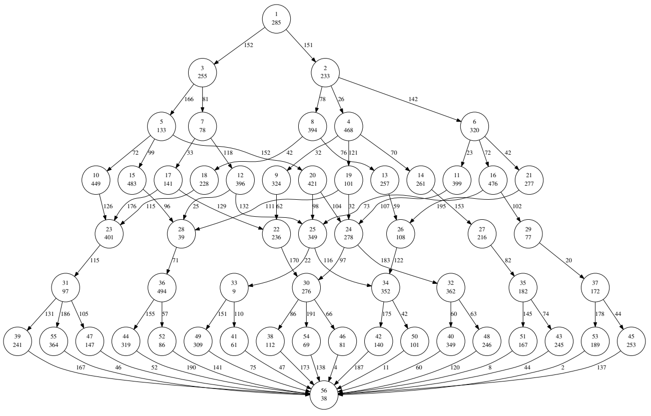
Fig. 1. Computation of *delay* and *total delay* values for given task *t*

4 Experimental Results

During experiments we have used two sets of synthetic graphs and the graph of a medical application – ART algorithm (reconstruction of tomographic scans [9]). The first set of synthetic graphs consists of seven randomly generated graphs (*gen-1...3, gen-3a...d*), with layered structure, Fig. 2(a). Each task (node) of this graph represents a mixed float- and integer-based generic computation (random generation of matrix of doubles, then floating-point matrix-by-vector multiplication, then rounding to integers and integer sort) with execution time defined by node weight (the weight controls the number of iterations of the generic computation). The second set of synthetic graphs consists of two hand-made graphs with known optimal mappings (*gen-m1, gen-m2*), with a general structure similar to the structure of randomly generated graphs, Fig. 2(b).

The following extremal optimization algorithm variants have been used during the experiments: **eo-a, eo-b, eo-c,** based on described in the paper local fitness functions (local fitness function LFFa, LFFb and LFFc respectively). For the comparative experiments we used a list scheduling algorithm with the ETF (Earliest Task First) heuristics. The ETF implementation is based on the description from [7]. We used a cluster of 7 homogeneous dual core processor nodes for program graph scheduling and program execution under ProActive.

Comparison of real execution times of an exemplary application, scheduled by different methods is presented in Fig. 3(a) and Fig. 3(b). The different variants of EO method obtained similar quality of initial mappings of applications. For synthetic graphs, the best results are obtained by **eo-b** algorithm, however the **eo-c** method is only marginally worse. For ART application graph, the **eo-c** method is the best one among the different EO variants. The typical execution time increase, comparing the EO and ETF algorithm, is below 10% (the only exception is *gen-m1* graph, for which ETF was able to find the optimal


(a) gen-3b

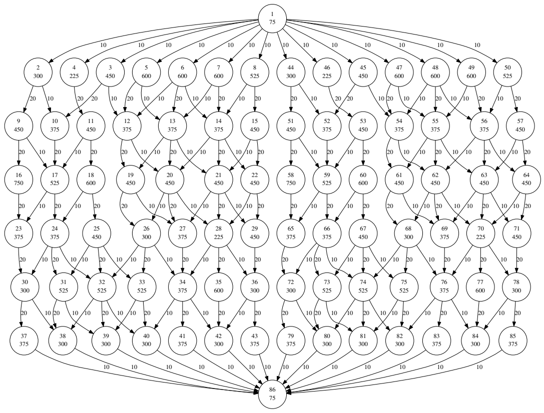
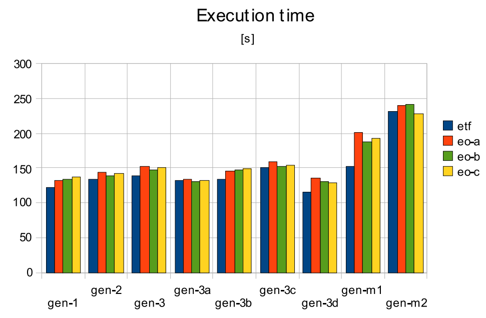

(b) gen-m2

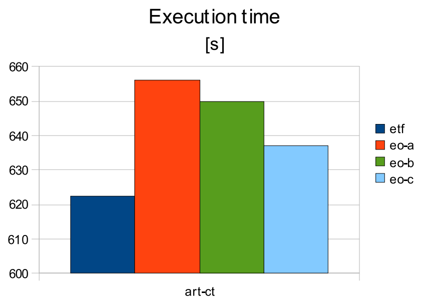
Fig. 2. The structure of a synthetic exemplary application graphs

solution). The experimental results show that EO technique is able, in general, to draw the same level of the quality of results as classical scheduling and mapping approaches like ETF algorithms. In our opinion, it is a quite good result, taking into account the simplicity of the basic principle of extremal optimization method.

In another experiment we empirically extrapolated the actual time complexity of presented optimization algorithms. For this purpose we used a set of large, randomly generated graphs (the number of nodes from 350 to 7000), which were scheduled by EO and ETF algorithms. The actual running times of the optimization algorithms confirmed theoretical complexity of EO and ETF methods, which is approximately $C(n^2)$ for EO (it is determined by the time complexity of the graph execution simulation, see Algorithm 2) and $C(n^3N)$ for ETF (see [7] for details), where n is the size of the graph, and N is the number of computing nodes. Although time complexity of EO method is lower than that of



(a) synthetic graphs



(b) ART graph

Fig. 3. The real execution times of the scheduled program graphs for different scheduling algorithms

ETF, the actual running times of different kinds of the EO algorithm for small graphs were much longer than the running times of ETF algorithm. It can be considered the main drawback of EO method. However, for large task graphs (tens of thousands of tasks in our experiment), EO-based methods are faster than ETF algorithm. Among investigated EO variants, **eo-a** and **eo-c** are the fastest, since **eo-b** method introduces additional run-time overhead due to the dominant sequence tracking. Another advantage of EO methods are very small memory requirements.

Experimental results indicate that extremal optimization technique can be useful for large mapping and scheduling problems when we will pay special attention to run-time optimization of EO algorithm. It is due to the low time complexity and memory consumption of EO methods. For small sizes of application graphs, it is advised to use classic scheduling methods, as ETF list scheduling.

5 Conclusions

The paper has shown how to optimize a distributed program schedule by using the extremal optimization technique. For homogeneous systems the extremal optimization algorithm has delivered results comparable to the ETF algorithms. The execution times of the scheduled programs determined by simulation were close to the real execution time of the synthetic programs corresponding to the scheduled graphs.

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. Journal of Physics: Conference Series 180 (2009)

2. Baude, et al.: Programming, Composing, Deploying for the Grid. In: Cunha, J.C., Rana, O.F. (eds.) GRID COMPUTING: Software Environments and Tools. Springer, Heidelberg (2006)
3. Boettcher, S., Percus, A.G.: Extremal optimization: methods derived from coevolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999), pp. 825–832. Morgan Kaufmann, San Francisco (1999)
4. Boettcher, S., Percus, A.G.: Extremal optimization: an evolutionary local–search algorithm. In: Bhargava, H.M., Ye, N. (eds.) Computational Modeling and Problem Solving in the Networked World. Kluver, Boston (2003)
5. Dong, F.: A taxonomy of task scheduling algorithms in the Grid. Parallel Processing Letters 17(4), 439–454 (2007)
6. Fiolet, V., Laskowski, E., Olejnik, R., Masko, L., Toursel, B., Tudruj, M.: Optimizing Distributed Data Mining Applications Based on Object Clustering Methods. In: Parallel Computing in Electrical Engineering (PARELEC 2006), pp. 257–262. IEEE (2006)
7. Hwang, J.-J., Chow, Y.-C., Anger, F., Lee, C.-Y.: Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. Siam J. Comput. 18(2), 244–257 (1989)
8. Jimenez, J.B., Hood, R.: An Active Objects Load Balancing Mechanism for Intranet. In: Workshop on Sistemas Distribuidos y Paralelismo, WSDP 2003, Chile (2003)
9. Kak, A.C., Slaney, M.: Principles of Computerized Tomographic Imaging. IEEE Press, New York (1988)
10. Kwok, Y.-K., Ahmad, I.: Benchmarking and Comparison of the Task Graph Scheduling Algorithms. J. Parallel Distrib. Comput. 59(3), 381–422 (1999)
11. Laskowski, E., Tudruj, M., Olejnik, R., Toursel, B.: Java Programs Optimization Based on the Most–Often–Used–Paths Approach. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 944–951. Springer, Heidelberg (2006)
12. Laskowski, E., Tudruj, M., Olejnik, R., Toursel, B.: Byte-code scheduling of Java programs with branches for Desktop Grid. Future Generation Computer Systems 23(8), 977–982 (2007)
13. Olejnik, R., Alshabani, I., Toursel, B., Laskowski, E., Tudruj, M.: Load balancing in the SOAJA Web Service Platform. In: 4th Workshop on Large Scale Computations on Grids (LaSCoG 2008), pp. 459–465. IEEE (October 2008)
14. Sneppen, K., Bak, P., Flyvbjerg, H., Jensen, M.H.: Evolution as a self–organized critical phenomenon. Proc. Natl. Acad. Sci. 92, 5209–5213 (1995)
15. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. IEEE Trans. on Parallel and Distributed Systems 13, 260–274 (2002)
16. Toursel, B., Olejnik, R., Bouchi, A.: An object observation for a Java adaptative distributed application platform. In: International Conference on Parallel Computing in Electrical Engineering (PARELEC 2002), pp. 171–176 (September 2002)
17. Yamaguchi, S., Maruyama, K.: Autonomous Load Balance System for Distributed Servers using Active Objects. In: 12th International Workshop on Database and Expert Systems Applications, Munich, Germany, pp. 167–171 (September 2001)

Software Environment for Parallel Optimization of Complex Systems

Ewa Niewiadomska-Szynkiewicz[1,2] and Michal Marks[1,2]

[1] Research and Academic Computer Network (NASK),
Wawozowa 18, 02-796 Warsaw, Poland
[2] Institute of Control and Computation Engineering,
Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
{ens,mmarks}@ia.pw.edu.pl

Abstract. The paper is concerned with parallel global optimization techniques that can be applied to solve complex optimization problems, and are widely used in applied science and in engineering. We describe an integrated software platform EPOCS (Environment for Parallel Optimization of Complex Systems) that provides the framework and tools which allow to solve complex optimization problems on parallel and multi-core computers. The composition, design and usage of EPOCS is discussed. Next, we evaluate the performance of methods implemented in the EPOCS library based on numerical results for a commonly used set of functions from the literature. The case study – calculating the optimal prices of products that are sold in the market is presented to illustrate the application of our tool to a given real-life problem.

Keywords: Software systems, numerical solvers, global optimization, parallel optimization, price management.

1 Introduction

Many issues related to applied science and engineering require the solution of optimization problems. Traditionally, these problems are solved using linear and nonlinear solvers, which normally assume that the performance function and a set of admissible solutions are convex and known in analytical form. In practice, however there are many problems that cannot be described analytically and are nonconvex, often with many extrema. During last decades, however, many theoretical and computational contributions helped to solve such type of problems arising from real-world application [1–3, 7, 10, 12]. Most of these techniques are based on global exploration of the domain. They are generally complex and usually involve cumbersome calculations, especially when consider simulation-optimization case when we have to perform simulation experiment in every iteration of the algorithm to estimate the value of a performance function [9]. The restrictions are caused by demands on computer resources – CPU and memory. On the other hand most of global techniques are easily adaptable to a parallel environment. Hence, the direction, which should bring benefits is parallel computing where the whole task is partitioned between several cores, processors or computers. Parallel implementation allows to reduce the computation time, improve the accuracy of the solution, and to execute large program which cannot be put on a single processor.

Recently a number of software packages with numerical solvers for global optimization have been developed, and can be found in the Internet. They support sequential and parallel programming. Publicly available implementations of interval analysis and branch-and-bound schemes, various types of techniques for global optimization (deterministic and stochastic) are provided in `http://www.solver.com/technology5.htm`. The goal of the COCONUT project (`http://www.mat.univie.ac.at/neum/glopt/coconut`) was to integrate the currently available techniques from mathematical programming, constraint programming, and interval analysis into a single discipline, to get algorithms for global constrained optimization. The authors of [4] report the results of testing a number of existing state of the art solvers using COCONUT routines on a set of over 1000 test problems collected from the literature. The Global World (`http://www.gamsworld.org/global/index.htm`) is a forum for discussion and dissemination of all aspects of global optimization problems. It provides links to libraries of solvers and a library of academic and practical test problems.

In this paper we describe our software package EPOCS. The contribution is organized as follows. First the composition, implementation and usage of our software is described. Next, the numerical algorithms supplied in EPOCS are presented. Finally, the results of numerical tests with solvers from EPOCS are discussed.

2 Description of EPOCS System

EPOCS (Environment for Parallel Optimization of Complex Systems) is an open source software framework that provides tools for solving complex optimization problems. EPOCS supplies the library of solvers for local and global optimization. It provides tools for research and education, and can be used to solve the optimization problems on parallel and multi-core machines. EPOCS implements two approaches to user-system interactions, i.e. EPOCS/CON and EPOCS/GUI. In the EPOCS/CON version batch processing is assumed. This type of user-system operation is dedicated to the complex tasks, where values of the objective function are calculated based on simulation (simulation-optimization scheme). EPOCS/GUI is dedicated mainly to education and research concerned with testing various algorithms and tuning their parameters. It supplies the graphical environment for optimization problem definition and results presentation. The graphical editor, symbolic expression analyzer and tools for dynamic, on-line monitoring of the calculation results are provided. The following presentation techniques are available: leaves of the function values (graphical) and a table of numbers. The results presentation is fitted to the optimization method (points, lines, grids). The visualization of a multidimensional problem is achieved by

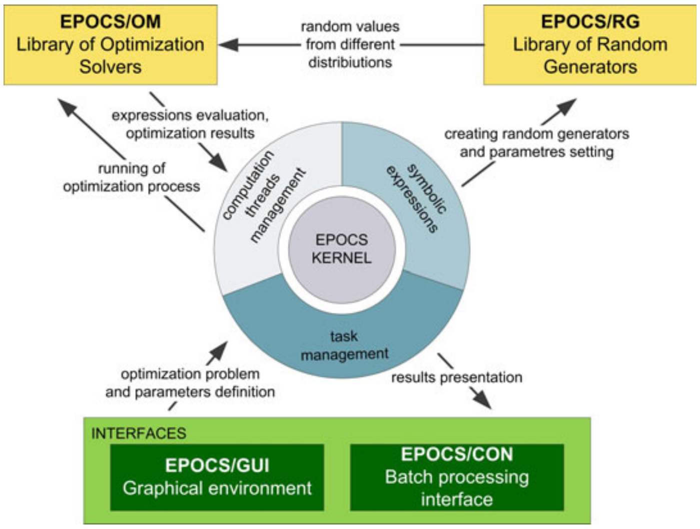

EPOCS/OM
Library of Optimization Solvers
random values from different distribiutions
EPOCS/RG
Library of Random Generators
expressions evaluation, optimization results
running of optimization process
creating random generators and parametres setting
computation threads management
symbolic expressions
EPOCS KERNEL
task management
optimization problem and parameters definition
results presentation
INTERFACES
EPOCS/GUI
Graphical environment
EPOCS/CON
Batch processing interface

Fig. 1. Components of the EPOCS system

displaying in the separate windows the leaves for each pair of variables, under the assumption that all other variables are fixed.

The EPOCS system is composed of three main components: *system kernel*, *user interfaces* and *library of numerical methods* (see Fig. 1). The system kernel provides support for symbolic expressions and is responsible for computation threads management and tasks management. The kernel provides also runtime infrastructure and manages communication between calculation processes and user interfaces. The user interfaces provide a set of tools mainly to support the interaction with the user and the runtime monitoring. It is obvious that the level of interaction and monitoring depends on the selected interface and is much more sophisticated for Graphical User Interface (GUI). The last, and core component of EPOCS – the library of numerical methods – provides a set of numerical algorithms divided into two components: the library of optimization solvers in sequential and parallel versions and the library of random generators.

EPOCS is based on C++ language and uses *Qt* – cross-platform application and UI framework. All numerical methods are implemented in uniform form as C++ classes. The optimization algorithms that are built upon EPOCS classes have hierarchical structure. The hierarchy of classes is well defined. Three fundamental classes: *task*, defining the considered optimization problem to be solved, *algorithm*, the basic class of all optimization methods and *generator*, for random numbers generation are supplied. New algorithms can be implemented applying classes defined in EPOCS. The open design of the system architecture, and its extensibility to include other open source modules, was chosen in the hope that the system will be a useful platform for researchers and students. The code is currently available for MS-Windows and Linux operating systems.

3 Numerical Methods Library

3.1 Library Overview

The numerical library consists of two components containing, respectively a collection of local and global optimization solvers and random numbers generators. The following algorithms for random numbers generation have already been implemented: uniform, normal (Gaussian), Beta, Cauchy and three quasi-random sequences: Halton, Sobol, Fauer [6]. Several techniques for one and multidimensional local and global search are provided. The current version of the system offers global optimization solvers from two groups [1–3, 7, 10] – deterministic: chaotic movement, branch-and-bound and clustering techniques, and stochastic: random search (pure and population set based direct search methods), simulated annealing (SA), genetic algorithms (GA), evolutionary strategies (ES), and hybrid techniques combining SA, GA and methods for solving local minimum [5, 11]. All listed algorithms are implemented in a few versions. Controlled random search algorithms (CRS2, CRS3, CRS4, CRSI, CRS6 [1]), simulated annealing and genetic algorithms in both simple and hybrid versions can be executed in parallel. The algorithms from the EPOCS library can be used to solve unconstrained and constrained optimization problems. The inequality constraints are accounted for in the minimized performance function using simple penalty terms for constraints violation.

3.2 Parallel Implementation

The implementation of parallel versions of optimization algorithms from the EPOCS library is based both on native system threads and the OpenMP tool. The OpenMP (Open Multi-Processing, http://www.openmp.org/) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in Fortran, C/C++ and Java on many architectures. It consists of a set of compiler directives, library routines, and environment variables that influence runtime behavior. An application can run on multiprocessor and multi-core machines. The objective of parallel implementations of the EPOCS system was to speed up the calculations and improve the accuracy of the solution. Several ways of methods parallelism were considered, which resulted several more and less complicated, and at the same time more and less effective variants of each algorithm. In general we propose to incorporate parallelism on two levels – parallel implementation of the EPOCS platform and parallel implementations of the numerical solvers from the EPOCS library. Thanks to parallel implementation of the EPOCS platform it is possible to execute different tasks and different solvers simultaneously. In general, the EPOCS system is designed as an application with master-slave communication model (see Fig. 2). We distinguished the master and the group of slave threads. The goal of a *master* (the main thread) which is initiated by the EPOCS systems, is to read all parameters from GUI, start the calculations and create slave threads. The new *slave* thread is created after selection the "run" command. The goal of the slave thread is to

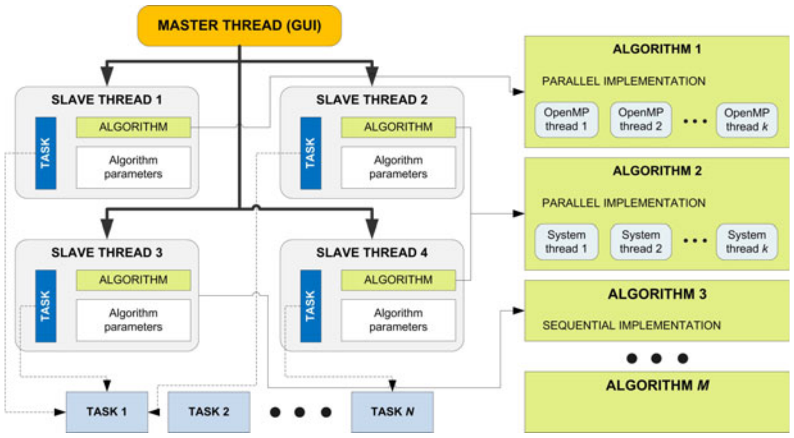


Fig. 2. Parallel implementation of the EPOCS system

perform the selected numerical algorithm on a selected task. Each thread consists of three components: the task, the algorithm and the algorithm parameters. Hence, it is possible to run multiple experiments using the same algorithm and the same task, but different algorithm parameters or input data. This is very useful feature of EPOCS system especially from educational point of view. In the same way the user can solve simultaneously different tasks using a given solver or use different solvers to solve a given task. Each thread (master or slave) can be executed on a separate processor.

In addition, EPOCS supplies parallel implementations of solvers. The system threads and the OpenMP programming interface were used to implement the parallel versions of optimization algorithms. For different algorithms different attempts to parallelization were considered but with focus on coarse-grained parallelism. Below we describe the parallel implementation of selected algorithms.

CRS Algorithms Parallel Implementation. CRS methods are population set based random search algorithms. The basic random search consists of three main steps: 1) generate the initial set of points P from the domain, 2) transform the population P, 3) check the assumed stopping condition. In principle, CRS methods were designed as a combination of local optimization algorithms with a global search procedure. In case of parallel implementation of all CRS methods supplied in EPOCS it is assumed that several independent computational threads transform the same, globally available population of points P. The current best point x_L is global for all computational threads, while the worst points x_H are local – one for each thread. Computational threads are initialized by the slave thread, which provides access to *Task* object and manages population of P points. The system threads are used (*QThread*) to calculation parallelization. The computation scheme for CRS is presented in Figure 3.

Genetic Algorithm and Simulated Annealing Parallel Implementation. Three versions of genetic algorithms were implemented: simple GA described in [7]

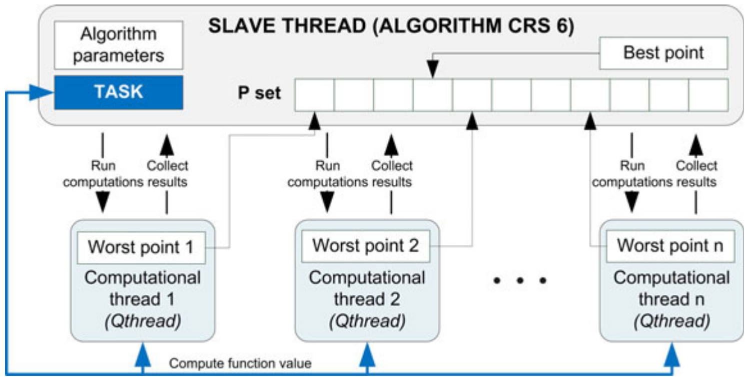


Fig. 3. The CRS6 algorithm – parallel implementation

and two hybrid techniques: genetic algorithm with the local tuning (GOD) and the combination of GA and SA called SA/SOS (Synchronous Approach with Occasional Solution Exchanges). The GOD method [11] integrates the simple GA with the well known downhill method (NM), developed by Nelder and Mead, and implemented in [6]. GOD operates as follows. In each step a new population is generated by genetic operations, and a subset by NM method. In order to speed up the convergence, the proportion of points generated by both methods varies as the global optimum is approached. The more iterations have been executed, the more points are created using the NM method.

The method called SA/SOS [5] combines two algorithms: SA and GA. SA/SOS operates as follows. Computation is performed on the initial set of points. SA is used to transform the population. After assumed number of steps, a crossover operator as implemented in GA is applied to the selected points. The randomly selected coordinates of chosen points are modified. The overall solution quality is measured as a sum of fitness of all points in the population. The control parameter T (temperature) in SA does not change in case when the overall solution is improved. Otherwise, it is changed according to a chosen cooling scheme.
In case of parallel versions of GA, GOD, SA and SA/SOS several independent instances of a given algorithm are executed, each on a separate processor with own set of input data. The slave thread is responsible for calculation processes initialization and communication with computational threads. After each assumed number of iterations the migration of the best individuals is coordinated by the slave thread. The OpenMP interface was used to implement parallel versions of all these methods. The comparison of given algorithms execution for described above two attempts to algorithms parallelization is presented in Figure 4.

4 Numerical Results

To verify and test the efficiency of solvers supplied in the EPOCS system we performed numerical experiments for commonly used test functions collected

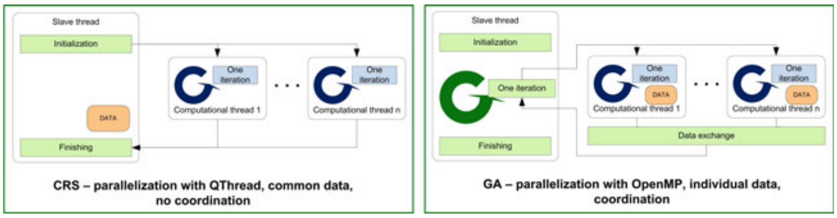


Fig. 4. Comparison of parallel CRS and GA execution (calculation schemes)

from literature. Next, we used EPOCS to solve several real-life optimization problems in economy and environmental systems. In this paper we present the case study concerned with a price management, and discuss the results achieved for stochastic search methods and heuristic techniques. All tests, which results are reported here were curried out on the Sun Fire V440 computer equipped with four processors.

4.1 Comparison of EPOCS Solvers

The goal of the first series of tests was to compare the performance of selected solvers supplied in the EPOCS library. We present the comparison of the simple Genetic Algorithm (GA) and Simulated Annealing (SA) with two hybrid techniques GOD and SA/SOS described in the section 3.2. All methods were tested on a set of commonly used test functions: Ackley (AC), Levy (LE), Rastrigin (RA) and Griewank (GR) described in [1], all with the global minimum equal 0. The sequential and parallel versions of selected solvers were considered. The focus was on benefits of parallel implementation. Three main criteria that determined the performance of the compared algorithms – the quality of the final result, running time and number of function evaluation – were taken into consideration.

Sequential computation. Two series of tests were performed. The GOD algorithm was compared with the genetic algorithm (GA), and the SA/SOS method was compared with the simulated annealing (SA). The calculations were terminated when the difference of performance values for a few trial points was less than 0.1. Table 1 shows the average results over series of 5 trials for four methods: GA, GOD, SA/SOS and SA. The values collected in the adequate columns denote: fun – test function, n – problem dimension, f^* – average value of the performance function calculated for 5 runs of the algorithm, f_{eval} – number of performance function evaluation needed to find the solution with the assumed accuracy. The numerical results collected in table 1 indicate that the GOD method is better both in terms of speed of convergence to the solution and calculated solution's accuracy than GA. The SA/SOS method achieves much better solution than SA with the similar computational burden. Hence, we can say that the proposed modification makes GA and SA far more effective.

Table 1. Comparison of four solvers: GA, GOD, SA and SA/SOS

		GOD		GA		SA/SOS		SA	
fun	n	f^*	f_{eval}	f^*	f_{eval}	f^*	f_{eval}	f^*	f_{eval}
AC	5	0.87	256	7.07	1600160	2.96	130481	3.61	120815
	10	0.93	1654	13.03	1660080	5.18	384525	5.27	362591
	20	3.34	16973	15.86	1700160	6.74	1270940	9.43	1200570
RA	5	0.87	1700	4.31	1400160	2.87	1317	6.69	121641
	10	1.51	23260	28.84	3200160	23.61	389842	30.98	362186
	20	17.5	655179	106.15	3500160	91.92	1282865	116.16	1201664

Table 2. Comparison of sequential and parallel GA and GOD

Size & threads		GA						GOD					
		GR			RA			GR			RA		
n	nt	f_{best}	f^*	f_{worst}	f_{best}	f^*	f_{worst}	f_{best}	f^*	f_{worst}	f_{best}	f^*	f_{worst}
5	1	1.5	1.64	1.87	3.76	4.31	4.8	0.06	0.08	1.01	0.79	0.87	1.01
	2	0.95	1.55	1.93	0.58	0.91	1.08	0.05	0.08	0.09	0.65	0.75	0.83
	3	0.94	1.25	1.44	0.31	0.71	1.38	0.04	0.07	0.08	0.62	0.71	0.80
	4	0.74	1.19	1.48	0.33	0.51	0.74	0.02	0.04	0.07	0.41	0.45	0.52
20	1	26.68	28.26	29.37	101.42	106.15	111,05	0.76	0.98	1.24	16.50	17.53	18.03
	2	20.62	24.63	29.33	77.57	83.47	92.62	0.51	0.66	0.75	15.91	16.52	17.99
	3	13.07	22.41	30.32	67.49	72.71	77.20	0.41	0.52	0.65	13.93	14.98	16.78
	4	12.75	16.98	23.35	62.10	64.90	72.53	0.38	0.44	0.54	13.01	13.99	15.97

Parallel computation. Parallel GA and GOD were tested for various numbers of threads. Final quality of result and computational effort measured by number of function evaluations were compared. The goal of parallelization was to calculate the solution with high level of accuracy in the time equal to sequential solver execution. The results of numerical experiments are presented in table 2. The values collected in the table denote: n – problem dimension, nt – number of threads (each thread was executed on a separate processor), f^* – average value of the performance over a serie of 5 trials, f_{best} and f_{worst}, respectively the best and worst solution. In general, the improvement of final results for multiple threads was observed. In most cases the accuracy of the solution increased in 10-20%. However, parallel GA found results even 50% better than its sequential version.

The goal of the last series of tests was to compare the efficiency of sequential and parallel SA executed on two processors. The goal was to speed up the calculations. The calculation times (in seconds) of Levy (LE) test function ($n = 1, \ldots, 400$) minimization are presented in Fig. 5. The calculation speedup after code parallelization was from 1.2 to 1.7, and it increased with the problem dimension.

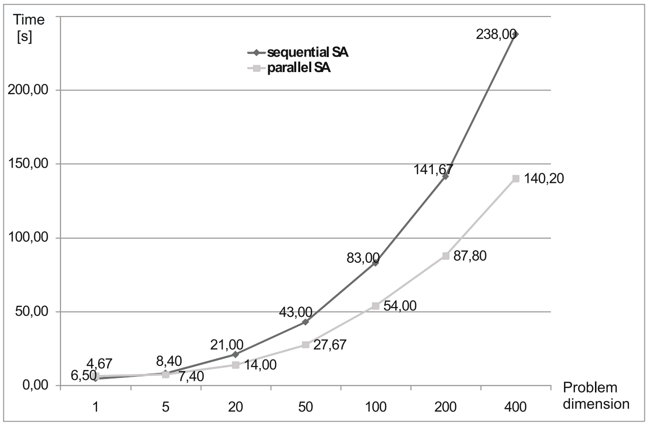


Fig. 5. Calculation time – minimization of Levy test function ($n = 1$ to $n = 400$)

5 Case Study Results

The EPOCS software was used to calculate the optimal prices for products that are sold in the market. The goal was to maximize the long-term profit PR equal to revenue minus cost. Let us consider n products that are sold in the market. Assume that $x = [x_1, x_2, \ldots, x_n]$ denotes a vector of prices of n products; x_i is the price of the i-th product. The relationship between prices and sales is called *price response function* $S(x)$ [8]. Several sales models are defined in literature. In our research we considered the model taking into account the elasticity of sales of one product with respect to the prices of other products. The expected sales was described as follows: $S_i(x) = \alpha_i \prod_{j=1}^{n} x_j^{\beta^{ij}}$, where x_j denotes the price of product j, α_i is the scaling factor for sales of product i, β^{ij} is the elasticity of sales of product i with respect to the price of product j (β^{ii} is referred to as the direct elasticity and β^{ij}, $i \neq j$ is the cross elasticity). The following constraints for price, sale and cash of each product are usually considered in optimal prices calculation: $x_i^{min} \leq x_i \leq x_i^{max}$, $S_i^{min} \leq S_i(x) \leq S_i^{max}$, $C_i^{min} \leq x_i S_i(x) \leq C_i^{max}$ for $i = 1, \ldots, n$. In addition, constraints for total sale and cash are usually taken into account: $TS^{min} \leq \sum_{i=1}^{n} S_i(x) \leq TS^{max}$, $TC^{min} \leq \sum_{i=1}^{n} x_i S_i(x) \leq TC^{max}$. In these constraints x_i^{min} and x_i^{max} denote minimal and maximal prices of product i, S_i^{min}, S_i^{max} minimal and maximal sale, C_i^{min}, C_i^{max} minimal and maximal cash, TS^{min}, TS^{max} minimal and maximal total sale, TC^{min}, TC^{max} minimal and maximal total cash.

Finally, the optimization problem was formulated. The goal was to calculate the prices for products that maximize the global profit PR, subject to constraints, and price response function listed above:

$$\min_{x_i, i=1,\ldots,n} \left[PR(x) = \sum_{i=1}^{n} \left(\frac{x_i}{1+v_i} - d_i \right) S_i(x) \right] \tag{1}$$

where v_i and d_i are given constants corresponding to the market entities of VAT and cost per product. We solved problem (1) for several sets of real data collected from IT company, containing various groups of products offered in supermarkets. All parameters in presented formulas were calculated based on market investigation. CRS2 and CRS6 algorithms [1] supplied in the EPOCS library were used to calculate the optimal prices. The efficiency of sequential and parallel versions of both algorithms were compared. The goal of parallelization was to speed up calculations. The optimization results of price management problem for various number of products are presented in tables 3 and 4. The assumed accuracy of the optimal point calculation was 1E-4. The average solution for the experiments with 15 products, number of function evaluations and calculation time for 5 runs of each algorithm are collected in table 3.

Table 3. Calculation results, 15 products, CRS2 and CRS6 methods (1 – 4 processors)

Number of	CRS2		CRS6	
processors	PR	Time [s]	PR	Time [s]
1	1236.74	1.23	1241.27	2.02
2	1236.92	0.64	1241.27	1.07
3	1236.94	0.47	1241.27	0.77
4	1237.13	0.44	1241.27	0.63

The speedup for the 15-dimension price management problem and parallel CRS2 and CRS6 (versions with four threads, each executed on separate processor) are as follows:

$$\frac{1\_thread\_time_{CRS2}}{4\_threads\_time_{CRS2}} = \frac{1.23}{0.44} = 2.8, \qquad \frac{1\_thread\_time_{CRS6}}{4\_threads\_time_{CRS6}} = \frac{2.02}{0.63} = 3.21$$

The values of speedup show the effectiveness of parallel implementation of both solvers. The results of experiments for 15, 31, 53 and 76 products, CRS2 and CRS6 methods, sequential and parallel implementations are presented in table 4. The solutions provided by both algorithms are compared with the best. In this table T_s and T_p denote calculation time for sequential and parallel solvers. The complexity of task (1) increases with the number of products and values of elasticity of sales of given products that influence the shape of function $S(x)$,

Table 4. Calculation results, 4 threads, different number of products, CRS2 and CRS6

Number of	The best	CRS2				CRS6			
products	PR	PR	T_s [s]	T_p [s]	Speedup	PR	T_s [s]	T_p [s]	Speedup
15	1241.27	1236.74	1.23	0.44	2.80	1241.27	2.02	0.63	3.21
31	830.75	809.21	12.92	3.79	3.41	830.75	25.31	6.54	3.87
53	544.65	526.05	21.82	6.20	3.52	544.65	66.39	16.99	3.91
76	1898.66	1836.11	42.43	11.50	3.69	1898.66	338.48	85.64	3.95

and finally the performance function. The results of experiments collected in the table 4 indicate that the calculation speedup increases with the problem complexity. Based on achieved results, as a conclusion we propose the following strategy: in cases when accuracy of the solution is the crucial the CRS6 method is suggested; when it is crucial that the problem is solved quickly the simpler method, i.e. CRS2 should be used. Both in case of CRS2 and CRS6 the parallel implementation provides better solutions with respect to sequential realization, and speeds up the calculations.

6 Conclusions

We presented the EPOCS framework for the complex systems sequential and parallel optimization. We can say that our tool can be successfully used for solving complex global optimization problems. The open design of the system architecture and extensibility to include new numerical methods make EPOCS a useful tool for researchers and students. EPOCS is continuously supported, maintained, and improved. Our comparative study of sequential and parallel implementations of the selected global techniques shows that the parallel calculations can improve the effectiveness of a given global method but the results strongly depend on their implementation and assumed attempt to parallelization.

References

1. Ali, M.M., Torn, A.: Population set-based global optimization algorithms: Some modifications and numerical studies. Computers and Operations Research 31, 1703–1725 (2004)
2. Horst, R., Pardalos, P.: Handbook of Global Optimization. Kluwer (1995)
3. Michalewicz, Z., Fogel, D.B.: How to Solve it: Modern Heuristcs. Springer, Heidelberg (2000)
4. Neumaier, A., Shcherbina, O., Huyer, W., Vinko, T.: A comparison of complete global optimization solvers. Mathematical Programming 103, 335–356 (2005)
5. Onbasoglu, E., Ozdamar, L.: Parallel simulated annealing algorithms in global optimization. Journal of Global Optimization 19, 27–50 (2001)
6. Press, W.H., Tukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C. The Art of Scientific Computing. Cambridge University Press (1992)
7. Schaefer, R.: Foundations of global genetic optimization. Springer, Heidelberg (2007)
8. Simon, H.: Price management. North-Holland (1989)
9. Spall, J.: Introduction to Stochastic Search and Optimization. John Wiley & Sons (2003)
10. Weise, T.: Global Optimization Algorithms: Theory and Application. e-book (2009), http://www.it-weise.de/projects/book.pdf
11. Yang, R., Douglas, I.: Simple genetic algorithm with local tuning: Efficient global optimizing technique. Journal of Optimization Theory and Applications 98, 449–465 (1998)
12. Zhigljavsky, A.A., Zilinskas, A.: Stochastic Global Optimization. Springer Optimization and its Applications, vol. 9. Springer, New York (2008)

Parallel Programming in Morpho

Snorri Agnarsson

Faculty of Engineering and Natural Sciences, University of Iceland
snorri@hi.is

Abstract. Morpho is a multi-paradigm programming language developed at the University of Iceland that supports parallel programming using both fibers (coroutines) and concurrently executing tasks (threads). Communication between both tasks and fibers is through channels. Morpho is open source and an alpha version is available[1]. Morpho can be used to augment Java with massively scalable multitreading, with orders of magnitude more concurrent computations than is possible with regular Java threading. Morpho supports polymorphic modules using a unique method based on substitutions rather than parametrization.

Keywords: concurrency, process-oriented programming, functional programming, fibers.

1 Introduction

The main motivation for Morpho is to develop a full-featured language supporting polymorphic modules based on modules as substitutions rather than on parametrization of modules. However, that aspect is not the main thrust of this paper. Rather this paper concentrates on the parallel programming aspects of Morpho and promotes the thesis that effective and scalable parallelism can conveniently be based on lightweight processes that can best be achieved by languages that allocate activation records on the heap using garbage collection for their management.

The Morpho programming language is designed to be a massively scalable scripting language. It runs on top of Java, can be invoked from Java, and Morpho programs have direct access to the functionality of Java and vice versa. Morpho is, however, distinguished from Java in various ways.

- The memory footprint of Morpho tasks is orders of magnitude smaller than that of Java threads, which have corresponding functionality.
- Morpho has fibers (coroutines) which are even more lightweight than tasks. The number of fibers and tasks that a Morpho program can run is orders of magnitude greater than the number of threads that a corresponding Java program can run.
- Morpho supports functional programming as well as imperative programming and object-oriented programming.

[1] http://morpho.cs.hi.is

  - Morpho has lexical scoping with nested functions and functions as first class values that can be assigned, returned as values and passed between system components.
  - Morpho has full tail-recursion removal of function calls and method invocations.
  - Morpho supports lazy streams for incremental (and parallel) evaluation of sequences of arbitrary and infinite length.
- Morpho has a module system that supports polymorphic modules. The module system is based on the ideas described in [1]. This corresponds loosely to Java generics, but is more powerful in some respects.
- Morpho uses run-time typing rather than compile-time typing. Variables in Morpho have no type, whereas values have types.
- Morpho has channels as a way of communicating and synchronizing.

2 Channels

Strategies for parallelization are intimately related to strategies for sharing information and passing information safely between system components. Strategies using features such as semaphores, locks and monitors are mainly intended for synchronization between components that share memory. Communication channels have a more general utility. The main reason channels were selected as the principal communication and synchronization mechanism in Morpho is to have the same method for supporting concurrency both within the same computer and between different computers on a network. Communication channels were developed by C.A.R. Hoare and others (see [11]) as part of their theory of communicating sequential processes (CSP). They were proposed as a programming language feature to facilitate parallel programming and communication. Pike's Newsqueak programming language implements channels [9]. Channels in CSP and in Newsqueak are synchronized. A receiver and a sender have a rendezvous to pass a value and we have, therefore, a notion of exact concurrency. In Morpho channels are not synchronized in this fashion because we want to use them not only within a single computer but also between computers on a network. Synchronized channels are not practical for network communication because network latency would make them much too slow. We do, however, have a notion of time-ordering since a receiver can only receive a value *after* the sender sends it. The syntax used in Morpho is similar to the one used in Newsqueak.

3 Advantages of Morpho

Morpho can be compared to other languages with small-grained parallelism features such as Scala, Erlang and Stackless Python.

3.1 Morpho and Scala

Morpho shares with Scala the feature of easy interoperability with Java. There are major differences, however, leading to some advantages offered by Morpho.

– Scala is compiled to Java bytecodes whereas Morpho is compiled to indirect threaded code (see [5,6]) that is interpreted by Java. Thus Morpho completely escapes serious limitations imposed by the architecture of the Java runtime, in particular the assumption that activation records are stored on a stack and that each thread has a separate stack.
– Activation records in Morpho are allocated on the heap, and more importantly in comparison to Scala the whole control chain of a running Morpho task is on the heap. This allows Morpho to support real lightweight thread-like tasks instead of simulated ones as in Scala. Scala offers two kinds of actors (see [7]). Both are comparable to a kind of combination of a Morpho channel and a Morpho task or fiber. The Scala actors are either thread-based or event-based.
  - The thread-based actors consume a lot of memory and the number of such actors is limited to some thousands on 32-bit hardware.
  - The event-based actors, on the other hand, are lightweight and it is possible to use hundreds of thousands or even millions of such actors "concurrently". But the event-based actors do not run separately but are rather implemented as closures that are "piggy-backed" on the threads of those communicating with them. They are therefore of more limited use than Morpho tasks in off-loading work to separate threads or CPU cores.

  A more minor point is that while Scala does support tail-recursion removal for single methods it does not support tail-recursion removal for mutually recursive methods. Morpho does full tail-recursion elimination, which is easily done due to the fact that activation records are stored on the heap. That is another consequence that demonstrates the extra power gained in the language run-time from heap-based activation records.

Compared to Morpho, Scala does not really offer lightweight threads but instead offers a limited capability to simulate such threads for message handling by chaining closures. Thus, the event-based actors of Scala do not directly leverage the additional capability offered by multi-core computers whereas the thread-based actors do not scale well due to their large memory footprint. A simple task creation benchmark in Scala (or Java) can generate and run about 5,000 threads on a 32-bit computer using default settings for the stack size in Java. A similar task generation benchmark in Morpho can generate and run millions of tasks before running out of memory.

3.2 Morpho and Erlang

Erlang does offer comparable parallel programming capability to that of Morpho, or even better, since Erlang supports easy parallelization of multiple computers on a network (see [4]) whereas the current implementation of Morpho has no such direct capability yet. Erlangs message passing is currently 3–4 times faster than that of Morpho as measured using the Erlang ring benchmark, implemented in Morpho for comparison. However, Morpho has some advantages that can be significant.

- Morpho interoperates directly with Java and Morpho programs can leverage functionality available in Java (and vice versa).
- Morpho supports imperative and object-oriented programming styles as well as functional programming.
- Morpho automatically supports any platform that has Java support.

Supporting streams as well as channels for synchronization in Morpho enables Morpho to be used in a functional programming style using both the parallel programming style supported by the strict (non-lazy) functional language Erlang (i.e. using channels) and the style promoted for the pure (lazy) functional language Haskell (i.e. using streams, see [8]).

3.3 Morpho and Stackless Python

Tasklets in Stackless Python correspond to Morpho fibers and offer very good scalability of parallelism for single processor machines. Stackless Python also supports channels for communication between tasklets. But Stackless Python currently does not support anything comparable to Morpho tasks. Thus Stackless Python does not leverage the additional computing power offered by recent multi-core processors.

4 The Implementation of Morpho

Morpho uses an object-oriented variant of indirect threaded code (see [5,6]). Each operation in the Morpho virtual machine is a Java object which has a method, `execute()` that executes the operation. A running Morpho task has a corresponding Java object that runs operations in an array of operations. The Morpho interpreter in its simplest form is the single line of code shown below, where `code` is the array of operations being executed, `pc` is the current program counter, and `this` is the current task.

```
    for(;;) code[pc++].execute(this);
```

The run-time overhead of interpreting the Morpho operations is comparatively small because the operations being executed are high-level ones and the dispatching of each operation is very simple, as seen above, but Morpho is slowed down a little by the fact that it is intended to be a high-level scripting language and therefore uses run-time typing rather than static or compile-time typing. Nevertheless, the performance of Morpho as measured by counting the number of messages passed through channels per second, is about the same as that of compiled Java code. And Morpho supports orders of magnitude more tasks and fibers than Java supports threads.

4.1 Morpho Tasks and Fibers

Morpho supports both tasks and fibers. Tasks are comparable in funtionality to Java threads in that they run concurrently and independently of each other. Each task contains a set of fibers (also called coroutines). Only one fiber in each task is running at any one time. This makes synchronization of fibers inside the same task quite easy. Because activation records in Morpho are stored on the heap it is trivial to suspend fibers by simply storing a pointer to their current activation records and related state. The state that needs to be saved is quite small, on the order of 20 bytes, but of course it mostly consists of pointers to existing data. Similarly, creating a new fiber or a new task in Morpho is a quite trivial and fast operation.

A set of tasks in a Morpho machine share a set of Java threads that take turns in executing the tasks. Typically the number of Morpho tasks is much larger than the number of Java threads servicing them. A task whose fibers are all blocked, for example by waiting on a channel (see below), releases its Java thread until it is ready again. This leads to a natural and efficient load-balancing between cores in multi-core machines. Tasks also regularly yield their threads even if not blocked.

One might think that storing activation records on the heap would slow down the run-time considerably. However, there is good analytical and empirical evidence that this strategy should not significanty reduce performance (see [2,3]). As memory sizes of computers become larger there is even reason to believe that this strategy could lead to faster systems than those relying on stack allocation for activation records.

5 Using Channels

- Channels are buffered and can contain at least one full value, making them asynchronous since a fiber or a task can write a value to a channel and then continue processing even if no other fiber or task has yet received the value.
- The expression `makeChannel()` creates a new channel and returns it.
- The expression `closeChannel(c)` closes a channel `c`, after which reads from the channel will eventually return a channel EOF.
- The expression `channelEOF()` returns the channel EOF value, which is a special value in Morpho.
- The expression `c<-e` writes the value `e` to the channel `c`. If the channel was in a writable state then this operation may succeed immediately and the executing fiber then immediately continues executing. Otherwise the executing fiber will wait until the channel becomes writable.
- The expression `<-c` reads a value from the channel `c` and returns it. If the channel is in a readable state (already contains a value) then this may succeed immediately, otherwise there is a wait, as in the above write operation.

A Morpho channel can be shared by multiple Morpho fibers and Java threads. Multiple channels can be monitored using a channel selector object.

The Morpho function in figure 1 reads all the values from a channel and returns their sum.

```
;;; Use:  s = sumChannel(c);
;;; Pre:  c is a channel that generates numbers.
;;; Post: All the numbers have been read from the channel and
;;;       s is their sum.
sumChannel =
  fun(c) {
    val eof = channelEOF();
    var sum = 0, x = <-c;
    while x != eof {
      sum = sum + x;
      x = <-c;
    };
    return sum;
  };
```

Fig. 1. Summing a channel

6 Locking with Channels

Since the channels in Morpho are not as synchronized as channels in CSP and Newsqueak we need a bit more work to use channels for locking in Morpho. The code in figure 2 shows a Morpho object definition of a simple lock object. A system component (normally a task) can acquire the lock and be certain that no one else has the lock and the same component should then release it once and only once. Only one component can possess the lock at the same time. Such locks are rarely (or perhaps never) needed in Morpho since the channels themselves are easier to use directly in solving concurrency problems.

7 Streaming Channels

Sometimes the imperative programming of channels becomes inconvenient and even confusing, for example when we need to navigate back and forth in the communication stream or when we need to use the same stream of values multiple times or in multiple places in the system. It then often is better to use a functional programming style to package a channel into a stream. Streams are functional programming constructs that stand for sequences of values. Streams may be infinite, if needed, and have the nature that a value in the stream is not evaluated until it is first used, whereupon it is memoized and not evaluated again upon further accesses.

```
;;; Use:  l = makeLock();
;;; Post: l is a new lock, in the free/unlocked state.
makeLock =
  obj() {
    var c = makeChannel();
    var tok;

    ;;; The channel c receives lock requests.  When
    ;;; locked, tok contains a channel that should
    ;;; eventually receive the lock release message.

    ;;; initialization:
    {
      ;;; Start an independent task as a guard.
      startTask(
        fun() {
          for(;;) {
            ;;; The lock is free/unlocked
            tok = <- c;     ;;; Wait for request
            ;;; A lock request tok has been received
            tok <- null;    ;;; Acknowledge request
            ;;; The lock is locked
            <- tok;         ;;; Wait for release
            ;;; The lock is free again
            tok = null;
          };
        }
      );
    };

    ;;; Use:  l.lock();
    ;;; Pre:  l is a lock.
    ;;; Post: l has been acquired.
    msg lock() {
      val req = makeChannel();
      c <- req;    ;;; Send lock request
      <- req;      ;;; Wait for acknowledgement
    };

    ;;; Use:  l.unlock();
    ;;; Pre:  l is an acquired lock.
    ;;; Post: l has been released.
    msg unlock() {
      tok <- null;  ;;; Send lock release
    };
  };
```

Fig. 2. Simulating a lock

- The expression `#[]` stands for the empty stream.
- The expression `#[ head $ tail ]` creates a stream whose first value (the head) is `head`, and whose tail (the rest of the values in the sequence) is the result from evaluating `tail`, which may be any expression. As with lazy streams in functional languages (see for example the pure functional Haskell programming language [8]), the tail expression is only evaluated if and when the tail is needed, so there is nothing to prevent us from creating infinite streams.
- The expression `streamHead(s)` returns the head of the stream `s`.
- The expression `streamTail(s)` returns the tail of the stream `s`.

Figure 3 shows how to transform a channel into a stream.

```
;;; Use:  s = streamOfChannel(c);
;;; Pre:  c is a channel that generates some finite or
;;;       infinite sequence of values.
;;; Post: s is a stream of the values generated by c.
streamOfChannel =
  fun(c) {
    val head = <-c,
    eof = channelEOF();
    if head == eof  { return #[]; }
    else { return #[head $ streamOfChannel(c)]; };
  };
```

Fig. 3. Transforming a channel into a stream

We can re-implement the `sumChannel` function as shown in figure 4. We can also transform streams into channels as shown in figure 5.

8 Scalability

Morpho tasks and fibers offer one to three orders of magnitude better parallel scalability than Java threads. A Morpho program can run well over a million concurrent tasks and fibers on a 32-bit personal computer, which is likely to be sufficient for the scalability needs of most current systems. On current 64-bit hardware this number has been verified to go up to tens of millions. The major reason for this improved scalability is that separate fibers and tasks in Morpho do not need separate run-time stacks. Activation records (sometimes called stack frames) in Morpho are not allocated on stacks but rather on the heap. This causes the memory footprints of fibers and tasks to be in proportion to the average length of the control chains instead of being in proportion to the maximum length of the control chains or worse. Also, because of Morpho's

```
;;; Use:  s = sumChannel2(c);
;;; Pre:  c is a channel that generates numbers.
;;; Post: All the numbers have been read from the channel
;;;       and s is their sum.
sumChannel2 =
  fun(c) {
    var s = streamOfChannel(c),
    sum = 0;
    while s != #[] {
      sum = sum + streamHead(s);
      s = streamTail(s);
    };
    return sum;
  };
```

Fig. 4. Summing a channel again

```
;;; Use:  c = channelOfStream(s);
;;; Pre:  s is a stream of values. s need not be finite.
;;; Post: c is a new channel that generates all the values
;;;       in s.
;;; Note: s is unchanged.
channelOfStream =
  fun(s) {
    val c = makeChannel();
    startFiber(
      fun() {
        while s!=#[] {
          c <- streamHead(s);
          s = streamTail(s);
        };
        closeChannel(c);
      }
    );
    return c;
  };
```

Fig. 5. Transforming a stream into a channel

automatic elimination of tail-recursion, the control chains of running fibers are likely to stay quite short, further reducing the memory footprint.

Morpho programs interface in a natural fashion, through channels, with the new event-driven IO features of Java (Java.nio, see [10]), which gives us the best of both worlds, the scalability of event-driven IO as well as the convenient imperative and functional programming styles of Morpho channels and streams.

Java.nio offers one to two orders of magnitude better scalability than regular socket programming in Java, with respect to the number of concurrent connections that can be served.

9 Benchmarks

A well-known benchmark, the ring benchmark, was used to compare the performance of Morpho with Erlang and Stackless Python. The code for Erlang and Stackless Python was found on the web (see [12]). The benchmark creates a chain of tasks connected by channels and passes messages through the chain. Once a chain has been created in Morpho we have two channels, `c0` and `cN`. Sending a message into `c0` will eventually result in the message coming out of `cN` having passed through the whole chain of tasks.

The measurements were done on a 32-bit dual-core Pentium running Windows Vista with 4 gigabytes of RAM. The ring benchmark was set up to use a chain of 10,000 tasks and to pass 10,000 messages through the chain. The elapsed time taken in Morpho was 3.7 times the time using Erlang (version V5.7.5) and 3.1 times that using Stackless Python (Python version 3.1.2, Stackless version 3.1b3 060516). The elapsed time durations using Erlang and Stackless Python were quite similar, in contrast to the measurements reported on the web page where the code for the Erlang and Stackless Python was found, where Stackless was reported to have $\frac{1}{12}$ the speed of Erlang. The reason for the discrepancy is unknown.

The Morpho ring benchmark was also run on a dual-core Windows machine with the added twist that each task was made to perform some CPU-time consuming computation on each message. Two different setups were used for this benchmark, one where both cores were made available and one where only one core was used. These measurements showed a 65%–75% performance increase by utilizing both cores. The same tests were run on a lightly loaded 32-core Linux computer and showed a speedup by a factor of over 13 times, which was the ratio between best throughput with all 32 cores used and best with one core used. The versions of Erlang and Stackless Python used for testing did not benefit from access to multiple cores. Stackless Python does not support multi-core processing but some versions of Erlang do.

The ring benchmark was also used to test how many Morpho tasks, Erlang processes and Stackless Python tasklets could be used on Windows. Erlang failed at 100,000 processes. While some Erlang implementations support more than 100,000 processes, other implementations use a 15-bit process identifier, which puts a hard upper limit on the number of Erlang processes. Morpho easily supported 1,000,000 tasks and Stackless Python had no problem with 1,000,000 tasklets. Stackless Python failed (barely) at 2,000,000 tasklets whereas Morpho succeeded (barely) with 2,000,000 tasks. Hence Morpho and Stackless Python seem very comparable in task/tasklet scalability. These tests were done on a 32-bit Windows Vista computer with 4 gigabytes of RAM. Furthermore, Morpho succeeded with a ring of 32,000,000 tasks on a 64-bit 32-core Linux computer with 128 gigabytes of RAM.

10 Conclusion

Morpho can be used with Java to improve scalability and programming convenience in massively parallel system development. The improvements are largely due to a huge reduction in the memory footprint of Morpho tasks and fibers relative to Java threads.

References

1. Agnarsson, S., Krishnamoorthy, M.S.: Towards a Theory of Packages. SIGPLAN Notices 20, 117–130 (1985), http://doi.acm.org/10.1145/17919.806833
2. Appel, A.W.: Garbage Collection Can Be Faster Than Stack Allocation. Inf. Process. Lett. 25(4), 275–279 (1987)
3. Appel, A.W., Shao, Z.: An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. Journal of Functional Programming 6 (1994)
4. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
5. Bell, J.R.: Threaded code. Commun. ACM 16(6), 370–372 (1973)
6. Dewar, R.B.K.: Indirect threaded code. Commun. ACM 18(6), 330–331 (1975)
7. Haller, P., Odersky, M.: Actors that Unify Threads and Events. In: Murphy, A.L., Ryan, M. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 171–190. Springer, Heidelberg (2007), http://lamp.epfl.ch/~phaller/doc/haller07coord.pdf
8. Peyton-Jones, S.: Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Engineering Theories of Software Construction, pp. 47–96. Press (2002)
9. Pike, R.: The Implementation of Newsqueak. Softw., Pract. Exper. 20(7), 649–659 (1990)
10. Pugh, W., Spacco, J.: MPJava: High-Performance Message Passing in Java using Java.nio. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 323–339. Springer, Heidelberg (2004)
11. Roscoe, A.W., Brookes, S.D., Hoare, C.A.R.: A Theory of Communicating Sequential Processes. Journal of the ACM (3), 560–599 (1984), http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/4.pdf
12. Stawarz, C.: Erlang vs. stackless vs. multitask: The ring benchmark showdown (September 2007), http://pseudogreen.org/blog/erlang_vs_stackless_vs_multitask.html (last accessed: December 3, 2010)

Extending Distributed Shared Memory for the Cell Broadband Engine to a Channel Model

Kenneth Skovhede, Morten N. Larsen, and Brian Vinter

Department of Computer Science, University of Copenhagen,
Universitetsparken 5, DK-2100 Copenhagen, Denmark
{skovhede,momi,vinter}@diku.dk
http://www.diku.dk

Abstract. As the performance gains from traditional processors decline, alternative processor designs are becoming available. One such processor is the CELL-BE processor, which theoretically can deliver a sustained performance close to 205 GFLOPS per processor[10]. Unfortunately, the high performance comes at the price of a quite complex programming model. In this paper we present an easy-to-use, CSP-like, communication method, which enables transfers of shared memory objects. The channel based communication method can significantly reduce the complexity of massively parallel programs. By implementing a few scientific computational cores we show that performance and scalability of the system is acceptable for most problems.

Keywords: CSP, CELL-BE, DSMCBE, channel communication.

1 Introduction

All current computers are fundamentally based on a Von Neumann architecture, and all suffer from the "Von Neumann bottleneck"[1] which boils down to the limited speed of transfers between the memory and the processor. The bottleneck problem was defined by John Backus in 1977, and so far no solutions have been provided to totally eliminate it, only approaches to help hide it.

The CELL-BE processor attempts to overcome the "Von Neumann bottleneck" using highly specialized processors, controlled by a more conventional processor core. Conceptually the CELL-BE is not that different from a small cluster-computer on a chip. Each of the specialized processors run their own program, has their own small memory and communication among the processors are explicit. The processor cores exchange information through reading and writing main memory, but with DMA (direct memory access) operations more similar to IO than memory access. From the perspective of the specialized cores, the main memory is more like a shared storage in a cluster than a conventional main memory. The CELL-BE processor is a heterogeneous multi-core processor consisting of nine cores. The primary core is an IBM 64 bit power processor (PPC64) with two hardware threads, which is linking the operating system and the eight powerful working cores, called the synergistic processing elements

(SPEs). The PPC and the eight SPEs are connected trough a 204.8 GB/s Element Interconnect Bus (EIB)[5]. The computing power of a CELL-BE chip is well investigated[13,9], and a single CELL blade with two CELL-BE processors is reported to yield 460 GFLOPS. This is achieved at a rate of one GFLOPS per second per watt[8].

As the CELL-BE architecture can be viewed as a cluster on a chip with eight nodes, SPEs, and a PPE front-end, splitting an application into smaller tasks are done in the same way as it would be done with traditional clusters. However, due to the limited amount of local storage (LS) on the SPE units available for both code and data, 256 KB, one has to consider the size of each task. This means that an application which would be best parallelized by a bag-of-task model in a cluster setup, might be best parallelized by a pipelined set-up on the CELL-BE. If it does not fit into the LS, it might be useful to split it up into two, four or eight pipelined stages depending on the size of the code.

The DSMCBE[11] system helps address this exact problem. With named memory regions, each SPE can obtain a copy, possibly exclusive, of a given memory region. An operation that also works as a coordination mechanism. That is, the process may choose to wait until the region is obtained before it continues. The technique is simple since the only difference between mutex programming and DSMCBE is that the locking also migrates the data into the active LS.

The simplicity and flexibility of the DSMCBE model means that it not only works within a single address space for a CELL-BE system but also allows inclusion of SPEs on other CELL-BE processors in a cluster, LAN or even WAN environment. Making the DSMCBE system an integrated distributed shared memory system as well as a system for programming CELL-BE processors.

By using DSMCBE, the challenges with the CELL-BE becomes easier to overcome, but it does not lead to flawless concurrent programs. Developers must still be very aware of dead-locks, live-locks and race conditions, which most of the times are very hard to discover at compile time and even at runtime. To overcome these problems C.A.R. Hoare introduced the CSP (Communicating Sequential Processes) data model in 1978[7]. This model is using explicit communication through well defined channels and a concept of processes, each with their own set of private variables. A CSP process can, from a programmers perspective, be seen as a sequential program. Finally it is possible to mathematically prove that a program is free of deadlocks and livelocks[6]. CSP has been implemented on multiple architectures and in many programming languages e.g., occam, C++CSP[12], JCSP[14] and PyCSP[3], but not yet on the CELL-BE. This paper describes how to utilize the functionality of DSMCBE to make CSP communication channels for the CELL-BE.

2 Work

One essential property of CSP, is the ability to move data to the processes that require it. As we have previously made a DSM system for the CELL-BE, we have used that as a base for the CSP channel implementation.

2.1 DSMCBE

DSMCBE is an usermode library that offers a simple API for working with shared memory regions. The user explicitly calls the three functions `create`, `acquire` and `release`. Using these three functions, DSMCBE can implement release consistency, which makes it possible for programs to share memory regions reliably. The underlying transport of data is completely hidden from the usercode, giving the programmer the illusion that memory is shared. Performance and scalability are shown to be quite good for most computational problems[11].

The DSMCBE library consist of several elements. The most central element is a single processing thread called the request-coordinator. Most of the other elements of DSMCBE can communicate with the request-coordinator by supplying a target for the answer. Using this single thread approach makes it simple to execute atomic operations and reduce the total number of locks required. The request-coordinator cannot determine if a participant is a PPC, SPE or the network[1]. The main drawback for this simple single thread design is that the thread may become a bottleneck.

Figure 1 shows the logical parts of the DSMCBE library.

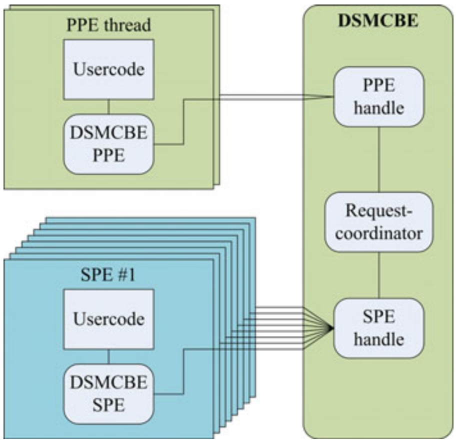


Fig. 1. DSMCBE internal structure

2.2 Channels

Using a DSM system as the base for a CSP channel implementation makes the data transfer operations simple, as they can use the DSM system directly. As CSP channels are blocking, the DSM system must support blocking in some way. Even though the DSMCBE system supports blocking multiple writers, it

[1] The network component is not shown in figure 1.

does not support resizing or deleting of shared regions. Therefore we have implemented a blocking create option to DSMCBE, so that multiple create calls are blocked until the shared region can be re-created. To ensure that the create calls can be executed we have also implemented a delete operation that works as an `acquire` call in write mode, but marks the object as deleted. This also enables the processes to re-create the object with a different size. Using the blocking `create` and `acquire-delete` calls, it is possible to implement a simple CSP channel. Since it is possible to create a shared region without a pending `acquire`, it is essentially a channel with a buffer size of one. This means that it is not possible for the programmer to define a unbuffered channel.

Figure 2 shows the DSMCBE library extended with the channel wrapper.

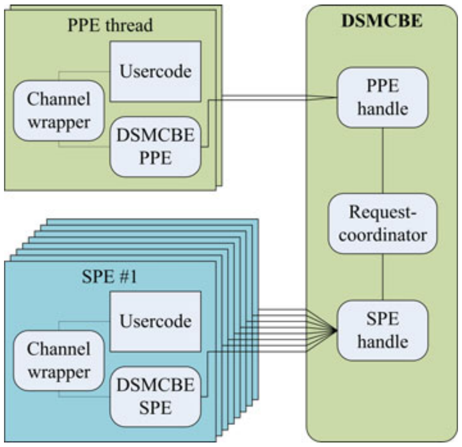


Fig. 2. DSMCBE with channel wrapper internal structure

2.3 Implementation

To support a channel based communication, we have modified the `create` call for the DSMCBE system to accept a blocking strategy for `create` call to items that already exist. To ensure that the `get` function call is atomic, we have implemented a `delete` function that removes and returns an existing item. As the channel model means that the `get` call is expected to block, the `delete` call also blocks until an item is created.

Rather than implement specific control structures for the `get` and `put` requests, they are merely implemented as simple wrapper functions that call the relevant version of the `delete` or `create` request respectively. As the `create` and `delete` calls are ordinary DSMCBE calls, they are actually completely asynchronous, simply forwarding a control structure. To adhere to the expected behavior of channels, the `get` and `put` operations will immediately block until a response is received, thus appearing to be normal synchronous calls.

As shown in figure 1 all requests from a SPU is forwarded to a *SPU handler* on the PPE, which is done using mailboxes. The *SPU handler* then forwards this request to the *request-coordinator*, which is in charge of maintaining the state and request queue for each item. When a request can be serviced, the *request-coordinator* will send a response back to the *SPU handler*, which will initiate DMA requests and ultimately respond to the SPU.

Internally in the DSMCBE model, each request is recorded in a simple FIFO queue, making most operations run in constant time. This ensures that the number of pending `put` requests do not affect the execution time.

3 Results

Using the simple channel communication system, we have implemented a few experiments, that show how well the system performs and scales. For each application there are some peculiarities that stem from the special hardware construction that is found in the CELL-BE. To establish a realistic scenario for the intended library usage, we have chosen one CSP experiment and two representative scientific computational cores. The first experiment is the CommsTime which is widely used to measure the overhead in CSP implementations. The second experiment (Prototein folding) is used to measure the implementations capability of scaling. The Prototein folding application is a basic bag-of-tasks solution that can be classified as an embarrassingly parallel application. The final experiment is the Heat Equation application which has an entirely different type of communication pattern and is an instance of the successive over-relaxation solutions. This experiment is used to see how the implementation reacts on heavy communication. The code for all experiments presented, as well as the code for the DSMCBE and CSP model can be found on the DSMCBE website `http://code.google.com/p/dsmcbe`

3.1 CommsTime

To measure the overhead of using the implemented CSP channels, we have chosen CommsTime which is a well-known method to benchmark CSP systems. CommsTime is used to compute the cost of a single channel communication operation, and thereby it is possible to compare this implementation against other CSP implementations.

As the CELL-BE processor has the eight available SPEs, we have executed CommsTime with two to eight SPEs participating. The first SPE will run the delta process that forwards the message and outputs the clock signal. The second SPE will run the prefix process and inject the value to send around. Any additional SPEs will run a delta process with one output, and just forward the message. Each additional SPE will add a communication channel. The PPE reads the clock signal from a channel and measures the time between each clock signal. Figure 3 shows the conceptual setup with four participating SPEs.

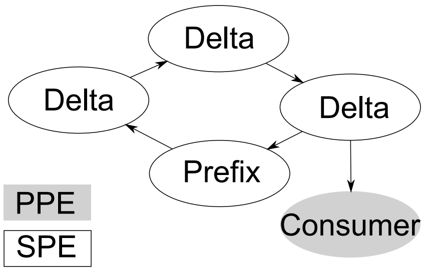


Fig. 3. CommsTime conceptual diagram with four SPEs

Figure 4 shows how the CommsTime application performs and shows that the CommsTime improve after two SPEs. This happens because the first SPE outputs the delta signal, which means that it has two channels, where any additional process has only one outbound channel. After this initial stage, the times are fairly constant, with a slight increase in time. This happens because the actual computation time on each SPE is very limited, and each SPE thus awaits a PPU service most of the time. As there is only one PPU thread to service the SPEs, the time between the service calls increase with the number of SPEs. To remove bottleneck in the PPE from the system requires a change in the DSMCBE. To help remove this bottleneck, one could try to use multiple threads in the DSMCBE *SPE handler* module. This could improved the performance because the SPE handler is in a spin-loop the most of the time. However, multiple threads also requires more synchronization which do not improved performance. A better solution would be to remove the spin-lock, but this is not possible due to the hardware structure of the communication between the PPE and SPEs.

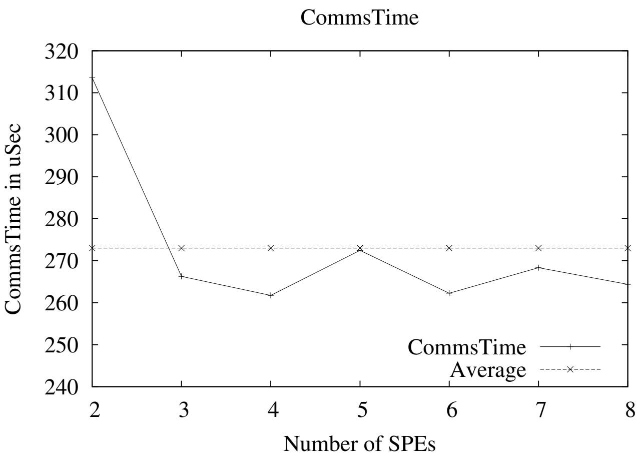


Fig. 4. CommsTime performance using one to eight SPEs

A peculiar effect of the test is that the CommsTime problem runs a little slower when then number of participating processes are odd. We expect that this is due to a transfer aliasing effect, and is subject of further investigation.

Table 1 shows the CommsTime measure for several systems, and reveals that the communication time is quite high compared to other CSP implementations. This large communication time is mainly caused by the physically separated memory blocks that require memory transfers. The number of messages passed internally in the DSM system is also larger than the required minimum. This is because the DSMCBE system supports multiple readers, which is not used in the channel communication scenario.

Table 1. Performance of one CSP channel communication using several CSP frameworks [4]

CSP Framework	Time per iterations (microSec)
CSP for CELL-BE (avg.)	273
OCCAM (KRoC 1.3.3)	1.3
C++CSP	5.0
JCSP (JDK 1.4)	230

One important difference between the CELL-BE CSP channel implementation and the other implementations, is that for the other implementations, the measured overhead is execution time on the processor that runs both user code and CSP library code. In the CELL-BE implementation this is not the case, as the CSP user code runs on the SPE units and the library code runs (mostly) on the PPU. This means that even though the overhead is large compared to the other implementations, the overhead runs in parallel with the user code, and will thus be hidden in many real-life applications.

3.2 Prototein

A prototein is a model of a protein that only contains two amino acids and only folds in 90 degree multiples. Folding a large prototein is a computationally intensive task, but is embarrassingly parallel since the subtasks have no interdependencies. We have implemented a single channel to dispense the subtasks, making it a bag-of-tasks type of implementation, with a single writer and multiple readers. The PPU is responsible for the initial problem division, and writes partially folded prototeins into the channel. Each participating SPU reads the partially folded prototeins from the same channel and completes the fold. Each SPU reports the best possible fold back to the PPE, which then picks the overall best fold. Figure 5 shows a conceptual setup of the communication pattern in the prototein folding.

As seen in figure 6, the prototein problem scales close to perfectly. The channel based implementation is slightly faster than the DSM implementation, because the bag(-of-tasks) is a shared object in the DSM model, where this is handled by

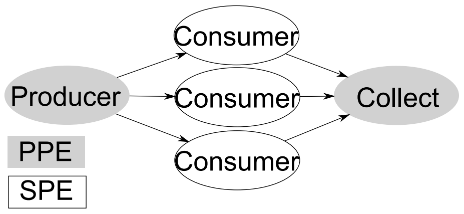


Fig. 5. Prototein conceptual diagram with three SPEs

blocking the requests in the channel version. This also means that the channel based model scales marginally better. The PPU service problem is not as present as in the CommsTime test because the SPEs actually do some computation and do not require the PPU service as often.

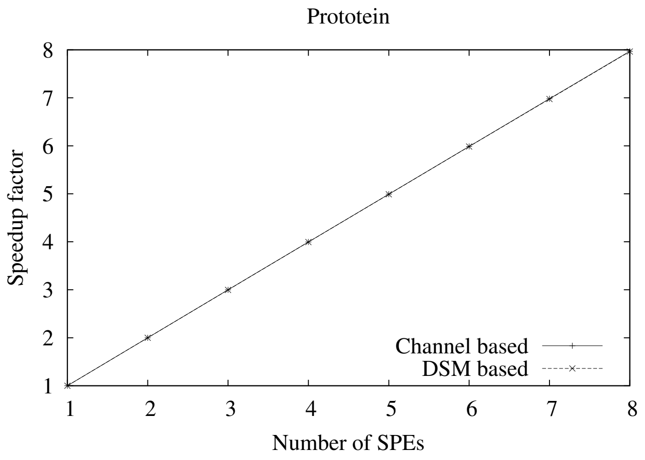


Fig. 6. Prototein folding has near optimal performance scaling

3.3 Successive over-Relaxation

In the final experiment we have implemented an example of successive over-relaxation (SOR) where a large block is initialized with a temperature of zero in the middle and -273.15 on the sides. The temperatures movement in the block is then simulated by successive over-relaxation in discrete time steps. After 1000 iterations, the simulation is stopped.

Each SPU is responsible for handling a fragment of the total simulated area, which is managed with double buffered transfers through the DSMCBE system. The shared boundaries between the fragments are coordinated through a channel for each SPU pair. The first and the last SPU has a single channel, where all others have two channels, one for the upper shared boundary, and one for the lower. Each half iteration consists of applying a SOR for one half of the points, followed by an exchange of boundaries. A full iteration is performed with repeating the half-iteration twice, each with a different half of the points. Figure 7 shows the conceptual communication pattern for the SOR sample application, and illustrates the lock-step setup.

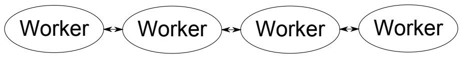


Fig. 7. HeatEquation conceptual diagram visualizing worker dependence

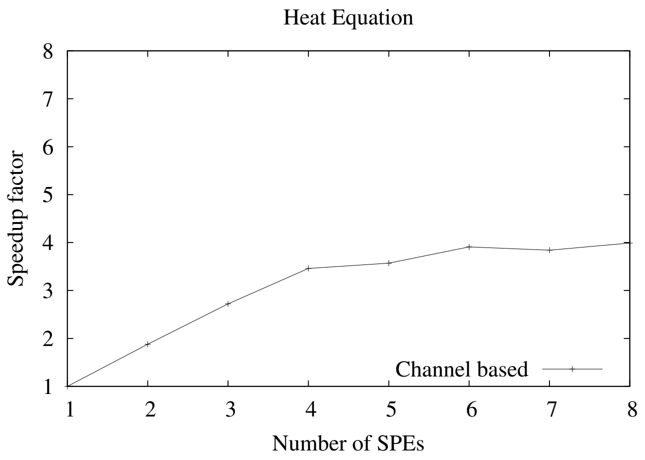


Fig. 8. HeatEquation suffers from contention on PPU resources

Figure 8 shows how the CSP model scales with one to eight SPE units, utilizing two PPU threads. As can be seen the problem does not scale well beyond four SPEs. The hard time constraints inherent in the problem, means that before each round can begin, all the processes must synchronize. This results in the slowest process preventing faster processes from completing.

Since the objects that we simulate are so large that they cannot fully fit in memory on the SPE units, we use the DSMCBE system to load memory regions onto the SPE units. This loading uses the same system as the channel communication, which naturally influences the measured times.

The overall problem with scalability in this example is the amount of service required from the PPU. Even though the SPE units handle the actual computation, they rely on the PPU for transferring both shared and private blocks. When the number of participating SPEs increase, so does the number of requests to the PPE. As a single slow SPE blocks all other SPEs, any delay in handling the requests propagate, resulting in the scalability problem seen in figure 8.

4 Conclusion

The absence of race-conditions in a channel based communication pattern, significantly reduces the program complexity. The reduced complexity comes at the expense of a drop in performance. Even though the overhead in the channel model is fairly high, the scalability is quite good. The scalability combined with the massive processing power offered by the SPE units, makes the library attractive for a number of tasks such as protein folding.

Clearly problems that are computationally intensive but have low memory requirements scale very well on the CELL-BE. When the problems have large memory requirements, the data transfers become the bottleneck. As DSMCBE is capable of performing asynchronous transfers, the transfer times can be hidden by performing overlapped execution. Hopefully overlapped execution can be exploited when buffered channels are implemented. There exists a boundary for the smallest number of computations done on a single byte, before the bytes transfer time can be hidden. What this boundary is has not been examined, but is excepted to be around 20-30 floating point operations per byte.

All work on DSMCBE and the CSP implementation, including the experiments, is released as open source under the LGPL license, and is available from the website: `http://code.google.com/p/dsmcbe`

5 Future Work

The underlying DSM system is optimized for multiple readers and multiple writers, which means that the object is recorded and managed in multiple places. Since channel objects can only exist in one place, there is a large potential for optimization. Once the basic channel system is in place, it would be desirable to implement a full CSP model with alternating channels. To ease the use of the channels it would be desirable to use a common standard for channels e.g., the one used in PyCSP, JCSP.

With CSP channels for the CELL-BE the next step is to make CSP processes. With the DSMCBE functionality to transfer data among the CELL-BEs, it would make good sense to use DSMCBE to transfer the CSP processes (user code). Once the code has arrived at the designated execution unit it can be executed. Some work relating to relocation of processes has already been done, but making a "real" thread library for the SPEs is not a trivial task. The fact that the SPEs do not have more then 256KB of LS, means it is possible that threading is not a feasible solution when we want many (100+) threads. Some work have however been done in this field[2], but at the time being it only supports a small number of threads.

Acknowledgments. The authors acknowledge the Danish National Advanced Technology Foundation (grant number 09-067060) and the innovation consortium (grant number 09-052139) for supporting this research project. Furthermore the authors acknowledged Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

References

1. Backus, J.: Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. Commun. ACM 21(8), 613–641 (1978)
2. Beltran, V., Carrera, D., Torres, J., Ayguade, E.: CellMT: A Cooperative Multithreading Library for the Cell/B.E. In: The Proceedings of the 16th Annual IEEE International Conference on High Performance Computing, HiPC 2009 (December 2009)
3. Bjørndalen, J.M., Vinter, B., Anshus, O.: PyCSP - Communicating Sequential Processes for Python
4. Brown, N., Welch, P.: An Introduction to the Kent C++CSP Library. Slides
5. Chen, T.: Cell Broadband Engine Architecture and its first implementation - A Performance View (2005), http://www.ibm.com/developerworks/power/library/pa-cellperf/ (accessed July 26, 2010)
6. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (1978)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
8. IBM: IBM doubles down on cell blade (2007), http://www-03.ibm.com/press/us/en/pressrelease/22258.wss (accessed July 26, 2010)
9. Jowkar, M.: Exploring the Potential of the Cell Processor for High Performance Computing (2007), http://www.diku.dk/~rehr/cell/docs/mohammad_jowkar_thesis.pdf (accessed July 26, 2010)
10. Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., Dongarra, J.: Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (2006), http://www.netlib.org/lapack/lawnspdf/lawn175.pdf (accessed March 29, 2010)
11. Larsen, M.N., Skovhede, K., Vinter, B.: Distributed Shared Memory for the Cell Broadband Engine (DSMCBE). In: International Symposium on Parallel and Distributed Computing, pp. 121–124 (2009)
12. Mcewan, A.A., Schneider, S., Ifill, W., Welch, P., Brown, N.: C++CSP2: A Many-to-Many Threading Model for Multicore Architectures (2007)
13. Rehr, M.: Application Porting and Tuning on the Cell-BE Processor (2008), http://dk.migrid.org/public/doc/published_papers/nqueens.pdf (accessed July 26, 2010)
14. Schaller, N.C., Hilderink, G.H., Welch, P.H.: Using Java for Parallel Computing - JCSP versus CTJ. In: Communicating Process Architectures 2000, pp. 205–226 (2000)

Global Asynchronous Parallel Program Control for Multicore Processors

Janusz Borkowski[1], Marek Tudruj[1,2], Adam Smyk[1], and Damian Kopanski[1]

[1] Polish-Japanese Institute of Information Technology,
86 Koszykowa Str., 02-008 Warsaw, Poland
[2] Institute of Computer Science, Polish Academy of Sciences,
21 Ordona Str., 01-237 Warsaw, Poland
{janb,tudruj,asmyk,damian}@pjwstk.edu.pl

Abstract. The paper discusses implementation issues of a distributed program design tool based on monitoring of application global states. It is shown how the experience from the PS-GRADE parallel program design tool with controlling distributed programs at the process level, based on the extensive use of signals, can be transferred at the level of threads. A programming technique is proposed to combine the use of process level communication libraries like MPI or sockets with the use of thread level libraries like OpenMP or pthreads. It enables designing a graphical parallel program development framework which uses signals at the level of distributed threads executed in multiple cores of processors. Viable implementation of global state monitoring and involved control data transmissions at the level of threads are discussed.

Keywords: parallel program design tools, distributed program execution control, global program states monitoring, multithreading technique.

1 Introduction

The paper is concerned with the implementation of a novel control paradigm in distributed program design tools. This new control design paradigm and the underlying program design tool extend current practice and assume constructing program execution control based on evaluation of predicates on application global states [5,8,9]. Control predicates implement a generalized synchronization of distributed program elements and are used to generate global control signals which influence execution of parallel processes and threads in programs.

The idea of global control of processes in parallel and distributed programs has been considered in literature and has some implementations. A classic example is the Linda environment [1] which uses a common global tuple space and simple primitives to write and read to/from it. Another approach is present in coordination languages. Manifold and Reo [2] enabled binding software components into coordinated structures. Components interacted with neighboring components using channels, however without a notion of a global state introduced. Global states are used as foundations of the Meta system [3] meant to glue together the components of distributed programs. In Meta, the processes were able to report their states through so called sensors, the states were combined to form consistent global states on which global predicates were analyzed. A complicated notation based on guards is used to program a desired behavior. There is a try to simplify this notation in the Lomita language [3]. Lomita has proved to be inefficient due to very costly ordered broadcasts used to guarantee proper message order delivery. Global control constructs for the OCCAM language are proposed in [4] based on replication of state variables.

A complete environment for designing program execution control based on predicates computed on global application states was implemented as the PS-GRADE framework [10,12]. PS-GRADE offers a graphical user interface taken from [11] and includes a system infrastructure which enables programmers to design the global program control based on application states monitoring. The control is de-coupled from a program computational code and located in special processes called synchronizers. PS-GRADE synchronizers collect local state information from processes, construct global consistent application states and evaluate on them control predicates. The predicates represented control conditions, which if fulfilled, asynchronously influence process behavior. To achieve such asynchronous control, specific signals were dispatched from synchronizers to application processes. Such a methodology improves standard program practice where the code responsible for synchronization and control is usually scattered in the program code in a chaotic unstructured way. PS-GRADE was built for single core processors and no global control means at thread level have been embedded in this system.

This paper discusses implementation issues of a new program design tool which aims at the design of distributed programs execution control based on monitoring of global application states both at the process and thread levels. It requires new programming techniques to comply with process level communication libraries like MPI or sockets and thread level libraries like OpenMP or pthreads in programs executed in manycore processors. The new design framework additionally extends the global control in distributed programs towards high level control constructs at the level of processes, governed by predicates evaluated on global application states.

The paper presents an analysis of viable implementation methods of involved control at the level of processes and threads for global control in parallel programs. The paper is composed of three parts. In the first part, program global control principles based on predicates computed on global application states are explained. The second part deals with the program execution control based on Unix signals in multithreaded environments. The third part presents an example of an application design based on the proposed program execution control development framework.

2 Global Predicate-Based Program Control

The parallel program control method based on global application state monitoring implemented in the PS-GRADE system was meant for execution on a distributed system with partially synchronized processor clocks. Special synchronizer processes collect messages about local states of application processes accompanied by the timestamps measured with a known accuracy of clock synchronization, construct consistent or observed global (or regional) states [6,7] and evaluate control predicates on them. A synchronizer is connected with, observed and controlled process by message passing channels, see Fig. 1(left).

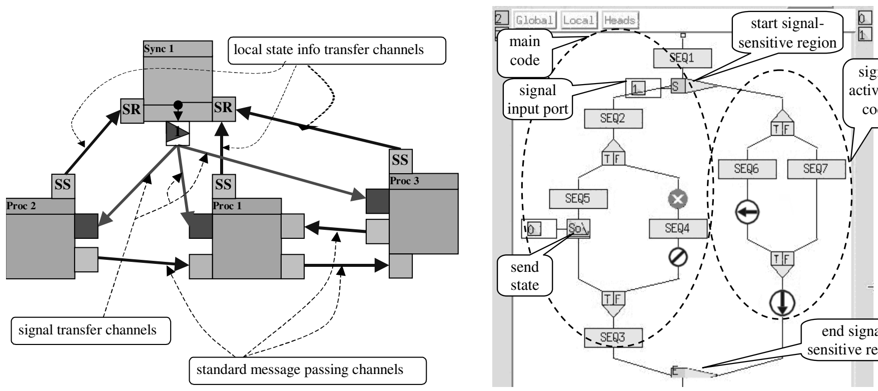


Fig. 1. A simple PS-GRADE application (left), a process control flow in PS-GRADE (right)

Three lower boxes represent application processes, which can be run on a separate computer. The arrows depict message passing channels (for data exchange between processes) and communication channels with the synchronizer. A programmer defines control predicates as required for a particular application. If a predicate is satisfied, control signals are dispatched by a synchronizer to processes, causing activation of the code associated with a given signal, or canceling current computations and omitting a section of the program code. An example of a PS-GRADE process control flow diagram is shown in Fig. 1(right). SEQ boxes contain sequential C code, TF branches represent conditional execution, other instructions like loops, send, receive are represented by other pictograms. The control flow is divided between the main branch, which is executed normally, and a number (one on the example) of branches activated by a signal reception.

Inter-node communication from synchronizers has been implemented through message passing, using messages containing information about signal type, signal parameters and the receiver. Dedicated processes called "dispatchers" were responsible at each node for local signal handling, Fig. 2. Dispatchers translated messages to Unix-type signals, which were the only available technical means to implement asynchronous reactions. The use of real–time signals dispatched with the sigqueue() call let us associate a parameter with a signal. The parameter has been used as a pointer to a structure containing all parameters required by the activated code and obtained from a synchronizer. Code execution cancellation used also UNIX signals.

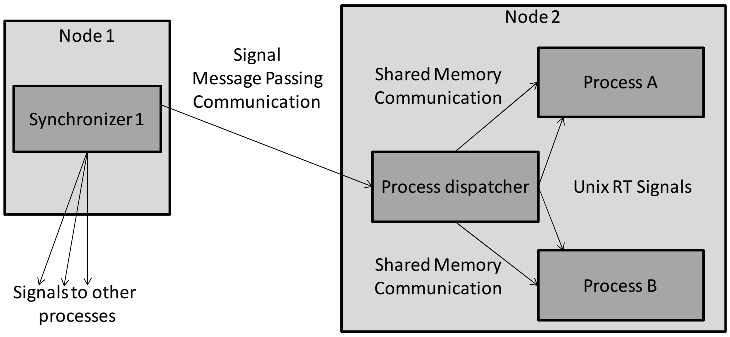

Node 1
Synchronizer 1
Signals to other processes
Signal Message Passing Communication
Node 2
Shared Memory Communication
Process A
Process dispatcher
Unix RT Signals
Shared Memory Communication
Process B

Fig. 2. Asynchronous signal transmission is PS-GRADE

Based on the PS-GRADE experience, we are currently building a new programming framework in which the program execution control based on global application states monitoring has been extended to influence the flow of control at the global level of processes and the internal behavior of threads. We introduce an additional graphical program specification level where we define a global control flow graph of a program. The control flow graph contains program code blocks, control flow switch blocks, global synchronization blocks and predicates blocks that participate in the design of the global control in the program. The blocks are interconnected by control flow edges to represent graphical simple or replicated control flow constructs. At this level, an image of block control interactions is specified by means of a list of predicate arguments in terms of graph blocks and a list of signal targets, separated by a semi-colon. Inter-block edges, represent the flow of control signals generated by predicate blocks. To simplify program graphs, transfers of local block states are not represented graphically at this level.

An exemplary simple control flow graph in the extended program design framework is shown in Fig. 3 (left). Here we have n program blocks (P1,..., Pn) embedded in a PARALLEL WHILE DO construct. The flow of control in the loop is determined by a global predicate GP1. It is evaluated on the basis of a global state built of local states of B1,..., Bm program blocks (processes or threads), included in the control flow diagram of the program. GP1 sends control signals to the switch SW which directs the flow of control in the graph. Actions of the switch stimulate actions of a hidden Execution Control (EC) system process, which co-ordinates execution and manages process creation, activation and synchronization. In Fig. 3 (left) actions of EC correspond to parallel control constructs: parallelize (PAR) and synchronize all blocks completion (JOIN). The internal behavior of processes P1,..., Pn is asynchronously controlled by the global predicate GP2 based on a global state composed of local states of P1,..., Pn in the similar way as in PS-GRADE. Nested control structures are allowed in the graph. By opening a new graphical window a programmer can insert a high level control construct into an existing program block. Ports at the block boundaries are generated to enable graphical design of a consistent flow of control.

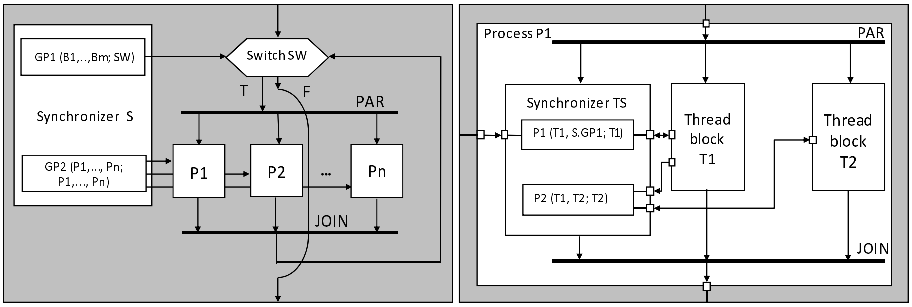


Fig. 3. PARALLEL WHILE DO construct with a control flow predicate GP1 and an asynchronous control predicate GP2 (left), process P1 internal window (right)

Terminal program blocks, which do not contain nested high level control constructs, represent application processes. The processes are next composed of parallel thread blocks and thread level synchronizers using a lower level graphical window, Fig. 3 (right). The blocks in this window are next specified using the C/C++ language and the pthreads, OpenMP and MPI libraries in special text editing windows. At this level, a programmer can define asynchronous global thread execution control based on thread level synchronizers. The synchronizers are implemented as special threads which collect state information from standard application threads, determine control predicates on constructed global thread states and send signals to threads to stimulate asynchronous reactions. For thread assignment to processor cores inside thread blocks the OpenMP thread_affinity and pthreads_affinity facilities are used [13].

Global control predicates used in program execution control, can be extensively applied to optimized scientific computations. An important problem in scientific computations is run-time optimization of the use of computing system resources. An important kind of such optimization is load balancing of processors in clusters, which leads to heuristic reduction of the total program execution time. The infrastructure for the proposed control method – application state reporting, global state analysis, control signal dispatching, asynchronous reaction to signals – suites very well implementation of load balancing in scientific computations. A user needs to define a processor load measure to be sent as the current process state, predicates to analyze the load imbalance to take some load control decisions and finally the code activated by control signals which works for the elimination of loading imbalance by transferring load to less charged processors or migrating parts of the application code. Each of the mentioned components can be easily adapted to specific scientific computing problem needs by programming the synchronizer-related fragments of the code. It enables flexible user-defined load balancing in parallel scientific applications.

Very frequently scientific computations show irregular parallelism features, where the irregularity is caused by dynamic nature of control and data dependencies between parallel tasks. As an effect, the processes may need to frequently exchange values of calculated parameters using an unpredictable and changing communication pattern. They may need to perform dynamic load balancing to maintain good efficiency. They may also need to stop and resume often, depending on the state of other processes. In a classical design, a programmer has to program all the control code and logic from scratch, and deal with a plethora of messages, some of them having control meaning, some carrying data. In our approach, a programmer gets a complete global infrastructure, on top of which the control strategy suitable for a particular irregular problem can be realized. By making an application global state available for a programmer we facilitate the use of such control. Current dynamic dependencies between processes can be easily analyzed, as they are reflected in process states included in the global states. Then, control signals and the code activation/cancellation mechanism enable an easy creation of sophisticated control patterns. The efficiency of the proposed control method for irregular numerical computations was shown in [12], by examples of a branch and bound computation for the Travelling Salesman Problem and an adaptive numerical integration. The obtained parallel speedup was generally not worse, and mostly, considerably better than that without the proposed control method.

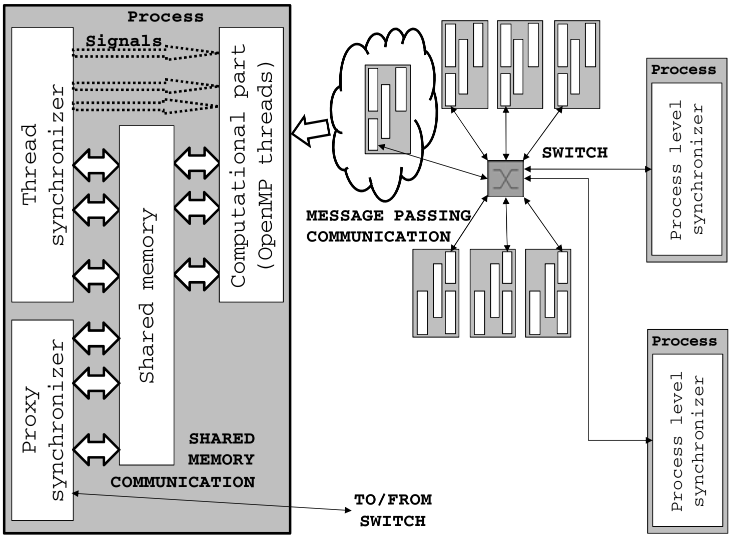

Process
Signals
Thread synchronizer
Shared memory
Computational part (OpenMP threads)
Proxy synchronizer
SHARED MEMORY COMMUNICATION
TO/FROM SWITCH
MESSAGE PASSING COMMUNICATION
SWITCH
Process
Process level synchronizer
Process
Process level synchronizer

Fig. 4. Control communication for global states synchronization

3 Signal-Based Asynchronous Global Thread Execution Control

Communication involved in distributed program execution control governed by global application states monitoring includes control data communication at process level and at thread level, Fig. 4. A process usually contains a number of parallel computational thread blocks and some thread synchronizers used to control process threads. To implement asynchronous global control at the level of threads we have used shared memory communication and Unix signals mechanism. Thread synchronizers collect state information from computational threads via shared memory, evaluate control predicates and send signals to threads to modify their behaviour. Thread synchronizers can also send state information to - and to receive signals from synchronizers in other processes. For processes allocated to different processor nodes this control communication is done by message passing using the MPI library. All inter-processor transfers initiated by thread synchronizers are delegated to separate shadow proxy synchronizers, which exchange control data (representing states or signals), taken from the process shared memory, by MPI message passing. All message transfer requests generated by thread synchronizers are collected in a queue, which is inspected and serviced in MPI by a proxy synchronizer. One inconvenience is that all control information messages are serialized in a proxy synchronizer, however, several proxy synchronizers can be organized in parallel if network resources allow.

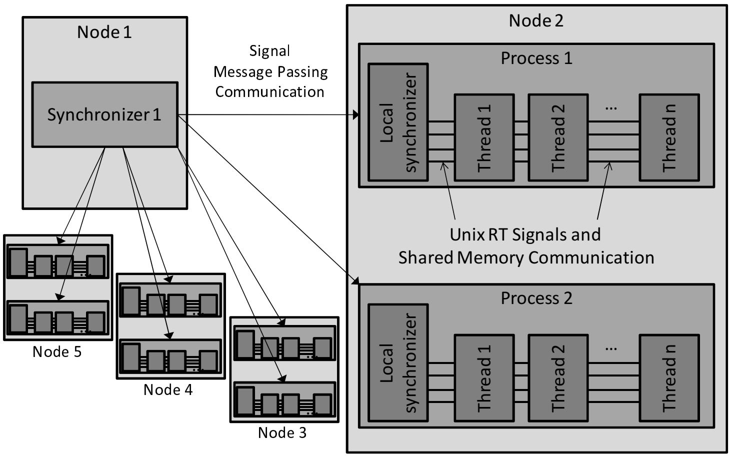


Fig. 5. Asynchronous signal transmission with presence of threads

Signal transmission from a thread synchronizer to a dedicated thread was a separate problem to be solved. We wanted to follow the concepts from the PS-GRADE environment (see Fig. 2). From a number of thread signal types in Linux, we decided to use real-time signals, because delivery of such a signal is guaranteed and it can carry a pointer of a data structure with parameters necessary to interpret the signal. We have examined the system level threads and pthread libraries, but, even with the use of system level functions, it was not obvious how to send a UNIX real-time signal to a specific system thread (pthread_kill() uses standard signals). There are 32 different real-time signals allowed in a Linux process. Each thread can block signal reception of all real-time signals except one. Now, by sending a proper real-time signal (one from 32 available) we can address a specific thread. This limits us to 32 synchronizer-controlled threads per process, but this limit seems reasonable for current implementation. Fig. 5 illustrates the new transmission method. Messages transmitting signals from synchronizers are addressed directly to processes (notice no dispatcher process from PS-GRADE) and stored in shared memory read by thread synchronizers.

OpenMP library is designed for using threads at a higher control level and it contains no methods whatsoever for handling signals, which we would like to provide in our framework. However, OpenMP implemented within the GCC compiler uses the pthread library to create and manage threads. Therefore, it seemed possible to treat OpenMP abstract threads as pthread threads in an asynchronous signal delivery code. This idea is illustrated in the skeleton example below.

```
1. void signal_handler(int signal_number) {
2.   int   id = omp_get_thread_num();
3.   printf("Interrupted thread no %d.\n", id);
4. }
5.
6. struct sigaction sa, oldsa;
7.
8. int main(int argc, char *argv[]) {
9.   sigset_t blocku1;
10.  sa.sa_handler = signal_handler;
11.  sigaction(SIGUSR1, &sa, &oldsa);
12.  sigemptyset(&blocku1);
13.  sigaddset(&blocku1, SIGUSR1);
14.  pthread_sigmask(SIG_BLOCK, &blocku1,NULL);
15.  omp_set_num_threads(4);
16.  #pragma omp parallel shared(blocku1)
17.  {
18.   #pragma omp single
19.   {
20.     printf("Program runs on %d threads.\n",
21.                           omp_get_num_threads());
22.   }
23.   if(omp_get_thread_num() == 2)
24.     pthread_sigmask(SIG_UNBLOCK, &blocku1, NULL);
25.
26.   do_something(omp_get_thread_num());
27.  }
28.  return 0;
29. }
```

In line 14 we block reception of signal USR1. In line 16 we create parallel threads using an OpenMP directive. The threads inherit their signal mask from the main thread. In lines 23 and 24 we select thread number 2 and let it unblock reception of the USR1 signal. Each of the 4 created threads executes function do_something(). The USR1 signal sent to the process containing these threads during execution of the do_something() function will be handled by the thread number 2 only. Thus, it is possible to mix an OpenMP code with the pthread-based signal handling when organizing an asynchronous global control of parallel execution of threads. We should notice that threads control through signals under OpenMP can be implemented in a straightforward way only for simple threads explicitly created by the #pragma omp parallel directive for noniterative computations. Parallel threads created for iterative computations by the directive #pragma omp parallel for are more cumbersome to be asynchronously controlled and to specify a signal handling code. Therefore, they are not suggested to be globally monitored and controlled by signals.

The described program design framework with the control features as above is under implementation under Linux in a cluster of quadcore Intel processors interconnected by a dual communication network, including the following features. Inter-processor control data communication for state reports and signals is performed by message passing in a separate Infiniband network. Computational data communication is performed by a separate Ethernet network. The C language with the MPI2, OpenMP and pthreads libraries are used for writing application programs and the framework control code. Processor clocks are synchronized using the Precision Time Protocol (PTP) [14] based on the use of the Infiniband network.

4 An Application Example Design in the Proposed Framework

Fig. 6 shows a parallel application scheme which a user can define in the proposed environment with program execution control based on global application states monitoring. The example concerns a branch and bound algorithm for the Travelling Salesman Problem.

The application is parallelized by the use of 3 worker processes which are assigned to processor nodes in the system. The nodes are multicore, so the processes are further parallelized at the level of threads assigned to cores. At the process level, we organize a loop controlled by the synchronizer GSync based on the predicate MoreIterations. This predicate takes into account the progression of the solution search and decides if the computations should proceed further. The predicate can be connected to other parts of the application (not shown in the picture) to get requirements from them concerning the quality of the solution. One of the processes starts computations by creating a number of sub-solutions to be further analyzed and developed. All processes report to the synchronizer GSync the number of subsolutions they have to process. The synchronizer decides (predicate LoadImbalance) if and what work (subsolutions) transfer should be done to have all processes working and no one idle. Using control signals it requests from processes to migrate pools of subsolutions to balance load in the system asynchronously to ongoing computations – a process / thread does not need to wait or periodically check if load balancing action is pending/is necessary. Additionally, upon finding a new local best solution a process reports it to GSync which confronts it to the best solutions known to him so far in the predicate NewBest. If the solution is the best, GSync broadcasts it by signals to other processes, so they are able to eliminate not perspective subsolutions.

At the thread level each process runs 3 parallel threads globally controlled by a local synchronizer ThSync. A similar control scheme as described at the process level is embedded within each process using a local thread level synchronizer. ThSyncs act as control interfaces between the global synchronizer GSync and local threads enabling better distribution and parallelization of control actions. Upon reception of control signals from GSync they direct the appropriate local signals to threads. When a thread reports a new state to ThSync it checks if the whole process enters a new global solution state. If this is the case, ThSync reports a new process state to the global synchronizer GSync.

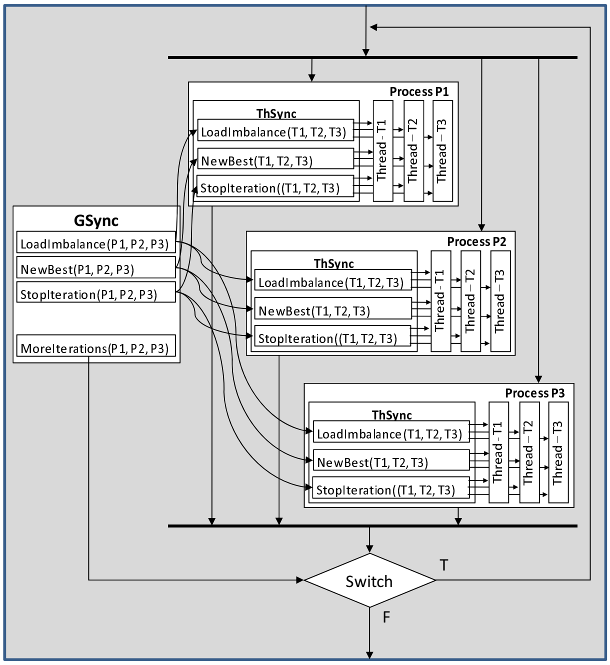

Process P1
ThSync
LoadImbalance(T1, T2, T3)
NewBest(T1, T2, T3)
StopIteration((T1, T2, T3)
Thread - T1
Thread - T2
Thread - T3
GSync
LoadImbalance(P1, P2, P3)
NewBest(P1, P2, P3)
StopIteration(P1, P2, P3)
MoreIterations(P1, P2, P3)
Process P2
Process P3
Switch
T
F

Fig. 6. The control flow graph of the application

Each thread runs a branch and bound computation code represented by the left-hand side of the detailed thread diagram in Fig. 7. In the main computational loop, subsolutions from a local pool are branched and evaluated (bounded) one by one. During the computations, local state (workload and found best solutions) are reported to a thread synchronizer. Whenever a signal is received, its handling code represented by the right-hand-side is activated. A signal handling can use parameters delivered with the signal, e.g. which process is under-loaded and waiting for work. When the signal handling is done, the control returns to the left hand side code, to the place it was interrupted. Depending on a variable sent with a signal, the standard computational loop can be broken and a desired work redistribution can take place.

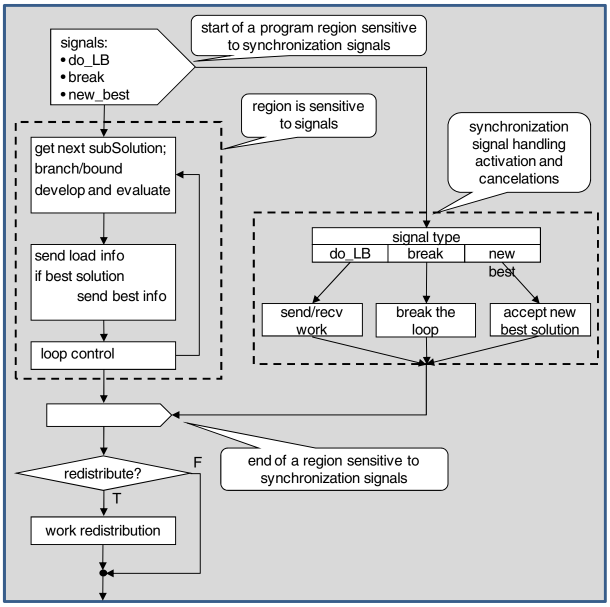

signals:
• do_LB
• break
• new_best
start of a program region sensitive to synchronization signals
region is sensitive to signals
get next subSolution; branch/bound develop and evaluate
send load info if best solution send best info
loop control
synchronization signal handling activation and cancelations
signal type
do_LB
break
new best
send/recv work
break the loop
accept new best solution
redistribute?
F
T
work redistribution
end of a region sensitive to synchronization signals

Fig. 7. The flow diagram of a thread

The global synchronizer GSync can break the branch and bound computations in all processes based on the predicate StopIteration when a good enough solution has been found, or when the asynchronous load balancing among processes is not satisfactory. In the latter case, a complete work redistribution should be performed. When leaving an iteration, the processes start waiting for next activation (next iteration) and the control decision is taken if another loop iteration should be done. The decision is exercised into the control flow action performed by the switch, based on the recently received signal from the MoreIteration predicate contained in GSync. If no breaking the loop signal has arrived, execution of next iteration is enabled.

5 Conclusions

Distributed programs execution control based on monitoring of global application states at the process and thread levels to be included in a new tool for program design, has been discussed. New ways of implementing the local state communication based on shared variables and signals have been proposed. Methods for combining the use of thread level libraries for organizing control in threaded programs with process level communication have been shown. The discussed methods and the underlying program design framework enable extending the distributed program control based on program global states monitoring the process level towards implementation in distributed threads executed in multiple cores of contemporary processors.

References

1. Carriero, N., Gelernter, D.: Linda in Context. Communications of the ACM 32(4), 444–459 (1989)
2. http://www.cs.ucy.ac.cy/courses/EPL441/manifold/tut.pdf, http://reo.project.cwi.nl/
3. Marzullo, K., Wood, D.: Tools for constructing distributed reactive systems. Technical report 14853, Cornell University, Department of Computer Science (February 1991), http://citeseer.nj.nec.com/145302.html
4. Tudruj, M.: Fine-Grained Global Control Constructs for Parallel Programming Environments. In: Parallel Programming and Java: WoTUG-20, pp. 229–243. IOS (1997)
5. Tudruj, M., Kacsuk, P.: Extending Grade Towards Explicit Process Synchronization in Parallel Programs. Computers and Artificial Intelligence 17, 507–516 (1998)
6. Babaoglu, O., Marzullo, K.: Consistent global states of distributed systems: fundamental concepts and mechanisms. In: Distributed Systems. Addison-Wesley (1995); Consistent global states of distributed systems: Fundamental Concepts and Mechanisms
7. Stoller, S.D.: Detecting Global Predicates in Distributed Systems with Clocks. Distributed Computing 13(2), 85–98 (2000)
8. Borkowski, J.: Global Predicates for Online Control of Distributed Applications. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2004. LNCS, vol. 3019, pp. 269–277. Springer, Heidelberg (2004)
9. Borkowski, J.: Interrupt and Cancellation as Synchronization Methods. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 3–9. Springer, Heidelberg (2002)
10. Tudruj, M., Borkowski, J., Kopanski, D.: Graphical Design of Parallel Programs with Control Based on Global Application States Using an Extended P-GRADE System. In: Distributed and Parallel Systems, Cluster and GRID Comp., vol. 777. Kluver (2004)
11. Toth, M.L., Podhorszki, N., Kacsuk, P.: Load Balancing for P-GRADE Parallel Applications. In: Proc. of DAPSYS 2002, Linz, Austria, pp. 12–20 (2002)
12. Borkowski, J., Tudruj, M., Kopański, D.: Global predicate monitoring applied for control of parallel irregular computations. In: 15th Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP 2007, Naples, Italy. IEEE (2007)
13. http://www.kernel.org/doc/man-pages/online/pages/man3/pthread_attr_setaffinity_np.3.html, http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/index.htm#optaps/common/optaps_openmp_thread_affinity.html
14. Precision Time Protocol (PTP), EndRun Technologies, http://www.endruntechnologies.com/pdf/PTP-1588.pdf

Highly Scalable Dynamic Load Balancing in the Atmospheric Modeling System COSMO-SPECS+FD4

Matthias Lieber[1], Verena Grützun[2], Ralf Wolke[3],
Matthias S. Müller[1], and Wolfgang E. Nagel[1]

[1] Center for Information Services and High Performance Computing,
TU Dresden, 01062 Dresden, Germany
matthias.lieber@tu-dresden.de
[2] Max Planck Institute for Meteorology, Hamburg, Germany
[3] Leibniz Institute for Tropospheric Research, Leipzig, Germany

Abstract. To study the complex interactions between cloud processes and the atmosphere, several atmospheric models have been coupled with detailed spectral cloud microphysics schemes. These schemes are computationally expensive, which limits their practical application. Additionally, our performance analysis of the model system COSMO-SPECS (atmospheric model of the Consortium for Small-scale Modeling coupled with SPECtral bin cloud microphysicS) shows a significant load imbalance due to the cloud model. To overcome this issue and enable dynamic load balancing, we propose the separation of the cloud scheme from the static partitioning of the atmospheric model. Using the framework FD4 (Four-Dimensional Distributed Dynamic Data structures), we show that this approach successfully eliminates the load imbalance and improves the scalability of the model system. We present a scalability analysis of the dynamic load balancing and coupling for two different supercomputers. The observed overhead is 6% on 1600 cores of an SGI Altix 4700 and less than 7% on a BlueGene/P system at 64Ki cores.

Keywords: atmospheric modeling, spectral bin cloud microphysics, scalability, dynamic load balancing, model coupling.

1 Introduction and Related Work

Cloud processes still represent one of the major uncertainties in current weather forecast, air quality, and climate models [1,3,24]. This, however, contrasts to their high importance to the atmosphere. It is obvious that future high-resolution atmospheric models require a more detailed description of cloud processes in order to achieve more realistic predictions of, e.g., extreme weather events. Most of today's atmospheric models describe cloud microphysical processes with a *bulk* approach. The so-called one-moment bulk schemes represent the hydrometeor classes (e.g. cloud water, graupel, and snow) by their bulk mass only and assume a prescribed size distribution of the particles. Two-moment [21] and multi-moment [16] schemes extend the description of each class by additional prognostic variables, such as the hydrometeor number density. This allows a better parameterization of the size distribution function. However, several studies emphasize the importance of a size-resolving approach [5,13]. Such *spectral* microphysics models explicitly characterize the size distribution of the hydrometeors by applying a bin discretization. Spectral microphysics schemes have been introduced in the PSU/NCAR Mesoscale Model (MM5) [13], the Weather Research and Forecasting Model (WRF) [12], and the COSMO model (Consortium for Small-scale Modeling) [6]. One of the challenges for the application of spectral bin microphysics schemes in atmospheric models is their enormous computational complexity. Thus, they have been applied for process studies only, but not for operational applications and it is very unlikely that such schemes will be used for numerical weather prediction or climate studies in the near future. Nevertheless, they are an interesting method for research applications, such as studies on the aerosol-cloud interaction [18], air quality modeling [7] or as benchmark for bulk schemes [22]. Because of their huge computational costs, a high scalability on high-performance computing systems is essential to use such models for comprehensive studies. However, this is complicated by severe load imbalances induced by the spectral microphysics: Cloudy areas of the model domain generate a substantially higher workload than cloudless areas. Such irregular workload variations require dynamic load balancing techniques [25], which readjust the partitioning periodically during the run time to maintain an equal distribution of the computational work. Note, that only a few of the widely-used atmospheric models support dynamic load balancing: parallel versions of MM5 [15] (discontinued) and the Regional Atmospheric Modeling System [26] (experimentally).

We propose a dynamic load balancing scheme for detailed cloud models. The basic idea is to decouple the partitioning of the cloud model from the atmospheric model's partitioning. Instead of creating data structures for the hydrometeors within the atmospheric model, these data are managed by a highly scalable framework, which dynamically balances the workload over the parallel processes. For this task we have developed the framework FD4 (Four-Dimensional Distributed Dynamic Data structures [9,11]). To our knowledge, such dynamic techniques have not yet been used for detailed cloud models. Due to the separation, both models need to be (re)coupled and thus form a system comparable to climate models in the way the coupled atmosphere and ocean model communicate regularly with each other.

Several software frameworks and tools have been developed to provide services for the parallel implementation of complex simulation codes, such as distributed data management and dynamic load balancing [4,25], adaptive mesh refinement [2,27], and model coupling [8,19]. FD4 integrates dynamic data management, load balancing, and coupling into a single framework to operate on the same data structures, which allows more performance optimizations compared to the utilization of separate software for these tasks. However, specialized frameworks offer more functionality than FD4, like grid interpolation for coupling, the selection of various partitioning methods, or adaptive mesh refinement. FD4 has been developed for the parallelization and coupling of detailed cloud models. For example, to account for the requirements of size-resolved cloud microphysics models, the framework is optimized for large numbers of values per grid cell. However, FD4 can also be used for other multiphase or multiphysics applications. FD4 is written in Fortran 95 and uses MPI-2 [14] for parallelization. It is available as open source software at http://www.tu-dresden.de/zih/clouds

The rest of the paper is organized as follows: In the next section we introduce the atmospheric modeling system COSMO-SPECS and explain why the detailed microphysics scheme causes load balance issues. In Sect. 3 we describe the dynamic load balancing approach applied in the recently developed COSMO-SPECS+FD4 and briefly introduce the framework FD4. Finally, in Sect. 4, we show performance results of a benchmark scenario on two different supercomputers comparing both versions of the modeling system.

2 The Atmospheric Modeling System COSMO-SPECS

The model system COSMO-SPECS [6] has been developed to study the interaction between aerosols, clouds, and precipitation with a high level of detail. It consists of the COSMO model (http://www.cosmo-model.org), a non-hydrostatic limited-area atmospheric model, and the spectral microphysics model SPECS (SPECtral bin cloud microphysicS [23]). From the implementation point of view, the cloud parameterization scheme of COSMO has been replaced by SPECS, which introduces 11 new variables to describe three types of hydrometeors (water droplets, frozen particles, and insoluble particles). These 11 variables are discretized into a predefined number of size classes (e.g. 66 for the case presented in Sect. 4), leading to a high amount of data that have to be allocated for each cell of the rectangular grid.

Since the cloud microphysical processes operate on much smaller time scales than the dynamical processes in COSMO, two different step sizes are applied for the time integration. The COSMO step size is about 10–100 s, whereas the step size for the microphysics is at most 1 s. This splitting amplifies the computing time proportion of SPECS and, consequently, the model system's run time is dominated by the microphysics computations. Additionally, the computing time of SPECS per grid cell varies strongly depending on the range of the present size distribution for the three hydrometeor types. Especially the existence of frozen particles, which triggers additional computations, leads up to a 10 times increase of the computational costs compared to clear sky. The relation between the concentration of cloud particles and the computing time is shown in Fig. 1. COSMO is MPI-parallelized using a static domain decomposition of the horizontal grid into regular rectangular partitions. Due to the mentioned variability of the computational costs of SPECS, severe load imbalances occur, which lead to a significant waste of resources and insufficient scalability.

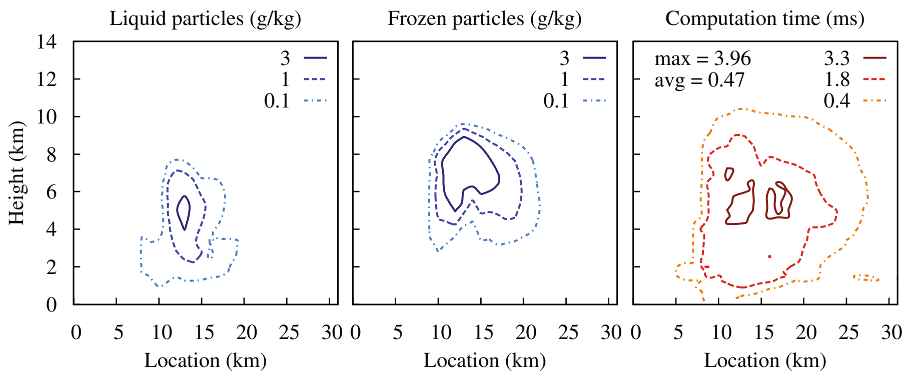


Fig. 1. Comparison of cloud particle mixing ratio and computing time of the spectral bin microphysics model SPECS for a vertical cross section through a simulated cumulus cloud. The plot on the right shows the computing time of one small time step of SPECS running on an SGI Altix 4700 (1.6 GHz Itanium 2 processor).

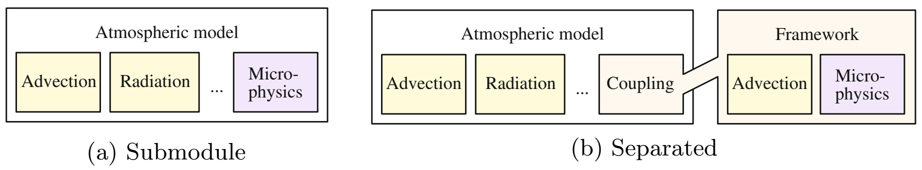
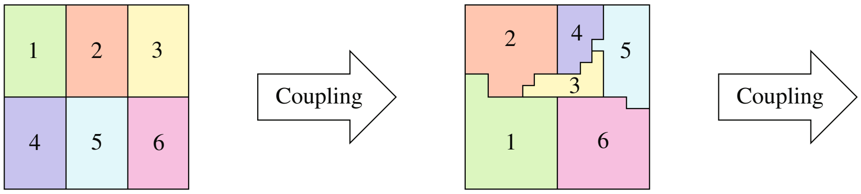


Fig. 2. Coupling concepts for cloud microphysics in atmospheric models: (a) Submodule based on data structures of the atmospheric model, (b) Separated data structures and decomposition using a data management framework

3 Load Balancing and Coupling Using FD4

In the original COSMO-SPECS implementation, the microphysics is incorporated as a submodule in the COSMO model, see Fig. 2(a). To enable the application of dynamic load balancing for the cloud model, we separated the hydrometeor data and related computations (microphysics and advection) from the COSMO model, see Fig. 2(b). These data are managed by the framework FD4 [9,11], which has been developed for the parallelization of multiphase cloud models. The program flow of one time step in COSMO-SPECS+FD4 is shown in Fig. 3. FD4 balances the microphysics computations and transfers coupling data between the different partitionings. The extensive hydrometeor data exist in the FD4 data structures only and are not exchanged with COSMO.

FD4 Data Structure. FD4 decomposes the regular grid in the three spatial dimensions into rectangular blocks, which consist of multiple grid cells. These blocks represent the smallest unit for load balancing. Consequently, their total number should be large enough to enable a fine-grained load balancing. FD4 allocates the data fields in the blocks according to a variable table that is specified



COSMO	**Coupling**	**SPECS**	**Coupling**
atmospheric dynamics, correction of wind fields to preserve mass balance	COSMO $\Downarrow$ SPECS	advection of hydrometeors, spectral bin microphysics, FD4 dynamic load balancing	SPECS $\Downarrow$ COSMO

COSMO	**Coupling**	**SPECS**	**Coupling**
atmospheric dynamics, correction of wind fields to preserve mass balance	COSMO $\Downarrow$ SPECS	advection of hydrometeors, spectral bin microphysics, FD4 dynamic load balancing	SPECS $\Downarrow$ COSMO

Fig. 3. Program flow of one time step in COSMO-SPECS+FD4 and exemplary illustration of the spatially different partitionings for COSMO and SPECS. The 6 parallel processes perform computations for COSMO and SPECS alternately. The partitions of each process for COSMO and SPECS are indicated by numbers 1–6. The SPECS computations are performed using a predefined number of smaller time steps per COSMO step.

by the user. An iterator is provided to traverse through the list of local blocks and access the data.

Dynamic Load Balancing. The blocks are distributed across the processes using space-filling curve (SFC) partitioning [25]. In general, SFCs provide a fast mapping from n-dimensional to one-dimensional space that preserves spatial locality. FD4 uses a Hilbert SFC [20] to reduce the three-dimensional partitioning problem to the contiguous partitioning of a one-dimensional array of block weights. For optimal load balance, the maximum load (*bottleneck value*) among all partitions has to be minimized. Several heuristics and exact algorithms exist for this problem [17]. FD4 uses a trivial parallel algorithm: Each process checks for a different bottleneck value whether a partitioning exists for it. Then, the minimum of the valid bottleneck values is identified and each process determines its own partitioning based on this value.

Performing dynamic load balancing involves costs for the calculation of a balanced partitioning and the redistribution of blocks. It is only beneficial (i.e. application run time is reduced), when the time-saving of a better balanced workload compensates these costs. This is addressed explicitly by FD4: The load balancing routine estimates the time required for load balancing and the time lost due to imbalance based on the elapsed steps and decides automatically whether load balancing is beneficial or not.

Coupling. FD4 facilitates to couple models based on FD4 to external models that have a different partitioning. It computes the overlaps of the external model's partitions with the FD4 block structure and transmits the data directly between the processes. Data can be exchanged in both directions between FD4 and the external models.

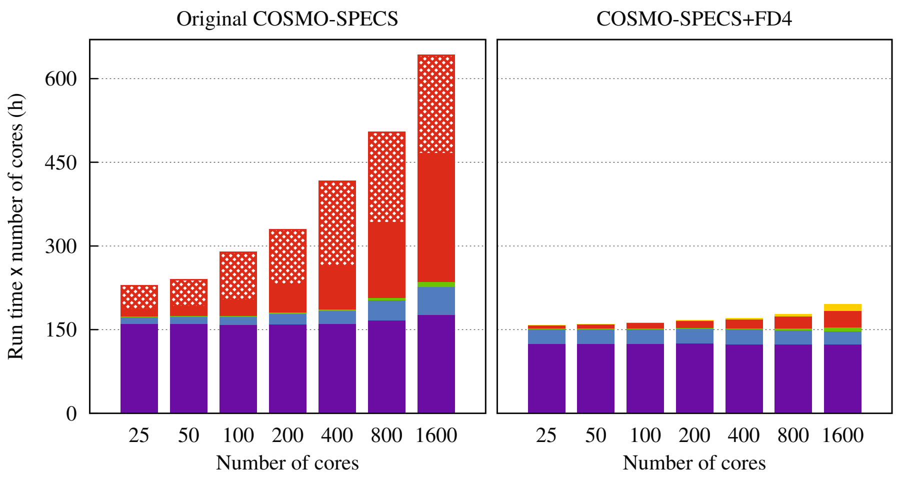
FD4 is based on the *sequential* scheduling [8] of the coupled models. Consequently, all processes perform computations for both COSMO and SPECS alternately. We expect this approach to perform better compared to the *concurrent* scheduling strategy, where the available cores are divided into fixed disjoint groups per coupled model. Since the total workload of SPECS varies strongly depending on the quantity and the type of clouds in the model domain, the latter approach would lead to load imbalances *between* the models [10].

4 Performance Results

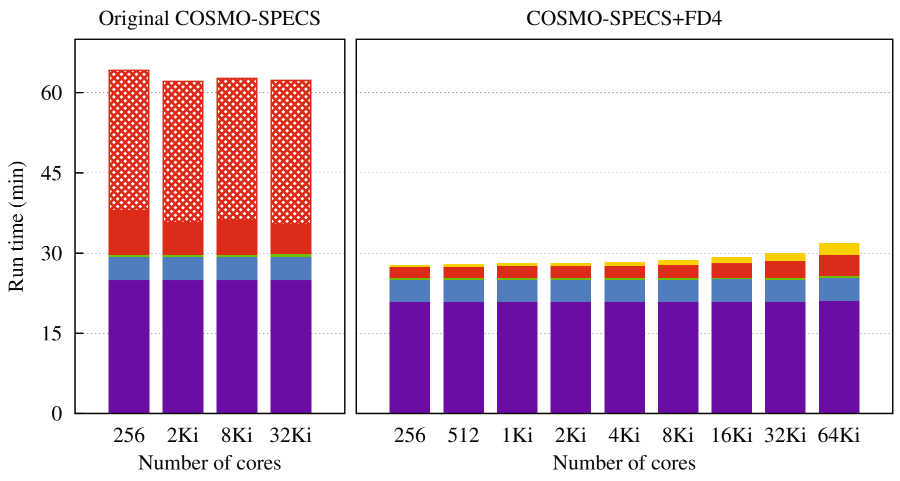
We compared the computational performance and scalability of the original COSMO-SPECS and its load balanced version COSMO-SPECS+FD4 using an artificial test scenario of a heat bubble over flat terrain [6]: A temperature perturbation, which is placed in the center of the horizontal grid, results in the growth of a precipitating mixed-phase cumulus cloud during the simulation period of 30 min. Additionally, we introduced a wind shear to the initial conditions. Figure 1 shows mixing ratios of liquid and frozen cloud particles after 30 min of simulation time. The resolution of the periodic horizontal grid was 1 km. The domain height of 18 km was discretized using 48 nonuniform height levels. The time step sizes were 10 s for COSMO and 0.5 s for SPECS, which results in 20 small microphysical steps per dynamical step. The original and the load balanced version yield identical simulation results except for small numerical deviations. All performance measurements are presented without model initialization time and output of simulation results.

4.1 Strong Scaling Benchmark on SGI Altix 4700

For this benchmark a fixed computational grid size of 80×80 cells with 48 height levels was used. The block size for the FD4 decomposition was 2×2×4, which results in a total number of 19 200 blocks. Figure 4(a) shows the performance results for 25 to 1600 cores on an SGI Altix 4700. Note, that the *overall* run time (wall clock time × number of cores) is shown, i.e. the total consumed CPU time. For a strong scaling benchmark, ideal scaling is achieved when the total consumed CPU time is constant with increasing number of cores. It is clear to see that the load balanced implementation scales much better. At 1600 cores, the original program took 24:10 min whereas the FD4 implementation required 7:22 min only, which is more than three times faster. The component breakdown of Fig. 4(a) reveals that the spectral microphysics consumes much more computation time than the COSMO model. However, with rising number of cores, the run time of the original COSMO-SPECS is increasingly dominated



(a) Strong scaling benchmark on SGI Altix 4700



(b) Weak scaling benchmark on IBM BlueGene/P

🟨 FD4 load balancing and coupling (COSMO-SPECS+FD4 only)
🟥 Ghost exchange of microphysical variables
▨ Waiting time due to load imbalance (original COSMO-SPECS only)
🟩 COSMO computations and communication, wind field correction
🟦 Advection of microphysical variables
🟪 SPECS spectral bin microphysics computations

Fig. 4. Performance results and breakdown into components for the strong scaling benchmark (top) and the weak scaling benchmark (bottom). Flat horizontal lines indicate perfect scaling.

by MPI communication and waiting times due to load imbalance. The reason for the increasing MPI communication costs was found to be an inefficient message exchange scheme for the ghost cells of the microphysical variables using many small messages instead of few big ones. At 1600 cores less then 40% of the overall time is used for computations. The optimized COSMO-SPECS+FD4 has a much smaller communication overhead which is slightly increasing due to a decreasing average load balance and increasing costs for the actual message transfer. The run time percentage of FD4's data management is relatively low. However, it increases from 0.1% for dynamic load balancing and 0.1% for coupling at 25 cores to 2.7% and 3.3%, respectively, at 1600 cores.

4.2 Weak Scaling Benchmark on IBM BlueGene/P

The complexity of the dynamic load balancing and coupling algorithms applied in FD4 depends on the total number of blocks and the number of MPI processes. This poses the question, if COSMO-SPECS+FD4 can run on 10^4 cores efficiently. Therefore, we performed weak scaling benchmarks on an IBM BlueGene/P system. To scale up the problem size (and workload) exactly in the same proportion as the number of cores, we use a *replication scaling* approach [27]. At model initialization, the horizontal grid is *virtually* subdivided into tiles of 32×32 cells. Each tile is initialized with identical conditions for the heat bubble test scenario. The horizontal grid resolution and the number of height levels are kept constant at 1 km and 48 levels, respectively. We scaled our benchmark from a 32×32 grid containing one cloud at 256 cores up to a 512×512 grid containing 256 clouds at 64Ki cores. With an FD4 block size of 2×2×4 cells, the average number of blocks per process is constant at 12. Thus, FD4 had to balance 786 432 blocks dynamically on 64Ki cores in the largest run. Note, that neither COSMO-SPECS nor FD4 take advantage of the replication. Figure 4(b) shows the measured run times for the original COSMO-SPECS and the tuned COSMO-SPECS+FD4 divided into components. Since the workload per core is kept constant, perfect scaling is achieved when the program's run time does not increase with rising number of cores. Both versions scale almost perfectly, but the load balanced version is approximately twice as fast as the original one. The plot for COSMO-SPECS+FD4 indicates that the slight increase of run time is due to the load balancing and coupling of FD4 as well as growing costs for the ghost exchange. The FD4 workload is mainly growing because of the above mentioned complexity of the algorithms. At 64Ki cores, the percentage of FD4 is less then 7%, which shows that COSMO-SPECS+FD4 can efficiently utilize more than 10^4 cores.

4.3 Analysis of Load Balance

In Fig. 5 the measured load balance of both model versions is plotted against the time steps of the benchmark simulation on 8192 cores. Load balance is defined here as the average computing time among all processes divided by the maximum computing time among all processes. The ideal case is a load balance of one and

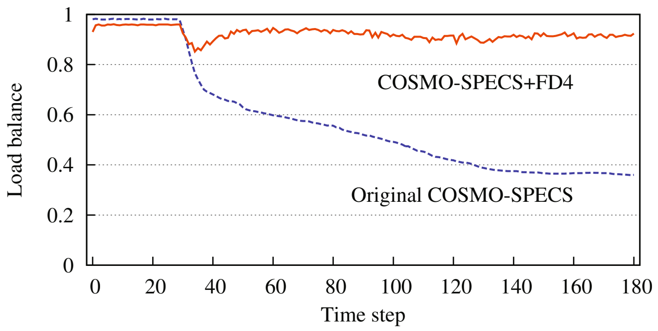


Fig. 5. Comparison of the load balance per COSMO time step between the original COSMO-SPECS and COSMO-SPECS+FD4 with dynamic load balancing. The measurement was performed for the weak-scaling benchmark case at IBM BlueGene/P on 8192 cores.

the worst case is the reciprocal of the number of processes. After 30 time steps, the load balance in the original COSMO-SPECS starts to drop notably, which indicates the beginning of the cloud growth. At the end of the simulation run, the balance is below 0.4. The load balance in COSMO-SPECS+FD4 drops down to 0.85 after 30 steps but stabilizes after 45 steps in the interval between 0.89 and 0.96 for the rest of the run. In the phase during steps 30–45, the load balancing approach to take the measured workload of the blocks as estimation for the next time step is not able to sufficiently keep pace with the high dynamics of workload variation. On average about 64% of the blocks have been migrated between the processes per COSMO time step during the COSMO-SPECS+FD4 run, which is very much. However, due to the costly microphysics computations, the relative communication overhead is very low. Furthermore, the communication pattern for the block migration is highly local: About 63% of the blocks were exchanged between direct neighbor MPI ranks in this run. Local communication patterns typically provide higher bandwidths than arbitrary patterns. The reason for this high locality is the SFC partitioning algorithm, which only shifts the process borders in the one-dimensional array of blocks.

5 Conclusion and Outlook

In this paper, we introduce a new way of coupling detailed cloud microphysics computations to atmospheric models, which allows dynamic load balancing. By using the framework FD4 to couple the mesoscale atmospheric model COSMO and the spectral bin microphysics model SPECS, a significant performance increase is achieved. Performance measurements on up to 64Ki cores show that the approach induces only little overhead for dynamic load balancing and coupling. While we expect the approach to be beneficial for other possibly less expensive spectral schemes, this most likely does not apply to two-moment or multi-moment schemes due to their much smaller number of variables and considerably lower computational costs.

The high scalability of the new system is an important requirement for the feasibility of practical applications with spectral microphysics in atmospheric models. Additional improvements could render this possible in the very near future. As a next step we are aiming to reduce the computational costs of the microphysics by dynamically deciding for each grid cell whether the fast bulk parameterization scheme is sufficient (clear sky) or the spectral model is required. Another important aspect is the proper selection of the time integration step for the microphysics. The time scales of cloud processes are very heterogeneous in time and space, and thus, multirate time integration schemes provide a further approach of saving computational costs.

Acknowledgments. We thank the Deutscher Wetterdienst (German Weather Service) for providing the COSMO model. Furthermore, we thank Jülich Supercomputing Centre, Germany, for access to the JUGENE BlueGene/P supercomputer and Leibniz Supercomputing Centre, Germany, for access to their SGI Altix 4700 system. This work was funded by the German Research Foundation (DFG), grant No. NA 711/2-1.

References

1. Builtjes, P., Fowler, D., Feichter, J., Lewis, A., Monks, P., Borrell, P. (eds.): The Impact of Climate Change on Air Quality. The 4th ACCENT Barnsdale Expert Workshop (2008)
2. Burstedde, C., Burtscher, M., Ghattas, O., Stadler, G., Tu, T., Wilcox, L.C.: ALPS: A framework for parallel adaptive PDE solution. J. Phys. Conf. Ser. 180(1), 012009 (2009)
3. Chin, M., Kahn, R.A., Schwartz, S.E. (eds.): Atmospheric Aerosol Properties and Climate Impacts. U.S. Climate Change Science Program and the Subcommittee on Global Change Research (2009)
4. Devine, K., Boman, E., Heaphy, R., Hendrickson, B., Vaughan, C.: Zoltan Data Management Services for Parallel Dynamic Applications. Comput. Sci. Eng. 4(2), 90–97 (2002)
5. Fahey, K.M., Pandis, S.N.: Size-resolved aqueous-phase atmospheric chemistry in a three-dimensional chemical transport model. J. Geophys. Res. 108(D22), 4690 (2003)
6. Grützun, V., Knoth, O., Simmel, M.: Simulation of the influence of aerosol particle characteristics on clouds and precipitation with LM–SPECS: Model description and first results. Atmos. Res. 90, 233–242 (2008)
7. Jacobson, M.Z., Ginnebaugh, D.L.: Global-through-urban nested three-dimensional simulation of air pollution with a 13,600-reaction photochemical mechanism. J. Geophys. Res. 115, D14304 (2010)
8. Larson, J., Jacob, R., Ong, E.: The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models. Int. J. High Perf. Comput. Appl. 19, 277–292 (2005)
9. Lieber, M., Grützun, V., Wolke, R., Müller, M.S., Nagel, W.E.: FD4: A Framework for Highly Scalable Load Balancing and Coupling of Multiphase Models. In: AIP Conf. Proc., vol. 1281(1), pp. 1639–1642 (2010)

10. Lieber, M., Wolke, R.: Optimizing the coupling in parallel air quality model systems. Environ. Modell. Softw. 23(2), 235–243 (2008)
11. Lieber, M., Wolke, R., Grützun, V., Müller, M.S., Nagel, W.E.: A framework for detailed multiphase cloud modeling on HPC systems. In: Parallel Computing, vol. 19, pp. 281–288. IOS Press (2010)
12. Lynn, B., Khain, A., Rosenfeld, D., Woodley, W.L.: Effects of aerosols on precipitation from orographic clouds. J. Geophys. Res. 112, D10225 (2007)
13. Lynn, B.H., Khain, A.P., Dudhia, J., Rosenfeld, D., Pokrovsky, A., Seifert, A.: Spectral (Bin) Microphysics Coupled with a Mesoscale Model (MM5). Part I: Model Description and First Results. Mon. Weather Rev. 133, 44–58 (2005)
14. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface (1997), http://www.mpi-forum.org
15. Michalakes, J.: MM90: A scalable parallel implementation of the Penn State/NCAR Mesoscale Model (MM5). Parallel Computing 23(14), 2173–2186 (1997)
16. Milbrandt, J.A., Yau, M.K.: A Multimoment Bulk Microphysics Parameterization. Part I: Analysis of the Role of the Spectral Shape Parameter. J. Atmos. Sci. 62(9), 3051–3064 (2005)
17. Pinar, A., Aykanat, C.: Fast optimal load balancing algorithms for 1D partitioning. J. Parallel Distrib. Comput. 64(8), 974–996 (2004)
18. Planche, C., Wobrock, W., Flossmann, A.I., Tridon, F., Van Baelen, J., Pointin, Y., Hagen, M.: The influence of aerosol particle number and hygroscopicity on the evolution of convective cloud systems and their precipitation: A numerical study based on the COPS observations on 12 August 2007. Atmos. Res. 98(1), 40–56 (2010)
19. Redler, R., Valcke, S., Ritzdorf, H.: OASIS4 - a coupling software for next generation earth system modelling. Geosci. Model Dev. 3(1), 87–104 (2010)
20. Sagan, H.: Space-filling curves. Springer, Heidelberg (1994)
21. Seifert, A., Beheng, K.D.: A two-moment cloud microphysics parameterization for mixed-phase clouds. Part 1: Model description. Meteorol. Atmos. Phys. 92, 45–66 (2006)
22. Seifert, A., Khain, A., Pokrovsky, A., Beheng, K.D.: A comparison of spectral bin and two-moment bulk mixed-phase cloud microphysics. Atmos. Res. 80, 46–66 (2006)
23. Simmel, M., Wurzler, S.: Condensation and activation in sectional cloud microphysical models. Atmos. Res. 80, 218–236 (2006)
24. Solomon, S., et al. (eds.): Climate Change 2007: The Physical Science Basis. Contribution of Working Group I to the Forth Assessment Report of the Intergovernmental Panel on Climate Change (IPCC). Cambridge University Press (2007)
25. Teresco, J.D., Devine, K.D., Flaherty, J.E.: Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations. In: Numerical Solution of Partial Differential Equations on Parallel Computers, pp. 55–88. Springer, Heidelberg (2006)
26. Tremback, C.J., Walko, R.L.: The Regional Atmospheric Modeling System (RAMS): Development for parallel processing computer architectures. In: 3rd RAMS Users Workshop (1997)
27. Van Straalen, B., Shalf, J., Ligocki, T., Keen, N., Yang, W.S.: Scalability challenges for massively parallel AMR applications. In: IPDPS 2009, pp. 1–12. IEEE Computer Society (2009)

Interactive Weather Simulation and Visualization on a Display Wall with Many-Core Compute Nodes

Bård Fjukstad, Tor-Magne Stien Hagen, Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, and Otto Anshus

Dept. of Computer Science, University of Tromsø, N-9037 Tromsø, Norway
{baard.fjukstad,phuong.hoai.ha,john.markus.bjorndalen,otto.anshus}@uit.no,
{tormsh,daniels}@cs.uit.no

Abstract. Numerical Weather Prediction models (NWP) used for operational weather forecasting are typically run at predetermined times at a predetermined resolution and a fixed geographical region. The period between each run is a function of waiting for observational data and the availability of compute resources. The resolution is a function of the geographical region, the available processing power and operational forecasting time constraints. The geographical region is defined by being a region with known need or interest for forecasts. These characteristics make it hard to interactively produce and visualize on-demand high-resolution forecasts for a small and arbitrarily located region. This paper documents a system achieving this, using a high-resolution tiled 22 mega pixel display wall, a 16 node PC cluster and a HP BL 460c blade server with two quad core processors. We document the performance characteristics experimentally. The results show that using 10 km resolution background data, the system produces a 6 hour forecast for a 117 x 123 km small region with 3 km resolution, in 3 minutes. Visualizing the forecast takes between 3 - 75 seconds. An informal survey among operational forecasters indicate that the majority is willing to wait up to 3 minutes for higher resolution forecasts. This paper identifies and documents some of the bottlenecks and computational challenges created by combining interactivity and traditional batch oriented computing. The main bottlenecks in the system are identified as the execution time of the NWP and the preparation of data for visualization.

Keywords: Interactive Numerical Weather Model, WRF, Tiled Display Walls, Live Data Sets, On-Demand Computation.

1 Introduction

Numerical Weather Prediction models for use in weather forecasting centers are often computed for a fixed static region at a fixed resolution. One example is the very high-resolution turbulence forecasting system called SIMRA [5], in daily operational use by the Norwegian Meteorological Institute [10]. The SIMRA system uses the wind field from a coarser model to estimate the detailed current turbulence levels around specific locations. Available compute resources limit the number of locations this model can make available to the weather forecaster. This reduces the number of airports where the forecaster can assess the current level of risk for severe turbulence. Therefore, only places of interest with a previously known high level of risk have pre-computed models available. At any given day, this may or may not be the actual trouble spots. Results from NWPs are interactively visualized on a typical PC display. Standard screen resolution is in the order of 1200x1024 pixels.

For any given area and selected parameter the visualization software often has to render the available data using a subset of the original data points. One example is viewing wind fields. These are often visualized using small arrows at each data point, which is not possible to do for a large area on a small screen without either reducing the readability of the plot or displaying only a subset of the available data plots. Using very large high-resolution displays gives the user the option of both viewing large areas, and at the same time all available data points. This has previously been shown to be advantageous using standard visualization software [6].

This paper presents WallWeather, an interactive system and approach for visualizing state-of-the-art numerical meteorological models using a wall-sized high-resolution tiled display [11]. The idea is that the user does not know a priori where high-resolution forecasts would be most useful, and that the user based on available coarser models, select the area and desired resolution. Initially the resolution is a function of the available background meteorological data. The user can select a region of interest by zooming in on that region and have NWP done on-demand for the selected area at the desired resolution. The many-core compute clusters will provide the on-demand weather forecasts for the selected areas.

The ability to select smaller regions of interest with high-resolution forecasts, combined with a display wall supported by on-demand computing, enables a close to interactive experience for the user, at resolutions orders of magnitudes larger than regular desktop displays.

WallWeather is a platform for further experimenting with various ways to divide the total workload and also to investigate the many bottlenecks such complex combined systems present. WallWeather is also a system that both generates and visualizes datasets on demand, as opposed to existing batch-oriented systems where datasets are created at pre-defined times.

This work is based on an idealized use case shown in Table 1.

2 The Numerical Weather Prediction model

In this paper the WRF NWP model [2] is used. WRF is currently a very popular research model for high-resolution weather forecasting systems. WRF is available in numerous settings and is extensively used in many meteorological research and operational centers [3].

A simpler downscaling of the wind field for each time-step, like the SIMRA system, may reduce the workload but does not provide the forecaster with the fully integrated set of parameters available from the NWP models.

To simplify the prototype, the resolution of the WRF model is limited to a to a fixed set of discrete resolutions. This is a necessity given the available topographical, meteorological and environmental data. NWP models are usually downscaled by a factor of 3-5, so that when using 50 km background data, 9.9, 3.3 and 1.1 km resolution models would be the natural levels for stepwise increase of the resolution for the NWP model. Even higher resolution models are possible with access to high-resolution background data. To ensure numerical stability of the model with the steep topography in the area of interest, the time step of the model must be reduced more than recommended.

In the prototype, a static set of background meteorological data from a date with locally severe weather in the area of interest where chosen. An independent start analysis using actual observations is not used in the system. For the small areas in which the NWP model is run, normally only a few actual observations would be available, and a long time-period is needed to include the necessary observation error statistics for the analysis. The prototype still incurs most of the workload that an operational system would require.

Figure 1 shows a possible scenario with several trouble spots. For Areas A, B and F the requested resolutions are large enough for running WRF directly using the background meteorological data. Areas D and E are requested with a higher resolution and require an intermediate step, area C, to be computed. Once area C is computed, all higher resolution areas that fall within C require no extra intermediate computations. The effect of these scenarios on the perceived latency for the user is shown in Figure 2.

3 Experimental Platform

3.1 The Display Wall

The display wall [12] consists of 28 projectors driven by 28 computers arranged in a 7x4 grid yielding a total resolution of 7168 x 3072 pixels. When using WallGlobe the user perceives the display wall as one single coherent display.

Table 1. Idealized use case

1	The forecaster browses a coarse resolution model for possible trouble spots.
2	The forecaster zooms in to view details and triggers a new NWP run.
3	The forecaster views the results from a high-resolution model for the specific area.
4	The forecaster pans the view to include nearby trouble spots, or zooms out and focuses on a new area.

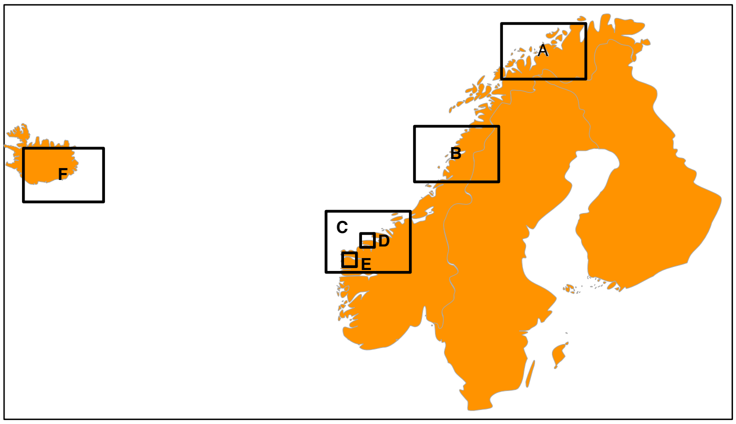


Fig. 1. Cases A, B, D, E and F are the trouble spots in this situation. A background model is assumed available in the whole area of interest.

Zoom and pan is implemented using a touch-free interface [11]. Figure 3 shows two persons interacting with the display wall and the WallGlobe visualization system.

3.2 WallScope

The work presented in this paper is implemented as part of the WallScope [7] visualization and computation platform. WallScope uses a Live Data Set (LDS) architecture. Visualization clients run on each computer in the display wall cluster, all synchronized by a separate state server. Each client requests data from the LDS, which initiates local or remote computations to satisfy the request. LDS may also return a cached copy if the computation has been performed earlier. The LDS architecture is shown in Figure 4. The architecture separates visualization from data management, and data management from the data producer. For interactive visualization of weather forecasts the WallScope system is extended with an on-demand simulation and visualization backend using the WRF NWP model.

The visualization system used in this paper is WallGlobe, a system for visualizing the Earth by combining data from different compute resources in the WallScope system. WallGlobe requests images of size 512x512 pixels from the LDS, which are used as part of the final rendering. Each tile in the display wall requests the images it needs to complete the visualization. Until the maximum zoom level is reached, each tile will at all times use images with a resolution as high as or higher than the resolution of the local resolution. The tile resolution is 1024x768 pixels. The worst-case scenario is that a tile has just reached a new zoom level and that the available images are offset as illustrated in Figure 5. In this case, 25 images have to be requested and retrieved from the LDS.

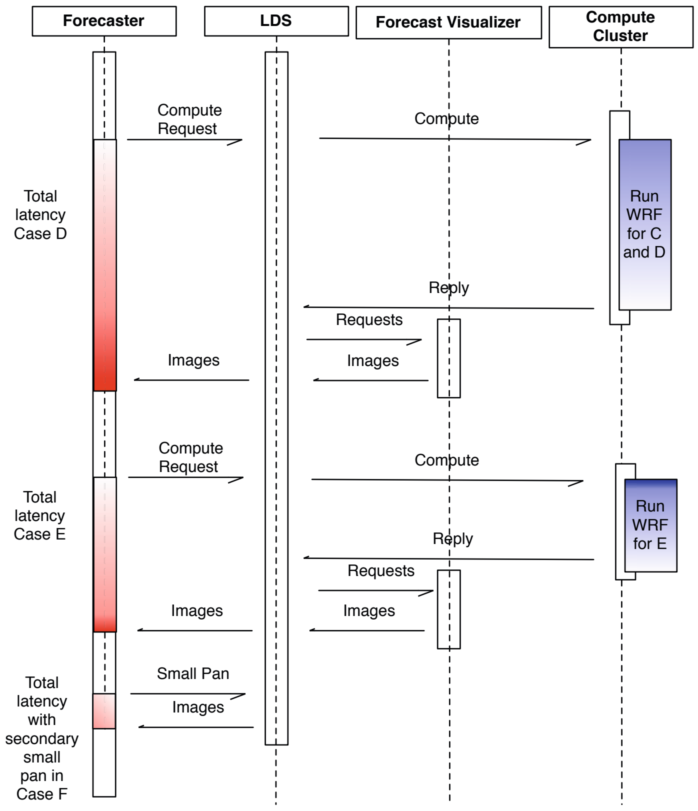


Fig. 2. Three different cases are shown. Case D with no intermediate level available, Case E, with the C area available, and Case F where the model is first run on a slightly larger area than requested so that minor Pans does not trigger a full generation of a new area.

3.3 Compute Clusters

The prototype utilizes two clusters. One is a local 32 node 3.2 GHz Pentium 4 cluster, "Rocks"; the other is a 704 node 1408 CPU 5632 core "Stallo" [4] high-performance cluster. The Rocks cluster is a dedicated cluster and jobs submitted are immediately executed. Stallo uses a standard batch job queuing system and is therefore not very well suited for interactive use. An express queue with limitations on the number of cores available for each job can be used for a near real-time interactive use.

3.4 Network

Every node in the display wall are interconnected using gigabit Ethernet. The display wall is connected to the compute clusters over a gigabit Ethernet link.

Fig. 3. Using the display wall and the WallGlobe visualization system

4 Experiments

4.1 Methodology

To evaluate the WallWeather system, two experiments were conducted.

In experiment one, a small informal survey of the operational forecasters at the Norwegian Meteorological Institute in Tromsø, Norway, was conducted, to establish a limit on how long a forecaster would be willing to wait for higher resolution forecasts for a selected area. 14 out of 18 possible participants responded to the questionnaire.

In experiment two, the actual total latency of the system was measured, using both compute clusters. These experiments showed the effect of running the data producing services on a multi node, multi-core platform. The WRF model is expected to scale well and perform well on these platforms [8].

The ECMWF ERA-Interim data used in this study have been obtained from the ECMWF data server [1]. A specific date with severe weather in the area were used for this study. The data has a spatial resolution of around 50 km. The model was run for a 6 hour forecast for a small 39x41 grid, 28 vertical levels with 9.9 km resolution using a time step of 30 sec. The timestep were shortened due to the very steep topography in the model area and to keep the model numerically stable.

The perceived latency after triggering a NWP model run, depends on the availability of the background data the model needs at the requested resolution. Figure 2 illustrate this. As explained in chapter 2, a run of the WRF model may require several steps with increasing resolution before the requested resolution is computed, figure 2 illustrates this. In the top part of Figure 2 area C has to be computed first, and then area D. If the next requested area falls inside the

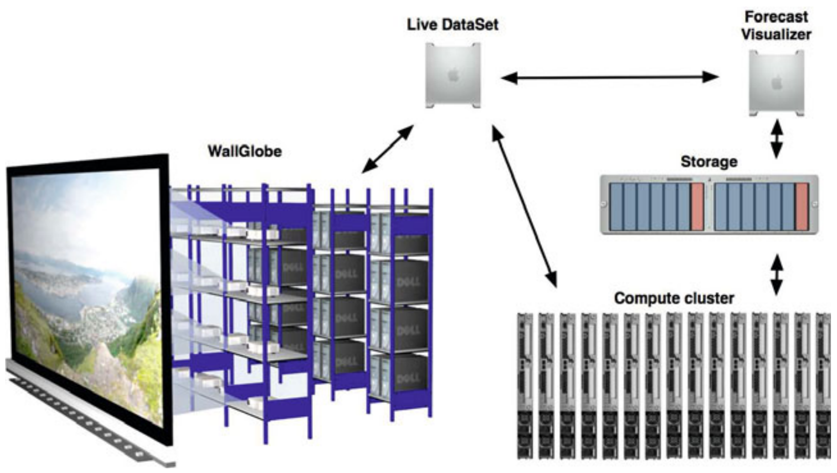


Fig. 4. Architecture with the communication paths indicated

already computed area of C, then the request can be answered by running the model only for the new area E. If the request is only for a small pan within the areas already computed and visualized, then the request will be satisfied from the LDS cache, as shown in the lowest part of Figure 2.

4.2 Results

Table 2 shows the results of experiment one. Almost 60% of the forecasters were willing to wait more than one minute for higher resolution forecasts. Less than 30% would wait more than 5 minutes.

Table 3 show the results of experiment two. For the actual computation, the times are in separate columns. Transferring the resulting data files and retrieving one parameter from the forecast visualizer is identical for both, and are therefore merged into one column.

Table 2. How long a forecaster is willing to wait for higher resolution forecasts

Time	Count
5-14 sec	2
15-44 sec	0
45 sec - 1 min	3
1-2 min	2
3 - 5 min	3
5 - 10 min	2
More than 10 min	2
Total N	14

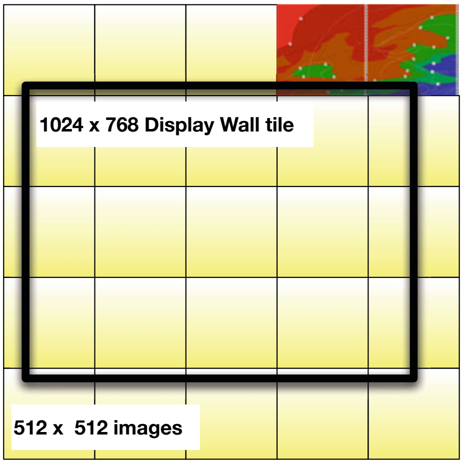


Fig. 5. Illustration of the parts needed for one tile of the display wall. Each image requested from the LDS is 512x512 pixels. Each display wall tile has a resolution of 1024x768 pixels. The WallGlobe will always use images with higher or equal resolution to the tile's resolution.

Table 3. Average run-times Case E using the Stallo cluster using 8 cores on 1 node and the Rocks cluster using 1 core on 16 nodes. Models domain is 39 x 41, 28 vertical levels, 9.9 km resolution, 6 hour forecast with 30 sec time steps.

Task	**Time on "Stallo"**	**Time on "Rocks"**
Running pre-processing on cluster front-end	13 sec	13 sec
Running the WRF model	56 sec	174 sec
Transferring result file to visualization host	0.4 sec	
Retrieving one parameter for visualization	3 sec	

5 Discussion

Table 3 indicates that the largest bottleneck is the execution of the WRF forecast model. When the numerical forecast model is completed, the next bottleneck is the generation of visualization data from the model output. The time listed for visualization in Table 3 is for one single image of size 512x512 pixels.

The system was not intended as a system for delivering high-resolution numerical forecasts each day or at a specific schedule for large areas. For such use the traditional batch oriented systems would be better. The system was created to provide additional high-resolution forecasts for smaller user-selected areas, based on existing coarser resolution NWP model data available to the forecaster.

The WallWeather system provides a practically interactive system even if the latency times for the user are longer than some operational use will tolerate. The system has the ability to display high-resolution visualizations from user defined areas using on demand numerical weather prediction models. This enables possible new insight into relevant meteorological problems, as well as better and more accurate forecasts.

One major bottleneck is the use of one single node for forecast visualization. When each image used by the LDS would come from a single visualizing node, all images needed for covering one single tile on the display wall would take up to 75 sec to retrieve. Since the LDS uses caches, most images that are shared with other tiles on the display wall would be retrieved much faster.

One observation is that using fixed grid sizes with variable spatial resolution in the NWP model, the workload on the computational components varies only with the spatial resolution and time steps needed in the model.

Based on experiment one the latency of the system falls within the acceptable waiting time for the forecasters.

6 Related Work

The triggered WRF forecasts part of the LEAD project [13], presents a similar use case to the WallWeather system. Higher resolution WRF model runs were generated automatically using positions of known severe weather systems from the NOAA NWS news feed. By changing the workflow brokering on a powerful computation cluster to increase the scheduling priority of the model run, timely forecasts were provided. The project identified several problems regarding reliability problems on the compute cluster and the effect on the lack of provided forecasts. No end-user latencies were reported.

7 Conclusions

This paper has presented a prototype of an interactive numerical weather model system, used for on-demand high-resolution visualization on a high-resolution display wall. New numerical weather prediction models are relatively easy to set up with a large range in resolutions, limited mostly by available environmental data, and available computing resources. The experiments conducted on the WallWeather system demonstrates that interactive running of NWPs on high-resolution display walls is coming closer to a practical solution for operational weather forecasting.

8 Future Work

Using GPUs in WRF may improve the runtime significantly [9]. Utilizing GPUs may also improve the visualization performance.

There are various obvious ways to speed up the forecast visualization part of the system. Implementing a distributed system using a compute cluster with single forecast visualization node on each compute node is one possible solution. Depending on the number of nodes, this may reduce the forecast visualization delay to 3 seconds.

Acknowledgements. The authors would like to thank the technical staff at the Computer Science department, University of Tromsø, and the HPC group at the Computer Center, University of Tromsø.

This work has been supported by the Norwegian Research Council, projects No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computers, Storage and Display Devices, and No. 155550/420 - Display Wall with Compute Cluster.

References

1. ECMWF, European Center for Medium Range Weather Forecasting, http://dataportal.ecmwf.int/data/d/interim_daily/
2. The WRF model, http://www.wrf-model.org
3. WRF Real Time Forecasting, http://wrf-model.org/plots/wrfrealtime.php
4. The Stallo cluster (2010), http://docs.notur.no/uit/stallo_documentation/user_guide/key-numbers-about-stallo
5. Eidsvik, K., Holstad, A., Lie, I., Utnes, T.: A prediction system for local wind variations in mountainous terrain. Boundary-Layer Meteorology 112(3), 557–586 (2004), http://dx.doi.org/10.1023/B:BOUN.0000030561.25252.9e
6. Fjukstad, B., Anshus, O., Bjørndalen, J.: High resolution numerical models on a Display Wall. In: The 7th Annual Meeting of the European Meteorological Society (EMS) and the 8th European Conference on Applications of Meteorology (2007)
7. Hagen, T.M.S., Stødle, D., Anshus, O.: On-demand high-performance visualization of spatial data on high-resolution tiled display walls. In: Proceedings of the International Conference on Information Visualization Theory and Applications, pp. 112–119 (2010)
8. Michalakes, J.: (2003), http://www.cisl.ucar.edu/dir/CAS2K3/CAS2K3
9. Michalakes, J., Vachharajani, M.: Gpu acceleration of numerical weather prediction. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–7 (April 2008)
10. Midtbø, K., Bremnes, J.B., Homleid, M., Ødegaard, V.: Verification of wind forecasts for the airports hammerfest, honningsvåg, sandnessjøen, ørsta, værnes, sandane and narvik (2008), http://met.no/Forskning/Publikasjoner/metno_report/2008/filestore/report02_20081.pdf
11. Stødle, D., Troyanskaya, O., Li, K., Anshus, O.J.: Tech-note: Device-Free Interaction Spaces. In: 3DUI 2009: Proceedings of the IEEE Symposium on 3D User Interfaces, pp. 39–42 (March 2009)
12. Wallace, G., Anshus, O.J., Bi, P., Chen, H., Chen, Y., Clark, D., Cook, P., Finkelstein, A., Funkhouser, T., Gupta, A., Hibbs, M., Li, K., Liu, Z., Samanta, R., Sukthankar, R., Troyanskaya, O.: Tools and applications for large-scale display walls. IEEE Comput. Graph. Appl. 25(4), 24–33 (2005)
13. Wilhelmson, R., Alameda, J., Rossi, A., Hampton, S., Jewett, B., Weber, D., Thomas, K., Wang, Y., Droegemeier, K.: Lead: Automatic triggering of high resolution forecasts in response to severe weather indications from the noaa storm prediction center. Urbana 51, 61801

The Algorithm of Multiple Relatively Robust Representations for Multi-core Processors

Matthias Petschow and Paolo Bientinesi

RWTH Aachen University, 52062 Aachen, Germany
`petschow@aices.rwth-aachen.de`
`http://www.aices.rwth-aachen.html`

Abstract. The algorithm of Multiple Relatively Robust Representations (MRRR or MR^3) computes k eigenvalues and eigenvectors of a symmetric tridiagonal matrix in $O(nk)$ arithmetic operations. Large problems can be effectively tackled with existing distributed-memory parallel implementations of MRRR; small and medium size problems can instead make use of LAPACK's routine `xSTEMR`. However, `xSTEMR` is optimized for single-core CPUs, and does not take advantage of today's multi-core and future many-core architectures. In this paper we discuss some of the issues and trade-offs arising in the design of MR^3–SMP, an algorithm for multi-core CPUs and SMP systems. Experiments on application matrices indicate that MR^3–SMP is both faster and obtains better speedups than all the tridiagonal eigensolvers included in LAPACK and Intel's Math Kernel Library (MKL).

Keywords: MRRR algorithm, tridiagonal eigensolver.

1 Introduction

Given a Hermitian matrix $A \in \mathbb{C}^{n \times n}$, the eigenproblem is finding solutions to the equation $Av = \lambda v$ with $\|v\| = 1$, where $\lambda \in \mathbb{R}$ is called an *eigenvalue* and $v \in \mathbb{C}^n$ an associated *eigenvector*. An eigenvalue together with an associated eigenvector are said to form an *eigenpair*. Since A is Hermitian, all the eigenvalues are real and n mutually orthogonal eigenvectors can be found.

Computing all the eigenpairs is equivalent to finding a matrix factorization $A = V \Lambda V^*$, where $\Lambda \in \mathbb{R}^{n \times n}$ is a diagonal matrix containing the eigenvalues as elements, and $V \in \mathbb{C}^{n \times n}$ is a unitary matrix whose columns are the associated eigenvectors. Any Hermitian matrix can be reduced to symmetric tridiagonal form by means of similarity transformations. In this paper we thus focus on the parallel computation of a subset or all the eigenpairs of a real symmetric tridiagonal matrix.

Several efficient and accurate methods exist for the symmetric tridiagonal eigenproblem. Among them, Bisection and Inverse Iteration (BI) [1], the QR algorithm (QR) [2,3], Divide & Conquer (DC) [4,5], and the algorithm of Multiple Relatively Robust Representations (MRRR) [6]. Until the introduction of the latter, the computation of all the eigenpairs required $O(n^3)$ flops in the worst case.

With the MRRR algorithm it is instead possible to compute all the eigenpairs in $O(n^2)$ flops. Moreover, similar to the method of Inverse Iterations, MRRR allows the computation of a subset of the eigenpairs at reduced cost: $O(nk)$ flops for k eigenpairs. In fact, the MRRR algorithm can be seen as a sophisticated variant of Inverse Iteration that does not require explicit orthogonalization, hence the quadratic complexity. An informative and detailed performance analysis of LAPACK's [7] implementations of the four algorithms can be found in [8].

As multi-core architectures have replaced uni-processors, our goal is to explore how MRRR, one of the fastest sequential algorithms, can make efficient use of today's multi-core and future many-core CPUs. A representative example is given in Fig. 1 *(left)*, where the execution time for a matrix of size $n = 4{,}289$ from quantum chemistry is shown as a function of the number of threads used. We present results for four routines: MKL's DC (`DSTEDC`), both MKL's and LAPACK's sequential MRRR (`DSTEMR`), and MR^3–SMP, the multi-core variant of the MRRR algorithm that we present in this paper.[1] BI with 487 seconds and QR with timings between 167 and 61 seconds are much slower and are not shown in the graph.[2] While DC casts most of the work in terms of `DGEMM` and can take advantage of parallelism by multi-threaded BLAS [9], MRRR's `DSTEMR` is sequential and therefore does not exploit any of parallelism of multi-core processors. As a result, DC can become faster than the sequential MRRR as the amount of available parallelism increases.

Tridiagonal matrices are common in applications, but they play a much bigger role as part of dense and banded eigensolvers. In these cases, when many of the eigenpairs are to be computed, the most common approach to solve the eigenproblem consists of three stages: 1) Reduction of A to a real symmetric tridiagonal matrix $T = U^*AU$ via a unitary matrix $U \in \mathbb{C}^{n\times n}$; 2) Solution of the symmetric tridiagonal eigenproblem $Tz = \lambda z$; 3) Back-transformation of the eigenvectors of T into those of A by $v = Uz$.

When MRRR is used for the tridiagonal stage, the reduction becomes the computational bottleneck in this procedure, requiring about $\frac{16}{3}n^3$ floating point operations (flops), only half of which can be cast in terms of fast BLAS-3 kernel routines. By contrast, the about $8n^3$ flops required by the back-transformation stage can be performed efficiently, since the computation can be casted almost entirely in terms of BLAS-3 routines.

Although negligible in a sequential execution, the time spent in the tridiagonal eigensolver becomes significant when multiple cores are used. Fig. 1 *(right)* shows the fraction of the total execution time that is spent on each of the three stages of the dense symmetric eigenproblem of size $n = 4{,}289$ for a varying number of threads. In the single-threaded execution, the tridiagonal eigensolver is indeed negligible, while in the multi-threaded executions it can take up to 40% of the total execution time. By contrast, with 24 threads our multi-core parallel algorithm MR^3–SMP accounts for about 7% of the execution time.

[1] More detailed information about the parameter of the experiment can be found in Section 4.

[2] For the timings of BI and QR Intel MKL 10.2 was used.

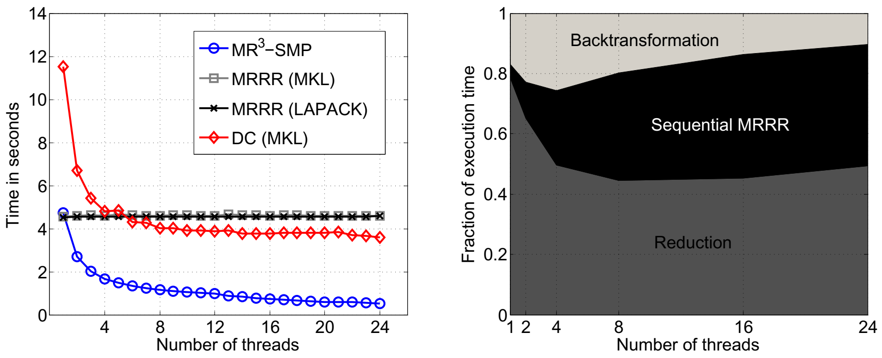


Fig. 1. *Left:* Timings as function of the number of threads used in the computation. Qualitatively, the graph is typical for the applications matrices that we tested. The Divide & Conquer algorithm becomes equally fast or even faster than the sequential MRRR algorithm. MR^3–SMP however is faster and obtains better speedups than DC. *Right:* Fraction of time spent in the solution of the corresponding dense symmetric eigenproblem for the reduction, tridiagonal eigenproblem, and back-transformation.

The paper is organized as follows: Section 2 contains a brief discussion of the MRRR algorithm. In Section 3 the design of the MRRR algorithm for shared-memory computer systems, MR^3–SMP, is described. In Section 4 the results of experiments evaluating the performance of MR^3–SMP are shown.

2 The MRRR Algorithm

In this section the algorithm of Multiple Relatively Robust Representations is briefly discussed. A detailed description and justification of the algorithm can be found in [6] and references therein.

Without loss of generality, the symmetric tridiagonal matrix $T \in \mathbb{R}^{n \times n}$ is assumed to be irreducible. That is, no off-diagonal element is smaller in magnitude than a certain threshold that warrants setting it to zero. Possible quantities for such a threshold are discussed in [10]. Otherwise, when T is reducible, each diagonal block can be treated separately.

The algorithm starts by computing a factorization of T, called a *Relatively Robust Representation* (RRR). From this factorization eigenvalue approximations are computed and finally the associated eigenvectors together with the possibly refined eigenvalue.

An RRR is a representation of T that has the property that small relative changes in its non-trivial entries only cause small relative changes in a specific set of eigenvalues [11]. Such an RRR is given by bidiagonal factorizations $T - \sigma I = LDL^T$ with $\sigma \in \mathbb{R}$, where L is lower unit bidiagonal and D is diagonal. The condition for the factorization to be an RRR is given in [11]. However, in the special case that $T - \sigma I = LDL^T$ is definite, the factorization is guaranteed to be an RRR for all eigenvalues.

After computing an RRR for all desired eigenvalues λ_j, it is possible to compute approximations $\hat{\lambda}_j$ to high relative accuracy, that is $|\lambda_j - \hat{\lambda}_j| = O(\varepsilon|\hat{\lambda}_j|)$, where ε denotes the machine precision. This can be achieved using $O(n)$ flops via bisection using a differential quotient-difference transform [12]. In the definite case, all eigenvalues can be computed to high relative accuracy via the dqds-algorithm in $O(n^2)$ arithmetic operations [13].

Once an eigenvalue $\hat{\lambda}_j$ is computed to high relative accuracy, the associated eigenvector $\hat{z}_j$ is computed by solving $(LDL^T - \hat{\lambda}_j I)\hat{z}_j = \gamma_r e_r$, where the right hand side of the system is a multiple of the r-th standard basis vector and γ_r the opportunely chosen scalar [12]. One of the features of the MRRR algorithm is that it is possible to compute an eigenvector $\hat{z}_j$, with $\|\hat{z}_j\|_2 = 1$, such that the residual norm satisfies

$$\|(LDL^T - \hat{\lambda}_j I)\hat{z}_j\|_2 = O(n\varepsilon|\hat{\lambda}_j|) \ . \tag{1}$$

By the gap theorem of Davis and Kahan [14] the error angle to the true eigenvector z_j is bounded by

$$|\sin \angle(\hat{z}_j, z_j)| \leq \frac{O(n\varepsilon)}{\operatorname{relgap}(\hat{\lambda}_j)} \ , \tag{2}$$

where the relative gap of $\hat{\lambda}_j$ is defined as $\operatorname{relgap}(\hat{\lambda}_j) := \min_{i\neq j}(|\hat{\lambda}_j - \lambda_i|/|\hat{\lambda}_j|) = \operatorname{gap}(\hat{\lambda}_j)/|\hat{\lambda}_j|$. This definition and (2) imply that, provided the gap of an eigenvalue $\hat{\lambda}$ is of the same order as its magnitude, the computed eigenvector $\hat{z}_j$ has an error angle of $O(n\varepsilon)$ to the real eigenvector of the RRR. In practice, when $\operatorname{relgap}(\hat{\lambda}_j) \geq tol$ the eigenvalue is called a *singleton* and the eigenvector is computed from the RRR.

Eigenvalues that are not singletons form *clusters*. For each cluster the algorithm aims at computing a new RRR for the eigenvalues of the cluster via the differential stationary qds (dstqds) transform [12]: $L_i D_i L_i^T = LDL^T - \sigma_i I$. The parameter $\sigma_i \in \mathbb{R}$ is thereby chosen in a way that at least one of the eigenvalues in the cluster becomes a singleton. The shifted eigenvalues $\hat{\lambda}_j - \sigma_i$ must be refined to relative accuracy with respect to the new RRR. Full accuracy is only needed for the singletons, so that eigenvectors with small relative residual norm can be computed. This procedure is than applied recursively until all eigenvectors are computed.

A main feature of the MRRR algorithm is that, although the eigenvectors might be computed from different RRRs, they are numerically orthogonal and no Gram-Schmidt procedure for orthogonalization has to be invoked. This property is discussed in detail in [6], where it is shown that the computed quantities satisfy

$$\|(T - \hat{\lambda}_j I)\hat{z}_j\| = O(n\varepsilon\,\|T\|) \ \text{and} \ |\hat{z}_k^T \hat{z}_j| = O(n\varepsilon), \ \ k \neq j \ . \tag{3}$$

3 The MRRR Algorithm for Multi-core Processors

In this section we discuss the design of MR^3–SMP, a parallel version of the MRRR algorithm specifically designed for multi-core and shared-memory architectures. One of the salient features of such systems is the capability of communication among processors at low costs thanks to shared caches and memory. As a consequence, both redundancy and costly data exchanges, characteristic to distributed-memory parallelizations [15,16], now can and should be avoided in favor of a fine grain parallelism.

MR^3–SMP is based on the routine DSTEMR of LAPACK Version 3.2 and makes use of *POSIX threads* for parallelization. A detailed description of DSTEMR and its design criteria can be found in [17].

3.1 Parallelization Strategy

MR^3–SMP achieves parallelism by dynamically dividing the computation into independent tasks. The tasks are placed into a work queue and can be executed by many threads in parallel. This form of parallelism may produce a bigger overhead than a static division of the work, but it is flexible and attains good load balancing among the processors.

After computing the root representation[3], the initial eigenvalue approximations are either computed by the dqds-algorithm or by parallel bisection, depending on the amount of parallelism available and the number of eigenvalues to compute. Bisection is used, when the number of cores c is greater than $12 \cdot \#eigenvalues/n$ [15]. The dqds-algorithm computes the eigenvalues to full accuracy, while bisection only does so when only the eigenvalues are desired.

The computation of the eigenvectors and a gradual refinement of the eigenvalues can be represented in form of a representation tree. The associated work can be naturally divided into tasks: Each node consists of an index set Γ_p of associated eigenpairs and depends on the RRR of its parent node. The computation that has to be executed depends entirely on the size of the index set $|\Gamma_p|$. In the case of $|\Gamma_p| = 1$ the node is a leaf node and represents a singleton. Otherwise, in the case $|\Gamma_p| > 1$, the node is considered a cluster. For both cases, singletons and clusters, we created a task type. A third task type is introduced to add the ability of splitting the work of refining eigenvalues into tasks. The three task types will be called *S-task*, *C-task* and *R-task* subsequently. The work associated to each task is discussed next:

1. ***S-task:*** As described in Section 2, the eigenvectors associated to singletons can be computed immediately. This leads to the following computational task: For a set of singletons $\Gamma_s \subseteq \Gamma_p$, compute the eigenvalues to high relative accuracy and the associated eigenvectors. This is done via Inverse Iteration with twisted factorizations and Rayleigh Quotient correction [17].
2. ***C-task:*** A task is created for each cluster $\Gamma_c \subset \Gamma_p$: Compute a new RRR for the eigenvalues in the cluster and refine the eigenvalues to relative accuracy with respect to the new RRR until a distinction between singletons and clusters is possible. At this point the new representation can be used to

[3] If the input matrix is reducible, there will be multiple root nodes and representation trees.

partition the computation into tasks recursively, that is creating S-tasks and C-tasks with $\tilde{\Gamma}_s \subseteq \Gamma_c$ and $\tilde{\Gamma}_c \subset \Gamma_c$, respectively.

3. ***R-task:*** R-tasks are created when it is advantageous to split and parallelize the refinement of eigenvalues forming a cluster. The R-tasks are therefore created during the execution of a C-task, after computing the new RRR of the cluster. The computation involved in an R-task is: Given an RRR, refine a subset of $\Gamma_i \subset \Gamma_c$ to relative accuracy with respect to the RRR via bisection.

3.2 The Work Queue

In order to execute the tasks in parallel, a work queue is established and filled with the three types of tasks. Each of the tasks can then be processed by any of the computing cores.

The work queue consists of three levels, one for each task type and implemented as a FIFO queue, with different priorities. Many different priority policies may be chosen. Our objective is to attain high performance while limiting memory consumption. In our policy R-tasks, S-tasks and C-tasks have high, medium and low priority, respectively. During the computation of the eigenvectors, each thread in the thread pool is dequeuing tasks from the work queue, processing tasks with higher priority first.

The computation is initialized by treating the root node as a special C-task. In this case there is no need to compute an RRR and refine its eigenvalues. Since the root representation is not overwritten, no special care has to be taken to resolve the data dependency of the newly created tasks at $depth = 1$ in the representation tree. Therefore the tasks are created fast and fill up the work queue. To achieve the same for clusters at higher depth, the parent RRR is copied into the output eigenvector matrix Z for cluster tasks. The task is therefore created faster than computing the RRR for the cluster first and store it in Z, as it is done in DSTEMR.

The organization of the work queue is among other things motivated to bound the amount of memory required during the computation. In the case of a single thread, the order of computation complies with the DSTEMR routine: at each level all the singletons are processed before the clusters.

3.3 An Example Matrix

Fig. 2 shows the execution traces of an exemplary eigenvector computation. The examined matrix of size $n = 12{,}387$ comes from a frequency response analysis of automobile bodies. Computing the eigenvectors took about 49.3 seconds sequentially and about 3.3 seconds with 16 threads. In the time-line graph green, blue and yellow sections correspond to the processing of S-tasks, C-tasks, and R-tasks, respectively. Everything considered as parallelization overhead is colored red.

On average, each thread spends about 66% of the execution time in computing the eigenvectors of singletons, 19% in computing new RRRs of clusters and to refine the associated eigenvalues, and additionally 15% for refining eigenvalues. Almost no overhead occurs during the computation, due to dynamic task scheduling.

The first time that the refinement of eigenvalues is split via R-tasks, a cluster of size 8,871 is encountered by the bottommost thread. Since the cluster contains a large part of the eigenvectors that are still to be computed, the refinement of its eigenvalues is split among all the threads. The number of eigenvalues to refine within a task is reduced in size when the tasks are created, so that load balancing among all the threads is achieved. The procedure of splitting the refinement among all threads is repeated two more times during the computation. Later during the computation there are also examples where the refinement of eigenvalues is split, but the computation is not distributed among all threads.

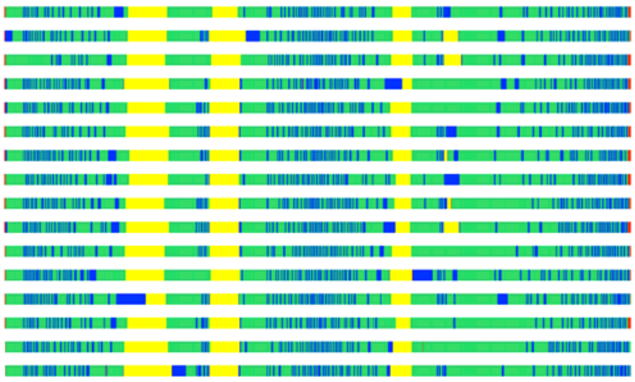


Fig. 2. Execution traces for a matrix of size $n = 12{,}387$, arising in a finite-element model of an automobile body. The colors green, blue, and yellow represent time spent in the execution of S-tasks, C-tasks, and R-tasks, respectively.

4 Experimental Results

In this section we present timing results for the following routines: MR^3–SMP, MKL's and LAPACK's `DSTEMR` (MRRR), and MKL's `DSTEDC` (DC).[4] All the experiments were run on a SMP system comprising four six-core *Intel Xeon 7460 Dunnington* processors, running at a frequency of 2.66 GHz. For both MR^3–SMP and LAPACK[5] routines we used the Intel compilers[6] *icc* and *ifort*, enabling the optimization level 3. LAPACK's `DSTEMR` was linked to MKL's BLAS, and MR^3–SMP to the reference BLAS.

In Table 1 we show results for a set of matrices arising in actual applications. In order make a fair comparison with DC, in all cases the entire eigenspectrum is

[4] For the all MKL routines Version 10.2 was used.

[5] Version 3.2.1.

[6] Version 11.1.

computed. However, we point out that MR^3–SMP allows for subset computation at reduced cost.

MKL's and LAPACK's DSTEMR routines are sequential and attain almost identical performance, therefore we only report timings for the single-threaded execution of LAPACK's MRRR. For DSTEDC and MR^3–SMP we instead report results for 1, 12, and 24 threads.

With the exception of the last matrix (Auto.d), DC executed with 24 threads is faster than the sequential MRRR; in all cases, MR^3–SMP is faster than DC.

Table 1. Execution times in seconds for a set of matrices arising in applications. The first four matrices are from quantum chemistry and the last four arise in finite element models.

Matrix	**Size**	**DC**			**MRRR**	**MR^3–SMP**		
		1	12	24	seq.	1	12	24
SiOSi$_6$	1,687	0.95	0.40	0.40	0.52	0.55	0.14	0.12
ZSM-5	2,053	1.47	0.58	0.56	0.94	0.97	0.23	0.15
Juel.k1b	4,237	11.57	3.69	3.60	4.49	4.72	0.97	0.51
Auto.a	7,923	63.50	19.32	17.72	19.51	20.53	3.69	1.92
Auto.b	12,387	219.56	65.70	33.61	56.43	59.10	11.63	4.84
Auto.c	13,786	233.94	70.94	36.32	54.27	60.31	10.63	5.46
Auto.d	16,023	324.47	98.83	92.02	90.27	97.51	20.56	8.31

Table 2. Speedup of the total execution time of routine DSTEDC and MR^3–SMP on a 24-core system. The reference for DSTEDC is its single threaded execution and for MR^3–SMP is the sequential DSTEMR. The last column shows the factor τ by which MR^3–SMP is faster than DC.

Matrix	**Size**	**DC** (MKL)	**MR^3–SMP**	τ
SiOSi$_6$	1,687	2.4	4.3	3.3
ZSM-5	2,053	2.6	6.3	3.7
Juel.k1b	4,237	3.2	8.8	7.0
Auto.a	7,923	3.6	10.2	9.2
Auto.b	12,387	6.5	11.7	6.9
Auto.c	13,786	6.4	9.9	6.6
Auto.d	16,023	3.5	10.9	11.0

Although the computation was executed on 24 cores, the speedup of MR^3–SMP against the sequential DSTEMR is far from 24. Why the obtained speedup is nonetheless close to the optimal is discussed through the example of matrix *Auto.b*. As it can be seen in Fig. 3 *(left)*, the fast but sequential dqds-algorithm is used to compute the initial eigenvalue approximations: it requires about 7.3 seconds, and it is used with up to 12 cores. If the dqds-algorithm were always be used, independently of the number of available cores, according to Amdahl's law the total speedup would be limited to $56.4/7.3 \approx 7.7$. This limit can be observed in Fig. 3 *(right)* for the total speedup. Instead, in MR^3–SMP bisection is used for more than 12 cores. The computation time for the eigenvalues decreases, but the input of the eigenvector computation changes. When in the initial eigenvalue approximation we force to always use either the dqds-algorithm or bisection, Fig. 3 *(right)* shows good scalability of the eigenvector computation. Notice that the graph of the total speedup in Fig. 3 *(right)* is not yet at a flat asymptote, and greater speedups can be expected with more parallelism.

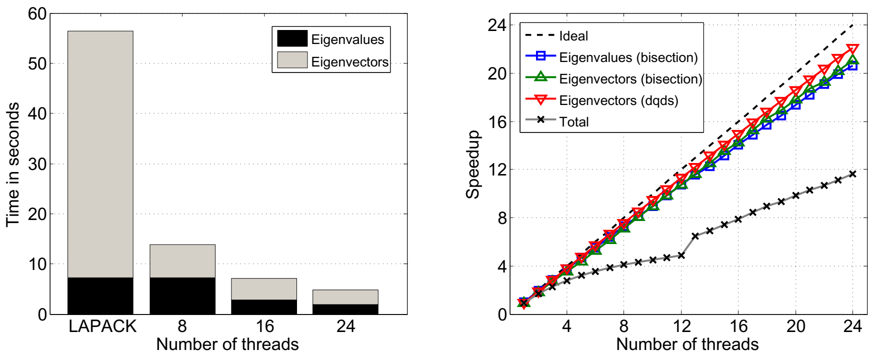


Fig. 3. *Left:* Time spent in the computation of the eigenvalues and eigenvectors for the matrix *Auto.b* of size 12,387. *Right:* Speedup for the eigenvalue and eigenvector computation. The total speedup is naturally limited since for up to 12 cores the initial eigenvalue approximation is performed by the sequential dqds algorithm.

For the sake of brevity, accuracy results are omitted, but we remark that in all tests the accuracy of MR^3–SMP is comparable to that of LAPACK's sequential routine `DSTEMR`.

5 Conclusion

We presented a design to adapt the algorithm of Multiple Relatively Robust Representations to shared-memory computer systems. The result, MR^3–SMP, is an algorithm specifically tailored for current multi-core and future many-core architectures, as well as SMP systems made out of them. We compared MR^3–SMP with all tridiagonal eigensolvers contained in LAPACK and Intel's MKL on a set of matrices arising in real applications: in all cases MR^3–SMP proved to be the fastest algorithm and attained the best speedups.

Acknowledgment. The authors wish to thank the Center for Computing and Communication RWTH Aachen for the resources to run all the experiments, Diego Fabregat and Roman Iakymchuk for discussions. Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged.

References

1. Wilkinson, J.: The Calculation of the Eigenvectors of Codiagonal Matrices. Comp. J. 1(2), 90–96 (1958)
2. Francis, J.: The QR Transform - A Unitary Analogue to the LR Transformation, Part I and II. The Comp. J. 4 (1961/1962)
3. Kublanovskaya, V.: On some Algorithms for the Solution of the Complete Eigenvalue Problem. Zh. Vych. Mat. 1, 555–572 (1961)
4. Cuppen, J.: A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem. Numer. Math. 36, 177–195 (1981)
5. Gu, M., Eisenstat, S.C.: A Divide-and-Conquer Algorithm for the Symmetric Tridiagonal Eigenproblem. SIAM J. Matrix Anal. Appl. 16(1), 172–191 (1995)
6. Dhillon, I., Parlett, B.: Multiple Representations to Compute Orthogonal Eigenvectors of Symmetric Tridiagonal Matrices. Linear Algebra Appl. 387, 1–28 (2004)
7. Anderson, E., Bai, Z., Bishof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. SIAM (1999)
8. Demmel, J., Marques, O., Parlett, B., Vömel, C.: Performance and Accuracy of LAPACK's Symmetric Tridiagonal Eigensolvers. SIAM J. Sci. Comp. 30, 1508–1526 (2008)
9. Dongarra, J., Du Cruz, J., Duff, I., Hammarling, S.: A Set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Software 16, 1–17 (1990)
10. Demmel, J., Dhillon, I., Ren, H.: On the Correctness of some Bisection-like Parallel Eigenvalue Algorithms in Floating Point Arithmetic. Electron. Trans. Numer. Anal. 3, 116–149 (1995)
11. Parlett, B., Dhillon, I.: Relatively Robust Representations of Symmetric Tridiagonals. Linear Algebra Appl. 309, 121–151 (1999)
12. Dhillon, I., Parlett, B.: Orthogonal Eigenvectors and Relative Gaps. SIAM J. Matrix Anal. Appl. 25, 858–899 (2004)
13. Parlett, B., Marques, O.: An Implementation of the DQDS Algorithm (Positive Case). Linear Algebra Appl. 309, 217–259 (1999)
14. Parlett, B.: The Symmetric Eigenvalue Problem. Prentice-Hall (1980)
15. Bientinesi, P., Dhillon, I., van de Geijn, R.: A Parallel Eigensolver for Dense Symmetric Matrices Based on Multiple Relatively Robust Representations. SIAM J. Sci. Comp. 21, 43–66 (2005)
16. Vömel, C.: ScaLAPACK's MRRR Algorithm. ACM Trans. on Math. Software 37(1), 1:1–1:35 (2010)
17. Dhillon, I., Parlett, B., Vömel, C.: The Design and Implementation of the MRRR Algorithm. ACM Trans. on Mathem. Software 32, 533–560 (2006)

Parallelization of Multilevel ILU Preconditioners on Distributed-Memory Multiprocessors

José I. Aliaga[1], Matthias Bollhöfer[2], Alberto F. Martín[1], and Enrique S. Quintana-Ortí[1]

[1] Dpto. de Ingen. y Ciencia de Computadores, Universidad Jaume I, Spain
{aliaga,martina,quintana}@icc.uji.es

[2] Institute of Computational Mathematics, TU-Braunschweig, Germany
m.bollhoefer@tu-braunschweig.de

Abstract. In this paper we investigate the parallelization of the ILUPACK library for the solution of sparse linear systems on distributed-memory multiprocessors. The parallelization approach employs multilevel graph partitioning algorithms in order to identify a set of concurrent tasks and their dependencies, which are then statically mapped to processors. Experimental results on a cluster of Intel QuadCore processors report remarkable speed-ups.

Keywords: Sparse linear system, iterative solver, preconditioner, ILU decomposition, MPI, distributed-memory multiprocessor.

1 Introduction

The solution of sparse linear systems is a computational bottleneck in many scientific computing application problems. While sparse direct methods have proven to be extremely efficient for a wide range of applications, the increasing size of the problems arising from 3D PDEs applications asks for fast and efficient iterative solution techniques. This in turn requires alternative techniques like approximate factorizations combined with Krylov subspace methods, because of their moderate computational and memory requirements [10]. Among these, ILUPACK[1] (*Incomplete LU decomposition PACKage*) is a software package mainly based on ILU factorizations with improved robustness in conjunction with Krylov subspace methods.

Although the application of a preconditioner has the potential of accelerating the convergence rate of iterative solvers, the computational cost per iteration increases. Moreover, the time of computing the preconditioner also needs to be taken into account. To compensate for this, high-performance computing techniques can be applied to speed-up the computation of both the preconditioner and the iterative procedure. The parallelization of ILUPACK-based preconditioners on shared-memory multiprocessors, and scaling studies with up to 16 cores, are discussed in previous work [1,2,3]. As in sparse direct methods [7], this parallelization is inspired by a nested-dissection hierarchy of the initial system

[1] http://ilupack.tu-bs.de

which allows to map independent tasks concurrently to cores within each level. This paper demonstrates that the same parallelization carries over to distributed-memory multiprocessors, reporting remarkable performance on up to 32 cores.

The paper is structured as follows. ILUPACK is briefly reviewed in Sect. 2. Details on the parallelization of this package on distributed-memory multiprocessors are given in Sect. 3. Finally, Sect. 4 contains experimental results collected from the parallel algorithm and offers a few concluding remarks.

2 Computation of Preconditioners in ILUPACK

Preconditioning in ILUPACK relies on the so-called *inverse-based approach*, which improves the robustness of classical ILU factorizations bounding the growth of the entries in the inverses of the triangular factors. To justify this, consider the ILU factorization

$$A = \tilde{L}\tilde{D}\tilde{U} + R \ , \tag{1}$$

where $\tilde{L}, \tilde{U}^T$ are unit lower triangular matrices, $\tilde{D}$ is diagonal, and R is the error matrix which collects those entries that were dropped during the factorization. Applying the preconditioner $M = \tilde{L}\tilde{D}\tilde{U}$, we obtain the preconditioned matrix

$$\tilde{L}^{-1}A\tilde{U}^{-1} = \tilde{D} + \tilde{L}^{-1}R\tilde{U}^{-1} \ . \tag{2}$$

Although dropping typically results in some “relatively small” error matrix R, both $\tilde{L}^{-1}$ and $\tilde{U}^{-1}$ may exhibit very large norms, so that application of the preconditioning can significantly amplify the size of the entries in R. This may directly impact the convergence rate of the preconditioned iterative solver.

The inverse-based preconditioning approach relies on approximate factorizations with “bounded” inverse triangular factors; i.e., factorizations with $\left\|L^{-1}\right\| \leq \kappa$ and $\left\|U^{-1}\right\| \leq \kappa$, for some prescribed small threshold $\kappa > 1$. In practical applications, an ILU factorization of the system at hand does not typically satisfy this requirement, so that pivoting is necessary to bound the inverse triangular factors during the computation. Pivoting is accommodated in a multilevel framework in order to construct a hierarchy of partial inverse-based approximations, as sketched in the following multilevel algorithm:

1. **Preprocessing step.** Matrix A is scaled by diagonal matrices D_l and D_r and reordered by permutation matrices P_l and P_r,

$$\hat{A} = P_l^T D_l A D_r P_r \ .$$

2. **Factorization step.** At each step of the Crout variant of the ILU factorization, the method is interlaced with a pivoting strategy which yields a nonexpensive estimation of the norm of a new row/column of the inverse factors. If the estimation exceeds the threshold κ, the current pivot is rejected and the corresponding row/column are moved to the bottom/right-end of the matrix. Otherwise, the pivot is accepted and dropping is applied to the current row/column before they are incorporated to the factors. This is illustrated

in Fig. 1. Collecting the permutations due to inverse-based pivoting on P, we obtain the following partial ILU factorization of a permuted matrix:

$$P^T \hat{A} P = \begin{bmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{bmatrix} \begin{bmatrix} \tilde{D}_B & 0 \\ 0 & \tilde{S}_C \end{bmatrix} \begin{bmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{bmatrix} + \begin{bmatrix} R_B & R_F \\ R_E & 0 \end{bmatrix} .$$

The method applies additional dropping to the approximate Schur complement $\tilde{S}_C$, so that we actually compute

$$\hat{S}_C = \tilde{S}_C + R_C = C - \left(\tilde{L}_E \tilde{D}_B \tilde{U}_F\right) + R_C .$$

3. **Restarting step.** Steps 1 and 2 are repeatedly applied to $A = \hat{S}_C$ until S_C is void or "sufficiently dense" to be efficiently factorized by a level 3 BLAS-based direct factorization kernel.

For a more detailed description of the numerical approach which lays the foundation of ILUPACK and its theoretical properties see [4,5].

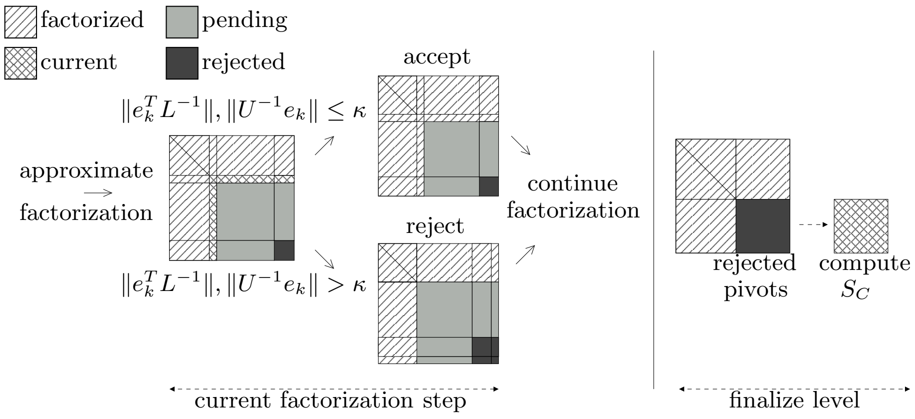


Fig. 1. ILUPACK pivoting strategy

3 Parallelization of ILUPACK

To design a parallel version of ILUPACK, we decompose the computation of the preconditioner into tasks, identify the dependencies among them, and apply static mapping to these operations.

For sparse linear systems, it is possible to apply graph-based symmetric reorderings to find a permutation Π such that

$$\Pi^T A \Pi = \left[\begin{array}{cc|c} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right] . \qquad (3)$$

Computing the ILU decomposition of the leading blocks A_{11} and A_{22}, we obtain the following partial approximation

$$\left[\begin{array}{cc|c} \tilde{L}_{11} & 0 & 0 \\ 0 & \tilde{L}_{22} & 0 \\ \hline \tilde{L}_{31} & \tilde{L}_{32} & \mathrm{I} \end{array}\right] \left[\begin{array}{cc|c} \tilde{D}_{11} & 0 & 0 \\ 0 & \tilde{D}_{22} & 0 \\ \hline 0 & 0 & \tilde{S}_{33} \end{array}\right] \left[\begin{array}{cc|c} \tilde{U}_{11} & 0 & \tilde{U}_{13} \\ 0 & \tilde{U}_{22} & \tilde{U}_{23} \\ \hline 0 & 0 & \mathrm{I} \end{array}\right] + \left[\begin{array}{cc|c} R_{11} & 0 & R_{13} \\ 0 & R_{22} & R_{23} \\ \hline R_{31} & R_{32} & 0 \end{array}\right] ,$$

where the approximate Schur complement is given by

$$\hat{S}_{33} = A_{33} - \left(\tilde{L}_{31}\tilde{D}_{11}\tilde{U}_{13}\right) - \left(\tilde{L}_{32}\tilde{D}_{22}\tilde{U}_{23}\right) + R_{33} \ ; \tag{4}$$

proceeding with the ILU factorization of $\hat{S}_{33}$, the ILU decomposition of $\Pi^T A \Pi$ is completed. The structure of $\Pi^T A \Pi$ allows the explotation of parallelism during this computation. In particular, we can disassemble $\Pi^T A \Pi$ into two submatrices

$$\left[\begin{array}{c|c} A_{11} & A_{13} \\ \hline A_{31} & A_{33}^1 \end{array}\right] , \left[\begin{array}{c|c} A_{22} & A_{23} \\ \hline A_{32} & A_{33}^2 \end{array}\right] , \ A_{33}^1 + A_{33}^2 = A_{33} \ , \tag{5}$$

so that the ILU decomposition of the leading block of both submatrices can be concurrently obtained,

$$\left[\begin{array}{c|c} A_{11} & A_{13} \\ \hline A_{31} & A_{33}^1 \end{array}\right] = \left[\begin{array}{c|c} \tilde{L}_{11} & 0 \\ \hline \tilde{L}_{31} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} \tilde{D}_{11} & 0 \\ \hline 0 & \tilde{S}_{33}^1 \end{array}\right] \left[\begin{array}{c|c} \tilde{U}_{11} & \tilde{U}_{13} \\ \hline 0 & \mathrm{I} \end{array}\right] + \left[\begin{array}{cc} R_{11} & R_{13} \\ R_{31} & 0 \end{array}\right]$$

$$\left[\begin{array}{c|c} A_{22} & A_{23} \\ \hline A_{32} & A_{33}^2 \end{array}\right] = \left[\begin{array}{c|c} \tilde{L}_{22} & 0 \\ \hline \tilde{L}_{32} & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c} \tilde{D}_{22} & 0 \\ \hline 0 & \tilde{S}_{33}^2 \end{array}\right] \left[\begin{array}{c|c} \tilde{U}_{22} & \tilde{U}_{23} \\ \hline 0 & \mathrm{I} \end{array}\right] + \left[\begin{array}{cc} R_{22} & R_{23} \\ R_{32} & 0 \end{array}\right] .$$

Then, we can also compute in parallel the Schur complements corresponding to both partial approximations

$$\hat{S}_{33}^1 = A_{33}^1 - \left(\tilde{L}_{31}\tilde{D}_{11}\tilde{U}_{13}\right) + R_{33}^1 \ , \ \hat{S}_{33}^2 = A_{33}^2 - \left(\tilde{L}_{32}\tilde{D}_{22}\tilde{U}_{23}\right) + R_{33}^2 \ .$$

However, the construction of (4) requires communication before the addition of these two blocks can be computed

$$R_{33} \approx R_{33}^1 + R_{33}^2 \ \rightarrow \ \hat{S}_{33} \approx \hat{S}_{33}^1 + \hat{S}_{33}^2 \ . \tag{6}$$

Finally, the sequential ILU factorization of $\hat{S}_{33}$ completes the parallel approximate factorization of $\Pi^T A \Pi$.

To expose a higher degree of parallelism, we need to identify a larger number of independent diagonal blocks. We can do this by applying a permutation similar to Π on the two leading blocks, and then reordering and renaming the blocks,

$$\left[\begin{array}{ccc|ccc|c} \hat{A}_{11} & 0 & \hat{A}_{13} & 0 & 0 & 0 & * \\ 0 & \hat{A}_{22} & \hat{A}_{23} & 0 & 0 & 0 & * \\ \hat{A}_{31} & \hat{A}_{32} & \hat{A}_{33} & 0 & 0 & 0 & * \\ \hline 0 & 0 & 0 & \bar{A}_{11} & 0 & \bar{A}_{13} & * \\ 0 & 0 & 0 & 0 & \bar{A}_{22} & \bar{A}_{23} & * \\ 0 & 0 & 0 & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} & * \\ \hline * & * & * & * & * & * & * \end{array}\right] \rightarrow \left[\begin{array}{cccc|cc|c} A_{11} & 0 & 0 & 0 & A_{15} & 0 & A_{17} \\ 0 & A_{22} & 0 & 0 & A_{25} & 0 & A_{27} \\ 0 & 0 & A_{33} & 0 & 0 & A_{36} & A_{37} \\ 0 & 0 & 0 & A_{44} & 0 & A_{46} & A_{47} \\ \hline A_{51} & A_{52} & 0 & 0 & A_{55} & 0 & A_{57} \\ 0 & 0 & A_{63} & A_{64} & 0 & A_{66} & A_{67} \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right] . \tag{7}$$

Figure 2 illustrates the dependency tree for the factorization of the diagonal blocks in the right-hand side of (7); there, the nodes which lie at the same height can be factored in parallel and the edges of the graph define the dependencies between the diagonal blocks, other words, the order in which the blocks of the matrix have to be processed. We identify three classes of nodes in the figure:

1. The **Leaf nodes**, which are responsible for the factorization of the four leading diagonal blocks in parallel.
2. The **Intermediate nodes**, which factorize in parallel the next two intermediate diagonal blocks, A_{55} and A_{66}. These blocks cannot be factorized unless the leading diagonal blocks corresponding to its children have been already factorized, i.e., A_{11} - A_{22} and A_{33} - A_{44} respectively.
3. The **Root node**, which sequentially factorizes the last diagonal block, A_{77}. This approximation can be only computed when all the preceding diagonal blocks have been processed.

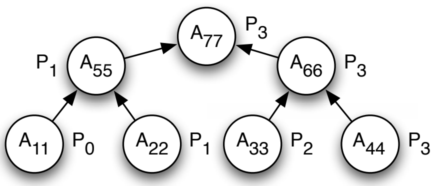


Fig. 2. Dependency tree of the diagonal blocks

The parallel computation of the preconditioner also commences by disassembling A, with one submatrix for each leaf node,

$$\left[\begin{array}{c|c|c} A_{11} & \multicolumn{2}{c}{A_{15}\ \ A_{17}} \\ \hline A_{51} & A^1_{55} & A^1_{57} \\ \cline{2-3} A_{71} & A^1_{75} & A^1_{77} \end{array}\right], \left[\begin{array}{c|c|c} A_{22} & \multicolumn{2}{c}{A_{25}\ \ A_{27}} \\ \hline A_{52} & A^2_{55} & A^2_{57} \\ \cline{2-3} A_{72} & A^2_{75} & A^2_{77} \end{array}\right], \left[\begin{array}{c|c|c} A_{33} & \multicolumn{2}{c}{A_{36}\ \ A_{37}} \\ \hline A_{63} & A^3_{66} & A^3_{67} \\ \cline{2-3} A_{73} & A^3_{76} & A^3_{77} \end{array}\right], \left[\begin{array}{c|c|c} A_{44} & \multicolumn{2}{c}{A_{46}\ \ A_{47}} \\ \hline A_{64} & A^4_{66} & A^4_{67} \\ \cline{2-3} A_{74} & A^4_{76} & A^4_{77} \end{array}\right],$$

$$A_{55} = A^1_{55} + A^2_{55}\ ,\ A_{66} = A^3_{66} + A^4_{66}\ ,\ A_{77} = A^1_{77} + A^2_{77}\ + A^3_{77} + A^4_{77}\ .$$

Thus, the partial factorization of these submatrices can be computed concurrently. For example, computing the ILU of A_{11}, we obtain the following partial approximation

$$\left[\begin{array}{c|c|c} \tilde{L}_{11} & \multicolumn{2}{c}{0\ \ \ \ 0} \\ \hline \tilde{L}_{51} & \mathrm{I} & 0 \\ \cline{2-3} \tilde{L}_{71} & 0 & \mathrm{I} \end{array}\right] \left[\begin{array}{c|c|c} \tilde{D}_{11} & \multicolumn{2}{c}{0\ \ \ \ 0} \\ \hline 0 & \tilde{S}^1_{55} & \tilde{S}^1_{57} \\ \cline{2-3} 0 & \tilde{S}^1_{75} & \tilde{S}^1_{77} \end{array}\right] \left[\begin{array}{c|c|c} \tilde{U}_{11} & \multicolumn{2}{c}{\tilde{U}_{15}\ \ \tilde{U}_{17}} \\ \hline 0 & \mathrm{I} & 0 \\ \cline{2-3} 0 & 0 & \mathrm{I} \end{array}\right] + \left[\begin{array}{c|c|c} R_{11} & \multicolumn{2}{c}{R_{15}\ \ R_{17}} \\ \hline R_{51} & 0 & 0 \\ \cline{2-3} R_{71} & 0 & 0 \end{array}\right].$$

When the partial factorizations of all the leaf nodes are completed, the processes in charge of these tasks send the local Schur complement to the corresponding intermediate node, which then accumulates them to continue the process,

$$\left[\begin{array}{c|c}\hat{S}^1_{55} & \hat{S}^1_{57} \\ \hline \hat{S}^1_{75} & \hat{S}^1_{77}\end{array}\right] + \left[\begin{array}{c|c}\hat{S}^2_{55} & \hat{S}^2_{57} \\ \hline \hat{S}^2_{75} & \hat{S}^2_{77}\end{array}\right] = \left[\begin{array}{c|c}\hat{S}_{55} & \hat{S}_{57} \\ \hline \hat{S}_{75} & \hat{S}^{12}_{77}\end{array}\right] ,$$

$$\left[\begin{array}{c|c}\hat{S}^3_{66} & \hat{S}^3_{67} \\ \hline \hat{S}^3_{76} & \hat{S}^3_{77}\end{array}\right] + \left[\begin{array}{c|c}\hat{S}^4_{66} & \hat{S}^4_{67} \\ \hline \hat{S}^4_{76} & \hat{S}^4_{77}\end{array}\right] = \left[\begin{array}{c|c}\hat{S}_{66} & \hat{S}_{67} \\ \hline \hat{S}_{76} & \hat{S}^{34}_{77}\end{array}\right] .$$

The matrix resulting from assembling these two submatrices presents the same structure as that defined in (3)

$$\left[\begin{array}{c|c}\hat{S}_{55} & \hat{S}_{57} \\ \hline \hat{S}_{75} & \hat{S}^{12}_{77}\end{array}\right] \oplus \left[\begin{array}{c|c}\hat{S}_{66} & \hat{S}_{67} \\ \hline \hat{S}_{76} & \hat{S}^{34}_{77}\end{array}\right] = \left[\begin{array}{cc|c}S_{55} & 0 & S_{57} \\ 0 & S_{66} & S_{67} \\ \hline S_{75} & S_{76} & S_{77}\end{array}\right] , \quad S_{77} = S^{12}_{77} + S^{34}_{77} , \qquad (8)$$

and the process continues as described above.

This procedure can be generalized to obtain the same number of leaf nodes as process/processors, so that the ILU factorization of each leaf node can be mapped to a specific process. The performance of the parallel computation of the preconditioner will be improved if the load is balanced among the leaf nodes is optimum and the communication time is reduced. Mapping nodes to processors as in Fig. 2, yields a high degree of parallelism if the computational cost is concentrated on the leaf nodes of the dependency tree, and the cost is evenly distributed among the leaf nodes. In practice, it is not possible to know the cost of the multilevel ILU factorization *a priori*, but we can estimate this cost from the number of rows/columns and nonzeros per node. Therefore, we must find a permutation of A that minimizes the number of rows/columns of non-leaf nodes, while simultaneously balancing those of the leaf nodes.

The MLND (*Multilevel Nested Dissection*) algorithm [8] is a recursive procedure that, at each step, splits a graph into two disjoint subgraphs connected by the nodes included in the separator. Some conditions hold for the result of this computation; e.g., the size of the separator may feature some minimum criteria and/or the size of the subgraphs can be made equal up to a certain degree. The recursion can be continued on the subgraphs until their size is relatively small. By viewing a sparse matrix as a graph, this procedure generates a reordered matrix similar to that shown in left-hand side of (7).

There exist several implementations of MLND (e.g., in METIS[2], SCOTCH[3]), which usually lead to balanced elimination trees that exhibit a higher degree of concurrency. There also exist parallel versions of these packages (ParMETIS [9] and PT-SCOTCH [6]) that exploit several types of parallelism during the computation of the permutation. To illustrate the quality of current partitioning packages, we applied ParMETIS to a benchmark matrix of dimension 10^6 (see Sect. 4 for details), in order to generate a tree with the structure shown in Fig. 2; the result is shown in Fig. 3.

[2] http://glaros.dtc.umn.edu/gkhome/views/metis

[3] http://www.labri.fr/perso/pelegrin/scotch

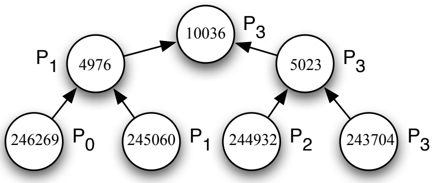


Fig. 3. Number of rows/columns of the dependency tree corresponding to a matrix of size 10^6 arising from the finite-difference discretization of the Laplace 3D PDE

The described parallel algorithm forces a certain order of elimination to expose a high degree of concurrence during the approximate factorization of the reordered matrix. In particular, the leaf nodes first factorize the leading diagonal blocks, and then the corresponding Schur complements are received and accumulated by the corresponding intermediate nodes. This idea is recursively applied till the root node is reached. However, this parallel process does not control the growth of the norms of the inverse triangular factors during the computation of the preconditioner. In order to accommodate the inverse-based preconditioning approach, we first restrict the **preprocessing** and **factorization** steps to the leading block of each submatrix, so that only the rows/columns of this block are preprocessed, and rejected rows/columns are moved to the bottom/right-end of the leading block. Thus, e.g., the **factorization** step applied to the left-most submatrix in (5) results in the following partial approximation,

$$\left[\begin{array}{c|c} P_{11} & 0 \\ \hline 0 & I \end{array}\right]^T \left[\begin{array}{c|c} A_{11} & A_{13} \\ \hline A_{31} & A_{33}^1 \end{array}\right] \left[\begin{array}{c|c} P_{11} & 0 \\ \hline 0 & I \end{array}\right] = \left[\begin{array}{cc|c} B_{11} & F_{11} & F_{13} \\ E_{11} & C_{11} & C_{13} \\ \hline E_{31} & C_{31} & A_{33}^1 \end{array}\right] =$$

$$\left[\begin{array}{cc|c} \tilde{L}_{B,11} & 0 & 0 \\ \tilde{L}_{E,11} & I & 0 \\ \hline \tilde{L}_{E,31} & 0 & I \end{array}\right] \left[\begin{array}{cc|c} \tilde{D}_{B,11} & 0 & 0 \\ 0 & \tilde{S}_{C,11} & \tilde{S}_{C,13} \\ \hline 0 & \tilde{S}_{C,31} & \tilde{S}_{C,33}^1 \end{array}\right] \left[\begin{array}{cc|c} \tilde{U}_{B,11} & \tilde{U}_{F,11} & \tilde{U}_{F,13} \\ 0 & I & 0 \\ \hline 0 & 0 & I \end{array}\right] + \left[\begin{array}{cc|c} R_{B,11} & R_{F,11} & R_{F,13} \\ R_{E,11} & 0 & 0 \\ \hline R_{E,31} & 0 & 0 \end{array}\right],$$

and then the multilevel process is recursively applied on the matrix

$$\left[\begin{array}{c|c} \hat{S}_{C,11} & \hat{S}_{C,13} \\ \hline \hat{S}_{C,31} & \hat{S}_{C,33}^1 \end{array}\right] = \left[\begin{array}{c|c} \tilde{S}_{C,11} & \tilde{S}_{C,13} \\ \hline \tilde{S}_{C,31} & \tilde{S}_{C,33}^1 \end{array}\right] + \left[\begin{array}{c|c} R_{C,11} & R_{C,13} \\ \hline R_{C,31} & R_{C,33}^1 \end{array}\right] . \tag{9}$$

Figure 4 illustrates the computation of the partial ILU factorization of A_{22} in (7), computed by a single leaf node of the dependency tree. The **restarting** step is also adapted, because it recursively applies the restricted steps until $\hat{S}_{C,11}$ in (9) is void or "sufficiently small". Finally, the intermediate node assembles the approximate Schur complement computed by its children as:

$$\begin{bmatrix} \hat{S}_{C,11} & 0 & \hat{S}_{C,13} \\ 0 & \hat{S}_{C,22} & \hat{S}_{C,23} \\ \hat{S}_{C,31} & \hat{S}_{C,32} & \hat{S}_{C,33} \end{bmatrix} = \left[\begin{array}{c|c} \hat{S}_{C,11} & \hat{S}_{C,13} \\ \hline \hat{S}_{C,31} & \hat{S}_{C,33}^1 \end{array}\right] \oplus \left[\begin{array}{c|c} \hat{S}_{C,22} & \hat{S}_{C,23} \\ \hline \hat{S}_{C,32} & \hat{S}_{C,33}^1 \end{array}\right] .$$

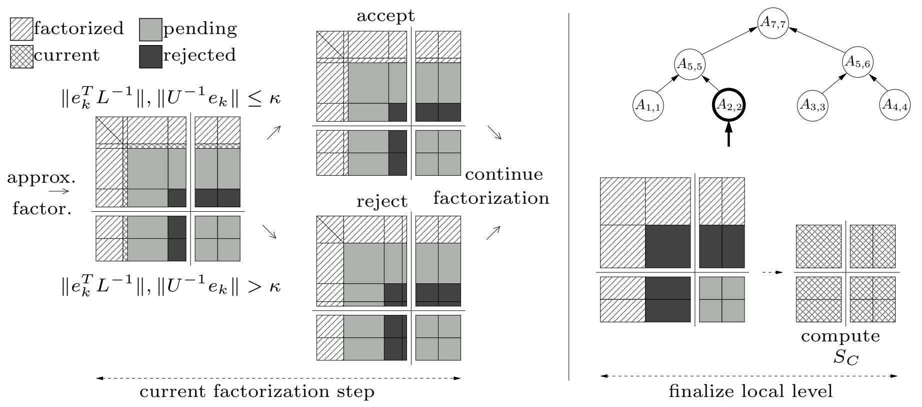


Fig. 4. Local incomplete factorization computed by a single node of the task tree

Unlike ([6]), the parent node must now consider the pivots rejected by its children, which are incorporated into the submatrix constructed by the former. This is illustrated in Tab. [1], which reports the structure of the parallel ILU factorization corresponding to the problem/tree in Fig. [3]. We can observe that the leaves and the root have to build several partial inverse-based ILUs. Moreover, the number of levels and the number of rejected pivots in each leaf can be different.

Table 1. Number of accepted pivots by inverse-based pivoting in each level, and number of rejected pivots which are pushed upwards for the tree in Fig. [3]

	Leaves						Intermediates		Root	
Proc	level1	level2	level3	level4	level5	rejected	level1	rejected	level1	level2
P_0	184977	33739	19065	8462		26				
P_1	184073	33529	17952	9291	215	0	5001	1		
P_2	183783	33123	18925	9070		31				
P_3	182922	33071	18608	9074		29	5083	0	9905	132
	735755	133462	74550	35897	215	86	10084	1	9905	132
	979879					86	10084	1	10037	

4 Experimental Results and Conclusions

All experiments in this section were obtained using IEEE double-precision arithmetic, on a cluster interconnected by an InfiniBand network with 4 nodes. Each node contains two Intel QuadCore *Nehalem* processors (8 cores), at 2.27 GHz and with 24 Gbytes of RAM. We used the OpenMPI message-passing library tuned for the InfiniBand network. The dependency tree was computed using ParMETIS (routine `ParMETIS_V3_NodeND` with defaults parameters).

We consider a standard benchmark problem for the solution of PDEs: the Laplacian equation $-\Delta u = f$ in a $3D$ unit cube $\Omega = [0,1]^3$ with Dirichlet boundary conditions $u = g$ on $\partial\Omega$. Although this regular problem is known to be best-suited for multigrid methods, we have selected it due to its large dimension and applicability. The problem is discretized using a uniform mesh of size $h = \frac{1}{N+1}$. The computational domain Ω is replaced by a grid $\Omega_h = \{(x_i, y_j, z_k) = (ih, jh, kh)|\ i, j, k = 1, \ldots, N\}$, and the differential operator is replaced by finite differences

$$\begin{aligned}\Delta u\left(x_i, y_j, z_k\right) \approx & \tfrac{1}{h^2}\left(- u_{i-1,j,k} - u_{i,j-1,k} - u_{i,j,k-1} + \right.\\ & \left. 6u_{i,j,k} - u_{i+1,j,k} - u_{i,j+1,k} - u_{i,j,k+1}\right)\ ,\end{aligned}$$

where $u_{ijk} \approx u(x_i, y_j, z_k)$. Because of the Dirichlet boundary conditions, any unknown u_{ijk} such that $i, j, k \in \{0, N+1\}$ is explicitly available and becomes part of the right-hand side vector. The resulting linear system $Au = b$ presents a sparse symmetric positive definite (SPD) coefficient matrix with seven nonzero elements per row, and $n = N^3$ unknowns. We set N=100, 126, 159, 200, and 252 in our experiments, which results in five SPD linear systems with roughly n =1, 2, 4, 8, and 16 millions of unknowns. We also consider four large-scale SPD benchmark matrices (bmwcra_1, af_shell3, ldoor and G3_circuit) from the UF sparse matrix collection[4]. We have selected these to evaluate the performance of our parallelization approach with irregularly structured problems.

Figure 5 shows the speed-up of the parallel ILU preconditioner for the different matrices, number of nodes, and number of cores per node. The total number of cores equals the product of the number of nodes and cores per node. The dependency tree is generated so that its number of leaves equals the number of cores. Thus, those combinations of "number of nodes-cores per node" which result in the same number of cores, utilize the same dependency tree to exploit parallelism, but a different mapping of MPI processes to cores. From this figure we can conclude that the parallel ILUPACK implementation exhibits reasonable strong scaling, as the parallel efficiency drops moderately as the number of cores grows. Moreover, the performance almost remains constant when the same number of cores are involved in the parallel computation, revealing a mild influence of the distribution of the cores among the nodes; only for the largest matrices, a small performance reduction is observed when using eight cores per node (see, e.g., for $N = 252^3$, drop from 4-4 to 2-8). We believe that this is due, to some extent, to contention caused by the fully utilization of the resources in a node.

At first glance, it might appear that the factor that contributes more to the drop in efficiency observed in Fig. 5 is the lack of parallelism in the higher levels of the dependency tree. Although this factor can (asymptotically) limit the strong scalability of our approach, in practice, we observed a moderate reduction of the computational cost concentrated on the leaves of the tree as the number of cores increases, so that even a single type of parallelism (e.g., tree parallelism) can provide a reasonable degree of parallelism for a multilevel ILU preconditioner. Table 2 reports the percentage of the aggregated computational cost which is concentrated on the leaves and the non-leaf nodes of the tree vs. the number of cores. This cost is defined as the aggregation of the computational cost of all

[4] http://www.cise.ufl.edu/research/sparse/matrices

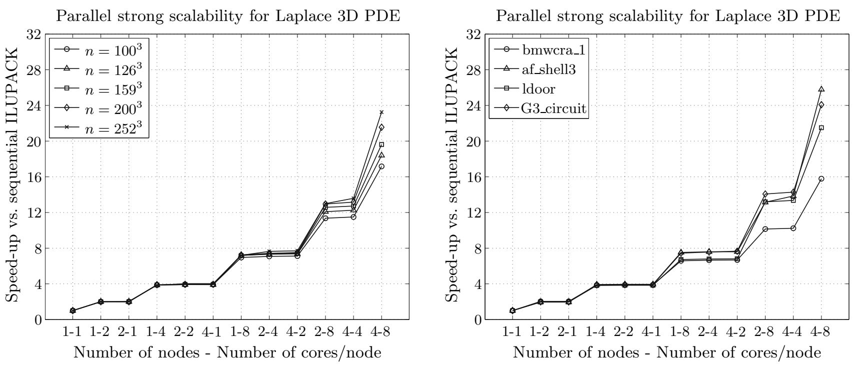


Fig. 5. Speed-up of the parallel multilevel ILU algorithm for the five Laplace 3D PDE matrices (left) and the four matrices from the UF sparse matrix collection (right)

tasks in the tree, so that overheads associated with parallelism (e.g., communication or idling) were not accounted in Tab. 2. The experiment clearly reveals a moderate reduction of the computational cost concentrated on the leaves with the number of cores for all matrices except bmwcra_1 (the smallest test matrix). For bmwcra_1, this reduction does not solely justify the performance drop observed in Fig. 5.

Table 2. Percentage of the aggregated computational cost which is concentrated on the leaves (left) and non-leaf nodes (right) of the dependency tree

matrix	1 core	2 cores	4 cores	8 cores	16 cores	32 cores
$n = 100^3$	(100.0,0.0)	(99.76,0.24)	(99.35,0.65)	(98.36,1.64)	(96.75,3.25)	(93.87,6.13)
$n = 126^3$	(100.0,0.0)	(99.84,0.16)	(99.46,0.54)	(98.71,1.29)	(97.25,2.75)	(95.20,4.80)
$n = 159^3$	(100.0,0.0)	(99.81,0.19)	(99.58,0.42)	(98.95,1.05)	(97.88,2.12)	(96.11,3.89)
$n = 200^3$	(100.0,0.0)	(99.90,0.10)	(99.63,0.37)	(99.15,0.85)	(98.24,1.76)	(96.93,3.07)
$n = 252^3$	(100.0,0.0)	(99.89,0.11)	(99.71,0.29)	(99.32,0.68)	(98.61,1.39)	(97.52,2.48)
bmwcra_1	(100.0,0.0)	(99.99,0.01)	(99.46,0.54)	(97.83,2.17)	(94.69,5.31)	(90.51,9.49)
af_shell3	(100.0,0.0)	(99.87,0.13)	(99.86,0.14)	(99.66,0.34)	(99.18,0.82)	(97.97,2.03)
ldoor	(100.0,0.0)	(99.96,0.04)	(99.74,0.26)	(99.27,0.73)	(98.76,1.24)	(97.58,2.42)
G3_circuit	(100.0,0.0)	(99.99,0.01)	(99.75,0.25)	(99.63,0.37)	(99.12,0.88)	(98.06,1.94)

We believe that the main key factor for the performance drop observed in Fig. 5 is the parallel overhead due to idle MPI processes, which in turn is caused by an unbalanced distribution of the computational work associated with the leaf nodes in the tree. Table 3 reports, for a parallel execution with 32 cores, how much computational time is concentrated on the most and least expensive computational leaves; this is expressed as a percentage relative to the parallel execution time in the rows labeled as $leaf_{max}$ and $leaf_{min}$. This table also reports the aggregated parallel overhead relative to the aggregated parallel execution time, with the latter defined as the product of the parallel execution time and the number of cores. The aggregated parallel overhead was estimated by substracting the aggregated parallel execution time and the aggregation of the computational costs of all tasks in the tree (i.e., the useful computation). The table clearly correlates load unbalance in the computation of the leaves and parallel overhead; see e.g., values for bmwcra_1. Future developments will require additional techniques to improve load balancing in the computation of the leaves.

Table 3. Amount of computational time concentrated on the most and least computationally expensive leaves, and relative aggregated parallel overhead

	100^3	126^3	159^3	200^3	252^3	bmwcra_1	af_shell3	ldoor	G3_circuit
$leaf_{max}$	65.28	68.61	73.28	77.75	80.80	60.27	87.67	82.50	84.38
$leaf_{min}$	50.00	53.28	58.40	64.03	68.98	27.40	53.42	62.50	71.88
overhead	39.15	35.83	32.18	26.47	23.01	54.88	36.77	26.80	19.63

Acknowledgments. This research has been supported by the CICYT project TIN2008-06570-C04-01 and by the Fundación Caixa-Castelló/Bancaixa and UJI project P1-1B2009-31.

References

1. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Parallelization of multilevel preconditioners constructed from inverse-based ILUs on shared-memory multiprocessors. In: Parallel Computing: Architectures, Algorithms and Applications. Advances in Parallel Computing, vol. 38, pp. 287–294. NIC (2007)
2. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Design, Tuning and Evaluation of Parallel Multilevel ILU Preconditioners. In: Palma, J.M.L.M., Amestoy, P.R., Daydé, M., Mattoso, M., Lopes, J.C. (eds.) VECPAR 2008. LNCS, vol. 5336, pp. 314–327. Springer, Heidelberg (2008)
3. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Exploiting thread-level parallelism in the iterative solution of sparse linear systems. Parallel Computing (2010) (in press, accepted manuscript)
4. Bollhöfer, M., Grote, M.J., Schenk, O.: Algebraic multilevel preconditioner for the helmholtz equation in heterogeneous media. SIAM Journal on Scientific Computing 31(5), 3781–3805 (2009)
5. Bollhöfer, M., Saad, Y.: Multilevel preconditioners constructed from inverse–based ILUs. SIAM J. Sci. Comput. 27(5), 1627–1650 (2006); special issue on the 8–th Copper Mountain Conference on Iterative Methods
6. Chevalier, C., Pellegrini, F.: PT-SCOTCH: A tool for efficient parallel graph ordering. Parallel Comput. 34(6-8), 318–331 (2008)
7. Davis, T.A.: Direct Methods for Sparse Linear Systems. SIAM (2006)
8. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. 20(1), 359–392 (1998)
9. Karypis, G., Kumar, V.: A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. J. Parallel Distrib. Comput. 48(1), 71–95 (1998)
10. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM Publications (2003)

CUDA 2D Stencil Computations for the Jacobi Method

José María Cecilia[1], José Manuel García[1], and Manuel Ujaldón[2]

[1] Computer Engineering and Technology Department, University of Murcia, Spain
{chema,jmgarcia}@ditec.um.es

[2] Computer Architecture Department, University of Malaga, Spain
ujaldon@uma.es

Abstract. We are witnessing the consolidation of the GPUs streaming paradigm in parallel computing. This paper explores stencil operations in CUDA to optimize on GPUs the Jacobi method for solving Laplace's differential equation. The code keeps constant the access pattern through a large number of loop iterations, that way being representative of a wide set of iterative linear algebra algorithms. Optimizations are focused on data parallelism, threads deployment and the GPU memory hierarchy, whose management is explicit by the CUDA programmer. Experimental results are shown on Nvidia Teslas C870 and C1060 GPUs and compared to a counterpart version optimized on a quadcore Intel CPU. The speed-up factor for our set of GPU optimizations reaches 3-4x and the execution times defeat those of the CPU by a wide margin, also showing great scalability when moving towards a more sophisticated GPU architecture and/or more demanding problem sizes.

Keywords: CUDA, GPGPU, Stencil Computation, Parallel Numerical Algorithms.

1 Introduction

The newest versions of programmable GPUs provide a compelling alternative to traditional CPUs, delivering extremely high floating point performance for scientific applications which fit their architectural idiosyncrasies [11]. Whereas hybrid clusters of SMPs where once the realm only of costly supercomputers, GPUs now turn nodes with this added complexity into a commodity. This fact has attracted GPUs to researchers in many fields [7], among which numerical methods constitute one of the most prolific ones.

A large number of numerical computing techniques use large multidimensional arrays as its primary data structure, which bring us a good opportunity to benefit from Single Instruction Multiple Data (SIMD) parallelism. In addition, such algorithms normally have an iterative nature, that is, they tend to converge through a number of steps towards the final solution until certain condition is fulfilled. Usually, parallelism is exploited within an iteration, where each processor can work on a different subsection of the global data to produce an output which is partially communicated to other processors. Then, data is rearranged to become the input to the next iteration, which prevents from parallelizing consecutive iterations.

In order to exploit SIMD parallelism in general-purpose computing, both Nvidia and AMD have released software components which provide simpler access to GPU computing power than that realized by treating the GPU as a traditional graphics processor. CUDA [3] is Nvidia's solution as a simple block-based API for SIMD programming; AMD's solution is called Stream Computing. We choose CUDA to program the GPU for being more popular and providing more mechanisms to optimize general-purpose applications. Nevertheless, both models are expected to converge in OpenCL [8] as a higher level standard shared by a wide set of GPU models. Among them, we have some high-end graphics cards aimed specifically at the scientific General Purpose GPU (GPGPU) computing market: the Tesla products [12] are from NVIDIA, and Firestream [6] is AMD's product line.

Stencil computations are those in which each computing node in a multi-dimensional grid is updated with weighted values contributed by neighboring nodes. These neighbors comprise the stencil, and multiple iterations across the array are usually required to achieve convergence or to simulate time steps. Among those stencil codes, our work focuses on the Jacobi method to solve Laplace's differential equation, which is a priori not an ideal partner for GPUs due to its low arithmetic intensity. We overcome this drawback by exploring a wide set of optimizations paths in CUDA: threads deployment, different uses of the shared memory, the effect of larger 2D stencils, the floating-point accuracy for single and double precision, and finally the scalability for our solution versus a multicore CPU. Our best version reaches a speed-up factor of 3-4x over a non-optimized GPU version, also showing great scalability when moving towards a more sophisticated GPU architecture and/or demanding problem sizes.

The rest of the paper is organized as follows. Section 2 explains the Jacobi method. Section 3 briefly introduces the specifics of the GPU programming with CUDA. Section 4 outlines our CUDA implementation, exposes the execution times and analyzes the results. Finally, Section 5 reviews some related work and Section 6 concludes.

2 The Jacobi Method

Jacobi [9] is a popular algorithm for solving Laplace's differential equation on a square domain, regularly discretized [5]. The kernel (see Figure 1) is based on the following idea: Let us consider a body represented by a 2D array of particles, each with an initial value of temperature. This body is in contact with a fixed value of temperature on the four boundaries, and Laplace's equation is solved for all internal points to determine their temperature as the average at all of the five stencil nodes (see Figure 1).

Taking this task as the computational core, a number of iterations are performed over the data to recompute average temperatures repeatedly, and the values gradually converge to a finer solution until the desired accuracy is reached. For experimental purposes, we consider a constant number of 4096 iterations. Note that iterations have to be serialized due to carried-loop dependencies, but parallelism is enabled within iterations because the computation at each particle

```
for (k=0; k<4096; k++) {
 for (i=0; i<N; i++)
  for (j=0; j<N j++)
   T[i][j] = 0.2*(A[i][j]+A[i-1][j]+
             A[i+1][j]+A[i][j-1]+A[i][j+1]);
 for (i=0; i<N; i++)
  for (j=0; j<N j++)
   A[i][j] = T[i][j]; }
```

Fig. 1. Jacobi's solver pseudocode

Table 1. GPU features for the Tesla cards used during our experimental analysis

Hardware feature	**Tesla C870**	**Tesla C1060**
Compute Capabilities	1.0	1.3
Number of streaming processor	128	240
Frequency of streaming processors	1.35 GHz	1.30 GHz
Global memory size and type	1.5 Gbytes GDDR3	4 Gbytes GDDR3
Global memory width and speed	384 bits @ 2x800 MHz	512 bits @ 2x800 MHz
Global memory bandwidth	76.8 Gbytes/sc.	102 Gbytes/sc.

is independent. Thus, the workload depends more on the number of iterations, whereas the amount of parallelism that can be extracted from the code relies more on the size of the 2D input matrix.

At the end, the Laplace equation, once discretized, leads to our Jacobi kernel. This kernel consists of three nested loops, with the two innermost being of length N (which is the matrix dimension), and the outermost being of length k (the number of iterations) - see Figure 1. The algorithm complexity can be expressed as $O(k \cdot N^2)$.

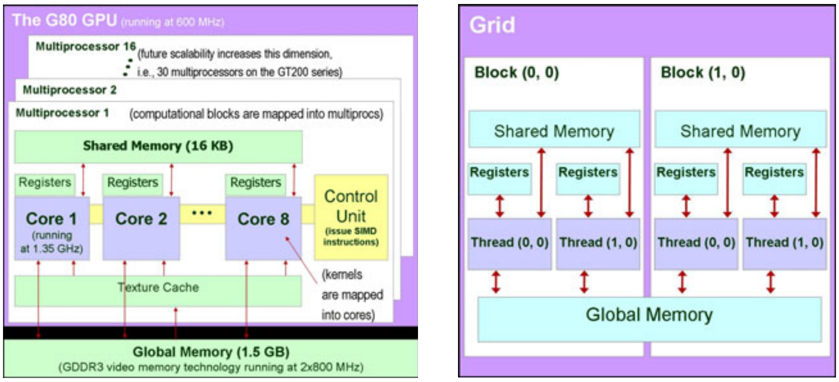
3 CUDA

CUDA (Compute Unified Device Architecture) [3] is a programming interface and set of supported hardware to enable general purpose computation on Nvidia GPUs and leverage special hardware features not visible to more traditional graphics-based GPU programming, such as small cache memories, explicit massive parallelism and lightweight context switch between threads.

The Tesla C870 and C1060 GPUs are respectively based on the G80 and GT200 architectures, whose major features are depicted in Table 1.

Each GPU multiprocessor can run a variable number of threads, and the local resources are shared among them. In any given cycle, each core in a multiprocessor executes the same instruction on different data based on its `threadId`, and communication between multiprocessors is performed through global memory (see Figure 2).

In the CUDA programming model, a program is decomposed into **blocks**, groups of threads runnning in parallel within a single multiprocessor where they share registers, memory and other multiprocessor's resources (see Figure 2.a).


The G80 GPU (running at 600 MHz)
Multiprocessor 16 (future scalability increases this dimension, i.e., 30 multiprocessors on the GT200 series)
Multiprocessor 2
Multiprocessor 1 (computational blocks are mapped into multiprocs)
Shared Memory (16 KB)
Registers
Core 1 (running at 1.35 GHz)
Core 2
Core 8
Control Unit (issue SIMD instructions)
Texture Cache
(kernels are mapped into cores)
Global Memory (1.5 GB) (GDDR3 video memory technology running at 2x800 MHz)
Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Registers
Thread (0, 0)
Thread (1, 0)
Global Memory

(a) Hardware interface (b) Programming model

Fig. 2. CUDA highlights for the G80 core used in the Tesla C870 board

Table 2. Major hardware and software limitations with CUDA. Constraints are listed for the G80 and GT200 GPUs, the ones inside our Teslas C870 and C1060 boards.

	G80	**GT200**
CUDA Compute Capabilities	1.0 and 1.1	1.2 and 1.3
Multiprocessors per GPU	16	30
Processors / Multiprocessor	8	8
32-bit registers / Multiprocessor	8192	16384
Shared Memory / Multiprocessor	16 KB	16 KB.
Threads / Warp	32	32
Thread Blocks / Multiprocessor	8	8
Threads / Block	512	512
Threads / Multiprocessor	768	1024

A **kernel** is a code function compiled to the instruction set of the device and executed by all of its threads. Threads run on different processors of the multiprocessors sharing the same executable and global address space, though they may not follow exactly the same path of execution. A kernel is organized into a **grid** as a set of **thread blocks** explicitly defined by the application developer and executed on a single multiprocessor.

Threads placed in different blocks from the same grid cannot communicate. This tradeoff between parallelism and thread resources must be wisely solved by the programmer to maximize execution efficiency on a certain architecture given its limitations. These limitations are listed for the case of our Tesla boards in Table 2.

3.1 Memory Optimizations

During our CUDA implementation, attention must be paid to how threads access the 16 banks of shared memory, since only when the data resides in different banks can all of the available ALU bandwidth truly be used. Each bank only

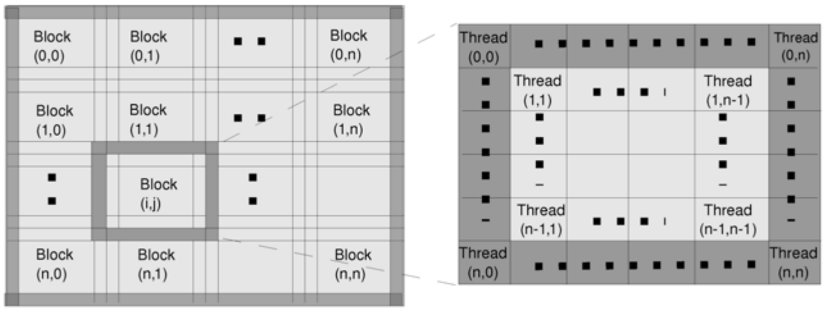


Fig. 3. Threads deployment for the CUDA parallelization strategy

Table 3. Execution times (in seconds) for our Jacobi baseline implementation

Matrix size	(threads deployment per block) (14x14)	(16x16)	(18x18)	(20x20)
1024^2	13.50	**13.16**	13.70	14.13
2048^2	52.73	**50.74**	52.57	52.43
4096^2	206.99	**203.35**	207.06	211.28
8192^2	843.55	**850.18**	899.46	852.26

(a) Tesla C870

Matrix size	(threads deployment per block) (14x14)	(16x16)	(18x18)	(20x20)
1024^2	3.27	**2.34**	3.24	3.071
2048^2	12.73	**8.72**	11.88	11.594
4096^2	50.36	**34.60**	46.28	44.402
8192^2	211.03	**144.02**	211.16	177.795

(b) Tesla C1060

supports one memory access at a time; simultaneous memory bank accesses are serialized, stalling the rest of the multiprocessor's running threads until their operands arrive. The use of shared memory is explicit within a thread, which allows the developer to solve bank conflicts wisely.

Another critical issue related to memory performance is *data coalescing*. A coalesced access involves a contiguous region of global memory where the starting address must be a multiple of region size and the k^{th} thread in a half-warp must access the k^{th} element in a block being read. This way, the hardware can serve completely two coalesced accesses per clock cycle, which maximizes memory bandwidth. It is programmer's responsability to organize memory accesses in such a way.

4 Implementation

4.1 Optimal Threads Deployment

Figure 3 shows the threads deployment for the parallelization of the Jacobi method using CUDA. Blocks and threads are deployed following a 2D layout to balance the decomposition of the computational domain on each matrix dimension. Adjacent blocks share data placed on boundaries, and each thread within a block is responsible for updating a single element on each iteration.

Among all possibilities concerning an input matrix of size NxN and a squared block of BxB threads, we have selected $N = 1024, 2048, 4096, 8192$ and $B =$

$14, 16, 18, 20$ for representing good choices after a preliminary survey. Table 3 shows that 16x16 constitutes the optimal number of threads per block, with a penalty around 5-10% in terms of the execution time for the other three cases. All remaining squared alternatives for the matrix of threads led to worse results.

4.2 Shared Memory Optimizations

Our CUDA baseline implementation does not use shared memory. All threads access the device memory to read an element together with its four matrix neighbors and later update its value with the average. From this departure point, three optimizations were incrementally developed:

1. Each input element read from device memory is stored into shared memory by the owner thread prior to the actual computation, and the output result is written back into device memory. The kernel length increases from 34 to 78 instructions, but this variant notably reduces the pressure on device memory, just requiring 18 GB/s of memory bandwidth compared to 122 GB/s in our baseline version.

   On the Tesla C870, 99.68% of the memory accesses to device memory are non-coalesced when running the code using CUDA Compute Capabilities 1.0 (CCC 1.0). On the Tesla C1060, things are very different, because this device uses coalescing rules based on CCC 1.3, leading to a 100% of coalesced accesses. Benefits are therefore larger on the Tesla C1060 GPU.
2. Our second optimization uses an internal register as substitute of the shared memory cell on each thread. This leads to a more homogeneous behavior of threads, which is exploited to omit certain __syncthreads() calls at block level, and also enables data prefetching as a positive side-effect. Nevertheless, these enhancements behave similarly on CCC 1.0 and CCC 1.3, and consequently are translated into minor improvements in the overall execution time.
3. The third optimization reduces the relative amount of shared memory used by each thread so that a block of threads can work with a greater data area. In other words, the block uses the same amount of shared memory, say 16x16, but with those data now is capable of managing a tile of 16x32 data in two iterations. Doing so, we can maximize the parallelism allowed on each CUDA platform. In CCC 1.0, the maximum number of threads assigned to a multiprocessor is 768, whereas in CCC 1.3 this number reaches 1024. In the first case, the amount of shared memory used by each block was reduced until 4120 bytes, so that we can assign three blocks of 256 threads to each multiprocessor. In the second case, we reduced this size even more until we could assign four blocks of 256 threads, which increases parallelism leading to slightly better results.

Table 4.a shows the execution times for all these versions on a Tesla C870 and Table 4.b does the same for the Tesla C1060 GPU. An average speed-up factor of 3.5x is roughly attained.

Table 4. Execution times (in seconds) for our Jacobi implementation using different optimizations. Between parenthesis, we show the speed-up factor versus the baseline implementation on the same platform. Threads deployment is 16x16 for all cases.

Matrix	**Baseline:**	**Optimiz. 1**	**Optimizs. 1+2**	**Optimizs. 1+2+3**
1024^2	13.16	3.77 (3.49x)	3.76 (3.50x)	3.88 (3.39x)
2048^2	50.74	14.49 (3.50x)	14.45 (3.51x)	14.71 (3.45x)
4096^2	203.35	55.60 (3.65x)	55.59 (3.65x)	57.45 (3.54x)
8192^2	850.18	243.00 (3.50x)	241.81 (3.51x)	241.81 (3.51x)

(a) Tesla C870

Matrix	**Baseline:**	**Optimiz. 1**	**Optimizs. 1+2**	**Optimizs. 1+2+3**
1024^2	2.34	0.73 (3.20x)	0.65 (3.60x)	0.63 (3.71x)
2048^2	8.72	2.79 (3.12x)	2.47 (3.53x)	2.42 (3.60x)
4096^2	34.60	11.45 (3.02x)	9.93 (3.48x)	9.66 (3.58x)
8192^2	144.02	45.70 (3.15x)	40.35 (3.57x)	40.29 (3.57x)

(b) Tesla C1060

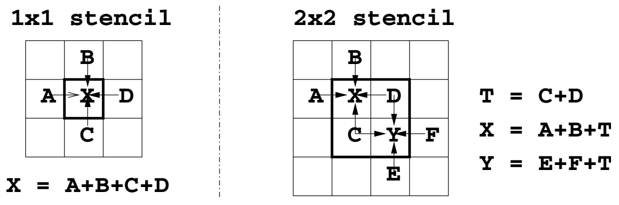


Fig. 4. Increasing the stencil size: some redundant operations may be saved

4.3 The Effect of Larger 2D Stencils

Our next alternative kernel tries to evaluate the effect of changing the 2D stencil size, which imposes a coarser granularity on SIMD parallelism. Instead of a single element, a 2x2 matrix of elements was assigned to every thread. Using this new stencil, partial sums on diagonal elements of the matrix can be reused for computing the output elements on the other diagonal (see Figure 4), saving two arithmetic operations and four memory accesses on each thread at the expense of using two registers for storing auxiliary values.

Execution times are shown in Table 5 on a Tesla C1060 GPU for different threads deployment (depicted on rows). The input matrix size is 4096^2 and our kernel uses shared memory without further optimizations. The execution is slowed down 30-40% on average with respect to the case in which each thread computes a single element, proving that context switch is free in CUDA: using a 1x1 stencil we require 341x341 calls with thread blocks, whereas using a 2x2 stencil, we just need 157x157 calls.

Table 5. Execution times (in seconds) on a Tesla C1060 GPU for different threads deployment (depicted on rows). The input matrix size is 4096^2 and the code version uses shared memory without further optimizations. The stencil size is the number of elements computed by each thread.

Threads	**Stencil size**		**Slowdown**
deployment	**1x1**	**2x2**	**factor**
14x14	13.91	19.31	38%
16x16	11.45	15.43	34%
18x18	13.27	18.16	36%
20x20	13.83	18.40	33%

Table 6. Execution times (in seconds) and empirical streaming bandwidth (in GB/s.) for the optimal version of our Jacobi implementation (that is, optimizations 1, 2 and 3 performed), using single and double precision in our Tesla C1060 GPU, where theoretical peak bandwidth is 102 GB/s. We run out of memory for the 8192^2 case on double precision. Threads deployment is 16x16 for all cases.

Matrix	**Single Precision**		**Double Precision**		**Slowdown**
size	**Exec. time**	**Bandwidth**	**Exec. time**	**Bandwidth**	**factor**
1024^2	0.90	104.25 GB/sec	1.70	56.69 GB/sec	89%
2048^2	3.36	114.49 GB/sec	6.55	14.70 GB/sec	95%
4096^2	13.41	117.00 GB/sec	26.33	3.66 GB/sec	96%

4.4 Floating-Point Accuracy and Performance

We now evaluate floating-point performance by comparing the GPU power on single and double precision versions of our optimal Jacobi implementation. According to Table 1, peak performance on the Tesla C1060 is 933 GFLOPS in single precision and 78 GFLOPS in double precision. However, Table 6 shows that times roughly double when switching from single to double precision. This basically means that our application is bandwidth limited and the fact that double precision numbers occupy 64 bits versus 32 bits for single precision explains the slowdown factor.

Table 6 also includes bandwidth data attained by our implementation. In certain cases, we reached values exceeding the theoretical peak bandwidth provided by the global memory (GDDR3 video RAM in our GPU), which can justify the presence of small cache areas in the underlying architecture, a feature which is often deliberately undisclosed by hardware vendors. Nevertheless, we believe these bandwidth numbers reflect a solid validation of outstanding performance for the kernels developed throughout this work.

4.5 GPU versus CPU Multi-core Performance

In our final experiment, we want to compare the many-core GPU with a multicore CPU in terms of scalability, parallel performance and the influence of the

Table 7. Execution times (in seconds) for different architectures and implementations

Input	**On a Tesla GPU**		**On an Intel Core 2 Quad Q9450 CPU**				
matrix	**C870**	**C1060**	**1 core**	**2 cores**	**4 cores**	**4 cores**	**4 cores**
size			**1 thread**	**2 threads**	**4 threads**	**8 threads**	**16 threads**
1024^2	3.88	0.63	12.30	6.04	3.08	3.91	4.26
2048^2	14.71	2.42	50.05	53.13	61.10	61.17	59.02
4096^2	57.45	9.66	200.56	220.02	252.92	251.84	251.44
8192^2	241.81	40.29	807.87	876.00	1003.01	1009.11	1000.43

memory hierarchy. Table 7 presents the execution times that we have obtained parallelizing the Jacobi method on the GPU using CUDA and the CPU using pthreads. For the multithreaded CPU version, the best performance was obtained by assigning entire rows to each CPU core as data partition.

We can see that the Tesla C1060 GPU is unbeatable, and the C870 is also more effective than the quad-core in most of the cases, overall when working with large matrices. It can also be seen how the CPU times are poorly scalable when the working set exceeds the L2 cache size (12 MB in our case), that is, from the 2048^2 case on. In other words, the CPU cores have to rely on caches to become effective, and the Jacobi method becomes even more bandwidth limited when running on multicore CPUs.

5 Related Work

APIs such as OpenMP are able to tile stencil loops at run-time and execute the tiles in parallel [10]. Researchers have investigated the best combination of tiling strategies that optimizes both cache locality and parallelism, and even propose automatic tuning for tiling stencil computations on multicores [4], GPUs [13] and the Cell [2]. Those techniques are usually based on the concept of ghost zones, which enlarge the tile with a perimeter overlapping neighboring tiles by multiple *halo* regions to reduce communications by replicating computation.

Stencil kernels on GPUs have recently gained attention by the scientist community. Listed in order of affinity with our work, we may select the following four contributions: Datta et al [4] tune a benchmark of 3D stencil kernels on GPUs and multicores, Christen et al. [2] consider a 7-point stencil kernel to be implemented on GPUs and the Cell BE, Amorim et al. [1] perform a comparison of the Jacobi method between a GPU parallelization using OpenGL and CUDA, and finally, Venkatasubramanian et at. [13] also implement the Jacobi method on GPUs and hybrid CPU/GPU systems.

Focusing on the work performed specifically on Jacobi method, Amorim et al. [1] use diagonal matrices and a different access pattern than ours to compare results against a CPU implementation on a quad-core AMD Phenom processor, obtaining a 78x speed-up factor.

On the other hand, the work in [13] was developed in parallel to ours with a similar methodology. Our implementation sacrifices two idle threads on each half-warp by replicating data placed at block boundaries, which helps us to succeed on a more homogeneous access to the device memory. This is rewarded on conflicts-free accesses to memory banks and particularly on coalesced accesses, overall in CUDA Compute Capabilities 1.3, where conditions for non-coalesced accessed were widely relaxed.

In all CUDA implementations, a crucial parameter for attaining the best performance is the thread block size. In [13], a 1.7x slowdown factor is reported when moving from a 64x8 to a 16x8 block size on a Tesla C1060 GPU using a 8x8 thread block. Though we agree on how sensitive the execution time is to the thread deployment, our best performance is achieved on a 16x16 thread block, with a clear benefit on lighter threads. This way, our key guidelines to the optimal Jacobi implementation were to reduce to the minimum (1) the number of target elements computed by each thread, and (2) the number of tiles computed by each thread block.

6 Summary and Conclusions

This paper explores CUDA on GPUs to optimize stencil computations using as benchmark the Jacobi method for solving Laplace's differential equation. Optimization paths are focused on data parallelism, threads deployment and the GPU memory hierarchy, with a clear influence of the stencil access pattern.

Experimental results show great success for our techniques on Teslas C870 and C1060 GPUs, achieving great scalability and good performance versus a quad-core Intel CPU. The speed-up factor for our set of GPU optimizations reaches 3-4x and the execution times defeat those of the CPU by a wide margin, also showing great scalability when moving towards a more sophisticated GPU architecture and/or more demanding problem sizes.

Streaming and arithmetic intensive kernels produce higher performance on the GPU to reach two orders of magnitude gain factors with respect to a multicore CPU. However, our kernel for Jacobi is bandwidth limited, preventing us from further optimizations. This behavior is also confirmed when comparing single to double precision performance, as the peak computational power is theoretically more than an order of magnitude higher for the single precision case and the execution time barely gets better by a factor of two.

Acknowledgments. This work has been jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, by the Junta de Andalucíía under project P06-TIC-02109, and also by the Spanish MEC and European Commission FEDER funds under grants Consolider Ingenio-2010 CSD2006-00046 and TIN2009-14475-C04-02.

References

1. Amorim, R., Haase, G., Liebmann, M., Weber dos Santos, R.: Comparing CUDA and OpenGL Implementations for a Jacobi Iteration. Technical Report, Graz University of Technology (December 2008)
2. Christen, M., Schenk, O., Neufeld, E., Messmer, P., Burkhart, H.: Parallel Data-Locality Aware Stencil Computations on Modern Micro-Architectures. In: Procs. IEEE Intl. Parallel and Distributed Processing Symposium, Rome (May 2009)
3. CUDA: http://developer.nvidia.com/object/cuda.html (accessed September 15, 2010)
4. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D.A., Shalf, J., Yelick, K.: Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In: Proceedings ACM/IEEE Supercomputing 2008, Austin, TX, USA, pp. 1–12 (November 2008)
5. Demmel, J.: Applied Numerical Linear Algebra. SIAM, Philadelphia (1997)
6. Firestream: AMD Stream Computing, http://www.amd.com/us/products/workstation/firestream/Pages/firestream.aspx (accessed September 15, 2010)
7. GPGPU: General-Purpose Computation Using Graphics Hardware (2010), http://www.gpgpu.org
8. The Khronos Group: The OpenCL Core API Specification. Headers and documentation, http://www.khronos.org/registry/cl (accessed September 15, 2010)
9. Lester, B.: The Art of Parallel Programming. Prentice Hall, Engl. Cliffs (1993)
10. OpenMP: The OpenMP API (2010), http://www.openmp.org
11. Owens, J., Luebke, D., Govindaraju, Harris, M., Kruger, J., Lefohn, A., Purcell, T.: A Survey of General-Purpose Computation on Graphics Hardware. Journal Computer Graphics Forum 26(1), 80–113 (2007)
12. Tesla: Nvidia Tesla GPU computing solutions for HPC, http://www.nvidia.com/object/personal_supercomputing.html (accessed September 15, 2010)
13. Venkatasubramanian, S., Vuduc, R.W.: Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. In: Proceedings ACM Intl. Conference on Supercomputing, New York, USA (June 2009)

Streaming Model Computation of the FDTD Problem

Adam Smyk[1] and Marek Tudruj[1,2]

[1] Polish-Japanese Institute of Information Technology,
86 Koszykowa Str., 02-008 Warsaw, Poland
[2] Institute of Computer Science, Polish Academy of Sciences,
21 Ordona Str., 01-237 Warsaw, Poland
{asmyk,tudruj}@pjwstk.edu.pl

Abstract. The Finite Difference Time Domain (FDTD) method enables computerized simulation of the electromagnetic wave propagation. We propose a streaming model for FDTD computations oriented towards a multicore processor architecture. FDTD computations are characterized by injection of small portions of data into computational nodes, processing them and returning the results into main storage. We can parallelize FDTD computations by combining the loop tiling approach and a communication mechanism based on a rotating buffers infrastructure. The described FDTD algorithm has been implemented using both of these techniques on a streaming architecture of the Cell/BE processor. The efficiency of FDTD computations has been estimated for different parameters of the assumed loop tiling algorithm and the rotating buffers mechanism.

Keywords: FDTD, loop tiling method, Cell/BE, streaming architecture, rotating buffers mechanism.

1 Introduction

Intensive development of system-on-chip and network-on-chip technologies as well as embedded and multicore systems enforces major changes in parallel processing methodology. In multicore systems, total computational performance is bounded due to limitations of processor clock speed caused by heat dissipation problems. Simultaneously, in many modern computational systems, DMA or RDMA communication is used instead of traditional sockets or other message passing methods. It enables using direct access to remote memory from each processing element, and hence, overlapping of computation and communication can be easily and efficiently applied. In order to obtain sufficient computational performance of parallel solving numerical problems, several supporting mechanisms can be used:

– loop tiling technique [1, 2]– to find, describe and implement data dependencies in computational expressions;
– rotating buffers communication [3] – to exploit a DMA communication infrastructure and create a data stream between all memory modules involved in a streaming computational process.

In next two sections, we describe how to combine these mechanisms to increase efficiency of computations.

2 Loop Tiling Approach

Loop tiling is one of many techniques [4] used by parallel compilers to maximize parallelism, improve memory hierarchy performance, decrease communication and synchronization time. Using this technique, only iterative problems can be solved. The FDTD simulation is an example of a stencil computation problem [5, 6]. In the FDTD method, the involved electromagnetic wave propagation area is transformed into its discrete form (a set of Yee cells) and computations are performed only for discrete points of the area. In the paper a two-dimensional version of the FDTD problem is considered. Electromagnetic wave propagation is described by time-dependent Maxwell equations [8]. For the FDTD method they assume the form presented below (1).

$$\nabla \times H = \gamma E + \varepsilon \frac{\partial E}{\partial t}, \qquad \nabla \times E = -\mu \frac{\partial H}{\partial t} \tag{1}$$

If the FDTD problem is considered in an isotropic environment, the Maxwell equations can be transformed into their differential forms (2)

$$\begin{cases} \bar{E}_z^n(i,j) = CA_z(i,j)\bar{E}_z^{n-1}(i,j) + CB_z(i,j) \cdot \\ \qquad\qquad [\bar{H}_y^{n-0.5}(i+1,j) - \bar{H}_y^{n-0.5}(i+1,j) + \\ \qquad\qquad\quad + \bar{H}_x^{n-0.5}(i,j-1) - \bar{H}_x^{n-0.5}(i,j+1)] \\ \bar{H}_x^n(i,j) = \bar{H}_x^{n-1}(i,j) + RC \cdot [\bar{E}_z^{n-0.5}(i-1,j) - \bar{E}_z^{n-0.5}(i+1,j)] \\ \bar{H}_y^n(i,j) = \bar{H}_y^{n-1}(i,j) + RC \cdot [\bar{E}_z^{n-0.5}(i,j-1) - \bar{E}_z^{n-0.5}(i,j+1)] \end{cases} \tag{2}$$

The whole iteration space of the equations (2) is divided into tiles. A tile represents an atomic piece of computation, defined by a set of nested loops, which solve three differential equations (2). Next, a data dependency set is created [1]. It contains a number of dependency vectors (3). They consist of combinations of distance and direction values, which describe all dependencies for all expressions for all nested loops. The shape of a tile directly follows from the dependency set. For a 2D FDTD problem (2), we can consider a space tiling that can be described by the following dependency set:

$$\begin{aligned} D_{ez} &= \{(0,0), (-1,0), (1,0), (0,1), (0,-1)\} \\ D_{hx} &= \{(0,0), (-1,0), (1,0)\} \\ D_{hy} &= \{(0,0), (0,1), (0,-1)\} \end{aligned} \tag{3}$$

The shape of a tile determines the volume of total communication between all processing nodes and all memory modules involved in computations. We have assumed a rectangular shape of a tile. All tile dependencies for the FDTD problem are expressed by a tile space graph, Fig.1. To reduce the total communication volume [2], appropriate mapping should be applied. In our case, we have used a mirror-rotating mapping according to the scheme presented in Fig. 1.

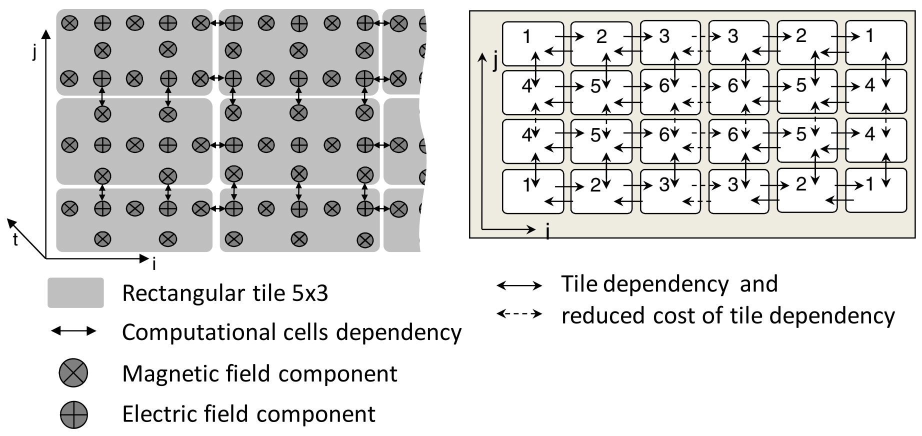


Fig. 1. Computational mesh of the 2D FDTD problem

3 Rotating Buffers Mechanism

In a streaming computational model, a small amount of data should be transferred from the main memory of computational system to local memories of processing elements. If data are already stored in a local memory, the computations will be started. When the computations are finished, all results are transferred back to the main memory. To reduce communication latencies, double or triple buffering techniques are used. During the computations of tiles already stored in a local memory, other tiles are transferred between local and main memory modules. Such overlapping of computations and communication significantly increases total processing efficiency. We have extended this approach onto a rotating buffers technique. We have assumed a number of tiles that can be transferred at the same time. Each tile is stored in a single buffer. To reduce control overhead, we have used a buffer rotation pointer. It points on a currently used buffer and simultaneously, it defines buffers whose contents is currently transferred to and from local memory. Such a simple multi-overlapping fully exploits the DMA facility potential for data transfers. It also enables adjusting the tile size and the tile shape of the FDTD problem for a given irregular simulation area and for a given system architecture.

In Fig. 2, the logical structure of the memory used in the rotating buffer method is presented. This infrastructure is used in Hitachi SR2201 supercomputer, but it can be easily transformed to various distributed parallel computational systems like Cell/BE PS3 for example. Local memory of the computational node is logically divided into two parts LAM (Locally Accessed Memory) containing data used only for local computations with direct access from application program level and GAM (Globally Accessed Memory) containing data used for data exchange. Access to GAM is available only through the rotating buffers memory infrastructure.

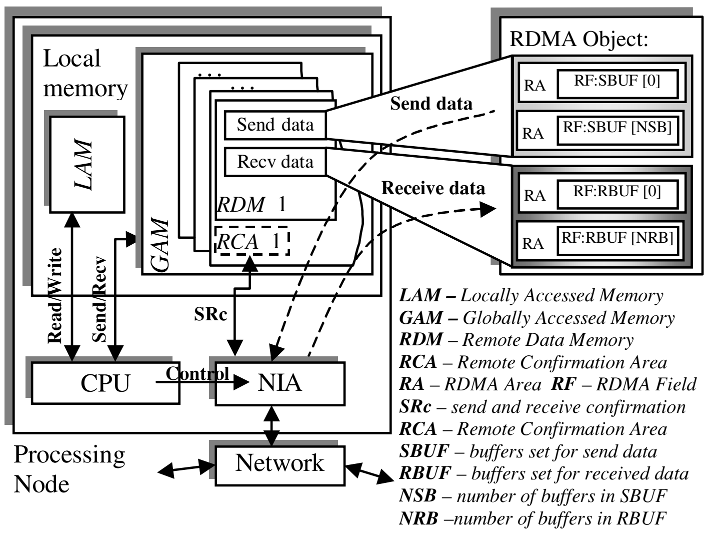

Local memory
LAM
GAM
. . .
Send data
Recv data
RDM 1
RCA 1
Read/Write
Send/Recv
SRc
CPU
Control
NIA
Processing Node
Network
RDMA Object:
RA RF:SBUF [0]
RA RF:SBUF [NSB]
RA RF:RBUF [0]
RA RF:RBUF [NRB]
Send data
Receive data
LAM – Locally Accessed Memory
GAM – Globally Accessed Memory
RDM – Remote Data Memory
RCA – Remote Confirmation Area
RA – RDMA Area RF – RDMA Field
SRc – send and receive confirmation
RCA – Remote Confirmation Area
SBUF – buffers set for send data
RBUF – buffers set for received data
NSB – number of buffers in SBUF
NRB –number of buffers in RBUF

Fig. 2. Memory structure in rotating-buffers method (for one processing node on the HITACHI SR2201 supercomputer)

GAM area is divided into *N* separate sub-area pairs: RDM (Remote Data Memory) and RCA (Remote Confirmation Area), where N is the number of remote processors. Each pair is used to perform communication between two given nodes. The numbers of rotating buffers in the send and receive parts are fixed and denoted by NSB and NRB, respectively. All buffers which are defined in a RDM area are used only for data transmission. To avoid possible data overwriting, an additional control has to be introduced. This control is based on the RCA areas which are assigned independently to each RDM area. Each RCA area is intended to send and receive additional control messages which determine if the buffers from a RDM are ready to receive new data. A RCA consists of only two sets of one buffer each (by analogy to a RDM where the number of buffers can change, here the NSB and NRB numbers are always equal to 1). They are used to exchange synchronized messages between two processors. The control flow in the rotating buffers method for one processing node is presented in Fig.3. Data are exchanged between a local node and a remote node K. On both these nodes, the described above control and communication infrastructure was created. Additionally, for each processor two pointers K.SPTR and K.RPTR are created. They are used to indicate a next free buffer in which new data (to be sent to node K) will be placed (K.SPTR) or new data just received from K will be written (K.RPTR). These two pointers determine a rotating access to available buffers and introduce periodical synchronization between two communicating nodes, which assures that no data which are transferred from one node to another, will be lost (overwritten).

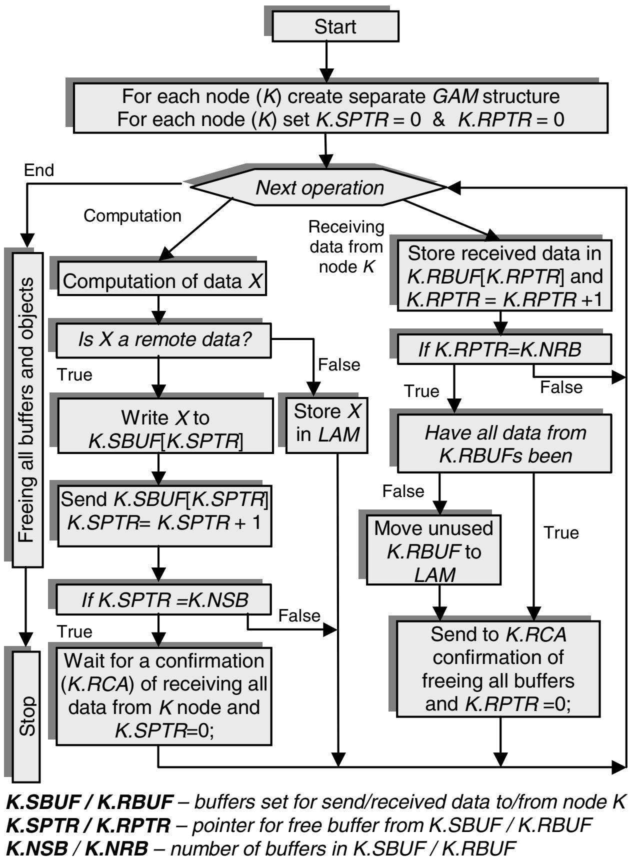

Start
For each node (K) create separate GAM structure
For each node (K) set K.SPTR = 0 & K.RPTR = 0
End
Next operation
Computation
Receiving data from node K
Computation of data X
Is X a remote data?
True
False
Write X to K.SBUF[K.SPTR]
Store X in LAM
Send K.SBUF[K.SPTR] K.SPTR= K.SPTR + 1
If K.SPTR =K.NSB
True
False
Wait for a confirmation (K.RCA) of receiving all data from K node and K.SPTR=0;
Freeing all buffers and objects
Stop
Store received data in K.RBUF[K.RPTR] and K.RPTR = K.RPTR +1
If K.RPTR=K.NRB
True
False
Have all data from K.RBUFs been
False
True
Move unused K.RBUF to LAM
Send to K.RCA confirmation of freeing all buffers and K.RPTR =0;
K.SBUF / K.RBUF – buffers set for send/received data to/from node K
K.SPTR / K.RPTR – pointer for free buffer from K.SBUF / K.RBUF
K.NSB / K.NRB – number of buffers in K.SBUF / K.RBUF

Fig. 3. Control flow of the rotating-buffers method for one processing node

4 Experimental Results

All experiments have been done on a Cell/BE PS3 processor equipped in one main processor (PPE – Power Processing Element) with 256MB of main memory, six worker units (SPE – Synergistic Processing Elements) with 256KB of local memory [9, 10, 11, 12, 13, 14, 15]. All processing units: PPE, SPE, main memory and the IO unit are interconnected by fast multi-ring network (EIB – Element Interconnected Bus). Communication can be done either by DMA transfers (for large-sized data) or by mailbox transfers (for very small-sized data). In our experiments, DMA transfers have been done by the rotating buffers infrastructure, where GAM on SPE processor contains only one pair of RDM and RCA areas and on PPE processor contains six pairs of RDM and RCA areas. It is because, the communication between SPE and PPE nodes is considered in this paper.

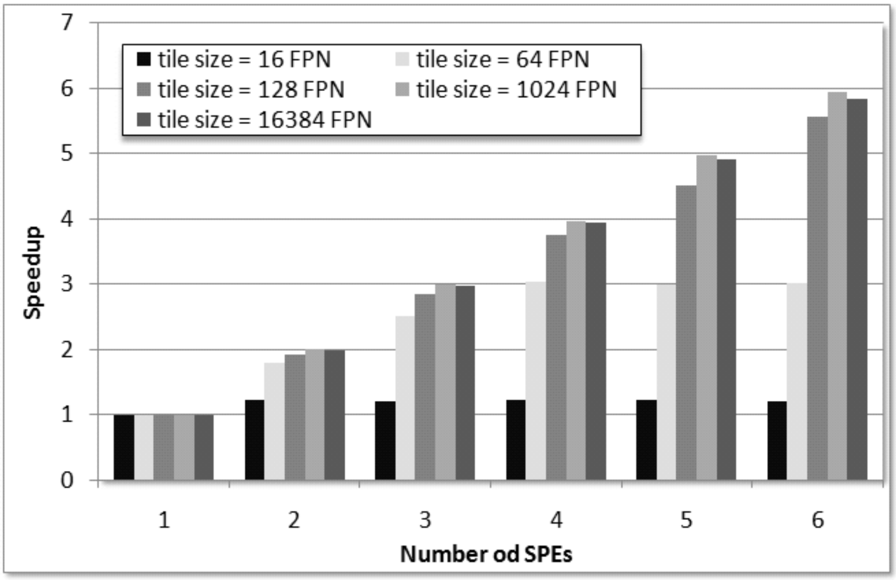


Fig. 4. Speedup of computations for different tile sizes

Firstly, the FDTD computations have been tested for a given shape of simulation area and 8000 rectangular tiles with various sizes expressed in floating point numbers (FPN). As we can see (Fig. 4), the speedup of computations depends on the number of SPE nodes involved in computations and on the size of a single tile. For large-size tiles (bigger than 128 FPN) the obtained speedup is close to the linear one. Unfortunately, for small-sized tiles (especially for 16 FPN) we have not been able to obtain any increase of efficiency. It means that the streaming model of computations implemented in Cell/BE architecture cannot be efficiently applied for fine grain computations (in our case, the FDTD problem cannot be successfully solved for strongly irregular simulation areas, where very small-sized tiles should be used). To solve this problem, we have implemented a rotating buffers infrastructure for the FDTD computations, see Fig. 5. We have tested the same simulation area for various sizes of tiles, various numbers of rotating buffers and various number of SPE units. In each test, we have used the same communication volume – the size of a tile multiplied by the number of tiles are the same in each test. As we can see, efficiency of computations depends on both the size of a tile and the number of rotating buffers. We can see again, that configurations with small size tiles work slower in comparison to large size configurations (10 times slower for the buffer size of 1 tile). If the size of the rotating buffers increases, the efficiency will increase as well. For configurations with 8 and 16 tiles in one buffer, the execution times are almost the same, respectively.

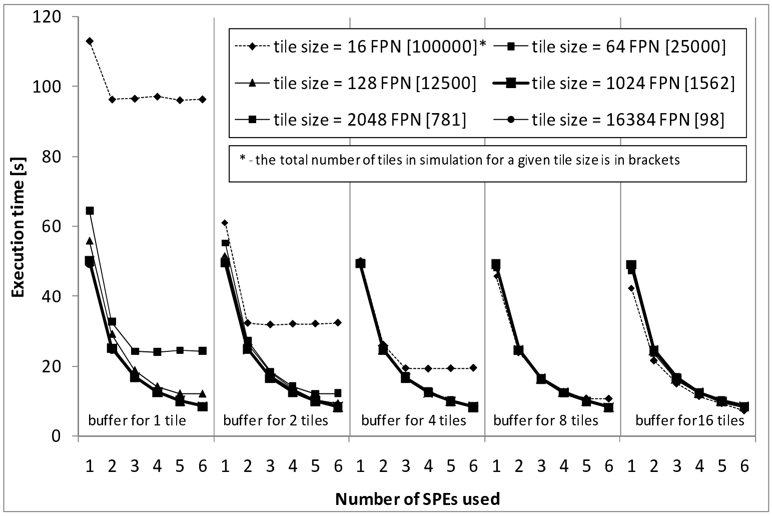


Fig. 5. Execution time of the FDTD computation for different configurations with rotating buffers infrastructure

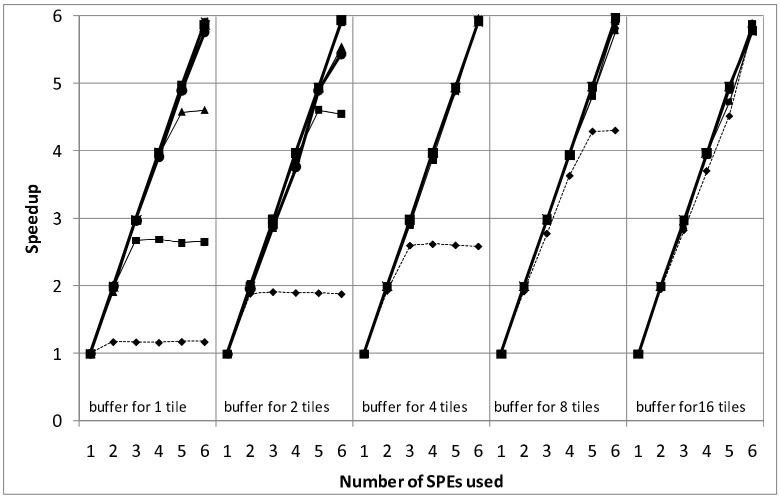


Fig. 6. Speedup of the FDTD computation for different configurations with rotating buffers infrastructure (curve notation as in Fig.5)

When the rotating buffers mechanism is combined with other communication optimization techniques (e.g. non-blocking mailbox synchronization) we can obtain a linear or even super linear speedup of the FDTD computation for some configurations (Fig. 6). It shows that for configurations with small sized tiles (16/64/128) and for 1 tile in rotating buffer configuration, speedup obtained for 6 SPEs does not exceed ˜1.2/˜2.7/˜4.6 in comparison to execution of these configurations on 1 SPE. For the rest of configurations (tile size 1024/2048/16384) we have obtained almost linear speedup. As the number of tiles in a rotating buffers infrastructure increased, the speedup (in comparison to execution of given configuration on 1 SPE) for small sized tiles configurations also increased and for configuration with 16 tiles in a buffer we have obtained almost linear speedup for all configurations. For configuration with 8 tiles and 6 SPEs we have obtained a slight super linear speedup. It shows that overlapping of the FDTD computation and simultaneous tiles transferring with the rotating buffers infrastructure in this case allows to obtain the best increase of the efficiency in comparison to other configurations.

General performance (for GCC -O3 compiler) obtained in the last experiment for tile size equals 16 vary from ˜3.3 GFLOP/s (configuration with 1 SPE) to ˜4.0 GFLOP/s (configuration with 6 SPEs). For tile size equals 2048, general performance vary from ˜7.7 GFLOP/s (configuration with 1 SPE) to ˜45.5 GFLOP/s (configuration with 6 SPEs). In our implementation we were oriented towards data communication improvement by the use of the rotating-buffers technique. It is possible to further improve the speedup of similar stencil computation by the use of the SIMD option inside a SPE offered by CELL/BE architecture, as in [asmykkey-16] where ˜12.4 GFLOP/s for 1 SPEs and ˜99.5 GFLOP/s for 8 SPEs on QS22 blade for Finite Difference application are reported. However, it requires special rearrangement of the computation structure inside a SPE to comply with the SIMD mode requirements. This will be the target ot further studies.

5 Conclusions

The combination of the tiling method and the rotating buffer memory infrastructure has turned out to be very efficient optimization technique for FDTD computations on streaming model architecture. The rotating buffer based on DMA communication has produced in some cases (configuration with 8 tiles in rotating buffers infrastructure) super linear speedup due to elimination of stalling of computations by constant providing new computational tiles to SPE processors. Efficient computations for configurations with small size tiles (fine grain parallelism) are very important in the case of very irregular computational shapes since big size tiles cannot be directly applied. Their direct mapping into computational area produces weak tiles (tiles with many unused computational cells), so some additional computational overhead will appear. Our optimizations have led to promising results for fine grain computations by reaching the maximal possible performance on the Cell/BE processors. The method presented in the paper can be easily adopted to other numerical problems that can be solved iteratively. The main problem is that the dependency set for a given iterative problem must be constant for large number of iterations, otherwise streaming computations will not be performed efficiently.

References

[1] Xue, J.: Loop tiling for parallelism. Kluwer Academic Publishers (2000)
[2] Manjikian, N., Abdelrahman, T.S.: Exploiting Wavefront Parallelism on Large-Scale Shared-Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems 12(3) (March 2001)
[3] Smyk, A., Tudruj, M.: RDMA Control Support for Fine-Grain Parallel Computations. In: PDP 2004, La Coruna, Spain (2004)
[4] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Austin, Texas, November 15-21 (2008)
[5] Adams, S., Payne, J., Boppana, R.: Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors. In: HPCMP Users Group Conference (2007)
[6] Orozco, D., Gao, G.: Mapping the FDTD Application to Many-Core Chip Architectures, CAPSL Technical Memo 087 (March 3, 2009)
[7] Taflove, A.: Computational Electrodynamics: The Finite-Difference Time-Domain Method. Artech House Antennas and Propagation Library (1996)
[8] Trikas, P.A., Balanis, C.A., Puerchine, M.P., Barber, G.C.: Finite-Difference Time-Domain Method for Electromagnetic Radiation, Interference, and Interaction with Complex Structures. IEEE Transactions on Electromagnetic Compatibility 35(2), ss. 192–ss. 203
[9] Buttari, A., Dongarra, J., Kurzak, J.: Limitations of the PlayStation 3 for High Performance Cluster Computing, Technical Report UT-CS-07-597, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, and as LAPACK Working Note 185 (May 2007)
[10] Gonzàlez, M., Vujic, N., Martorell, X., Ayguadé, E., Eichenberger, A.E., Chen, T., Sura, Z., Zhang, T., O'Brien, K., O'Brie, K.: Hybrid access-specific software cache techniques for the cell BE architecture. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, Toronto, Ontario, Canada (2008)
[11] http://www.us.playstation.com/PS3/
[12] Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a Programming Model for the Cell BE Architecture. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (2006)
[13] Kruijf, M., Sankaralingam, K.: MapReduce for the Cell B.E. Architecture, Technical Report #1625 (October 2007)
[14] Akhter, S., Roberts, J.: "Multi-Core Programming - Increasing Performance through Software Multithreading". IntelPress (2006)
[15] http://www.ibm.com/developerworks/power/cell/
[16] de la Cruz, R., Araya-Polo, M., Cela, J.M.: Introducing the *Semi-stencil* algorithm. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 496–506. Springer, Heidelberg (2010)

Numerical Aspects of Spectral Segmentation on Polygonal Grids

Anna Matsekh, Alexei Skurikhin, Lakshman Prasad, and Edward Rosten

Space and Remote Sensing Sciences Group, Los Alamos National Laboratory
Engineering Department, University of Cambridge
{matsekh,alexei,prasad}@lanl.gov,
er258@cam.ac.uk

Abstract. We present an implementation of the Normalized Cuts method for the solution of the image segmentation problem on polygonal grids. We show that in the presence of rounding errors the eigenvector corresponding to the k-th smallest eigenvalue of the generalized graph Laplacian is likely to contain more than k nodal domains. It follows that the Fiedler vector alone is not always suitable for graph partitioning, while the eigenvector subspace, corresponding to just a few of the lowest eigenvalues, contains sufficient information needed for obtaining meaningful segmentation. At the same time, the eigenvector corresponding to the trivial solution often carries nontrivial information about the nodal domains in the image and can be used as an initial guess for the Krylov subspace eigensolver. We show that proposed algorithm performs favorably when compared to the Multiscale Normalized Cuts and Segmentation by Weighted Aggregation.

Keywords: image segmentation, spectral graph partitioning, symmetric eigenvalue problem, generalized graph Laplacian.

1 Introduction

Image segmentation methods often rely on the use of the spectral graph partitioning techniques [1,2], when an image is represented by a simple weighted undirected graph. In this setting, graph vertexes represent image primitives, such as pixels, while its edges describe relationships between the neighboring image primitives. On the subsequent steps, graph clustering objective, seeking to partition image graph into a set of disjoint coherent segments, is set up.

Spectral graph partitioning presents a number of numerical challenges. The most notable problem is the complexity reduction of the underlying eigenvector computations – the bottle neck of all spectral segmentation methods [3]. This problem is typically addressed either within an algebraic multigrid or multilevel graph partitioning framework. This approach has been successfully adapted for the solution of image segmentation problems in multiscale [4,5,6] and algebraic multigrid-like frameworks [7]. In the current study, we replace an image with its compressed version on an unstructured polygonal grid, assuming that compression preserved main properties of the eigenvalue spectrum of the graph Laplacian.

Sometimes, the authors construct spectral segmentation in multidimensional spaces spanned by a few eigenvectors, corresponding to the smallest eigenvalues of the graph Laplacians [8], rather than relying on recursive bisection of the Fiedler vector alone – the theoretical optimal solution to the graph partitioning problems [1,2]. However, as far as we can tell, there has never been given a clear explanation why recursive bisection of the Fiedler vector alone may not always produce satisfactory segmentation. We try to give this explanation by analyzing the effects of the rounding errors on the finite precision solution to the problem in the Krylov subspace generated by a variant of the Cullum-Willoughby-Lanczos method [9,10].

2 Problem Setup

Let J be a pixel image and let P be its approximation on a polygonal grid obtained using VISTA software [11]. Polygonized image P consists of n primitives p_i obtained by the application of the Canny edge detector on the image J, followed by the application of an algorithm that superimposes Constrained Delaunay Triangulation (CDT) over the detected edges, assigning average colors to the triangles by Monte Carlo sampling of the pixels in each triangle [11]. The triangulated image is then further coarsened by combining neighboring triangles representing coherent patches, into polygons $p_i,\ i = 0, 1, \ldots, n-1$ [11]. Each polygon p_i is attributed with color u_i which is an aggregate of the colors of the triangles polygon p_i is comprised of. Typically, the number of polygons will be significantly smaller than the number of the pixels (*e.g.* see Figure 1). We will treat approximation P as a compressed image J, replacing J with P on the consecutive steps.

Our goal is to segment the polygonized image P into coherent visual scenes using the Normalized Cuts graph partitioning objective function [13] that seeks to minimizes graph cuts relative to the weighted degrees of its clusters. This problem formulation provides valuable information about the weighted degrees of graph nodes that we exploit when solving the underlying sparse symmetric eigenvalue problem. We represent P as a simple weighted undirected graph $G(V,\, E)$ with vertex set $V = (v_0, v_1, ..., v_{n-1})$ and edge set $E = \{e_{ij} \mid e_{ij} = v_i\, v_j\}$. Graph vertexes v_i describe polygons p_i, while its edges e_{ij} connect vertexes that represent neighboring polygons p_i and p_j. Graph G is then described by the weighted adjacency, or affinity, matrix $W \in \mathbb{R}^{n\times n}$, $W = W^T$ with elements w_{ij} representing weights associated with the edges e_{ij}.

Let $u_i = (u_i^0, u_i^1, u_i^2)$ be the aggregate color assigned to the polygon p_i in the CIELAB color space. Let (x_i^k, y_i^k), $k = 0, 1, \ldots, n_{i-1}$ be the coordinates of its n_i vertexes. We call two polygons p_i and p_j neighbors if they share at least one

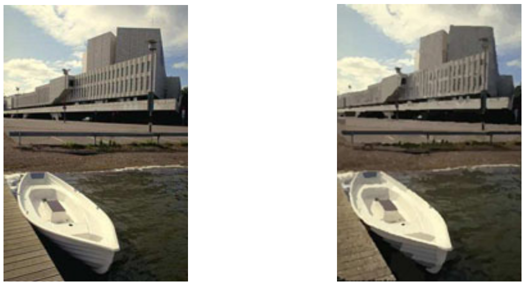


Fig. 1. Geometric image coarsening with VISTA: (a) original 321 × 481 pixel image from the Berkeley Segmentation Data Set [12]; (b) polygonized image consisting of 4521 primitives

Canny or CDT-detected edge. Let $\hat{u}_{ij} = (u_i - u_j)$. We can compute the elements of the affinity matrix W as follows:

$$w_{ij} \overset{\text{def}}{=} \begin{cases} \dfrac{e^{-c\,\hat{u}_{ij}^T\,\Sigma^{-1}\,\hat{u}_{ij}}}{|\Sigma|^{1/2}}\; e^{-f(g_{ij})}, & i \neq j \wedge j \in P^i_{\text{nbr}} \\ 0, \quad i = j \vee j \notin P^i_{\text{nbr}} \end{cases} \tag{1}$$

where $i = 0, 1, \ldots, n-1$, P^i_{nbr} is the list of indexes of the neighbors of v_i, Σ is the color sample variance-covariance matrix, $c > 0$ is an image-dependent constant and $f(g_{ij})$ is a function describing geometric affinity of p_i and p_j.

The Normalized Cuts graph clustering objective [13] is typically considered in its weak formulation

$$L\, z_i = \lambda_i\, D\, z_i \tag{2}$$

with the set of continuous solutions (λ_i, z_i), $i = 0, 1, \ldots$, where $L = D - W$, $L = L^T$ is the generalized graph Laplacian [2] and $D = \text{diag}(d_0, d_1, \ldots d_{n-1})$ is the diagonal matrix of weighted degrees $d_k = \sum_{i \neq k} w_{ki}$ of the graph nodes v_k. The generalized symmetric eigenvalue problem (2) is easily reduced to the spectrally equivalent symmetric eigenvalue problem

$$A\, x_i = \lambda_i\, x_i, \tag{3}$$

where $A = I - D^{-1/2}\, W\, D^{-1/2}$ is the normalized affinity matrix, I is the identity matrix and $x_i = D^{1/2}\, z_i$. Matrix A is symmetric positive semidefinite and consequently has nonnegative real spectrum. The smallest nonzero eigenvalue λ_1 of A corresponds to the optimal solution to the relaxed Normalized Cuts problem (2). Fiedler [1] calls λ_1 the algebraic connectivity of graph, as the corresponding eigenvector z_1, often called Fiedler vector [2], is the new representation of the original image, consisting of positive and negative components that correspond to the two nodal domains in graph G. Fiedler vector is typically used in spectral segmentation methods for recursive graph bipartitioning.

3 Numerical Findings

Although recursive application of the Fiedler vector gives analytically optimal solution to the segmentation problem, some authors often use a few additional vectors to perform spectral graph partitioning. For instance, Malik et al. [8] use a two-stage segmentation procedure that requires to partition the subspace consisting of the 11 eigenvectors corresponding to the 11 smallest nonzero eigenvalues in order to construct an oversegmented image followed by the Fiedler-based segmentation of the graph of the oversegmented image. We would like to understand whether, in finite precision, it is sufficient to compute high quality segmentation with the Fiedler vector alone, or whether we do need to generate multidimentional segmentation of the eigenvector subspace corresponding to the few of the smallest eigenvalues.

In order to understand how the analytical properties of the graph Laplacians change when their spectral characteristics are computed in the presence of rounding errors, we will analyze the first two smallest distinct eigenvalues and the corresponding eigenvectors of the affinity matrix. Our choice of the eigenvalue solver is the Lanczos Method with Guaranteed Accuracy (LMGA) [10] – an implementation of the Cullum-Willoughby-Lanczos method [9] that uses eigenvalue intervals computed using Interval Bisection method and the two-sided Sturm sequences [14,10,15] to accurately identify and discard spurious and numerically multiple eigenvalues. LMGA relies on the latest version of the two-sided Sturm sequence-based implementation of the Inverse Iteration method [10,15] – Inverse Iteration with Guaranteed Accuracy (IIGA) – to compute the eigenvectors of the underlining tridiagonal symmetric eigenvalue problem.

Let $(\tilde{\lambda}_i,\, \tilde{z}_i),\ i = 0, 1, \ldots$ be the finite precision solution to the generalized eigenvalue problem (2) computed using the solution $(\tilde{\lambda}_i,\, \tilde{x}_i)$ to the equivalent standard eigenvalue problem (3). In exact arithmetic, zero eigenvalue $\lambda_0 = 0$ is the trivial solution to the problem (2), as the corresponding eigenvector $z_0 = c\,(1, 1, \ldots, 1)^T$, where $c \neq 0$ is a scalar, contains equal components that would assign all graph vertexes to one nodal domain, or one segment. This is the property of all generalized graph Laplacians, whose eigenvectors behave very similarly to the modes of a vibrating string, with the first mode, corresponding to $\lambda_0 = 0$, assigning all graph vertexes to one nodal domain, the second mode splitting graph nodes into two domains, the third – into three domains, and so forth [16]. Due to the presence of rounding errors, approximate eiegnvector $\tilde{z}_0$ has distinct components $\tilde{z}_0^k$ dependent on the weighted degrees d_k of v_k:

$$\tilde{z}_0^k = \tilde{x}_0^k / \sqrt{d_k},\ k = 0, 1, \ldots, n-1. \tag{4}$$

We can model $\tilde{z}_0$ as the vector $\bar{z}_0$ with the components

$$\bar{z}_0^k = 1.0 + \zeta\, \|A\|_E\, \varepsilon_{\text{mach}} / \sqrt{d_k}, \tag{5}$$

where $\varepsilon_{\text{mach}}$ is the unit round-off error, $\|A\|_E$ is the Euclidean norm of A and $\zeta > 1$ is a scalar that accounts for the error of the algorithm used to compute the eigenvectors. We compute the eigenvectors $\tilde{x}_k,\ k = 0, 1, \ldots, n-1$ of the underlining tridiagonal problem using Interval Bisection followed by the IIGA method. The error in the eigenvectors $\tilde{x}_i$ will be at least as high as the error of the computed eigenvalues, which is guaranteed not to exceed $\sqrt{3}\,\|A\|_E\,\varepsilon_{\text{mach}}$ [10]. In our case, ζ is at least $\sqrt{3}$, which means that $\bar{z}_0$ is likely to contain errors that slightly exceed the unit round-off. This means that, in general, components $\bar{z}_0^k$ are not identical, each inheriting information about the weighted degree

$$d_k = \sum_{i \neq k} w_{ki} \tag{6}$$

of the corresponding graph node v_k through the error term. Weighted degree d_k of the node v_k is the sum of the weights of the graph edges that node v_k is incident to. This means that d_k accumulates a wealth of information about adjacent graph nodes and incident edges of the node v_k. If we are looking for a nontrivial initial guess for our Lanczos eigensolver, vector $\bar{z}_0$ with an appropriately chosen parameter ζ certainly makes a good choice.

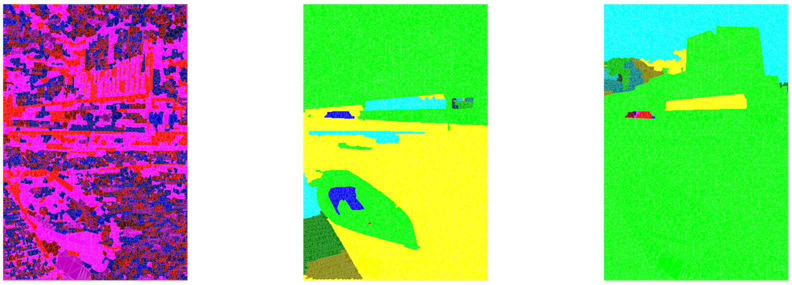
It is reasonable to assume that the error term $\zeta\,\|A\|_E\,\varepsilon_{\text{mach}}/\sqrt{d_k},\ \zeta > 1$ will be present in the components of all $\tilde{z}_i$. We can model all eigenvectors of the graph Laplacian as

$$\bar{z}_i^k = (\tilde{x}_i^k + \zeta\,\|A\|_E\,\varepsilon_{\text{mach}})/\sqrt{d_k}, \tag{7}$$

where $k = 0, 1, \ldots, n-1,\ i = 0, 1, \ldots$. It is clear that the error term will contribute additional information about the weighted degrees of the graph nodes into the floating point solution $\tilde{z}_i$. Consequently, we can expect that all $\tilde{z}_i$, including the Fiedler vector, may contain more than i nodal domains. Although the presence of the error term is desirable for computing an initial guess for the Krylov subspace eigensolver, it is likely to contaminate the Fiedler vector with the information about more than two nodal domains. This is probably the reason why in floating point arithmetic Fiedler vector alone often fails to deliver clear-cut results. As far as the use of the multidimensional eigenvector spaces for graph partitioning, the presence of the error term in the eigenvectors should not be damaging, as we are no longer looking for just two nodal domains. As a matter of fact, the information about more than i nodal domains in the i-th eigenvector in such a subspace may help to identify all nodal domains of interest without having to include a large number of eigenvectors.

We conducted a number of numerical experiments in order to verify our theory. We discovered that straightforward use of the formula (5) with $\zeta = 1.0$ produces extremely noisy results that can be seen in Figure 2(a), where we recursively bipartitioned vector $\bar{z}_0$ of the polygonized image presented in Figure 1(b). At the same time, it is clear that $\bar{z}_0$ has nontrivial components in the direction of the solution and we can take $\bar{z}_0$ as an initial guess for an iterative eigenvalue solver that can be computed in linear time. This means that we can solve the problem running LMGA to solve (3) with the initial iterate $\bar{x}_0$ with the following components

$$\bar{x}_0^k = \sqrt{d_k} + \zeta\,\|A\|_E\,\varepsilon_{\text{mach}}, \quad k = 0, 1, , \ldots, n-1 \tag{8}$$



(a) initial iterate $\bar{x}_0$ (b) first eigenvector $\hat{z}_0$ (c) Fiedler vector $\hat{z}_1$

Fig. 2. (a) Initial iterate $\bar{x}_0$ for LMGA, (b) rescaled eigenvector $\hat{z}_0$ ($\tilde{\lambda}_0 = 1.88469e-16$), (c) rescaled Fiedler vector $\hat{z}_1$ ($\tilde{\lambda}_1 = 1.53525e-06$)

that, due to the nature of the Normalized Cuts objective, incorporate information about the weighted degrees d_k of the corresponding graph nodes v_k.

Let $(\tilde{\lambda}_i, \tilde{z}_i)$, $i = 0, 1, \ldots, m-1$ be a set of m eigenpairs that we managed to compute with the LMGA such that $\tilde{\lambda}_0 < \tilde{\lambda}_1 < \cdots < \tilde{\lambda}_{m-1}$. We would like to examine the Fiedler vector $\tilde{z}_1$ and to verify our hypothesis that $\tilde{z}_0 \neq c\,\mathbf{1}$, $c \in \mathbb{R}$ may contain nontrivial information about the polygonized image P. In order to visualize our results we rescale components of $\tilde{z}_i$ as follows: $\hat{z}_i^k = (\tilde{z}_i^k - \min_j \tilde{z}_i^j)/(\max_j \tilde{z}_i^j - \min_j \tilde{z}_i^j)$ where $i = 0, \ldots, m-1$, $k = 0, \ldots, n-1$, so that $\hat{z}_i^k \in [0, 1]$. Next, we will apply a recursive bisection procedure to the rescaled vectors $\hat{z}_0$ and $\hat{z}_1$, splitting the vectors in two subvectors relative to the sample mean of its components. In Figure 2 we present results that we obtained in double IEEE-754 precision after 12 steps of the bisection procedure used to split and visualize vectors $\bar{z}_0$, $\hat{z}_0$ and $\hat{z}_1$ generated for the polygonized version of the image from the Berkeley Segmentation Data Set [12] shown in Figure 1. As expected, due to the presence of rounding errors, both the first (trivial) eigenvector $\hat{z}_0$, and the Fiedler vector $\hat{z}_1$, have more than two nodal domains, however, neither $\hat{z}_0$ (Figure 2(b)) nor $\hat{z}_1$ (Figure 2(c)) produce satisfactory segmentation. The corresponding eigenvalues are well separated: $\tilde{\lambda}_0 = 1.88469e-16$ represents numerical zero, while $\tilde{\lambda}_1 = 1.53525e-06$ is the nontrivial eigenvalue corresponding to the Fiedler vector.

Alternatively, we can attempt to construct segmentation using a set of the few of the lowest eigenvectors of the graph Laplacian. A multidimensional partitioning method can be used to split the computed eigenvector subspace into segments representing coherent visual scenes in an image. The larger the eigenvalue is that the eigenvector corresponds to, the more nodal domains in the image it will reveal. Note that the use of eigenvectors corresponding to larger eigenvalues has its downside – it may result in an oversegmentation of an image [8] due to the presence of a large number of nodal domains. Additionally, we can include the first, trivial, eigenvector into this set.

Depending on the implementation, multidimensional partitioning methods, such as the K-means and Mean-Shift [17], have computational complexity comparable and even higher than that of sparse eigensolvers, including LMGA. As a low complexity alternative, we developed an implementation of the Mean-Shift algorithm that we call 'Spatially Truncated Mean Shift' (STMS). We define the STMS iteration as follows. Let $Z = (\hat{z}_0, \hat{z}_1, \ldots, \hat{z}_{m-1})$ denote rescaled eigenvectors corresponding the first m smallest eigenvalues of the matrix W (2). Let $Y = Z^T = (y_0, y_1, \ldots, y_{n-1})$, where $y_i \in \mathbb{R}^m$. Then Spatially Truncated Mean Shift iteration takes the following form:

$$y_i^{(\tau+1)} = \sum_{l \in P^i_{\text{nbr}}} \frac{e^{-\|y_i^{(\tau)} - y_l\|_2^2/(\sqrt{2\pi}\,\sigma_{il})}}{\sum_{j \in P^i_{\text{nbr}}} e^{-\|y_i^{(\tau)} - y_j\|_2^2/(\sqrt{2\pi}\,\sigma_{ij})}}\, y_i, \tag{9}$$

where $y_i^{(0)} \overset{\text{def}}{=} y_i,\ i = 0, 1, \ldots, n-1;\ \tau = 0, 1, \ldots;\ \sigma_{ij}$ is sample variance of the shifted iterate $y_i^{(\tau)} - y_j$ and P^i_{nbr} is the set of indices of the neighbors of the data point y_i representing graph vertex v_i. Spatial truncation amounts to computing weights using only neighboring data points y_i instead of all $n-1$ data points, that is, we are computing locally a reweighted solution. Since the $\hat{z}_i$ are distinct representations of the same image and each can be separately treated as a suboptimal solution to the segmentation problem, we expect that this scheme will suffice for the identification of distinct nodal domains. The computational complexity of the Spatially Truncated Mean Shift is $O(n\, p_{\text{nbr}}\, k\, \tau)$, where p_{nbr} is the size of the largest set of neighbors P_i^{nbr}, while the computational complexity of the classical Mean Shift method is $O(n^2\, k\, \tau)$. Since $p_{\text{nbr}} \ll n$ we should see a significant speedup.

Our segmentation procedure can be summarized as an algorithm that applies Spatially Truncated Mean Shift (9) to the eigenvector subspace generated with the Lanczos Method with Guaranteed Accuracy on a VISTA-preprocessed image. We can formalize this algorithm that we call 'Spectral Segmentation on Polygonal Grids', or SSPG, as follows:

Algorithm 1 (Spectral Segmentation on Polygonal Grids (SSPG)).

1. *Construct graph affinity matrix W of a VISTA-polygonized image.*
2. *Apply LMGA to solve the eigenvalue problem* (3) *with the initial iterate set to* (8) *to compute 5 of the smallest distinct eigenvalues and the corresponding eigenvectors of W.*
3. *Apply STMS* (9) *to partition appropriately rescaled eigenvectors, including the eigenvector corresponding to the trivial solution.*
4. *If the number of identified segments is smaller than the number of requested segments, return to the step 2, increasing the number of eigenvectors to be computed by 5.*
5. *Visualize the results by assigning the same color to the pixels that belong to the same cluster identified by STMS.*

Note that for the BSDS images Algorithm 1 finds the requested number of segments in one or two steps, while for more challenging large images, such as images from remote sensing applications, it typically converges in about three to five iterations. Iteratively repeating step 8 is relatively inexpensive as the LMGA does not use reorthogonalization.

Table 1. Quantitative evaluation of segmentation results

Image	SSPG $(\rho_d, \rho_q, \rho_{cc}, \rho_{\mathbf{g}})$	MNC $(\rho_d, \rho_q, \rho_{cc}, \rho_{\mathbf{g}})$	SWA $(\rho_d, \rho_q, \rho_{cc}, \rho_{\mathbf{g}})$
1(a)	(0.47, 0.72, 0.90, **0.67)**	(0.53, 0.65, 0.97, **0.69)**	(0.65, 0.72, 0.62, **0.66)**
4(a)	(0.33, 0.64, 0.97, **0.59)**	(0.26, 0.59, 0.69, **0.47)**	(0.56, 0.68, 0.19, **0.41)**
4(b)	(0.70, 0.93, 0.19, **0.50)**	(0.70, 0.95, 0.28, **0.57)**	(0.50, 0.85, 0.89, **0.72)**
4(c)	(0.45, 0.51, 0.97, **0.58)**	(0.55, 0.85, 0.95, **0.76)**	(0.55. 0.76, 0.37, **0.53)**
4(d)	(0.41, 0.86, 0.83, **0.66)**	(0.32, 0.81, 0.92, **0.62)**	(0.55, 0.88, 0.12, **0.39)**

4 Experimental Results

In order to evaluate the performance of the SSPG algorithm we carried out a set of tests that compares its C/C++ implementation to the Multiscale Normalized Cuts (MNC) [6] and to the Segmentation by Weighted Aggregation (SWA) [7]. We ran our experiments on a 2.4 GHz two dual-core AMD Opteron 64-bit workstation with 16 GB of RAM, and we used test images 1(a) and 4(a) – 4(d) from the Berkeley Segmentation Data Set (BSDS) [12]. We report SSPG and MNC execution times in the CPU seconds and in the wall-clock seconds for SWA, as it does not provide a built-in CPU timer. We report the number of clusters found by SSPG and MNC algorithms and the level at which SWA results had been produced. In Figure 3, we present segmentation of the 321 × 481 pixel BSDS image 1(a) produced by the SSPG, MNC and SWA algorithms along with the execution times of these algorithms in seconds. Likewise, in Figure 4 we present segmentation results and execution times for for the 481 × 321 pixel BSDS images 4(a) – 4(d).

Additionally, we carried out a quantitative evaluation of the segmentation results produced by the SSPG, MNC and SWA algorithms using the BSDS human segmentation (Figures 3(a), 4(e) – 4(h)) as the reference images. Our test results are summarized in Table 1, where for each test case we computed

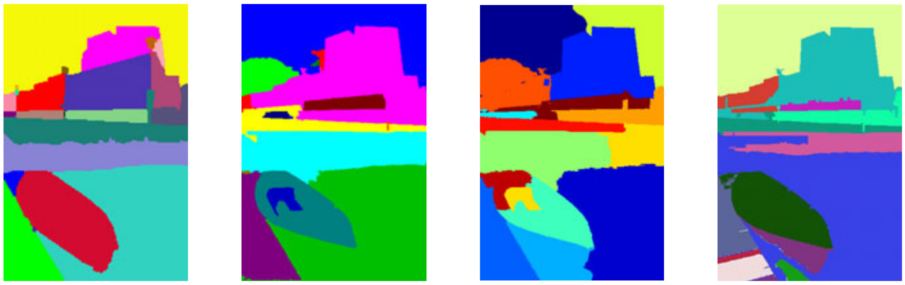
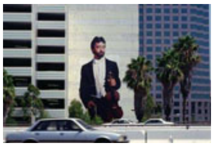
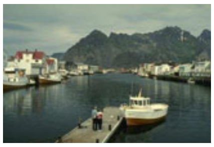
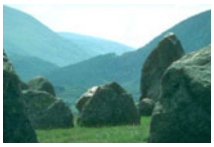
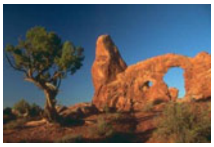

(a) reference18 clust (b) SSPG 4 sec; 14 clust (c) MNC 92 sec; 15 clust (d) SWA 4 sec; 11th level

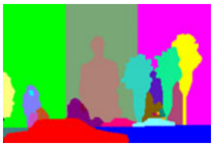
Fig. 3. (a) Test reference image (human segmentation from BSDS); (b) SSPG segmentation results; (c) MNC segmentation results; (d) SWA segmentation results

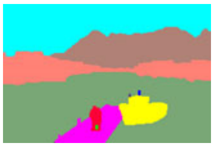

(a) 481 × 321 pixel image

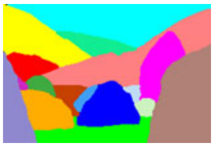

(b) 481 × 321 pixel image

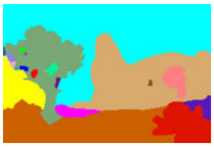

(c) 481 × 321 pixel image

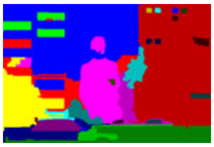

(d) 481 × 321 pixel image

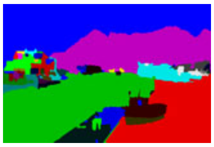

(e) reference 17 clust

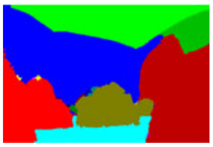

(f) reference8 clust

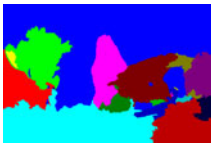

(g) reference17 clust

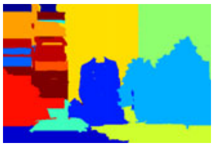

(h) reference15 clust

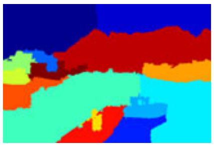

(i) SSPG 3 sec; 26 clust

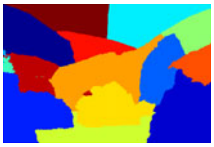

(j) SSPG 3 sec; 26 clust

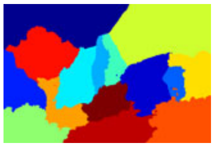

(k) SSPG 3 sec; 23 clust

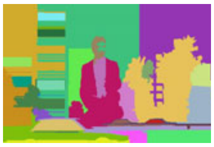

(l) SSPG 3 sec; 19 clust

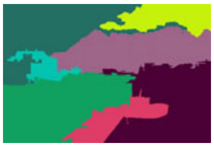

(m) MNC 106 sec; 15 clust

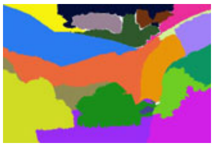

(n) MNC 103 sec; 15 clust


(o) MNC 55 sec; 15 clust


(p) MNC 61 sec; 15 clust


(q) SWA 4 sec; 10th level


(r) SWA 4 sec; 11th level


(s) SWA 4 sec; 10th level

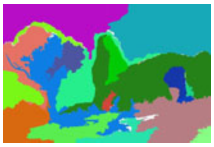

(t) SWA 4 sec; 10th level

Fig. 4. (a) – (d) Original BSDS images; (e) – (h) test reference images (human segmentation from BSDS); (i) – (l) SSPG segmentation results; (m) – (p) MNC segmentation results; (q) – (t) SWA segmentation results the following characteristics, commonly used for segmentation evaluation [18]: the detection rate $\rho_d = \sharp(S \cap R)/\sharp R$, the quality rate $\rho_q = |S \cap R|/|S \cup R|$, and the connectivity coefficient $\rho_{cc} = 2\ \min(\sharp S, \sharp R)/(\sharp S + \sharp R)$, where R is the reference image, $S = \cup S_i$ is a set of computed segments S_i such that $|S_i \cap R|/|S_i| > 0.5$, $\sharp(A)$ is the number of segments in A and $|A|$ is the number of pixels in A. Additionally, we report geometric mean $\rho_g = (\rho_d\, \rho_q\, \rho_{cc})^{1/3}$ as a characteristic of the overall quality of segmentation [18].

Upon the examination of the Figures 3 and 4 it is clear that the SSPG and SWA algorithms produce segmentation roughly 20-30 times faster than MNC. SSPG computes meaningful segmentation in time comparable and even superior to that of SWA. Analyzing Table 1, we can see that the quality of SSPG segmentation is overall comparable to that of MNC and SWA. Low values of ρ_{cc} produced in a few instances by all three algorithms are the consequence of the fact that some of the identified clusters may consist of a few disjoint segments.

5 Conclusions

We show that on polygonal grids the eigenvector subspace, corresponding to just a few of the lowest eigenvalues of the graph Laplacian, contains sufficient information necessary for obtaining meaningful segmentation, while the Fiedler vector alone is not always suitable for spectral segmentation. We also show that the eigenvector, corresponding to the trivial solution, often carries nontrivial information about the nodal domains in the image and can be used as an initial guess for an iterative eigensolver. Favorable performance of the developed algorithm is achieved through the polygonal coarsening at the preprocessing steps, the choice of the initial iterate for the LMGA, and the use of the STMS method on the postprocessing stage.

References

1. Fiedler, M.: Algebraic Connectivity of Graphs. Czechoslovak Math. J. 23(98), 298–305 (1973)
2. Biyikoğlu, T., Leydold, J., Stadler, P.F.: Laplacian Eigenvectors of Graphs. Perron-Frobenius and Fraber-Krahn Type Theorems, vol. 1915. Springer, Heidelberg (2007)
3. Dhillon, I.S., Guan, Y., Kulis, B.: Weighted Graph Cuts without Eigenvectors: a Multilevel Approach. IEEE Transactions on Pattern Analysis and Machine Intelligence 29(11), 1944–1957 (2007)
4. Yu, S.X., Shi, J.: Multiclass Spectral Clustering. In: International Conference on Computer Vision, Nice, France, pp. 11–17 (2003)
5. Yu, S.X.: Segmentation using multiscale cues. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 70–77 (2004)
6. Cour, T., Benezit, F., Shi, J.: Spectral Segmentation with Multiscale Graph Decomposition. In: IEEE International Conference on Computer Vision and Pattern Recognition, CVPR (2005), http://www.seas.upenn.edu/~timothee/software/ncut_multiscale/ncut_multiscale.html

7. Sharon, E., Galun, M., Sharon, D., Basri, R., Brandt, A.: Hierarchy and Adaptivity in Segmenting Visual Scenes. Nature 442, 810–813 (2006), http://www.wisdom.weizmann.ac.il/~swa/index.html
8. Malik, J., Belongie, S., Leung, T., Shi, J.B.: Contour and Texture Analysis for Image Segmentation. International Journal of Computer Vision 43(1), 7–27 (2001)
9. Cullum, J.K., Willoughby, R.A.: Lanczos algorithms for large symmetric eigenvalue computations, vol. 1. Society for Industrial and Applied Mathematics, Philadelphia (2002)
10. Matsekh, A.M., Shurina, E.P.: On Computing Spectral Decomposition of Symmetric Matrices and Singular Value Decomposition of Unsymmetric Matrices with Guaranteed Accuracy. Optoelectronics, Instrumentation and Data Processing 43(2), 159–169 (2007)
11. Prasad, L., Skourikhine, A.N.: Vectorized Image Segmentation via Trixel Agglomeration. Pattern Recognition 39, 501–514 (2006)
12. Martin, D., Fowlkes, C., Tal, D., Malik, J.: A Database of Human Segmented Natural Images and its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics. In: Proc. 8th Int'l Conf. Computer Vision, vol. 2, pp. 416–423 (2001)
13. Shi, J.B., Malik, J.: Normalized Cuts and Image Segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence 22(8), 888–905 (2000)
14. Godunov, S.K., Antonov, A.G., Kiriljuk, O.P., Kostin, V.I.: Guaranteed accuracy in numerical linear algebra. Kluwer Academic Publishers Group, Dordrecht (1993); ISBN 0-7923-2352-1. Translated and revised from the 1988 Russian original
15. Matsekh, A.M.: The Godunov-Inverse Iteration: a Fast and Accurate Solution to the Symmetric Tridiagonal Eigenvalue Problem. Applied Numerical Mathematics 54(2), 208–221 (2005)
16. Demmel, J.: Lecture Notes S267. U.C. Berkeley (1996), http://www.cs.berkeley.edu/~demmel/cs267
17. Comaniciu, D., Meer, P.: Mean Shift: A Robust Approach toward Feature Space Analysis. IEEE Trans. Pattern Analysis Machine Intell., 603–619 (2002)
18. Sturm, U., Weidner, U.: Further investigations on segmentation quality assessment for remote sensing applications. In: ISPRS Hannover Workshop 2009 High-Resolution Earth Imaging for Geospatial Information, Hannover, Germany (2009)

Parallel Kriging Algorithm for Unevenly Spaced Data

Jacek Strzelczyk and Stanislawa Porzycka

Department of Geoinformatics and Applied Computer Science,
Faculty of Geology, Geophysics and Environmental Protection,
AGH University of Science and Technology, Krakow

Abstract. In this work a new parallel kriging algorithm is presented. It is aimed at dealing with unevenly spaced data. The algorithm takes into consideration the spatial distribution of data points. The value in the interpolation points with similar neighbors' groups of sampled data points is calculated only once and then copied from one to another. The algorithm was tested for Permanent Scatterers Interferometry Synthetic Aperture Radar (PSInSAR) data. Efficiency results were compared with the previous version of proposed parallel kriging algorithm for unevenly spaced data.

Keywords: parallel kriging, unevenly spaced data, PSInSAR.

1 Introduction

The recently developed remote sensing systems provide us with a large amount of various data. These data carry very precise information about the objects or phenomena on the Earth surface and in this way they become an irreplaceable tool in environmental studies. They can be useful for geologists and geophysics to monitor or predict natural hazards like earthquakes, ground displacements or floods. The satellite data very often support decision-making process but before that they have to be thoroughly analyzed. A lot of satellite systems measure the particular parameters only at specific points on the ground. It may lead to the problem of interpolating the values of a parameters at locations without measurements. In case of data that are spatially or temporally correlated one of the best method of interpolation is *kriging*. This geostatistical technique provides a possibility to obtain highly accurate results. The main problem is that the kriging method is very computationally expensive especially when there is a need to apply it for a large number of unevenly spaced data points. To overcome this obstacle parallel computations can be used. In this work a parallel kriging algorithm is developed. The presented solution takes into consideration strong irregularity of the location of sampled data points. It was tested for satellite PSInSAR data that derive information about velocity of slow ground deformations.

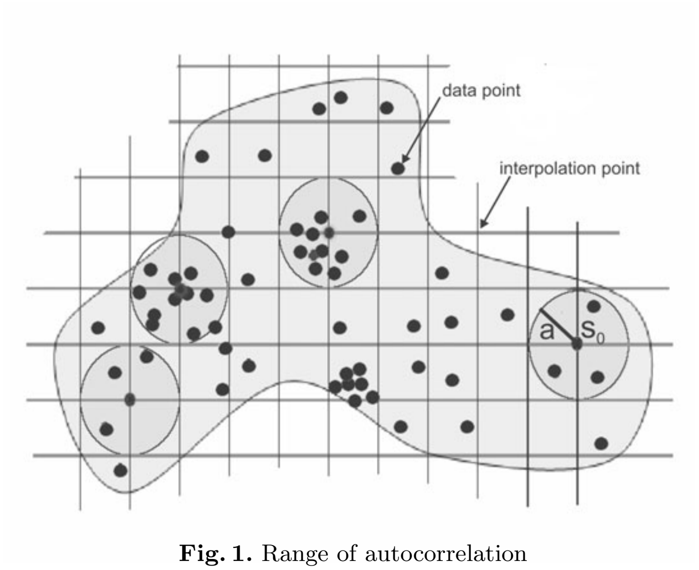


Fig. 1. Range of autocorrelation

2 Kriging Algorithm

Geostatistics is a branch of applied mathematics which was developed in the early fifties in the mining industry. Nowadays, the geostatistics is used in a wide range of scientific and industrial fields where the need to study spatially and/or temporally correlated data is essential. The geostatistical methods can be used to interpolate values of a particular parameter at unsampled locations. Interpolation, together with preliminary analysis, constitute a four-step procedure. In the first step, the statistical distribution of data has to be described. Then, in the second step, the spatial variability of data is modeled. Based on the obtained results the interpolation of parameter is performed in the third part of analysis. In the last step the estimation variance is computed for each interpolation point. In this work the authors focus on the third step of the described geostatistical procedure. This step concerns application of the interpolation algorithm.

Kriging is a geostatistical method of interpolation. It gives a possibility to interpolate values of parameter for regular grid based on both - evenly and unevenly spaced data. The estimation of a parameter value at location s_0 is based on the values of n neighboring data points s_i [1]. Number of neighboring data points (n), that have influence on the parameter value estimated in s_0, depends on the range a of spatial (temporal) autocorrelation of data (Figure 1).

In geostatistics the spatial (temporal) autocorrelation is determined using semivariogram function. The semivariogram is made as Equation 1

$$\widetilde{\gamma}(\widetilde{h_j}) = \frac{1}{2N}\sum_{i=1}^{N}(z(s_i) - z(s_i + h))^2, \forall h \in \widetilde{h_j} \qquad (1)$$

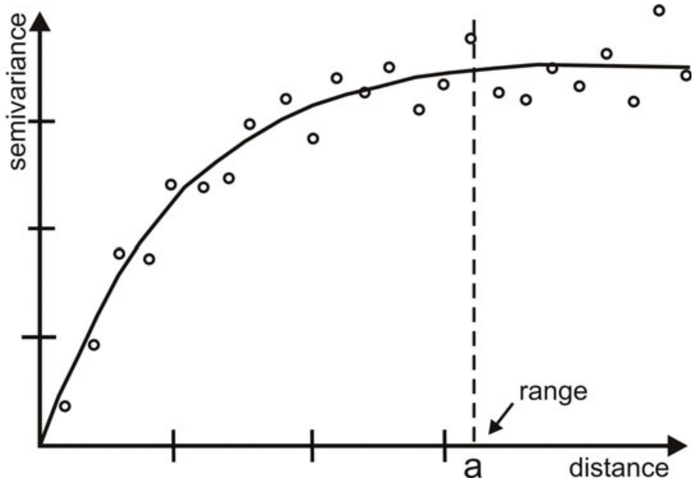


Fig. 2. Empirical semivariogram fitted by theoretical model

where:

$\widetilde{\gamma}$ - value of the semivariogram,
h_j - range of the distance,
N - number of sampled data point pairs that are located within the distance h,
$z(s_i)$ - values of parameter at location s_i,
h - distance.

Semivariogram plots the semivariance between two data points as a function of distance. The empirical values of semivariogram fitted by theoretical model (spherical model) were presented in the Figure 2. The range of autocorrelation a can be clearly seen on the semivariogram. The data within the distance larger than a are not correlated.

One of the most widely used type of kriging is ordinary kriging. The prediction is made as in Equation 2

$$\widehat{z} = \sum_{i=1}^{n} w_i(s_0) z(s_i) \tag{2}$$

where w_i are kriging weights. They are also determined based on the semivariogram function. In order to perform parameter interpolation for each interpolation point s_0 the kriging weights have to be determined by solving the equation system shown in Equation 3

$$\begin{bmatrix} w_0(s_0) \\ : \\ w_n(s_0) \\ \varphi \end{bmatrix} = \begin{bmatrix} \gamma(s_1,s_1) & \dots & \gamma(s_1,s_n) & 1 \\ : & : & : & : \\ \gamma(s_n,s_1) & \dots & \gamma(s_n,s_n) & 1 \\ 1 & 1 & \dots & 0 \end{bmatrix}^{-1} \begin{bmatrix} \gamma(s_0,s_1) \\ : \\ \gamma(s_0,s_n) \\ 1 \end{bmatrix} \tag{3}$$

where $\gamma(s_i, s_j)$ is the semivariance between data points s_i and s_j and φ is the *Langrange* multiplier.

In the case of large data sets the solution of the linear system for each interpolation point is a computationally expensive procedure. In the previous research [2] it has been shown that kriging computational time grows with the number of interpolation points (linear dependence) and, what is more significant, with the number of neighboring data points (power dependence). For large data sets the kriging algorithm is too much time consuming to be run on a single PC, so that parallel computations have to be used.

3 Parallel Solution

3.1 Existing Parallel Kriging Algorithms

Several parallel kriging algorithms have been proposed in the literature ([4], [5]), but they are designed to deal with rather equally spaced data (Figure 3). They use a classic domain decomposition idea, where the whole interpolation area is divided into equal parts containing similar number of data points and equal number of interpolation points. Each part of the area is then assigned to a single CPU which computes the values of a parameter using the kriging algorithm. At the end, the results from all the CPUs are gathered to create a single map of the estimated values. In the case of unevenly distributed data this approach is not effective because in each interpolation subarea a different number of the data points is located. Since the complexity of the kriging algorithm depends on the number of data points in the neighborhood of the interpolation point, the amount of computations on each CPU is different.

3.2 Proposed Solution

In the current work the authors propose an improved parallel kriging algorithm designed to efficiently work with unevenly spaced data. The first version of this algorithm was described in [6]. As opposed to the parallel kriging solutions mentioned above it did not divide the interpolation area into equal subareas. At the beginning the algorithm sequentially calculated the number of the data points in the circular neighborhood of each interpolation point. The radius of the neighborhood a was determined by the semivariogram model. After calculating the number of neighboring data points, the algorithm grouped the interpolation points into ten classes, where points with the similar number of neighbors were assigned to the same class. As a result, points with similar computation times were gathered together in ten classes (Figure 4). In the next step, each class was scattered over all CPUs (Figure 5). Each processor got nearly the same number of points to interpolate. Since those points were grouped into classes on the base of the neighbors number, approximately equal amount of computations needed to be executed on each CPU. The interpolation results were then gathered from all processors and saved as one area by the master CPU. Application of this algorithm considerably decreased the interpolation time for unevenly spaced data. Nevertheless, the total interpolation time can still be quite long for a real data set.

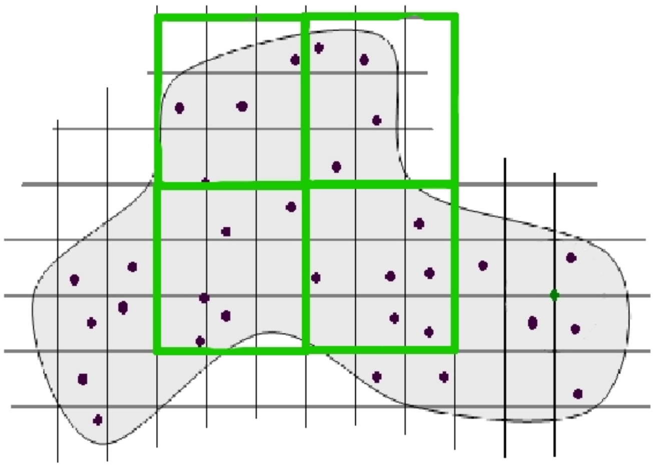


Fig. 3. Classical decomposition of spatial data

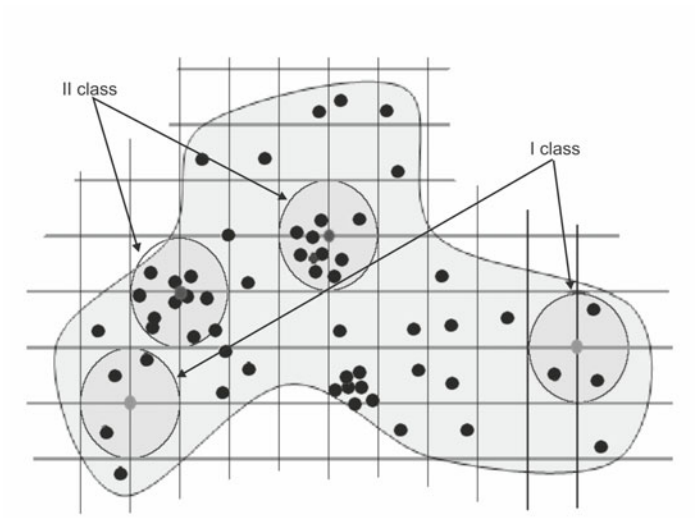


Fig. 4. Points with similar number of neighbors grouped into classes (only two first classes of ten presented)

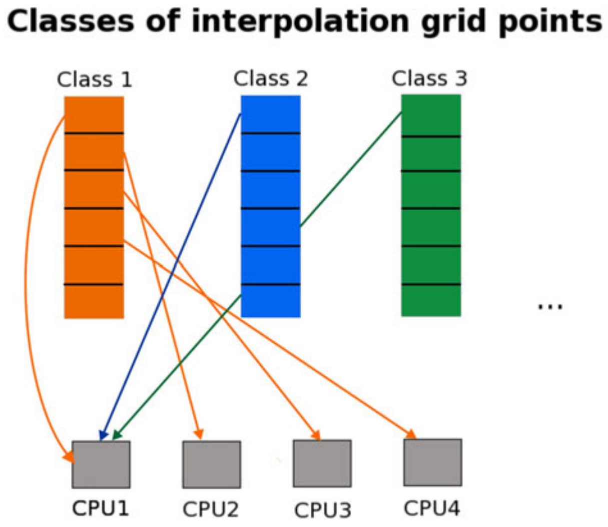


Fig. 5. Distribution of interpolation points' classes between CPUs

To shorten the calculation time even more, the authors propose a modification of the described parallel kriging algorithm. Some interpolation points have very similar set of neighboring data points. In such case, the results of interpolation obtained in two (or more) interpolation points differ very slightly ([5]). Therefore, the authors propose to perform the calculation of the searched parameter only in one of the interpolation points with similar neighborhood. The interpolated value (e.g. ground deformation velocity) is then copied to the other points. To perform this operation one condition has to be fulfilled - *Sorensen similarity index* describing the similarity of the sets of neighbors should be greater than the assumed threshold. The *Sorensen similarity index* is described by the Equation 4

$$QS = \frac{2 * C}{A + B} \tag{4}$$

where A is a number of neighboring data points for the first interpolation point, B is a number of neighboring data points for the second interpolation point and C is the number of neighboring data points common for both interpolation points. Increase of the threshold decreases the error, but elongates the calculation time. The authors set the threshold at the value of 0.95, which is a compromise between accuracy and calculation time.

The proposed modification can slightly change the results of the interpolation in comparison to the previous algorithm without copying. To investigate on this effect the values copying was implemented firstly in the 10th class, and then added to 9th, 8th, etc, since the last classes contain most computationally demanding interpolation points. The sums of differences between the values interpolated by the algorithm with copying in the consecutive classes' sets and the previous algorithm without copying were calculated and are shown on Figure 6.

Based on those results the authors decided to implement the values copying only in the last three classes 8th, 9th and 10th.

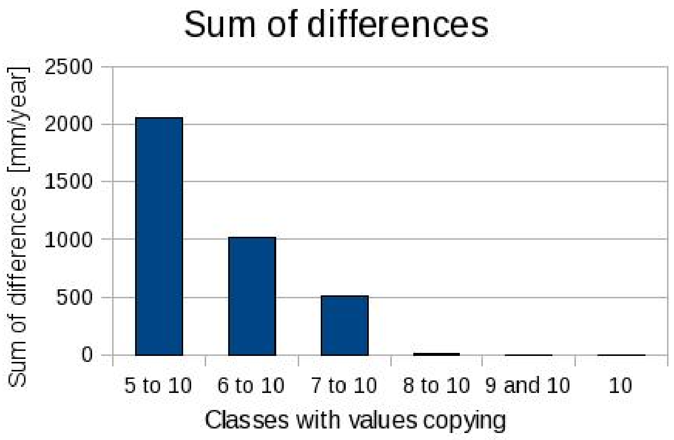


Fig. 6. Sum of differences between results estimated by the algorithm without values copying and algorithm with values copying in the consecutive classes sets

4 Application of the Proposed Algorithm for the PSInSAR Data

In this work both algorithms - previous parallel kriging solution described in [6], and the improved algorithm presented here - were tested for PSInSAR (Permanent Scatterers Interferometry Synthetic Aperture Radar) data. In the PSInSAR method the large sets of the satellite radar images are used to detect slow, long-term ground deformations [3]. The PSInSAR technique derives information about terrain deformations only at PS (Permanent Scatterers) points that are a stable radar targets. These points correspond mostly with a man-made objects on the ground like buildings, bridges, viaducts etc. As a result of the PSInSAR method application the map of PS points with determined values of deformations velocity (mm/year) is obtained. The spacing of a PS points is usually irregular and their density can be very high. In the urban areas it can be even higher than 100 points per km^2. The PSInSAR data used in this work provide information about slow deformations in the area of Upper Silesian Coal Basin (south Poland). In this region that covers 1200 km^2 about 120,000 unevenly spaced PS points were identified. The both versions of the parallel kriging algorithm (with and without values copying) were tested using the samples of 5,000, 10,000 and 15,000 data points.

Table 1 shows the distribution of interpolation points among the classes based on the number of neighboring data points. Class 1 groups interpolation points with the smallest number of neighbors, while class 10 groups points with the largest number of neighbors. It is clearly visible, that only about 4-5% of all interpolation points are in the last three classes, where the values copying was implemented. But since those points have the largest numbers of neighbors, their calculations are most complex and time consuming. Table 1 also shows the maximum number of neighbors in each class.

Table 1. Percentage of PS points and maximum number of neighbors in each class for different size of input data

5,000 PS			10,000 PS			15,000 PS		
Class	Max of neighbors	% of points	Class	Max of neighbors	% of points	Class	Max of neighbors	% of points
1	276	28.27	1	473	26.76	1	575	45.18
2	518	23.17	2	947	25.62	2	1150	28.17
3	800	15.98	3	1414	15.67	3	1727	12.57
4	1076	10.58	4	1867	10.93	4	2303	4.11
5	1234	8.10	5	2354	8.41	5	2870	1.48
6	1626	5.33	6	2808	5.20	6	3435	1.58
7	1846	4.47	7	3300	3.49	7	4023	1.74
8	2192	2.48	8	3781	2.45	8	4583	1.74
9	2432	1.11	9	4227	0.85	9	5170	1.71
10	2765	0.51	10	4733	0.62	10	5755	1.71

The calculation times of the previous version of the proposed algorithm (without values copying) and the new one (with values copying) are presented in Table 2. All computations in this study were conducted using 8 CPUs.

Table 2. Computation times of two parallel kriging algorithms and speedup of the newly proposed one

PS points	Algorithm without copying [s]	Algorithm with copying [s]	Speedup
5000	1772	1033	**1.72**
10000	13272	8096	**1.64**
15000	23943	14714	**1.63**

The results shows that the proposed modification of the parallel kriging algorithm can noticeably speedup the interpolation procedure. The time spent on the calculations using improved version of the parallel kriging algorithm was approximately 1.65 times shorter than in the version without copying.

In the current work the effectiveness tests of the algorithm with values copying were conducted. In those tests the calculation times of the newly proposed algorithm for different number of CPUs were measured. The results are shown on the Figure 7.

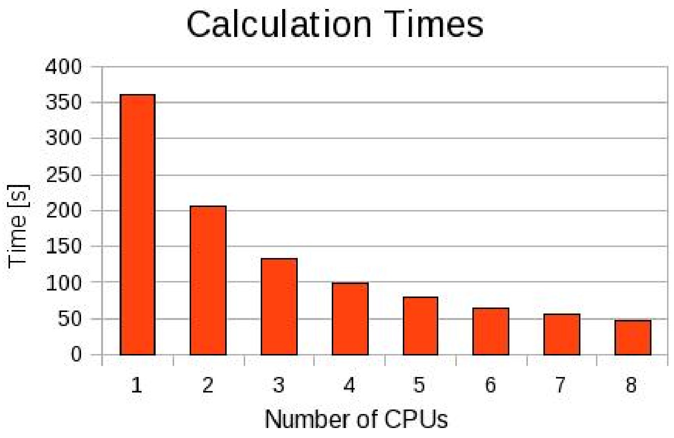


Fig. 7. Calculation times of the new parallel kriging algorithm for different number of CPUs

5 Conclusions

This study shows that the newly proposed solution can significantly decrease the interpolation time in case of unevenly spaced data. There are no major differences in the estimation results between the previous version of the algorithm and the newly described solution with values copying in 8th, 9th and 10th class. The computation time decreases with the increasing number of CPUs, but this relation slowly faints. Adding 7th and 8th CPU has no significant impact on the calculation time. Those results are in keeping with the Amdhal's Law, which says that the effectiveness of the algorithm decreases with the number of computation nodes. But in this case the speed of the decrease is acceptable.

References

1. Weckernagel, H.: Multivariate Geostatistics. Springer, Heidelberg (1995)
2. Lesniak, A., Porzycka, S.: Geostatistical Computing in PSInSAR Data Analysis. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009. LNCS, vol. 5544, pp. 397–405. Springer, Heidelberg (2009)
3. Ferretti, A., Prati, C., Rocca, F.: Permanent Scatterers in SAR Interferomenty. IEEE Transaction on Geoscience and Remote Sensing 39(1), 8–20 (2001)
4. Kerry, K.E., Hawick, K.A.: Kriging Interpolation on High-Performance Computers. In: Bubak, M., Hertzberger, B., Sloot, P.M.A. (eds.) HPCN-Europe 1998. LNCS, vol. 1401, pp. 429–438. Springer, Heidelberg (1998)
5. Gebhardt, A.: PVM Kriging with R. In: Proceedings of the 3rd International Workshop on Distributed Statistical Computing, DSC 2003 (2003), `http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Drafts/Gebhardt.pdf`
6. Strzelczyk, J., Porzycka, S., Leniak, A.: Analysis of ground deformations based on parallel geostatistical computations of PSInSAR data. In: Proceedings of 17th International Geoinformatics Conference (Fairfax 2009) (2009)

Parallel Particle-in-Cell Monte-Carlo Algorithm for Simulation of Gas Discharges under PVM and MPI

Christoph Schwanke, Andreas Pflug, Michael Siemers, and Bernd Szyszka

Fraunhofer Institute for Surface Engineering and Thin Films IST,
Bienroder Weg 54e, 38108 Braunschweig, Germany

Abstract. The simulation of complex problems in the field of plasma deposition technology requires the usage of parallel code running on modern multicore architectures. The inhouse developed *Particle-in-Cell Monte-Carlo* (PIC-MC) simulation environment has recently been ported from PVM towards MPI, which is the de-facto standard for parallelization by message passing. We measured a shorter latency time of MPI in comparison with PVM and determined the impact on the PIC-MC performance.

Keywords: Plasma-Simulation, Magnetron-Sputtering, Parallel-Computing, Message-Passing-Interface, D-Grid.

1 Introduction

Low pressure gas discharges play an important role in various industrial deposition methods such as magnetron sputtering and plasma enhanced chemical vapor deposition (PECVD). Due to continuously increasing demands on productivity and precision of such coating lines, the development of future deposition sources by pure empirical approaches becomes more and more elaborate.

To support the development of novel plasma sources, a parallel *Particle-in-Cell Monte-Carlo* (PIC-MC) simulation environment has been developed at Fraunhofer IST [1,2], which allows for description of transport phenomena and gas discharges in the low-pressure regime. *Low-pressure* in this context means, that the application of continuum dynamics for description of the transport and gas dynamics is not appropriate due to the large mean free paths.

This software has recently been ported from the parallel library PVM towards MPI, which is the de-facto standard for parallelization by message passing. Furthermore, an interface compatible with *Grid computing* has been implemented in order to address large problems with heavy CPU usage.

2 The PIC-MC Method

A detailed treatment of the density and the velocity distribution function suitable for the low-pressure regime is given by the Boltzmann transport equation,

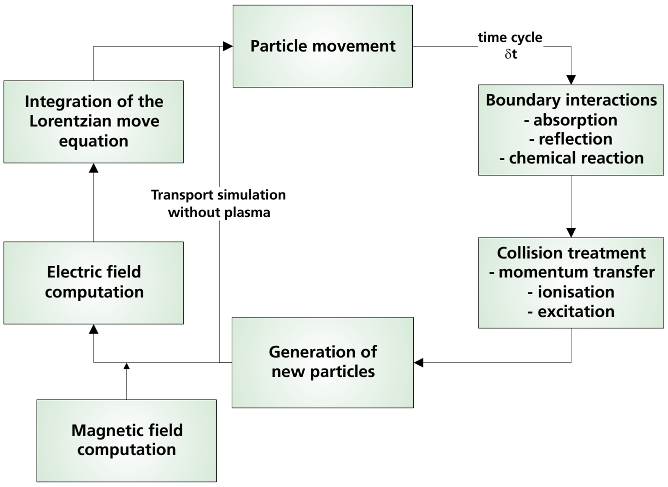

Particle movement
time cycle
δt
Boundary interactions
- absorption
- reflection
- chemical reaction
Collision treatment
- momentum transfer
- ionisation
- excitation
Generation of new particles
Magnetic field computation
Electric field computation
Integration of the Lorentzian move equation
Transport simulation without plasma

Fig. 1. Schedule of the main loop in a Particle-in-Cell Monte-Carlo (PIC-MC) simulation run

which can be solved statistically by a particle-based approach as given in [3]. For transport of neutral as well as charged particles, we combined this approach with electric and magnetic field computation modules, which is referred to as PIC-MC algorithm as described in detail in [4].

The main principle of this algorithm is to separately perform motion and collision of representative particles within discrete time steps δt as sketched in Fig. 1. *Representative* means, that one simulation particle represents a larger number of real particles depending on a weighting factor. For transport of neutral as well as charged particles, we combined this approach with electric and magnetic field computation modules.

We can apply any 3D CAD data of industrial coating setups with a hybrid geometry approach: A finite element mesh of arbitrarily shaped surfaces is embedded into a Cartesian grid for particle movement.

An example of a PIC-MC simulation result is shown in Fig. 2, where the simulated electron density distribution within a two-dimensional model of a bipolar magnetron discharge is plotted. The model consists of two sputtering targets at the bottom, a substrate on floating potential as well as a surrounding conducting chamber wall. The magnetron discharges operate at alternating current at a frequency of 100 kHz and target voltage amplitudes of ±300 V. In this simulation run, electrons, neutral Ar, ionized Ar+ as well the excitation process of metastable Ar* are considered.

3 Parallelization Scheme

The PIC-MC simulation software is parallelized by domain decomposition with a master-slave approach. A master process reads the input data comprising the

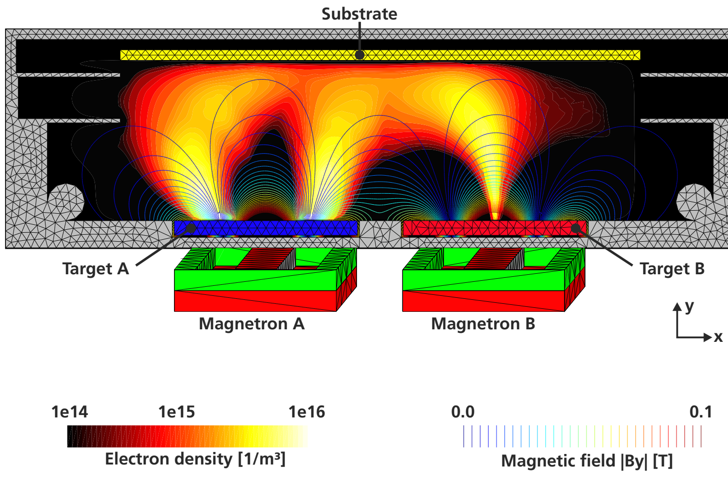


Fig. 2. Simulated electron density in a two-dimensional model for bipolar magnetron sputtering at a frequency of 100 kHz and a target voltage of ±300 V

whole geometrical information and all simulation parameters and manages a number of slave processes. The slave processes perform the computation of motion, collision and electrical field solving for a specified fraction of the simulation domain. Every time step the slaves exchange simulation particles among each other and get synchronized with the master.

All computations are performed on a Linux cluster based on the distribution "Debian Lenny" with 64bit. The cluster consists of seven nodes, whereof each node is equipped with Dual-Quadcore *Intel Xeon*® CPUs E5430 at 2.66 GHz and 16 GB of RAM. The communication between the nodes is realized with a GBit Ethernet switch.

4 Message Passing Interface

Our initial version of the simulation code uses PVM 3.4.5 for message passing. In the course of a public funded project "Plasma Technology Grid" within the *D-Grid* initiative[1], we ported this software towards OpenMPI 1.3.3. Nowadays, MPI is the de-facto standard for parallelization based on message passing and is most likely available on any external Grid resource.

In the master-slave architecture, the PIC-MC code uses two different modes to send a message. It can be passed from the master to every slave process or directly from slave to slave (e.g. transfering particle packages). For measuring the performance of both message passing interfaces, we started with a pure communication benchmark of these two modes (Fig. 3). On the one hand, a 32 byte sized message is passed 10^5 times from master to N slaves and backwards,

[1] `http://www.D-Grid.de`

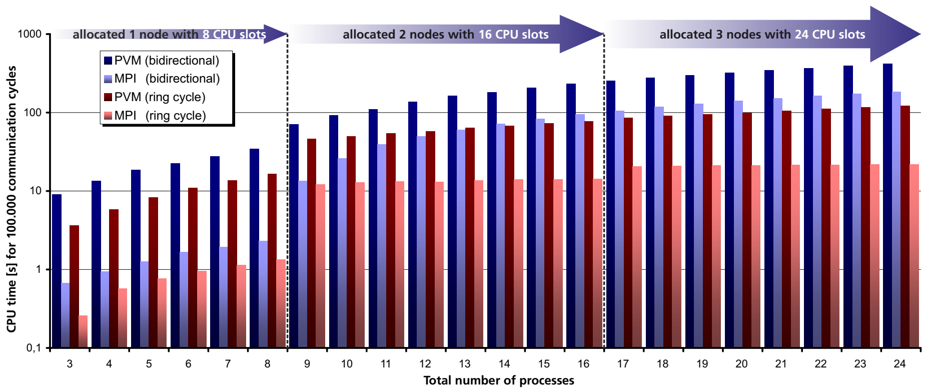


Fig. 3. Comparison of both message passing interfaces: Benchmark of a bidirectional master-slave (blue colored) and a ring-cycle slave-slave (red colored) communication

which is called the *bidirectional* mode. On the other hand N slaves are arranged in a ring, passing the message 10^5 rounds, called the *ring cycle* mode.

The result in Fig. 3 shows a significant performance advantage of the OpenMPI library over PVM. If only one node of the cluster is used (up to 8 CPUs), OpenMPI bypasses the network interface resulting in an even higher performance increase. In this case a difference of more than one order of magnitude between OpenMPI and PVM was measured. With more than 8 CPUs some messages will be transferred through the network, which immediately slows down the performance. Therefore the advantage of MPI's shorter latency time is getting detracted, but MPI still remains 2-5 times faster than PVM.

However, the OpenMPI tasks always consume 100% of a CPU core, respectively, even when in blocking receive mode, while the PVM tasks seem to produce a CPU load only outside blocking receive calls. The reason is that OpenMPI obviously tries to minimize latency while polling messages at the price of an always high CPU load, while PVM allows some additional latency during blocking receive.

We analyzed the impact of these findings on the new PIC-MC version based on OpenMPI via comparative benchmarks and found a performance increase of up to 90% in case of a rarefied gas flow simulations and about 30% in case of simulated magnetron sputter discharges. We assume that the performance difference between the PVM- and the OpenMPI based version is mainly due to the different latency times during message passing. To confirm this assumption we introduced an artificial latency time of 1 μs within the receive procedure of the OpenMPI based version. In this case, the performances of both versions were almost equal.

As another indication of our hypothesis, the performance difference is increased for simulation runs with high communication load, which is e. g. the case for gas flow simulation at low pressure. This is shown in Fig. 5, where the

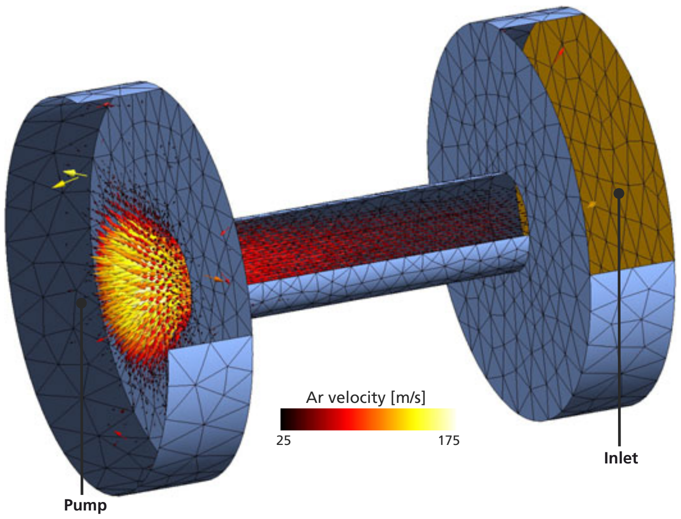


Fig. 4. Cylindrical 3D tube model, demonstrated with Argon velocity distribution at 100 mPa inlet pressure, as test case for benchmarking the particle-based gas flow simulation

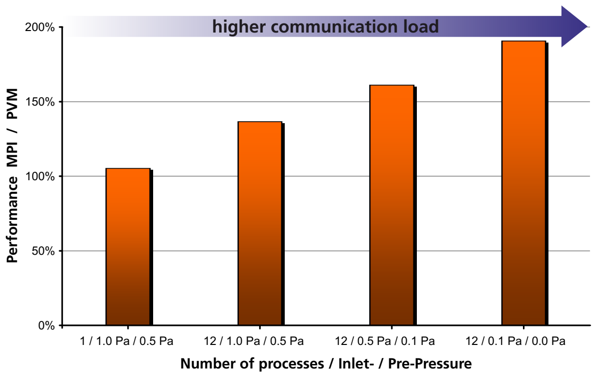


Fig. 5. Benchmark result of the 3D tube gas flow simulations (Fig. 4) using different pressures

relative performance difference is measured for an Argon flow through a cylindrical tube (Fig. 4): For an inlet pressure of 1.0 Pa, the performance boost of MPI using 12 slave processes is in the range of 40% while it increases up to 90% for a reduced inlet pressure. In the latter case, the computational load per process is reduced due to the decrease in collision activities. Thus the time fraction for particle data exchange and thus for message passing becomes higher.

Table 1. Simulation parameters of the particle-based gas flow simulation (Fig. 4)

Parameter	Quantity
Simulation grid	
Cell dimension Δx, Δy, Δz	20 mm
Number of cells	12,000
Particle species	
Ar weighting factor	$5x10^{12}$
Ar inlet-pressure p_{inlet}	0.1-1.0 Pa
Ar pre-pressure p_{pre}	0.0-0.5 Pa
3D Tube model	
Length	40 cm
Diameter	10 cm
Wall temperature	300 K
Simulation schedule	
Time step δt	1 µs
Total simulation time t_{max}	100 ms

5 Conclusion

For simulation of rarefied gas flows and low-pressure gas discharges we implemented a parallel Particle-in-Cell Monte-Carlo simulation environment, where the parallelization is realized with the PVM 3.4.5 library.

For compatibility with *Grid computing*, the porting of this software towards MPI was required; we used the library OpenMPI 1.3.3 for this purpose.

Comparative benchmarks demonstrated a performance boost of the MPI based version of up to 90% for applications with high communicative load. We attribute this to the more aggressive message polling strategy of OpenMPI focused on minimizing latency times.

The testing of the software within a *D-Grid* environment will be an issue of future work. Further planned developments are on a better integration of field solver modules into the particle transport code in order to exploit the CPU resources more efficiently.

Acknowledgment. The authors acknowledge the financial support of this project by the *Bundesministerium für Bildung und Forschung* (BMBF) under Contract No. 01IG09001A-I.

References

1. Pflug, A., Siemers, M., Szyszka, B.: Parallel DSMC Gasflow Simulation of an In-Line Coater for Reactive Sputtering. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 383–390. Springer, Heidelberg (2006)
2. Pflug, A., Siemers, M., Szyszka, B.: Design tools and simulations for plasma processing in large area coaters. In: Proc. 52nd SVC Annual Technical Conference Proceedings, pp. 364–369 (2009)
3. Bird, G.A.: Molecular Gas Dynamics and the Direct Simulation of Gas Flows. In: Monography. Clarendon Press, Oxford (1994)
4. Birdsall, C.K., Langdon, A.B.: Plasma Physics via Computer Simulation. In: Monography. Taylor & Francis Group (2004)

Monte Carlo Simulations of Spin Systems on Multi-core Processors

Marco Guidetti[1], Andrea Maiorano[2], Filippo Mantovani[3], Marcello Pivanti[1], Sebastiano F. Schifano[4], and Raffaele Tripiccione[1]

[1] Dipartimento di Fisica, Università di Ferrara and INFN Sezione di Ferrara, I-44100 Ferrara Italy
[2] Dipartimento di Fisica, Università di Roma "La Sapienza", I-00100 Roma Italy
[3] Deutsches Elektronen-Synchrotron DESY, D-15738 Zeuthen Germany, and INFN Sezione di Ferrara, I-44100 Ferrara Italy
[4] Dipartimento di Matematica, Università di Ferrara and INFN Sezione di Ferrara, I-44100 Ferrara Italy

Abstract. We optimize codes implementing Monte Carlo simulations of spin-glass systems for some multi-core CPU and GPU architectures. We consider both the binary *Ising* and floating-point *Heisenberg* spin-glass models in 3 dimensions. We provide performance figures for the Intel Nehalem quad-core and the IBM Cell/BE CPUs and the Nvidia Tesla C1060 GPU; for the binary model we also draw a comparison with the performance of dedicated computers, such as the *Janus* machine.

Keywords: Monte Carlo Simulations, Multi-core Architectures, Spin-Glass Systems.

1 Introduction

Spin models are ubiquitous in statistical mechanics; they are important tools to model and describe the properties of several condensed-matter systems and – equally importantly – they are emerging as fundamental paradigms of complexity [1]. The study of the properties of these systems relies critically on Monte Carlo simulations. The computational effort associated to these simulations is huge: a state of the art investigation requires computing resources in the order of tens of thousands of CPU- years. This huge effort is made manageable by the exploitation of the large amount of available parallelism, that can be exposed –up to some degree– on conventional architectures with reasonable programming effort. Over the years, carefully optimized application-driven machines have also been developed[2,3,4,5,6]; they have exploited available parallelism at unprecedented levels making very large simulations possible [7]. While application-driven solutions are still the most performing, the advent of new generation processors, such as multi-core CPUs and GPUs, offers new opportunities in this area. In this paper we assess the efficiency of these architectures for the computational problem at hand and compare with the performance of dedicated systems.

2 Spin Models

Spin models are defined in terms of generalized-spins, variables defined on the sites i of a discrete hyper-cubic D-dimensional lattice of linear size L. For discrete models, spins have values in a finite (usually small) set of values; in the simplest case (the Ising model) they take just two values, $s_i = \pm 1, i = 1 \cdots N = L^D$. For continuous models, spins are real variables; in the Heisenberg model, each spin is a 3D vector of unit length $\mathbf{s}_i$. A configuration C_k is an assignment of a spin value $s_i^{(k)}$ at all lattice sites. The energy of each configuration is a simple function of all spin values:

$$E(C_k) = -\sum J_{ij}\mathbf{s}_i^{(k)}\mathbf{s}_j^{(k)}, \tag{1}$$

the sum being taken only on nearest neighbor pairs of sites on the lattice, characterized by interaction parameters J_{ij}. The term $\mathbf{s}_i\mathbf{s}_j$ is a suitably defined product (e.g., the usual scalar product for Heisenberg spins or the ordinary arithmetic product for the Ising model). At finite temperature T, the Boltzmann distribution gives the probability that the system is in any given configuration:

$$P(C_k) \sim e^{-\beta E(C_k)}, \tag{2}$$

$(\beta = 1/T)$, so the energy function determines the properties of the system. Physically interesting informations on these systems are measured by statistical averages of configuration-dependent observables, $O(C_k)$, formally defined as

$$\langle\, O \,\rangle \;=\; \sum_{C_k} O(C_k)P(C_k). \tag{3}$$

Monte Carlo algorithms explore the space of configurations, generating sequences distributed in accordance with (2) (see e.g., [8] for an overview of Monte Carlo algorithms), so the averages of (3) are simply estimated as

$$\langle\, O \,\rangle \;\simeq\; \sum_{C_M} O(C_M), \tag{4}$$

where C_M labels configurations produced by the Monte Carlo procedure.

Monte Carlo algorithms proceed by changing the values of the spin at each and all sites of the lattice, following appropriate rules (see again [8]) that guarantee that Eq. 2 is respected; they depend only on the contribution of that spin to energy.

For Ising-like models, one specific Monte Carlo procedure, the so-called Metropolis algorithm, either leaves the spin value untouched or flips it, with probability depending on the value of the current spin and its neighbors. The spin-flip probability depends on the corresponding energy cost $\Delta E(s_i \to -s_i)$; from Eq. 1 we have:

$$\Delta E(s_i \to -s_i) = 2\sum_j J_{ij}s_i s_j \tag{5}$$

where, as already remarked, the sum extends only to neighbors of site i. The flip probability is unity if $\Delta E \leq 0$; otherwise it is $\exp(-\beta \Delta E)$.

For Heisenberg-like models the procedure must generate one point on the unit sphere, with probability modulated by the values of the neighbor spins. In this case fairly complex floating-point mathematics is necessary, including the computation of special functions.

The key point in either case is that energy, and the corresponding energy dependent distribution, is a function of the value of the spin and of its nearest neighbors only, so the algorithm can be applied concurrently on many grid points, exposing a very large amount of parallelism: if we consider a checkerboard partition of the grid we can update in parallel all white (or black) sites.

The values of the couplings J_{ij} characterize the properties of the system. If all J_{ij} are equal (and positive) these models describe the well-known behavior of a ferromagnetic material. If on the other hands the J_{ij} are randomly extracted, we have a true spin-glass model. A ferromagnet at zero temperature will run into one of the two fully magnetized states, with all spins aligned in the same direction. At finite temperature it still has two opposite partially magnetized states corresponding to free energy minima, and can eventually flip between them. For the spin-glass cases, where J_{ij} are randomly selected, the situation is much more complex: due to concurrent couplings the energy landscape becomes rugged, i.e. many almost degenerate local minima are present, and the number of Monte Carlo steps, and consequently the Monte Carlo time, needed to overcome energy barriers between these minima grows exponentially with system size; this is the main reason why simulations are so time consuming. Among several possible choices, we use a bimodal distribution $J_{ij} = \pm 1$ for the Ising model, and a Gaussian distribution with zero mean and unit variance for the Heisenberg model.

There are two different opportunities for parallelism in the simulation of these systems: first, we are physically interested in averages of the properties of the system over a large number of independent instantiations of the couplings (we call each such instantiation a *sample*); this implies a large number of independent simulations of systems that do not interact among them. We call this trivial but useful parallelism *external*. The second avenue for parallelism, we call it *internal*, exploits the opportunities – described before – of the Monte Carlo dynamics of *each* system. The challenge is then combining both opportunities in the most efficient way for each architecture. This has been recognized since many years by the spin glass community; simulation codes for tradition architectures have been carefully simulated, and even application-driven systems have been developed. Most of the state-of-the-art results in the last ten years critically depend on this computer related developments (see, among many others, the physical results of [7,9]).

In the following we proceed on this line of research, targetting new computer architectures (GPUs and multi-core CPUs) that have recently become available; the results we have obtained apply to this application and cannot be directly generalized to other scientific problems.

Algorithm 1. Bit-wise algorithm to update one spin. See the text for a definition of X_i.

```
Require: $\rho$ pseudo-random number
Require: $\psi = \min\{3, \text{int}(-(1/4\beta)\log\rho)\}$ {coded on two bits}
Require: $\eta = (\text{ not } X_i)$ {coded on two bits}
1: $c_1 = \psi[0] \text{ and } \eta[0]$
2: $c_2 = (\psi[1] \text{ and } \eta[1]) \text{ or } ((\psi[1] \text{ or } \eta[1]) \text{ and } c_1)$
3: $\sigma_i' = \sigma_i \text{ xor } (c_2 \text{ or } \text{ not } X_i[2])$
```

3 Simulation Algorithm

In this section we discuss ways to exploit the parallelism outlined in the previous section. A first approach to internal parallelism is *instruction-level*, using SIMD instructions within each computing-core, processing several spins in parallel. A further level of internal parallelism exploits *data-level* parallelism, partitioning the whole lattice across the cores, and updating each sub-lattice in parallel. In the following, we show how these options can be best combined for a few target architectures, for both discrete and continuous models.

3.1 Simulation Algorithms for the Ising Spin-Glass

Spin values for binary models can be coded in just one bit, while CPUs operate on long words of k bits (e.g., $k = 32, 64, 128$), so it is useful to combine internal and external parallelism. We proceed by mapping V ($V = 2, \ldots, 16$) binary-valued spins of $w = k/V$ lattice samples on a single CPU-word. We then use SIMD instructions to update in parallel V spins for each of w independent lattices, so we compound an *internal* parallelism of degree V and an *external* parallelism of degree w. We have used this approach first for the IBM/Cell processor (see [10] for details) and then ported it to the Intel architecture, using SSE instructions.

The actual steps of the algorithm (apart from random number generation) are first formally reduced to just a handful of logic operations, described by algorithm 1; the main advantages of this scheme are that it exploits the SIMD capabilities of the architecture leveraging on internal and external parallelism and that it does not use conditional statements. The very simple structure of algorithm 1 derives from the following considerations: first, we perform bit-wise transformations for the spin value at site i and coupling at edge (i, j):

$$s_i \in \{-1, 1\} \longrightarrow \sigma_i = (s_i + 1)/2 \in \{0, 1\} \quad , \tag{6}$$

$$J_{ij} \in \{-1, 1\} \longrightarrow \lambda_{ij} = (J_{ij} + 1)/2 \in \{0, 1\} \quad ; \tag{7}$$

We now write the energy cost associated to a spin flip (equation (5)) as $\Delta E(s_i \rightarrow -s_i) = 4X_i - 12$, with

$$X_i = \sum_j \lambda_{ij} \text{ xor } \sigma_i \text{ xor } \sigma_j . \tag{8}$$

(X_i is an integer in $\{0, 6\}$). Considering the bit representation of $X_i = \sum_k X_i[k] \times 2^k$, a negative energy cost (that is a unit flip probability) corresponds to $X_i[2] = 0$ (case i)). If, on the other hand, $X_i[2] = 1$ (case ii), the spin flips if, given a random number ρ in $[0, 1]$, the relation $\rho \leq \exp(-\beta \Delta E)$ holds; in this case, considering the bit-wise transformation, the condition reads

$$-\frac{1}{4\beta} \log \rho - X_i \geq -3 \quad . \tag{9}$$

The last relation can be manipulated further to allow an easy bit-wise implementation. First of all, in a three bit representation, not $X_i = 7 - X_i$; second, the outcome of testing Eq. 9 must be *or*'ed with the outcome of testing case i) above, making it possible to limit the representation of X_i to two-bits only; also, in case ii), the spin always flips when $(\log \rho)/(4\beta) \geq 3$; in addition, only the integer part of the first term on the r.h.s of Eq. 9 is needed. It turns out that, being $\psi = \min[3, \mathrm{int}(\log \rho)/(4\beta)]$, the spin flips if at least one of the two conditions $X_i[2] = 0$ and $\psi + (\text{ not } X_i)\%4 \geq 4$ is verified; this last condition reduces to the value of the last carry when adding the two-bit quantities ψ and $(\text{ not } X_i)\%4$. See again [10] for a more detailed description of this efficient but cumbersome sequence of transformations.

The mapping into machine words of $k = w * V$ bits must also apply to the quantities ψ, $X_i[0]$, $X_i[1]$, $X_i[2]$; in this way, the set of logical operations described above and needed to update a single spin may be efficiently performed in parallel for a set of k independent spins.

The choice and implementation of the pseudo-random number generator has a non-negligible impact on overall performance. We choose the reliable Parisi-Rapuano algorithm [11]; when exploiting internal parallelism, we must use different random numbers for the update process of each spin of a given sample; on the other hand, we can tolerate a small amount of correlation by sharing the same random number among different samples. These considerations impact on the best choice of w and V.

So far, we have discussed how to exploit instruction-level parallelism within a core. We now consider how to exploit multi-core parallelism. We divide the whole lattice into sub-lattices, and assign each partition to a different core. We split the lattice in C sub-lattices of contiguous planes (C is the number of cores), and we map each sub-lattice of $L \times L \times L/C$ sites onto a different core. Each thread, running on a different core, executes the program defined by a loop in which it first updates all its white spins and then updates all the black ones. White

Algorithm 2. Program of each thread

```
1: update the boundaries half-plane (indexes (0) and ($(L^3/C) - 1$)).
2: **for all** $i \in [1..((L^3/C) - 2)]$ **do**
3:     update half-planes ($i$)
4: **end for**
5: exchange half-plane (0) to the *previous core* and half-plane ($(L^3/C) - 1$) to the *next core*.
```

Algorithm 3. 3D Ising spin-glass: program executed on the host for the GP-GPU simulation

```
1: loop {loop on Monte Carlo steps}
2:     MCupdate(black)
3:     updateBorder()
4:     MCupdate(white)
5:     updateBorder()
6: end loop
```

and black spins are stored in data-structures called *half-planes*, each housing $L^2/2$ spins. Each core updates the half-planes of one color performing the steps described by algorithm 2. Each core houses a sub-lattice plus the boundary planes with the adjoining sub-lattices, which have to be updated at end of a sweep of the sub-lattice. Before performing such operation, the cores must be synchronized.

On the Cell processor[12] the details of the implementation vary; if the full simulation data-base can not stays within the local stores of the cores, DMA operations have to be carefully scheduled to move data from/to main memory with non-negligible performance penalty. For this reason, performances vary significantly as a function of the lattice size (see later).

On the Intel Nehalem processor[13], memory is shared (through a common L3 cache) so the implementation is conceptually simpler. We have implemented and compared the simulation program against:

– different libraries to handle intra-cpu parallelism: openMPI, openMP and pthread.
– shared and distributed memory allocation. In the shared case, the threads share the same data structure allocated by the main program, while in the distributed case each core copies the data items that it needs into a private structure.
– different compilers: we have used both *gcc* and *icc*, the Intel C compiler.

We obtain the best results using the *pthread* library, a *distributed* memory allocation, and the *icc* compiler, (performance gain is of the order of $10 - 15\%$, compared to the other options). We also find that a synchronization performed by *active waiting* is slightly better than a synchronization by pthread mutex-variables. The program has been written using intrinsic functions to map operations directly to SSE instructions, and can be compiled with a variable number of threads and variable degree of *internal* parallelism.

Simulations on GP-GPU use the same update algorithm used for CPUs, but the structure of the computation is slightly different. Following CUDA [14] terminology, a 3D lattice of side L is divided into a 2D-grid of $(L/2 \times L/2)$ sub-lattices. Each sub-lattice is a 3D-grid of $(2 \times 2 \times L)$ sites. This partition has been chosen for the following main reasons:

– ensure that each block fits on registers and shared memory of the GPU *streaming multiprocessors* (SM), for the most relevant sizes of $L = 16 \dots 128$,

Algorithm 4. 3D Ising spin-glass: program executed on the GP-GPU

```
Require: color = {black, white}
1: load a set of $(4 \times 4 \times L)$ points
2: apply MC-step to bulk's points
3: save bulk to memory {required to sync blocks}
```

– generate many blocks to keep active as long as possible each SM and hide memory access latencies,
– generate a large number of *warps* (groups of 8 instructions that can be executed in parallel) per SM to exploit *memory coalescing*, in order to improve the bandwidth between the SM and the memory.

The main simulation-program runs on the host, copies the lattice on the global memory of the GPU – replicating the surface planes – and launches several GPU kernels as described by algorithm 3. The *MCupdate()* kernel is invoked to update the white or black spins. As we need to update $L^3/2$ sites, the kernel runs on a thread-array configured as a 2D-grid of $(L/2 \times L/2)$ blocks, where each block is configured as a 3D-array of $(2 \times 2 \times L)/2$ threads. This configuration allows to have all threads of the block running while the update step is performed. The *updateBorder()* kernel is invoked on a grid of L blocks of L threads; it updates the surface planes of the lattice by performing memory copies in parallel .

The code executed by the GPU is organized as shown by algorithm 4. Each block loads a sub-lattice of $(4 \times 4 \times L)$ sites, including the bulk of spins to update, plus the planes of neighbors necessary for the Monte Carlo update step. As each block has been configured as a 3D-grid of $(2 \times 2 \times L)/2$ threads, this step is performed by 8 coalesced memory read operations. Each thread then applies the Monte Carlo update step to a single site, and stores the new value to memory. In our implementation we also used the multi-spin coded approach described above. We used one 32-bit word to map one spin plus the value of three coupling variables. This mapping allows to run in parallel up to 8 different simulations.

3.2 Simulation of the Heisenberg model

The Heisenberg model, as outlined in the introduction, uses floating-point arithmetics. Accuracy on long simulations requires that double precision be used throughout. In our tests, we have used a slightly more complex energy function

$$E = -\sum s_i^\alpha J_{ij}^{\alpha\beta} s_j^\beta, \tag{10}$$

where the interaction term J is a 3×3 symmetric matrix (summation of repeated greek indexes is implied), allowing to model anisotropic interactions among the spins; we expect that these models will become very important from the physical point of view in the near future. As far as implementation is concerned, this model implies a larger amount of floating-point computation and needs larger storage and higher memory bandwidth.

As in the *Ising* model, we have exploited instruction parallelism by updating in parallel two non-adjacent spins using SIMD instructions, and data parallelism by dividing the lattice in C sub-lattices, where C is the number of cores, each one housing $(L \times L \times L/C)$ spins. We have developed one implementation for the Intel Nehalem processor and, one for the Nvidia Tesla C1060 GPU [15].

On the Intel processor we use the *pthread* library to manage parallelism among the cores and the gcc compiler, as it allows to define *vector* variables; operations on such variables are automatically mapped on SSE instructions that update in parallel two non-adjacent spins of the sub-lattice. We also use vector versions of the log and exp functions – heavily used in this code – using intrinsics instructions.

On the GPU we follow the same approach of the *binary* case, but the structure of the thread-array is slightly more complex. As in the binary case, each block updates a sub-lattice of $(2 \times 2 \times L)$ sites, but due to larger register and shared memory requirements to store the variables of the model, the update procedure is performed by dividing the sub-lattice in blocks of $(2 \times 2 \times k)$ sites. The value of k that uses up all available memory space on the GPU streaming multiprocessor depends on the lattice size L, however $k = 16$ is an acceptable choice for most physically relevant values of L.

4 Results, Comparison and Conclusions

In this section we compare performance results for the codes and architectures described above. Our benchmark for the binary case is the *Janus* special-purpose machine [6], currently the most powerful system available for this class of simulations (Janus, on the other hand, does not support floating-point arithmetics).

We present results using two metrics, both relevant for physics applications. The *System spin Update Time* (SUT) is the average time needed by the Monte Carlo procedure to update *one* spin of *one* system (neglecting the w parameter if greater than 1); improving the SUT reduces the wall-clock time of the simulation, so this parameter is relevant for long simulations on large lattices; in this case, one usually deploys many (e.g., $128 \cdots 256$) independent simulations to accumulate statistics. The *Global spin Update Time* (GUT) is the average time to process one spin of all simulated systems using multi-spin encoding (taking into account the w parameter); in other words, GUT is simply SUT divided by the number of *replicas* that the simulation handles concurrently. Programs optimizing GUT allow to run efficiently simulations of smallish lattices with high statistics using a small number of computing elements (it is important to simulate efficiently also small lattices, since it is useful for physics analysis to compare the same quantities measured on lattices of several different sizes). All in all, SUT is a measure of how well we exploit internal parallelism, while GUT measures the combined benefits of internal and external parallelism. According to the physics program underlying the simulation, either of the metrics (or both) are relevant.

Table 1. System update time for the 3D Ising spin-glass (binary) model. I-NH (8-Cores) identifies a dual-socket quad-core Intel Nehalem board, while CBE (16-SPE) is a dual-socket Cell board

3D Ising spin-glass Model SUT (ns/spin)					
L	Janus	I-NH (8-Cores)	CBE (8-SPE)	CBE (16-SPE)	Tesla C1060
16	0.016	0.98	0.83	1.17	–
32	0.016	0.26	0.40	0.26	1.24
48	0.016	0.34	0.48	0.25	1.10
64	0.016	0.20	0.29	0.15	0.72
80	0.016	0.34	0.82	1.03	0.88
96	–	0.20	0.42	0.41	0.86
128	–	0.20	0.24	0.12	0.64

We collect our results for the Ising spin-glass model in tables 1 and 2. For each lattice size, GUT values are the best SUT times divided by the number of systems updated concurrently.

Table 3 presents performance data for the Heisenberg model on an Intel dual quad-core Nehalem and on a Tesla C1060 board (as mentioned before the Janus system does not support floating-point arithmetics, and we have not implement a code for the IBM Cell so far). For the GPU case, we also quote results for single-precision versions of the code. Indeed, the Tesla C1060 system has rather poor support for double-precision, while the more recent *Fermi* [16] architecture supports it efficiently: we regard our single-precision results as educated guesses of what the Fermi architecture may achieve with double-precision.

Figure 1 presents the relative speed-up of the Ising spin-glass and Heisenberg programs on the Intel system, as a function of the number of cores. The Ising spin-glass code scales linearly up to 4 cores for all lattice sizes except for L=16 where the sub-lattice allocated on each core is too small, making synchronization and memory access overheads too high. Using 8 cores the code scales almost

Table 2. Global update time for the 3D Ising spin-glass (binary) model, for the same systems as in the previous table. The number of systems simulated in parallel (w parameter) in the multi-spin approach is shown in parentheses.

3D Ising spin-glass Model GUT (ns/spin)					
L	Janus	I-NH (8-Cores)	CBE (8-SPE)	CBE (16-SPE)	Tesla C1060
16	0.001 (16)	0.031 (32)	0.052 (16)	0.073 (16)	–
32	0.001 (16)	0.032 (8)	0.050 (8)	0.032 (8)	0.31 (4)
48	0.001 (16)	0.021 (16)	0.030 (8)	0.016 (16)	0.27 (4)
64	0.001 (16)	0.025 (8)	0.072 (4)	0.037 (4)	0.18 (4)
80	0.001 (16)	0.021 (16)	0.051 (16)	0.064 (16)	0.22 (4)
96	–	0.025 (8)	0.052 (8)	0.051 (8)	0.21 (4)
128	–	0.025 (8)	0.120 (2)	0.060 (2)	0.16 (4)

Table 3. Simulation performance for the 3D Heisenberg model (SP = single-precision, DP = double-precision)

3D Heisenberg Model SUT (ns/spin)		
L	I-NH (8-Cores)	Tesla C1060
16	55.4 (DP)	–
32	38.0 (DP)	34.4 (SP) / 139.0 (DP)
48	32.5 (DP)	29.6 (SP) / 134.8 (DP)
64	29.6 (DP)	31.0 (SP) / 131.5 (DP)
80	29.9 (DP)	28.3 (SP) / 130.7 (DP)
96	30.8 (DP)	29.2 (SP) / 129.6 (DP)
128	30.5 (DP)	28.9 (SP) / 129.1 (DP)

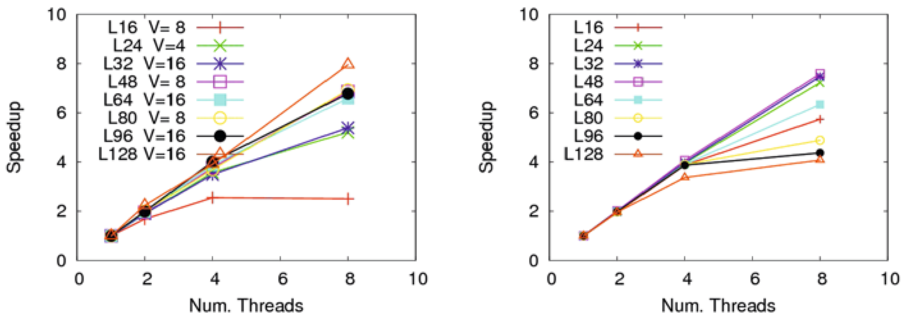


Fig. 1. Relative speed-up for the Ising spin-glass (left) and the Heisenberg (right) simulations

linearly and it has an almost perfect scaling for $L = 128$ and $V = 16$. With this combination of parameters each spin and each coupling-terms allocate one single byte, the lattice size fits the L3 cache and the run time is mainly dominated by the computation time and not by memory access. The Heisenberg code scales almost linearly up to 4 cores. Using 8 cores the scale behavior is linear for small lattice sizes, while for large lattices the overhead due memory access becomes relevant.

The good scaling behavior of our code for some lattice sizes as function of the number of threads/core (we use one thread per core), gives the opportunity to improve further the performances using quad-socket boards based on quad-, six- or eight-core CPUs. We have measured a similar behavior also on the Cell-based systems.

Some final comments are in order:

– Multi-core and GPU architectures have roughly similar performance levels for this class of applications (both binary and floating-point models), even if corresponding peak performances differ by one order of magnitude. We believe that this is due mainly to memory-bandwidth problem and synchronization overheads.

– Widely different multi-core architectures produce performances that hardly differ for more than a factor 4. Performance for the Cell/BE is strongly dependent on lattice size, as the schedule of data transfers to the local store has a strong impact; this effect is less severe for the Nehalem CPU (performance of the latter CPU however drops, in the binary case, as soon as the L3 cache is not large enough for the simulation data-base).
– New architectures narrow substantially the gap between special-purpose and commercial systems. However the former machines – in the cases in which they can be used – still have an edge of approximately two order of magnitudes; however, this edge is expected to shrink further in the not too distant future.

References

1. Marinari, E., Parisi, G., Ruiz Lorenzo, J.J.: Numerical Simulations of spin glass systems. In: Young, A.P. (ed.) Spin Glasses and Random Fields. World Scientific, Singapore (1998)
2. Condon, J.H., Ogielski, A.T.: Rev. Sci. Instruments 56, 1691–1696 (1985)
3. Ogielski, A.T.: Physics Review B 32, 7384–7398 (1985)
4. Cruz, A., et al.: Computer Physics Commmunication 133, 165–176 (2001)
5. Belletti, F., et al.: Simulating Spin Systems on JANUS. Computer Physics Commmunication 178, 208–216 (2008)
6. Belletti, F., et al.: JANUS: an FPGA-based System for High Performance Scientific Computing. Computing in Science and Engineering 11, 48–58 (2009)
7. Belletti, F., et al.: Simulating an Ising spin-glass for 0.1 seconds in Janus. Physical Review Letters 101, 157–201 (2008)
8. Landau, D.P., Binder, K.: A Guide to Monte Carlo Simulations in Statistical Physics. Cambridge University Press (2005)
9. Alvarez Banos, R., et al.: Static versus dynamic heterogeneities in the D=3 Edwards-Anderson-Ising spin glass. Physical Review Letters 105, 177–202 (2010)
10. Belletti, F., Guidetti, M., Maiorano, A., Mantovani, F., Schifano, S.F., Tripiccione, R.: Monte Carlo Simulations of Spin Glass Systems on the Cell Broadband Engine. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 467–476. Springer, Heidelberg (2010)
11. Parisi, G., Rapuano, F.: Effects of the random number generator on computer simulations. Physics Letter B 157, 301–302 (1985)
12. IBM Cell Broadband Engine Architecture, IBM web-site, http://www-128.ibm.com/developerworks/power/cell/documents.html
13. Inside Nehalem: Intel's Future Processor and System, Real World Technologies web-site, http://www.realworldtech.com/page.cfm?ArticleID=RWT040208182719
14. NVIDIA Corporation, NVIDIA CUDA C Programming Guide, NVIDIA web-site, http://developer.nvidia.com/page/home.html
15. NVIDIA's GT200: Inside a Parallel Processor, Real World Technologies web-site, http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242
16. Inside Fermi: Nvidia's HPC Push, Real World Technologies web-site, http://www.realworldtech.com/page.cfm?ArticleID=RWT093009110932

Software Environment for Market Balancing Mechanisms Development, and Its Application to Solving More General Problems in Parallel Way

Mariusz Kamola

[1] Institute of Control and Computation Engineering,
Warsaw University of Technology
[2] NASK (Research and Academic Computer Network)
`Mariusz.Kamola@nask.pl`

Abstract. A new software that supports research on algorithms for resource allocation in multi-commodity markets is presented. Thanks to a much more general data model used, and possibility of plugging in external optimization engines, various solvers can be used to extend the functionality of the platform. The software functionality is focused on supporting researchers in the algorithm engineering process, facilitating e.g. analysis and comparison of market strategies. An example application of the software to travelling salesman problem, solved by many agents, is presented.

1 Introduction

There exist many drivers for the growth of contemporary computation tasks complexity. The most obvious of them is the increase of data collection, storage and transmission capabilities of todays IT infrastructure. Less evident but equally grave is the fact that isolated systems get more and more connected, and start to exhibit completely different behaviour. Such phenomena are easy to appear, quite easy to expect, hard to foresee, difficult to simulate and mostly impossible to be described analytically. Networking and its intricate couplings has become a fact.

Due to the nonpolynomial complexity of many network problems it is usually very time-consuming, hence impractical, to look for their exact solutions. Let us consider, for example, travelling salesman problem (TSP) whose exact brute-force solving has complexity of $O(n!)$, n being the number of vertices. A much more common approach is to find a good heuristic algorithm, e.g. running in polynomial time [10]. The process of finding a good heuristics is a problem of its own, often consisting of tedious verifications of suites of similar algorithms, varying in details, over a number of test problems, their reference solution known. Managing data from such experiments in order to retrieve useful information can present difficulties, especially if the heuristic algorithm is to be run in parallel.

This paper presents capabilities of a novel software platform for agent-based distributed simulation and its usability for prototyping control strategies. Originally, such control was to be executed over market entities sell/buy offers, in order to achieve optimal allocation of transport network resources. However, it can be equally well applied for coordinating threads of any parallel algorithm, being particularly suitable for graph problems. In such generalized approach market entities become workers processing parallelized tasks, and exchanging data (formerly, 'offers') in a way managed by the coordinator (formerly, 'resource allocator'). The platform imposes a unified problem description and communication scheme, leaving modules implementation details to user's choice — not precluding further parallelism in their implementation.

The next section presents a formal complete model of all objects necessary to describe the process of offering and market clearing, M^3, presented in [2] and exploited by its authors in a number of applications. Then comes the description of the abovementioned software platform facilitating the research on algorithms. Next, the analysis of applicability of the platform in a few more general research scenarios is given, taking as an example the TSP problem solved in parallel by partitioning the set of all possible salesman paths. The paper concludes with studies of more advanced applications of the platform and the underlying problem description format, in research activities.

2 Multi-commodity Market Model

Multi-commodity market model, M^3, is a method and format for a formal description of a market where trade of resources takes place. It has been initially developed to describe offer structure in the energy market in Poland [3]. For its generality it has been next used to model IP network bandwidth trade [4]. With the research platform atop it may be successfully used to model, solve and investigate properties of virtually any graph problem.

M^3 defines the following basic entities and relations between them:

- network nodes and arcs, describing the topology of the network where capacity trade takes place,
- market entities (users, providers) that offer or sell resources (capacity),
- resources being offered, with their proper attributes,
- offers, i.e. bindings of market entities and resources, offered or demanded at a specific price.

It is also possible to define compound resources, i.e. containing simple resources and other compound resources. Analogously, one can define simple and compound offers and market entities. An UML graph representing offers is presented in Fig. 1, to show how flexible the M^3 model is. However, one can use it without being bothered by advanced features, like aggregation facilities. It is possible to declare only key values, *offeredPrice*, *min/maxValue* and *shareFactor* (1 for sell, -1 for buy offers), leaving other unset. The field *acceptedVolume* and *sell/buyPrice* parameters in *Commodity* structure contain results of the market balancing process.

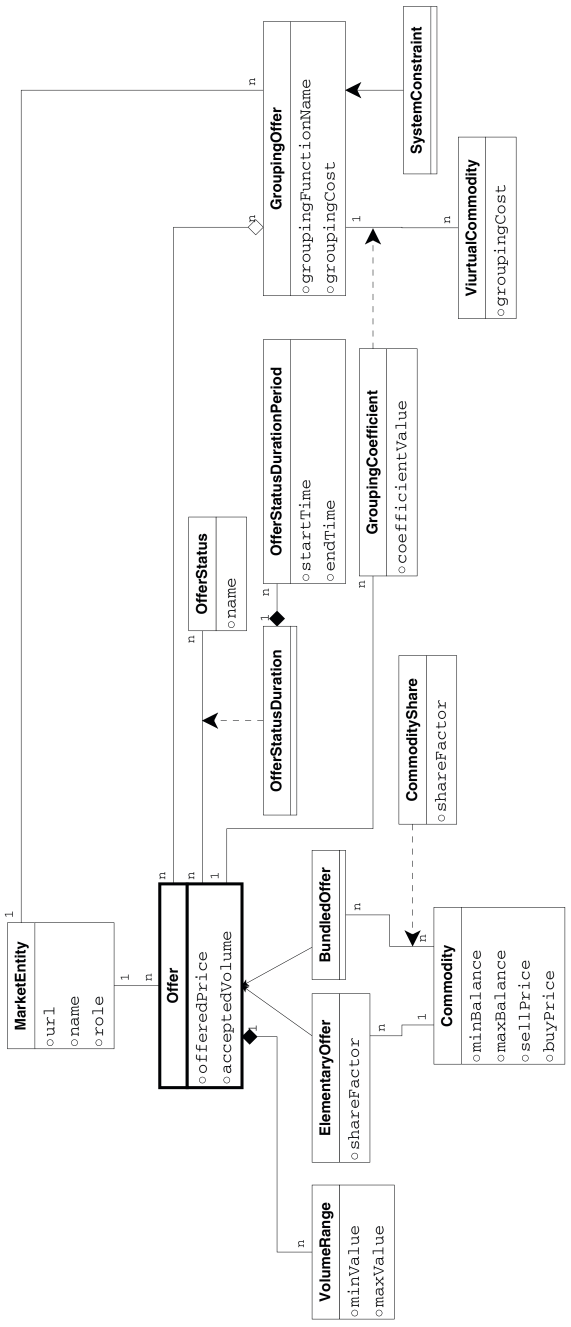


Fig. 1. Relations between offers and other key elements of M^3 model: market entities and commodities. The model allows defining complex offers recursively.

At first, one can perceive the M^3 model merely as a tool for network problem definition, containing placeholders for attributing nodes or arcs with parameters. M^3 defines only the minimum set of those attributes the ones needed in M^3 original applications. Those are defined in XSL dictionary files and, in fact, define the M^3 generic model completely. Concretisation of the generic model for a specific network (e.g. power grid or IP, or road network) is done by specifying extra node/arc attributes in *network kinds* XML file. Analogously, any specific data type describing market entities or offers can be made in proper XML files.

Having concretised the generic type, one can place the data regarding network topology, market entities and offers in appropriate three files. The content of these files defines one market case. Also, it contains placeholders where the solution (resources allocation and resource prices) can be put into. XML has been chosen as a format for M^3 because it makes possible easy transformation of network problems into formats supported by standard optimisation solvers e.g. GAMS, AMPL. This is done by means of XSLT [8].

3 The Platform for Research on Bidding and Resource Allocation Strategies

Good structural design and contemporary technology made M^3 quite popular in the local academic environment. A number of authors used it, mostly in pursuit of developing engines for clearing the market [5], [6]. However, due to variety of approaches to the problem (especially, various assumptions), their results were hardly comparable. It was only recently when they realised that creating a common research environment would be a good idea in several aspects. The major assumption was to store each numerical result in a common database in order to be capable of performing searches, queries and comparisons. This led to work on the formulation of a number of quantitative criteria for ranking the resource allocation algorithms in case of bandwidth trading scenario: economic (total transaction surplus, competition level for network resources) and technical ones (length of allocate network paths, ratio of contracted bandwidth to offered bandwidth) [7].

The platform implementation makes use of many technologies currently being considered industrial standards. The reason for using such standards was to make the platform itself a proof of concept for a to-be commercial trading system. There are three main use cases of the platform:

– Experiment definition. A mechanism designer prepares a set of M^3 files describing the simulation task. Also, the designer implements, if needed, own appropriate resource allocation mechanisms and user agents (or endpoint application for human users acting as agents, cf. Fig. 2).
– Simulation. A chosen experiment is being run: solver and agents' processed are spawned and they communicate alternately with the platform. The results are being stored into the database.
– Result analysis. A selected set of experiment results is selected, by advanced query, from the database. Their results are being compared and displayed in the GUI.

The software was created using technologies allowing application scaling, portability and flexibility. The platform runs as a web service, accessible and configurable via a browser. Such approach minimizes application maintenance costs. To accelerate application operation, some part of GUI processing is shifted to the browser thanks to Google Web Toolkit technology [9]. Links between application modules are managed by Spring framework, allowing for modules replacements without rebuilding of the whole application. The persistence layer is accomplished using recognised products: MySQL, now from Oracle Inc., and Hibernate. The communication with agents and the resource allocation module is managed reliably by Java Message Service. The web application container was Apache Tomcat. Apparently, all the presented technologies are freeware, well supported, popular and documented.

Performance of application three crucial components: XML/POJO serialization, JMS communication and RDBMS communication, has been checked for two test problems. The smaller problem data size was 22 kB, and the bigger problem data was 424 kB. The simulation was run on 4-core PC with Linux OS, using loopback interface instead of a real network. Execution times for different kinds of operations have been logged and analysed, with the following results:

– Serialization times were proportional to the problem sizes (measured in kilobytes).
– JMS communication times were also proportional both to problem sizes and the number of agents.
– RDBMS communication was worse than proportional to the problem sizes, n, but better than $O(n^2)$.

Therefore, the weak points of the architecture were the database and the communication queues. However, implementations of both the technologies come in large variety, and it would be relatively easy to replace e.g. inefficient JMS implementation with another one.

Since the users of the platform will not only run simulations but actually actively develop new agents, a friendly API has been designed, with the aim to keep Java entities structure and communication rules as simple as possible. Any user module, be it agent or solver, must implement methods *onInit* and *onAllocation*, defined by the *Endpoint* interface, requiring and returning *InitParams* and *RunParams*, respectively. *InitParams* contains module configuration data, while *RunParams* carries offers or allocations, depending on communication context.

To relieve platform user from laborious interaction with JMS, a user module skeleton class, *EndpointStub* was created. The class implements main JMS message processing loop, in particular activating *onInit* and *onAllocation* methods in a user-provided object. Also, an exemplary, do-nothing extension of *EndpointStub* was provided to users. The extension, *GenericEndpoint* echoes received parameters of *onAllocation* method; it is excellent for initial configuration testing, and as a base cllass for any user-specific implementation. It also provides a bunch of utility methods, e.g. for XML/POJO conversion.

From the point of view of the agents placing bids for resources, the platform remains rather a transparent thing (Fig. 2). Its only role is to merge individual

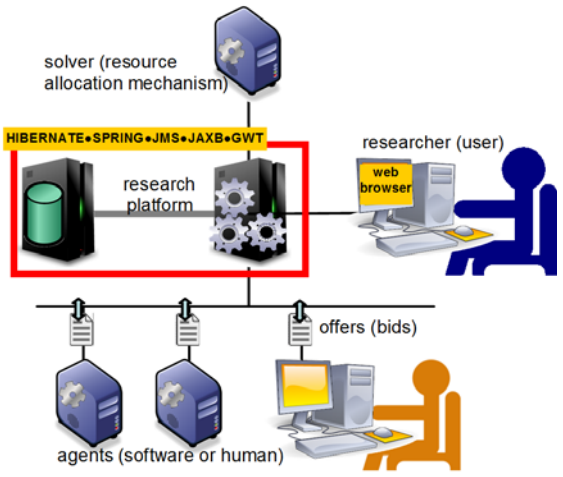

solver (resource allocation mechanism)
HIBERNATE•SPRING•JMS•JAXB•GWT
research platform
researcher (user)
web browser
offers (bids)
agents (software or human)

Fig. 2. Interaction of the platform with bidding agents and resource allocation engine (in simulation mode; message flow is up and down), and in result analysis mode (data flow is from left to right)

requests into a single M^3 document set, which is then presented to the resource allocation module. The content of an exemplary M^3 *offers* file is presented below. Attributes *offeredPrice* contain individual agents' price expectations.

```
<?xml version="1.0" encoding="windows-1250" ?>
<m3:offers xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation=http://www.openM3.org/m3 M3Offers.xsd
xmlns:m3=http://www.openM3.org/m3 xmlns:ia="http://www.ia.pw.edu.pl/m3">
<m3:Offer id="ia:o12345-67" offeredPrice="32000.00">
  <m3:description>Exemplary elementary offer</m3:description>
  <m3:offeredBy ref="ia:siekierki" />
  <m3:offerStatus status="m3:offer-open">
  <m3:durationPeriod startTime="2007-04-09T08:00:00"
    endTime="2007-04-09T09:59:59" />
```

Resource allocations are then sent back to the bidders (using *Commodity* section of M^3 model) and, optionally, the process is re-iterated until the allocations settle. This could be done easily without the platform altogether, save for the fact that all bids and allocations get stored in the database both in plain XML and in the structured form. Turning the original XSD definitions into an object hierarchy and a RDBMS scheme constituted the biggest part of the architectural and programming work. The information flow for the most important use case, the simulation, is presented in Fig. 3.

Once stored in the database, experimental results can be compared easily in a number of ways. A flexible system of experiment tagging allows searches across many dimensions: the author, allocation engine type, version and running

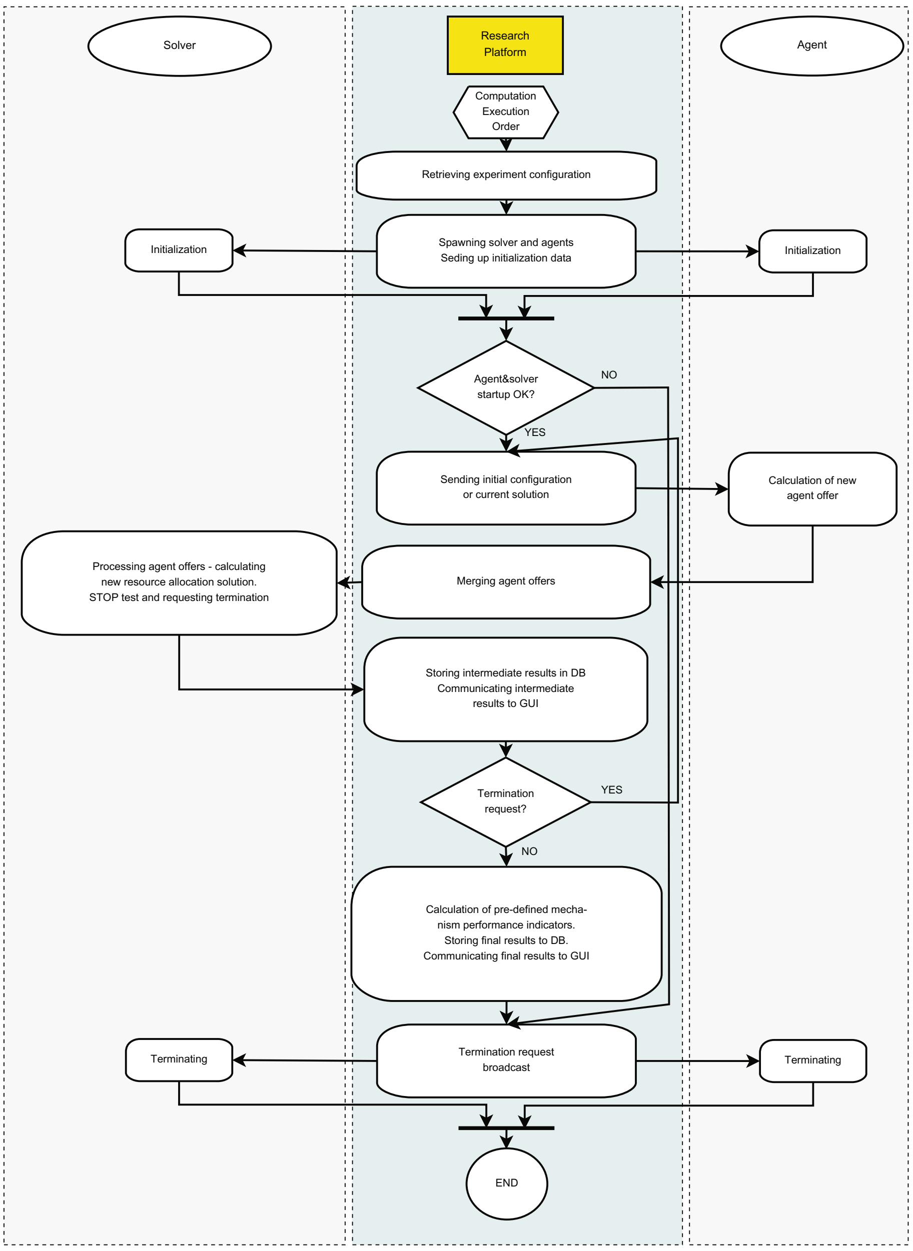
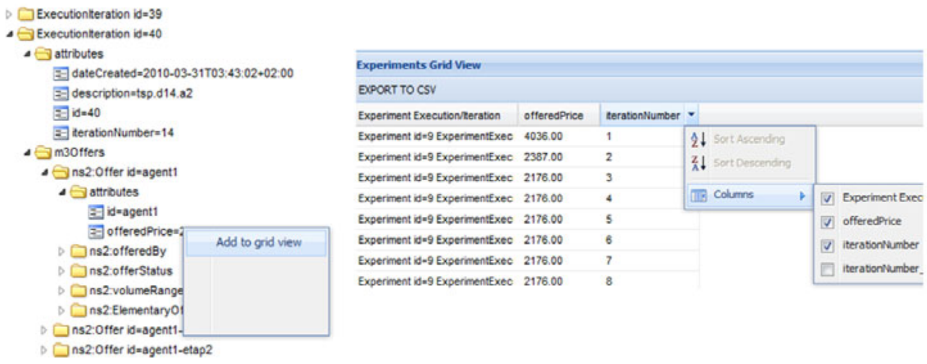

Solver
Research Platform
Agent
Computation Execution Order
Retrieving experiment configuration
Initialization
Spawning solver and agents Seding up initialization data
Initialization
Agent&solver startup OK?
NO
YES
Sending initial configuration or current solution
Calculation of new agent offer
Processing agent offers - calculating new resource allocation solution. STOP test and requesting termination
Merging agent offers
Storing intermediate results in DB Communicating intermediate results to GUI
Termination request?
YES
NO
Calculation of pre-defined mecha- nism performance indicators. Storing final results to DB. Communicating final results to GUI
Terminating
Termination request broadcast
Terminating
END

Fig. 3. Data flow diagram for the simulation use case



```
ExecutionIteration id=39
ExecutionIteration id=40
  attributes
    dateCreated=2010-03-31T03:43:02+02:00
    description=tsp.d14.a2
    id=40
    iterationNumber=14
  m3Offers
    ns2:Offer id=agent1
      attributes
        id=agent1
        offeredPrice=2
      ns2:offeredBy
      ns2:offerStatus
      ns2:volumeRange
      ns2:ElementaryOf
    ns2:Offer id=agent1-
    ns2:Offer id=agent1-etap2
```

Experiments Grid View

EXPORT TO CSV

Experiment Execution/Iteration	offeredPrice	iterationNumber
Experiment id=9 ExperimentExec	4036.00	1
Experiment id=9 ExperimentExec	2387.00	2
Experiment id=9 ExperimentExec	2176.00	3
Experiment id=9 ExperimentExec	2176.00	4
Experiment id=9 ExperimentExec	2176.00	5
Experiment id=9 ExperimentExec	2176.00	6
Experiment id=9 ExperimentExec	2176.00	7
Experiment id=9 ExperimentExec	2176.00	8

Fig. 4. Screenshots of results browse tree (left), with possibility to indicate variables to be exported or displayed as a table (right)

parameters, type of the problem etc. A number of specific data can be then extracted from such pre-filtered data: allocation or prices of a resource in n-th or last iteration, total number of iterations etc. Those can then be arranged freely in a table in order to draw a graph or to be exported for further processing outside the platform (Fig. 4). A number of pre-defined market mechanism performance indicators are calculated in each simulation iteration, and stored in the database. They include the total benefit of the operator of the market as well as a similar coefficient defined for the market participants. New algorithms for scoring simulation results, taking into account specifics of individual problems, can be easily implemented. Such freedom is also well handled by the database design, allowing any number of so obtained measures. Apart from the compiled-in functionality, the platform provides means to formulate and evaluate *XPath* expressions over some queried result set of performed experiments. In such way, platform user may calculate fancy measures for experiment results.

4 Application of the Platform to Parallel Computation Tasks

Inherently, the presented platform is able to support both distributed computation and subsequent analysis of results for any problems describable in the M^3 model. This means it can handle most of graph problems: TSP, coloring, transport etc. Also, it can be applied to massive parallel algorithms that are apparently not related to graphs, as evolutionary strategies. Noteworthy is the possibility of performing social networks analysis, using the platform. The most computationally demanding part of extracting knowledge from the social graph is to find communities, i.e. sets of vertices particularly mutually closely connected, in a sense. The complexity of exact solution finding is *NP*-hard. That is why approximate community-finding algorithms are being constructed. The platform can easily improve the process of good heuristic algorithm research in the field. This application scenario has been presented mostly for current popularity of social network analysis, applicable e.g. in marketing process, fraud detection, business/military/terrorist organisation structure analysis. Social network analysis is another topic of interest of the author of this paper, and the prospect of employing the platform in such research activities will hopefully be fruitful.

The platform has been successfully applied for an exemplary travelling salesman problem. The purpose was to demonstrate that coding of the problem can be simple, and that using the environment does not require any clumsy workarounds. One starts with adapting basic M^3 definitions in order to create data types matching ones problem. In our case no changes to the built-in market entity and network definitions were needed. The M^3 "commodity" term has been used in TSP problem to represent a single stage of a salesmans route. An extra property *stage_number* was created to identify stages. All stages are linked to an artificial network node, *n0*. Recall that offers tie stages and market entities. In the context of TSP, market entities are identical with computation units performing the work in parallel. To code the actual solution found by a unit, standard *minValue* and *offeredPrice* fields of the offer type. The first one indicates the concerned stage, while the second one indicates the node a salesman passed in that stage.

Computation units are started in this example as Java instances: they perform a specified part of the computational task, and report progress to the platform periodically. The resource allocation module implementation is a trivial one, selecting and recording the best solution found so far by the agents.

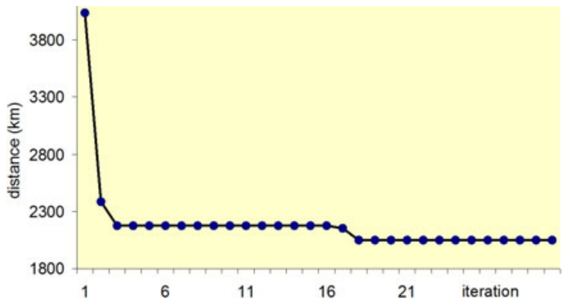


Fig. 5. TSP solution in subsequent algorithm steps

The example problem was defined for 14 major Polish cities. The graph of connections was full mesh, with geodesic distances. Rather than coding those distances as M^3 graph arcs, there were calculated dynamically by each agent using a dedicated Java package [1]. Thus, the M^3 network definition consisted only of cities names, latitudes and longitudes. No arcs were declared in order to accelerate communication. The exact solution of the problem was found correctly

(Fig. 5). There were two path calculation modules performing, one resource allocation module and the platform software, of course. In the case described no relevant information was passed back from the allocation to computation modules; however, passing some extra control data, e.g. adjusting their operations, would be easy to implement.

5 Conclusion

It was shown that the platform originally developed for investigating properties of resource allocation problem to the market can be successfully used to support research also in other areas. The presented example was a TSP problem. Platform's ability to incorporate external solvers and the design based on M^3 information model make it a convenient tool to perform comparative analysis of parallel tasks — a tool for a groupwork research.

The presented platform API was designed in abstraction of any programming paradigm. In fact, it does not impose any constraint on the user, save for the necessity of implementing the two methods: *onInit()* and *onAllocation()*. In particular, the user has the ability of accessing any object of the M^3 structures directly, by using POJO representation of the XML code. Alternatively, she or he may prefer to do high-level XSL transforms on the plain XML to complete problem definition with extra information (e.g. the objective function), difficult or impractical to be stored in M^3. Then, an external solver can be called. Also, the possibilities of performing any local parallelisation of the pre-assigned piece of the problem, are not limited in any way.

It should be stressed that the added value of the work presented in this paper is not related to equipment or algorithmic performance. Most of the work presented here was spent of mapping advanced modelling scheme, M^3, into a relational database, and on developing ergonomic programming and graphical interfaces, with the aim to create a teamwork scientific environment. Hopefully, this will make cooperation quicker and clearer, and the results — more objective.

Acknowledgments. The work presented in this paper was supported by the Polish Ministry of Science and Higher Education research grants no. PBZ-MNiSW-02/II/2007 and N N514 416934.

References

[1] Gavaghan, M.: GPS Receivers and Geocaching: Vincenty's Formula — a section of the web page, http://www.gavaghan.org/blog/category/codeproject (accessed March 30, 2010)

[2] Kacprzak, P., Kaleta, M., Pałka, P., Smolira, K., Toczyłowski, E., Traczyk, T.: M^3: Open Multi-commodity Market Data Model for Network Systems. In: Proceedings of the XVI International Conference on System Science, Wrocław (2007)

[3] Kacprzak, P., Kaleta, M., Pałka, P., Smolira, K., Toczyłowski, E., Traczyk, T.: Communication model for M^3— Open Multi-commodity Market Data Model. In: Proceedings of TPD 2007 Polish Conference, Poznań (2007)

[4] Kacprzak, P., Kaleta, M., Pałka, P., Smolira, K., Toczyłowski, E., Traczyk, T.: Application of open multi-commodity market data model on the communication bandwidth market. J. Telecommunications and Information Technology 4, 45–50 (2007)
[5] Karpowicz, M., Malinowski, K.: Network flow optimization with rational agents. NASK internal report (2009)
[6] Pałka, P., Kołtyś, K., Toczyłowski, E., Żółtowska, I.: Model for Balancing Aggregated Communication Bandwidth Resources. J. Telecommunications and Information Technology 3, 43–49 (2009)
[7] Stańczuk, W., Pałka, P., Lubacz, J., Toczyłowski, E.: Parametric pricing rule in bandwidth trade. In: Proceedings of 8th International Conference on Decision Support for Telecommunications and Information Society DIST 2009, Coimbra (2009)
[8] XSLT transformation "toAmpl-BCBTxsl" Documentation contained in archive available in Tools/XSLT files section of the web page, `http://www.openm3.org` (accessed March 30, 2010)
[9] Google Web Toolkit Overview, `http://code.google.com/webtoolkit/overview.html` (accessed March 30, 2010)
[10] Ausiello, G., Leonardi, S., Marchetti-Spaccamela, A.: On Salesmen, Repairmen, Spiders, and Other Traveling Agents. In: Bongiovanni, G., Petreschi, R., Gambosi, G. (eds.) CIAC 2000. LNCS, vol. 1767, pp. 1–16. Springer, Heidelberg (2000)

The Development of Fully Coupled Simulation Software by Reusing Segregated Solvers

Mika Malinen

CSC – IT Center for Science Ltd.,
P.O. Box 405, FI-02101 Espoo, Finland
mika.malinen@csc.fi
http://www.csc.fi

Abstract. Computational methods for solving coupled systems of partial differential equations can generally be divided into segregated and monolithic solvers. Monolithic solvers are often considered to be computationally expensive, and it is believed that their rewards are realized only in situations where segregated solvers have convergence problems due to the strong coupling. We give opposite empirical evidence by demonstrating that cost-effective monolithic solvers may be derived from using segregated solvers as preconditioners in the iterative solution of monolithic systems.

Keywords: monolithic solver, segregated solver, preconditioning, Elmer software, incompressible flow.

1 Introduction

A contemporary challenge in many fields of computational modeling is related to the design of efficient solution methods for simultaneous discrete systems which stem from a coupling between different physical phenomena described by partial differential equations. In many cases, sophisticated software to solve the constituent single-physics models are already available. Therefore, the simulation tools that enable the coupling of different models have usually been developed by using segregated solution strategies, where the solution is obtained iteratively by decoupling the equations and solving the resulting subproblems sequentially. A chief strength of such segregated solvers is that reusing the existing software is often simple, since only small modifications are usually required to implement the interaction terms. Also, given that the solution is obtained by solving simpler equations, they offer the potential for highly efficient implementations.

It is well known, however, that in cases where the coupling of the equations is strong the convergence of basic segregated solvers may be problematic. More robust solvers may generally be constructed by employing the monolithic solution strategy based on using fully coupled discretizations. In this approach, all discrete equations are assembled into a single system and the resulting algebraic equations are then solved simultaneously. Obviously, this strategy often requires the development of new software components and does not hence support the software modularity to the same extent as segregated solvers. As a result, many practitioners and software developers opt for using the segregated solution approach. This traditional strategy has also been the leading design paradigm in the development of our Elmer product [1], which is a versatile open source multiphysics simulation software based on finite elements.

The aim of this paper is to demonstrate that if a segregated solver exists, building an efficient solver for the fully coupled equations is not necessarily difficult. This judgment is based on a largely empirical observation that segregated solvers can be used to efficiently accelerate the convergence of a Krylov subspace method applied to the fully coupled discrete system, even when the basic segregated solution procedure fails to converge. Importantly, the Krylov acceleration can be advantageous also in regimes where the stability is not an issue. To provide an illustration of these points, we shall here consider an example fluid flow problem, which is solved with representative methods implemented into Elmer software.

2 Algorithmic Aspects

To address further the question of what is generally required by a monolithic solver in comparison with the sequential solution methods, consider a two-field multiphysics problem, the discretization of which leads to solving a linear system of equations

$$\begin{bmatrix} A & G \\ D & K \end{bmatrix} \begin{bmatrix} V \\ Q \end{bmatrix} = \begin{bmatrix} F \\ H \end{bmatrix}, \tag{1}$$

where the vectors V and Q contain the coefficients of finite element expansions of the unknown fields. Alternatively, a sequential solution method may be used so that assembling all the equations into the same system is avoided. A classical choice is the block Gauss-Seidel iteration

$$\begin{aligned} AV_{k+1} &= F - GQ_k, \\ KQ_{k+1} &= H - DV_{k+1}, \end{aligned} \tag{2}$$

with $k \geq 0$.

When the coupling of the equations is strong, the basic block Gauss-Seidel iteration may have only limited applicability due to convergence problems. Therefore, relaxation strategies are often employed. Then, if the relaxation is applied after solving (2), we come to updating the solution as

$$\begin{aligned} V_{k+1} &= V_k + \lambda_{k+1}(V'_{k+1} - V_k), \\ Q_{k+1} &= Q_k + \lambda_{k+1}(Q'_{k+1} - Q_k), \end{aligned} \tag{3}$$

where λ_{k+1} is an iteration parameter and (V'_{k+1}, Q'_{k+1}) is the basic Gauss-Seidel iterate.

We note that there is a close relationship between certain preconditioned versions of Krylov subspace methods for solving the fully coupled problem (1) and the iteration (3). For example, the preconditioned version of the GCR method based on the block triangular preconditioner

$$P = \begin{bmatrix} A & 0 \\ D & K \end{bmatrix} \tag{4}$$

is readily seen to produce updates of the form (3). The resulting method generates a sequence of minimal residual updates via determining the scalar λ_{k+1} at each iteration step such that the 2-norm of the residual

$$\left\| \begin{bmatrix} F \\ H \end{bmatrix} - \begin{bmatrix} A & G \\ D & K \end{bmatrix} \begin{bmatrix} V_{k+1} \\ Q_{k+1} \end{bmatrix} \right\| \tag{5}$$

is minimal over the set of corrections spanned by all search directions.

We conclude that if solvers for the subproblems associated with the update (2) exist, building a preconditioned solver for the fully coupled problem (1) is not particularly difficult. Basically, only the ability to perform the matrix-vector product associated with the linear algebra problem (1) is required. It is important to note that employing the block Gauss-Seidel preconditioning is not the focal point here. In practice any other sequential solution method could also be used to produce the search directions $V'_{k+1} - V_k$ and $Q'_{k+1} - Q_k$ in (3). Therefore, these ideas generally offer a way to reuse segregated solvers in connection with the fully coupled solution strategies.

3 An Example Problem: Unsteady Flow over a Cylinder

In addition to multiphysics problems, the segregated solution approaches have been used widely in the simulation of incompressible fluid flow. In order to compare the relative merits of traditional sequential solution approaches and the monolithic solution strategy based on the preconditioning, we consider here a standard benchmark problem of unsteady flow over a cylinder. The underlying partial differential equation model now consists of the incompressible Navier–Stokes equations expressed in the terms of the fluid velocity and pressure.

The semi-implicit Euler time integration of the spatially discretized version of the incompressible Navier–Stokes equations leads to solving linear systems that at each time step n take the form (1). As a prototype of segregated solvers, we consider the consistent splitting scheme [2], where the principal idea is to replace the incompressibility constraint by a consistent pressure Poisson equation. A fully discrete version of this scheme leads to solving systems of the form

$$\begin{bmatrix} A & 0 & 0 \\ D & M & 0 \\ D & C & S \end{bmatrix} \begin{bmatrix} V^{n+1} \\ \psi^{n+1} \\ Q^{n+1} \end{bmatrix} = \begin{bmatrix} F^{n+1} - GQ^n \\ 0 \\ SQ^n - DV^n \end{bmatrix}, \tag{6}$$

where ψ^{n+1} is an auxiliary variable, M is the discrete version of the identity operator on pressure space, while C and S are approximations of scaled Laplacian operators associated with the pressure Poisson equation. This algorithm can be viewed as a three-step procedure for advancing from time level n to $n+1$. Thus, in contrast with the example scheme (2), the iteration is not performed. This is motivated by the fact that an iteration is not necessary in order to maintain the time accuracy.

A preconditioned iteration for the fully coupled problem can be derived easily from (6). Then, instead of performing one update of the form (6), we compute similar updates repeatedly at the same time level in order to produce search directions for solving the coupled system by the minimal residual iteration. This iteration may seem as an unnecessary computational burden in comparison with the original consistent splitting scheme, where the solution is updated sequentially in a three-step fashion. However, we shall see that there is a considerable gain in using the additional iteration.

Some additional details regarding the example simulations are as follows. Both solvers employ the MINI finite element discretization on a two-dimensional body $\Omega = (0, 2.2 \text{ m}) \times (0, h) \setminus \Omega_C$, where $h = 0.41$ m and the boundary of the body Ω_C is a circle having the diameter $d = 0.1$ m and the center point at (x_1, x_2)=(0.2m,0.2m). On the inflow boundary $x_1 = 0$ a flow profile with components

$$\begin{aligned} v_1 &= 6\sin(\pi t/8)(1 - x_2/h)x_2/h \quad (\text{m/s}), \\ v_2 &= 0 \end{aligned} \tag{7}$$

is prescribed, while on the outflow boundary $x_1 = 2.2$ m the surface traction is assumed to vanish. In addition, a zero velocity condition is imposed at the top and bottom of the channel. Finally, the fluid density and viscosity are given by $\rho = 1$ kg/m^3 and $\mu = 0.001$ kg/(ms). The quantities we consider here are the drag and lift coefficients c_D and c_L over the cylinder for $t \in (0, 8]$. The time step size is taken to be $\Delta t = 1/720$, and the preconditioned GCR iteration to solve the fully coupled linear system is stopped when the ratio of the 2-norm of the linear system residual to the norm of the forcing vector is smaller than 10^{-5}.

It is noted that the average count of GCR iterations to solve the fully coupled linear system at each time step was 1.48, so the preconditioning strategy is indeed very effective. If we omit the costs of assembling the fully coupled matrix and performing the orthogonalization process to find the minimal residual update, one iteration of the fully coupled solver can be considered to be as expensive as one step of the basic consistent splitting algorithm. Given that the fully coupled iteration require about 3/2 iterations in average and thus the cost of the orthogonalization process is basically one matrix-vector product per iteration, we conclude that the overall computational cost of the preconditioned solution is comparable to that of the standard consistent splitting scheme. One might expect that the solution quality should also be the same.

The drag and lift coefficients obtained by using the basic consistent splitting algorithm and the preconditioned solver are now displayed in Figures 1 and 2. Interestingly, the computed lift coefficients are seen to differ significantly. It appears that the solution computed via the preconditioned algorithm has superior quality. This is evident from Figure 3 where the lift coefficient given

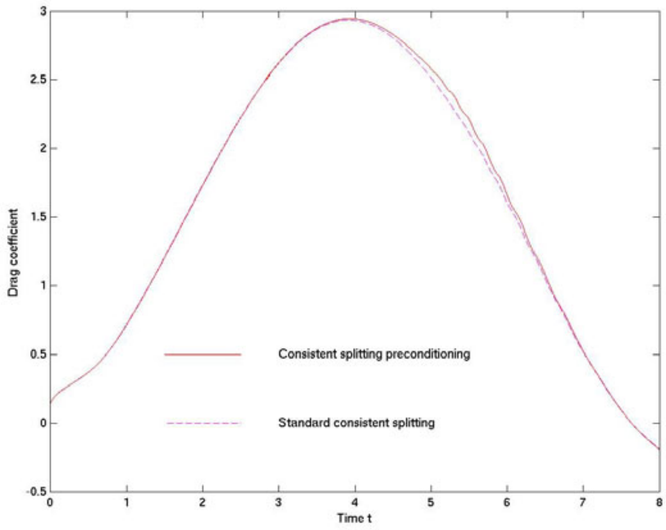


Fig. 1. The drag coefficients given by the consistent splitting scheme (*dashed curve*) and the preconditioned fully coupled solver (*continuous curve*) when the backward Euler time stepping with the time step size $\Delta t = 1/720$ is employed

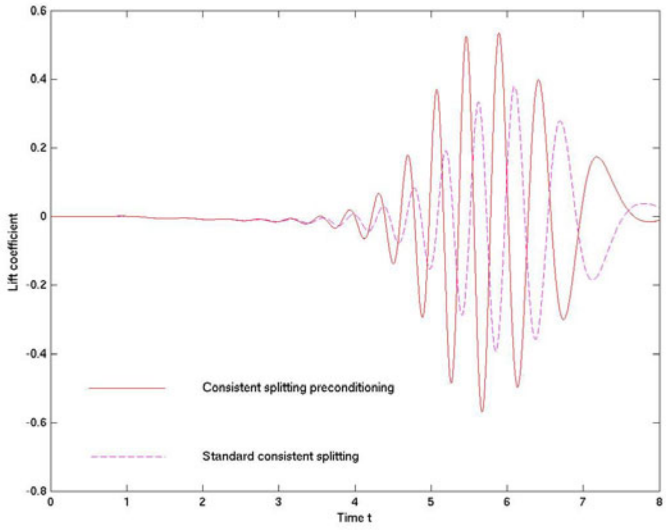


Fig. 2. The lift coefficients given by the consistent splitting scheme (*dashed curve*) and the preconditioned fully coupled solver (*continuous curve*) when the backward Euler time stepping with the time step size $\Delta t = 1/720$ is employed

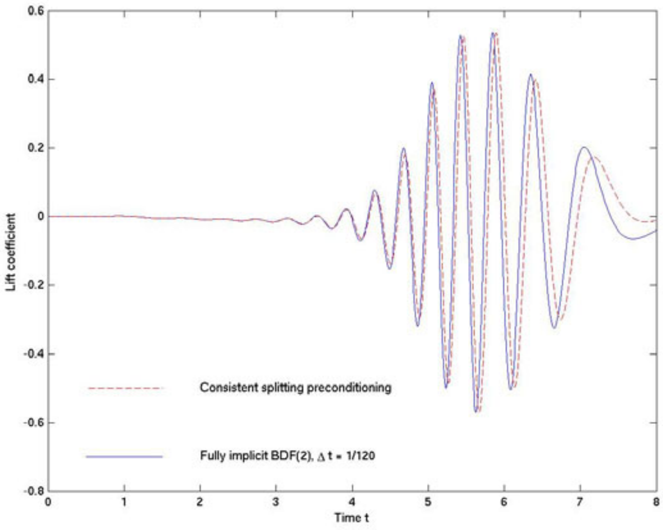


Fig. 3. The lift coefficients given by the preconditioned fully coupled solver based on the backward Euler time stepping with the time step size $\Delta t = 1/720$ (*dashed curve*) and the fully implicit monolithic solver based on BDF(2) with $\Delta t = 1/120$ (*continuous curve*)

by the preconditioned solver is compared with the reference solution, which is obtained by the fully implicit BDF(2) time integration with the time step size $\Delta t = 1/120$. We see that these results show good agreement and produce a fairly accurate approximation to the maximum value of the lift coefficient reported in the literature [3].

4 Concluding Remarks

It is widely believed that monolithic solvers are computationally expensive and that their rewards are realized only in situations where segregated solvers have convergence problems due to the strong coupling. The computational examples given here do not support this conclusion; for similar evidences in connection with fluid-structure interaction problems, see also [4]. Our general experience is that cost-effective monolithic solvers can often be obtained by utilizing segregated solvers as preconditioners in the iterative solution of monolithic systems. We have demonstrated here that if the segregated solver exists, a further step to build an efficient preconditioned solver for the fully coupled problem is not necessarily difficult. Rewards of this effort can be significant and may not be limited to obtaining improved robustness. Importantly, the possibility for reusing effective solvers for simpler subproblems also remains a natural option.

References

1. Elmer finite element software homepage, http://www.csc.fi/elmer
2. Guermond, J.L., Shen, J.: A New Class of Truly Consistent Splitting Schemes for Incompressible Flows. J. Comput. Phys. 192, 262–276 (2003)
3. Schäfer, M., Turek, S.: Benchmark Computations of Laminar Flow Around a Cylinder. In: Hirschel, E.H. (ed.) Notes on Numerical Fluid Mechanics, vol. 52, pp. 547–566. Vieweg (1996)
4. Heil, M., Hazel, A.L., Boyle, J.: Solvers for Large-Displacement Fluid-Structure Interaction Problems: Segregated vs. Monolithic Approaches. Computational Mechanics 43, 91–101 (2008)

Implementation and Evaluation of Quadruple Precision BLAS Functions on GPUs

Daichi Mukunoki and Daisuke Takahashi

Graduate School of Systems and Information Engineering, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan
mukunoki@hpcs.cs.tsukuba.ac.jp, daisuke@cs.tsukuba.ac.jp

Abstract. We implemented the quadruple precision Basic Linear Algebra Subprograms (BLAS) functions, AXPY, GEMV and GEMM, on graphics processing units (GPUs), and evaluated their performance. We used DD-type quadruple precision operations, which combine two double precision values to represent a quadruple precision value. On an NVIDIA Tesla C1060, our BLAS functions are up to approximately 30 times faster than the existing quadruple precision BLAS on an Intel Core i7 920. Additionally, the execution time of quadruple precision AXPY takes only approximately 2.7 times longer than that of double precision AXPY on the Tesla C1060. We have shown that quadruple precision BLAS operations are suitable for GPUs.

Keywords: quadruple precision BLAS, double-double precision, GPU.

1 Introduction

Floating-point operations have round-off errors. These errors may become a critical issue for some applications. For example, the convergence of iterative methods is heavily influenced by round-off errors, and there is a study to try to improve the convergence using quadruple precision operations [6]. In addition, the accumulation of round-off errors in large-scale computing has become a problem. On the other hand, double precision accuracy may not be sufficient for some scientific applications, especially the high-precision arithmetic demanded by scientific computations.

However, the hardware of most modern processors only supports up to double precision. Therefore, high-precision operations must be carried out via software emulation. Since this emulation is extremely computationally complex, it severely limits the performance of most traditional computers.

Recently, the performance of graphics processing units (GPUs) has increased more rapidly than that of CPUs. Furthermore, GPU programming has become easier with the development of GPU programming tools, such as NVIDIA's Compute Unified Device Architecture (CUDA). Thus, general purpose computing on GPUs (GPGPU) has been a major topic of research in recent years.

GPGPUs have two main characteristics. The first is a "many-core" and "multi-threaded" architecture, which makes them suited to highly parallel computations and vector operations. The second is that a GPU can act as an accelerator for the CPU via a peripheral bus, such as the PCI-Express (PCIe) bus. The NVIDIA Tesla C1060 has a peak double precision performance of 78 GFlops. However, a GPU connected by PCIe 2.0 x16 has a theoretical maximum bandwidth of only 8 GB/s. Thus only computationally-intensive applications are capable of effectively utilizing GPUs.

We have implemented three quadruple precision Basic Linear Algebra Subprograms (BLAS) functions, AXPY, GEMV and GEMM on GPUs. Quadruple precision operations using software emulation are highly computationally complex but require few memory references. Hence, quadruple precision BLAS operations are suitable for GPUs and are capable of attaining high performance.

2 Related Work

QD [1] is a quadruple and octuple precision floating-point arithmetic library. QD uses a double-double (DD) type algorithm for quadruple precision and a quad-double (QD) type algorithm for octuple precision operations. These algorithms represent quadruple and octuple precision values by combining two and four double precision values respectively. Our implementation uses the DD-type algorithm described in the next section.

XBLAS [8] is a well-known extended precision BLAS for CPUs. XBLAS uses the DD-type algorithm internally. However, it does not support full quadruple precision because the input and the output are double precision. MBLAS [10] is a multi-precision BLAS for CPUs. MBLAS uses two existing multi-precision libraries: the GNU Multiple-Precision Library (GMP) for arbitrary precision operations, and the QD for octuple and quadruple precision operations. The input and the output data are also multi-precision. MBLAS is the only multi-precision BLAS that fully supports quadruple precision operations.

There is some research on high-precision floating-point operations on GPUs. Graça [4] implemented double-float type double precision operations for GPUs that are not supported by hardware. Thall [12] also implemented double-float and quad-float type operations on GPUs. In addition, Lu et al. [9] implemented an implementation of the QD library on GPUs, called GQD.

On the other hand, CUBLAS [2] is a single and double precision BLAS on GPUs implemented using CUDA. However, there has been no research on implementing quadruple precision BLAS on GPUs. In this paper, we will present, "CUDDBLAS", our implementation of a DD-type quadruple precision BLAS for GPUs using CUDA. Like MBLAS, CUDDBLAS fully supports quadruple precision operations. The input and the output data are DD-type quadruple precision values.

3 Quadruple Precision Operations

There are two ways to store high-precision values using software emulation of high-precision floating-point operations: the first stores them in a format defined

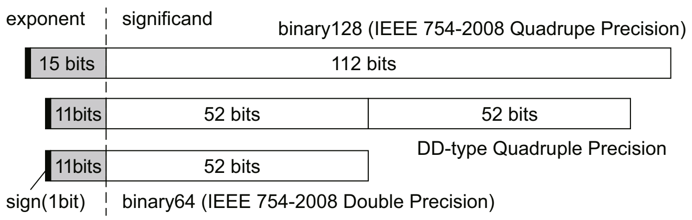

exponent
significand
binary128 (IEEE 754-2008 Quadrupe Precision)
15 bits
112 bits
11bits
52 bits
52 bits
DD-type Quadruple Precision
11bits
52 bits
sign(1bit)
binary64 (IEEE 754-2008 Double Precision)

Fig. 1. DD-type quadruple precision value

using integer values, and the second stores them using hardware-implemented floating-point formats. The first approach is used by GMP and is suitable for arbitrary precision. DD-type operations use the second approach. This approach is simple because it can use the exponent and the significand of the existing floating-point format, thus it has a speed advantage, but the exponent is not extendable. Therefore, this approach cannot represent a wide range of values, and it is not feasible for more than octuple precision. Thus we decided to use the DD-type algorithms for quadruple precision operations. We used the same algorithms as the QD library. For more details, refer to Hida et al. [7].

3.1 Quadruple Precision Format

In IEEE 754-2008, the quadruple precision format is defined as "binary128" with a significand part of 112 bits. In fact, the total precision is 113 bits because the significand has an implicit integer bit value of 1, and the format is approximately 34 decimal digits. The DD-type algorithm represents a quadruple precision value by combining two double precision values. In IEEE 754-2008, the double precision format is defined as "binary64" with a significand part of 52 bits; similarly, the total precision is 53 bits including the implicit bit. Therefore, the total precision of DD-type quadruple precision must be twice that, 106 bits. This is still 7 bits less than binary128's 113 bits and is approximately 32 decimal digits. Also, its exponent part is still only 11 bits (see Figure 1).

3.2 DD-type Algorithms

The DD-type algorithms are based on the error-free floating-point arithmetic algorithms. The given IEEE 754 double precision values are used with round-to-even for binary operations. The normal operations are represented as $\{+, -, \times, \div\}$, and the rounded operations are represented as $\{ \oplus, \ominus, \otimes, \oslash \}$.

The TwoSum algorithm (see Figure 2) produces an expansion $a+b$ such that $s+e = a+b$, where s is an approximation of $a+b$ and e represents the round-off error in the calculation of s. The QuickTwoSum algorithm (see Figure 3) also produces an expansion $a + b$ such that $s + e = a + b$, where $|a| \geq |b|$. These algorithms are also described in Shewchuk's paper [11].

```
TwoSum($a, b, s, e$){
    $s = a \oplus b$
    $v = s \ominus a$
    $e = (a \ominus (s \ominus v)) \oplus (b \ominus v)$
}
```

Fig. 2. TwoSum algorithm

```
QuickTwoSum($a, b, s, e$){
    $s = a \oplus b$
    $e = b \ominus (s \ominus a)$
}
```

Fig. 3. QuickTwoSum algorithm

```
SPLIT($a, h, l$){
    $t = (2^{27} + 1) \otimes a$
    $h = t \ominus (t \ominus a)$
    $l = a \ominus h$
}
```

Fig. 4. SPLIT algorithm

```
TwoProd($a, b, p, e$){
    $p = a \otimes b$
    SPLIT($a, a_H, a_L$)
    SPLIT($b, b_H, b_L$)
    $e = ((a_H \otimes b_H \ominus p) \oplus a_H \otimes b_L$
            $\oplus a_L \otimes b_H) \oplus a_L \otimes b_L$
}
```

Fig. 5. TwoProd algorithm

```
TwoProdFMA($a, b, p, e$){
    $p = a \otimes b$
    $e = fl(a \times b - p)$
}
```

Fig. 6. TwoProdFMA algorithm

In the SPLIT algorithm (see Figure 4), the double precision value a is split into two double precision values such that $a = h + l$, where h is the upper part and l is the lower part of each 26 bits.

The TwoProd algorithm (see Figure 5) produces $p + e = a \times b$, where p is an approximation of $a \times b$, and e represents the round-off error in the calculation of p.

However, the TwoProdFMA algorithm (see Figure 6) can be used only on machines that implement the double precision Fused-Multiply Add (FMA) instruction to calculate $a \times b + c$, with an intermediate result of 106 bits with no round-off error. This instruction is shown as $fl(a \times b - p)$ in Figure 6. The Tesla C1060 supports this FMA instruction (using the built-in function: `__fma_rn`). We can use it and reduce the operation count from 17 Flop to 3 Flop.

The DD-type quadruple precision operations consist of the aforementioned algorithms. The quadruple precision value a is represented as $a = a_H + a_L$ using two double precision values, a_H and a_L, where $|a_H| > |a_L|$.

The QuadAdd algorithm (see Figure 7) calculates the quadruple precision addition, $c = a + b$. The total number of operations are 20 Flop. The QuadMul algorithm (see Figure 8) calculates quadruple precision multiplication, $c = a \times b$. Its total operation count is 10 Flop when using TwoProdFMA.

```
QuadAdd($a_H, a_L, b_H, b_L, c_H, c_L$){
    TwoSum($a_H, b_H, sh, eh$)
    TwoSum($a_L, b_L, sl, el$)
    $eh = eh \oplus sl$
    QuickTwoSum($sh, eh, sh, eh$)
    $eh = eh \oplus el$
    QuickTwoSum($sh, eh, c_H, c_L$)
}
```

Fig. 7. QuadAdd algorithm

```
QuadMul($a_H, a_L, b_H, b_L, c_H, c_L$){
    TwoProd($a_H, b_H, p1, p2$)
    (TwoProdFMA($a_H, b_H, p1, p2$))
    $p2 = p2 \oplus (a_H \otimes b_L \oplus a_L \otimes b_H)$
    QuickTwoSum($p1, p2, c_H, c_L$)
}
```

Fig. 8. QuadMul algorithm

4 Implementation

We used CUDA to implement CUDDBLAS. We decided to target our implementation to the NVIDIA Tesla C1060, which is based on the GT200 architecture. The Tesla C1060 has 30 streaming multiprocessors (SM), each of which consists of 8 single-precision floating-point units (FPUs), and one double-precision FPU.

We implemented AXPY, GEMV, and GEMM. The BLAS operations are performed after the data transfer from the CPU to the GPU is finished. In other words, it does not overlap computations with data communications. The implementation techniques of these BLAS operations use the same general approach as double precision BLAS. Each element of a vector or a matrix is computed in parallel with each thread. These threads are arranged as a one- or two-dimensional structure. In addition, GEMV and GEMM use a blocking algorithm to use shared memory for data reuse. The shared memory is a fast on-chip memory of 16 KB per SM that can be shared among threads in the SM. The user can use this as scratch-pad memory. For GEMM, each of the 16×16 elements of a matrix of A and B is loaded. The number of the blocking size is decided by the group processing the threads and memory access, and the capacity of the shared memory. Each SM executes 32 threads concurrently, and 16 threads can access the memory concurrently.

Our implementation of GEMV uses parallel reduction operations [5] for inner product computations to avoid a bank conflict. Thus there is a small amount of error in the lower part of the DD-type value when compared to MBLAS, which does not use parallel reduction operations. However, if we do not use the reduction operation, the kernel-only executing performance drops by one-third.

We implemented DD-type quadruple precision operation functions, as well as the QuadAdd and QuadMul with TwoProdFMA algorithms using the FMA instruction. Quadruple precision operations are used in each thread. We note that there is no function call overhead because these DD-type operations were compiled using the CUDA compiler's inline expansion function. In addition, the compiler automatically replaces the multiply-add operation with the FMA instruction. In cases where the FMA instruction would alter the result of the DD-type algorithms, we used the built-in functions `__dmul_rn` and `__dadd_rn` to replace FMA with separate multiplication and addition instructions.

5 Performance Evaluation

5.1 Methodology

We compared the performance of our CUDDBLAS with some of the currently available BLAS implementations, shown in Table 1. GotoBLAS [3] is the optimized BLAS implementation for CPUs. CUBLAS is the BLAS implementation using CUDA by NVIDIA. Both are single and double precision BLAS. MBLAS is the only BLAS implementation that fully supports quadruple precision operations. It uses QD 2.3.7's quadruple precision operations. XBLAS is an extended precision BLAS using DD-type operations internally, but it does not support full quadruple precision since the input and the output are double precision.

Table 1. BLAS implementations used for performance evaluation

	Version	Hardware	Precision
GotoBLAS	2-1.00	CPU	Double
CUBLAS	2.3	GPU (CUDA)	
XBLAS	1.0.248	CPU	Extended
MBLAS	0.6.4	CPU	Quadruple
CUDDBLAS	–	GPU (CUDA)	Quadruple

We measured the performance of AXPY, GEMV, and GEMM. The performance of the double precision BLAS (GotoBLAS and CUBLAS) are measured in Flops. Meanwhile the DD-type BLAS (MBLAS, XBLAS and CUDDBLAS) are measured in DDFlops, which means DD-type operations per second. In the case of BLAS running on GPUs (CUBLAS and CUDDBLAS), these measurements do not include the time spent transferring data between the CPU and GPU via PCIe. To measure the performance accurately, we repeatedly executed each BLAS function for approximately one second, then computed the average execution time for one iteration of the function. All matrices are square matrices of size $N \times N$, and all vectors have length N and are composed of uniform random numbers. The evaluation environment is shown in Table 2.

Table 2. Evaluation environment

CPU	Intel Core i7 920 (2.67 GHz, Quad-Core, Hyper-Threading enabled)
RAM	12 GB (DDR3)
GPU	NVIDIA Tesla C1060
Video RAM	4 GB (GDDR3)
GPU Bus	PCI-Express 2.0 x16
OS	CentOS 5.3 (x86-64) kernel 2.6.18
CUDA	CUDA SDK 2.30, CUDA Driver 2.30
Compiler	gcc 4.1.2 (-O3) for CPU code, nvcc 2.3 (-O3) for GPU code

We note that GotoBLAS is performed in parallel on 4 threads with 4 physical cores. XBLAS does not support parallel processing. MBLAS supports parallel processing only for some level-1 functions, including AXPY, but not for GEMV and GEMM. MBLAS' AXPY is performed in parallel on 8 threads with 4 physical cores. MBLAS' GEMV and GEMM as well as all XBLAS functions were performed on a single thread.

5.2 Theoretical Peak Performance

The theoretical peak double precision performance of the NVIDIA Tesla C1060 is 78 GFlops (1.3 GHz $\times$ 30 double precision FPUs $\times$ 2 Flop with FMA = 78 GFlops). The theoretical peak double precision performance of the Intel Core i7 920 is 42.72 GFlops on 4 cores.

On the other hand, the theoretical peak performance of DD-type quadruple precision operation on the GPU is as follows: DD-type quadruple precision addition and multiplication operations consist of 20 and 10 Flop double precision operations, respectively. Among these, 2 Flop in the quadruple precision multiplication are performed using an FMA instruction. Therefore, the quadruple precision multiplication and addition are performed using 20 and 9 instructions, respectively. AXPY, GEMV and GEMM functions consist of multiply-add operations, and the ratio of multiplications and additions is $1:1$. Thus, on quadruple precision multiply-add operation, only one FMA instruction can performs 2 Flop, the rest of the 28/29 instructions cannot save a Flop by using FMA.

Hence, the theoretical peak performance of DD-type quadruple precision operation on the NVIDIA Tesla C1060 is as follows: First, 1.3 GHz$\times$30 FPUs $\times((1/29)\times 2$ Flop$+(28/29)\times 1$ Flop$)\approx 40.3$ GFlops of double precision operations. Then, 2 Flop of quadruple precision multiply-add operation equals 30 Flop of double precision operations. Therefore, 40.3 GFlops$/(30/2)\approx 2.69$ GDDFlops.

5.3 AXPY

Figure 9 shows the results of AXPY. When $N = 102400$, the performance of DDAXPY on CUDDBLAS is approximately 2.03 GDDFlops. This equals approximately 30.4 GFlops double precision operations, and approximately 39% of the theoretical peak performance of the GPU. However this performance is approximately 75% of the theoretical peak performance of DD-type quadruple precision operations on the GPU. Even though the computational cost of DD-type operations is 15 times that of the double precision operation, the execution time on CUDDBLAS is only approximately 2.7 times that of CUBLAS. This is because the performance of AXPY is bound by the memory bandwidth on double precision, since AXPY is a memory-intensive operation, $O(N)$ computations are performed on $O(N)$ load/store operations. On the other hand according to our analysis, when $N = 102400$, AXPY becomes computationally-bound when using quadruple precision on the GPU.

The actual measured memory bandwidth of the GPU and the CPU for a transfer of 1MB is approximately 67.5 GB/s and 12.5 GB/s, respectively. Thus,

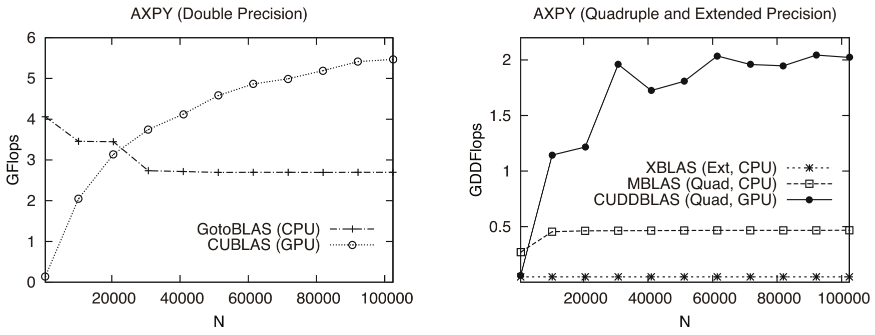


Fig. 9. Performance evaluation results of AXPY

the GPU's memory bandwidth is approximately 5.4 times as large as the CPU's. However when $N = 102400$ CUBLAS is only approximately 1.7 times faster than GotoBLAS, since GotoBLAS's data fits into L3 cache. For quadruple precision operations, CUDDBLAS is approximately 4.3 times faster than MBLAS. This is because the GPU's theoretical peak performance is much higher than the CPU's. Additionally, the GPU can use the FMA instruction to reduce quadruple precision multiplication from 24 Flop to 10 Flop, as described in Section 3.2.

5.4 GEMV

Figure 10 shows the GEMV results. When $N = 7680$, our GEMV on CUDDBLAS performs up to approximately 1.86 GDDFlops. This performance corresponds to approximately 36% of the theoretical peak performance of the GPU. However this performance is approximately 69% of the theoretical peak performance of DD-type quadruple precision operations on the GPU. Our GEMV is approximately 8.2% slower than our AXPY, even though GEMV performs $O(N^2)$ computations and $O(N^2)$ load/store operations, which is more computationally-intensive than AXPY. This is because AXPY becomes computationally-intensive when using quadruple precision.

For double precision, CUBLAS is approximately 5.5 times faster than GotoBLAS. However for quadruple precision, CUDDBLAS is approximately 21 times faster than MBLAS. The reasons for the growth of the performance gap between double and quadruple precision operations are as follows: GotoBLAS is performed using 4 threads, but MBLAS and XBLAS are performed on a single thread. Furthermore, the GPU can use the FMA instruction as we described in Section 5.3. Finally, we feel that MBLAS and XBLAS are not well optimized for speed when compared to GotoBLAS (e.g. GotoBLAS uses Intel's SIMD instruction).

5.5 GEMM

Figure 11 shows the results for GEMM. When $N = 4096$, our GEMM on CUDDBLAS attains approximately 2.63 GDDFlops and reaches approximately 98%

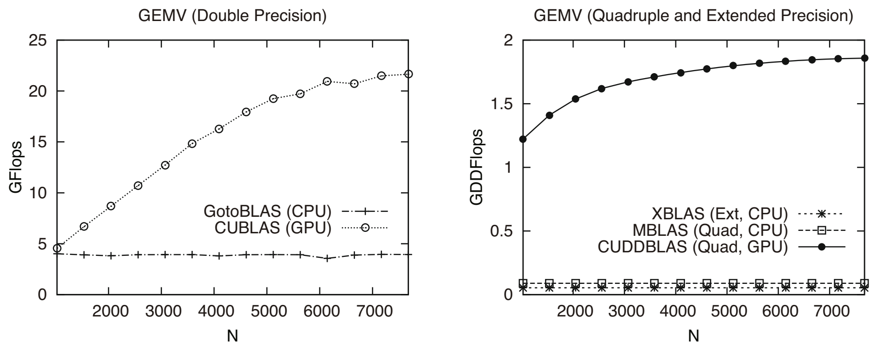


Fig. 10. Performance evaluation results of GEMV

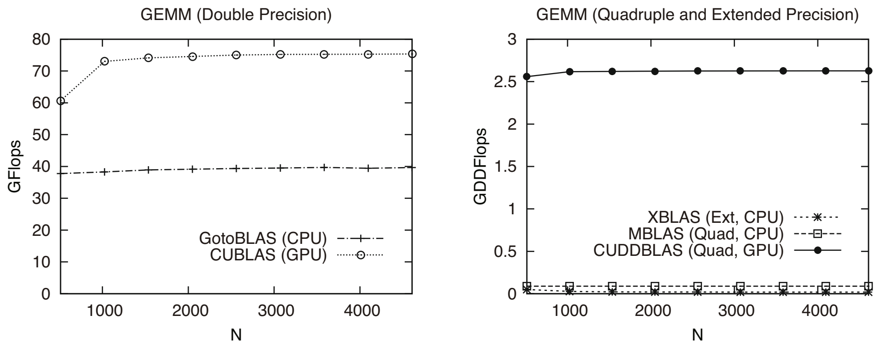


Fig. 11. Performance evaluation results of GEMM

of the theoretical peak performance of DD-type operations on the GPU. In general, GEMM's performance is close to a processor's theoretical peak (as seen in GotoBLAS and CUBLAS in our results). This is because GEMM is computationally-intensive, it performs $O(N^3)$ computations with only $O(N^2)$ load/store operations. Our GEMM performance of 2.63 GDDFlops corresponds to approximately 39.4 GFlops double precision operations, and approximately 51% of the theoretical peak performance of the GPU. This is because the theoretical peak performance of the GPU is calculated when using the FMA instruction, and most instructions of the DD-type algorithms cannot use the FMA instruction.

CUBLAS is approximately 1.9 times faster than GotoBLAS. However, CUDDBLAS is approximately 30 times faster than MBLAS, and approximately 134 times faster than XBLAS. The performance gap on quadruple precision operations is due to the same reasons we described in Section 5.4.

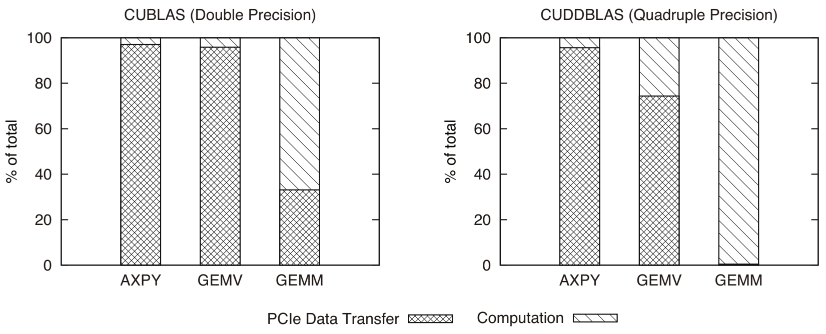


Fig. 12. The percentage of PCIe data transfer time of the total operation time (AXPY: $N = 102400$, GEMV: $N = 7680$, GEMM: $N = 4096$)

5.6 Effects of the PCIe Data Transfer

When a BLAS function on GPUs operates with data stored in the CPU's memory, it needs to transfer data from the CPU to the GPU via PCIe. However, for example for a 1MB transfer, the actual measured bandwidth of PCIe is approximately 5.3 GB/s (from the CPU to the GPU), whereas the bandwidth of the GPU memory is approximately 67.5 GB/s. Thus, the PCIe bandwidth is potentially a huge bottleneck for the GPU.

Figure 12 shows the PCIe data transfer time as a percentage of the total operation time. The total operation time includes the time spent transferring input and output data between the CPU and the GPU, and the memory allocation on the GPU. We note that CUBLAS and CUDDBLAS do not overlap computations with data communications. To measure the performance accurately, we repeatedly executed each BLAS function for approximately one second, then computed the average execution time for one iteration of the function (e.g. 800 and 450 repetitions for AXPY functions on CUBLAS and CUDDBLAS, respectively).

The PCIe data transfer time of AXPY on CUBLAS and CUDDBLAS accounted for approximately 97% and 96% of the total operation time, respectively. In GEMV, the data transfer time on CUBLAS and CUDDBLAS accounted for approximately 96% and 74%, and for GEMM it's 33% and 0.4% respectively. For quadruple precision operations, the computational time greatly increases, and therefore the percentage of time spent on transferring data via the PCIe bus decreases. The performance of quadruple precision GEMM in particular is not heavily affected by the PCIe bandwidth.

6 Conclusions

In this research, we implemented three DD-type quadruple precision BLAS functions, AXPY, GEMV, and GEMM, on GPUs using CUDA. We showed that GPUs can perform quadruple precision BLAS operations faster than CPUs. Our BLAS functions on the NVIDIA Tesla C1060 are up to approximately 30 times faster than the existing quadruple precision BLAS, MBLAS, on the Intel Core i7 920. On the GPUs, the performance of memory-intensive operations such as level-1 subroutines are not much slower than that of double precision. Computationally-intensive operation such as the level-3 subroutines are suitable for GPUs. However, DD-type quadruple precision operations perform only at approximately one-half of the peak performance of the GPUs since most of the DD-type algorithm instructions cannot be implemented via FMA instructions. Additionally, for quadruple precision operations, the computational time greatly increases, and therefore the percentage of time spent on transferring data via the PCIe bus decreases. We showed that quadruple precision BLAS operations are a computationally-intensive application suitable for GPUs.

We will focus on implementing all the BLAS functions and QD-type octuple precision operations. Additionally, we will evaluate the performance in actual applications and on the next generation Tesla GPU based on the Fermi architecture, which has a double precision peak performance of 515 GFlops. We expect that level-3 subroutines will be much faster than those on the Tesla C1060.

References

1. Bailey, D.H.: QD (C++/Fortran-90 double–double and quad-double package), http://crd.lbl.gov/~dhbailey/mpdist/
2. Corporation, N.: CUBLAS Library (including CUDA Toolkit), http://developer.nvidia.com/object/cuda_download.html
3. Goto, K.: GotoBLAS2, http://www.tacc.utexas.edu/tacc-projects/gotoblas2/
4. Graça, G.D., Defour, D.: Implementation of float-float operators on graphics hardware. In: Proc. 7th Conference on Real Numbers and Computers, RNC7 (2006)
5. Harris, M.: Optimizing Parallel Reduction in CUDA, http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf
6. Hasegawa, H.: Utilizing the quadruple-precision floating-point arithmetic operation for the Krylov Subspace Methods. In: Proc. SIAM Conference on Applied Linear Algebra, LA 2003 (2003)
7. Hida, Y., Li, X.S., Bailey, D.H.: Algorithms for Quad-Double Precision Floating Point Arithmetic. In: Proc. 15th Symposium on Computer Arithmetic (2001)
8. Li, X.S., Demmel, J.W., Bailey, D.H., Hida, Y., Iskandar, J., Kapur, A., Martin, M.C., Thompson, B., Tung, T., Yoo, D.J.: XBLAS – Extra Precise Basic Linear Algebra Subroutines, http://www.netlib.org/xblas/
9. Lu, M., He, B., Luo, Q.: Supporting Extended Precision on Graphics Processors. In: Proc. Sixth International Workshop on Data Management on New Hardware, DaMoN 2010 (2010)
10. Nakata, M.: The MPACK; Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK), http://mplapack.sourceforge.net/
11. Shewchuk, J.R.: Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Discrete and Computational Geometry 18, 305–363 (1997)
12. Thall, A.: Extended-Precision Floating-Point Numbers for GPU Computation. In: ACM SIGGRAPH 2006 Research Posters (2006)

Aggregated Pumping Station Operation Planning Problem (APSOP) for Large Scale Water Transmission System

Jacek Błaszczyk[1], Krzysztof Malinowski[1,2], and Alnoor Allidina[3]

[1] Research and Academic Computer Network (NASK),
ul. Wąwozowa 18, 02-796 Warsaw, Poland
jacek.blaszczyk@nask.pl, krzysztof.malinowski@nask.pl
[2] Institute of Control and Computation Engineering,
Faculty of Electronics and Information Technology,
Warsaw University of Technology,
ul. Nowowiejska 15/19, 00-665 Warsaw, Poland
[3] IBI-MAAK Inc.,
9133 Leslie Street, Suite 201,
Richmond Hill, Ontario, Canada L4B 4N1
alnoor.allidina@ibigroup.com

Abstract. Large scale potable water transmission system considered in this paper is the Toronto Water System (TWS), one of the largest existing potable water supply networks. The main objective of the ongoing Transmission Operations Optimizer (TOO) project consists of developing an advanced tool for providing pumping schedules for 153 TWS pumps, with all quantitative requirements with respect to system operation being met while the energy costs are minimized ("The aim of pump scheduling is to minimize the marginal cost of supplying water while keeping within physical and operational constraints, such as maintaining sufficient water within the system's reservoirs, to meet the required time-varying consumer demands." – [6]). It is assumed that TOO should produce detailed optimal schedules for all pumps. The following modules of TOO are being currently developed: demand forecasting module, energy rates forecasting module, pumping schedule optimizer and, finally, an assessment module consisting mainly of hydraulic, EPANET based, TWS simulator. This paper presents key component of the pumping schedule optimizer, namely, the Aggregated Pumping Station Operation Planning Problem (APSOP) and the approach to its solution.

Keywords: water supply, minimum cost operation planning, large-scale nonlinear programming.

1 Introduction: Hydraulic Model of TWS

For the purpose of optimizing a water supply system operation it is advisable to consider sufficiently accurate model of such system [2,5]. In case of TWS there are over 3500 pipes, 4000 nodes (1000 demand nodes) in the main network to be taken into account. There are 19 pressure zones in the system and purified (treated) drinking water is stored in 28 potable water reservoirs and elevated tanks, many of them having two or more cells. Treated water is provided from clearwells at four treatment plants and a number of ground sources (wells). Pumping is done at 29 pumping stations, where, for planning purposes, 153 pumps are grouped into 47 pump groups as logical pumping stations. Each group of pumps draws water at its suction side from a common node and discharges it into a common collector. Majority of pumps are constant speed units. There are also more than 700 valves of various types, the majority of which are throttle control valves (TCV). There are 20 pressure reducing valves (PRV) and 24 check valves (CV) outside of the pumping stations. The TWS is described roughly at Toronto Water site ({urlhttp://www.toronto.ca/water/) and its hydraulic model is provided in EPANET format [7].

2 Goal of the Optimization

For the purpose of operation optimization the pumping schedules and valve settings must be computed with at least a 24-hour planning perspective; in fact due to specific electrical energy tariffs within Ontario's power system, planning over an longer period may be required.

Aggregated Pumping Station Operation Planning Problem (APSOP), presented in this paper, was designed to provide desired aggregated flows, together with pressure (head) gains, at each logical pumping station (PS), and to provide such valve settings that are not predetermined. All these values should be specified as average values for hourly intervals over the 24-hour planning period. The input to APSOP consists of: initial and required final tank/reservoir volumes, forecasted demand at the various water supply nodes where water is being withdrawn from the network, and forecasted energy rates needed to compute the pumping costs.

The performance index to be minimized involves various cost components related to electrical energy usage over time. The actual energy consumption is charged based upon a special tariff, where a given fraction, (determined by contractual obligations), of the energy actually consumed is charged according to predetermined per unit cost and the remaining energy is charged according to current spot market price (rate); thus the need to forecast future spot rates (http://www.ieso.ca/imoweb) prior to solving APSOP. In addition to energy consumption there are other costs related, in particular, to peak power values, i.e., to peak kW and peak kVA values attained over the considered period. To compute power consumption at the pumping stations it is necessary to use aggregated pumping station efficiency curves. The resulting cost function is complex, non-convex and highly nonlinear.

3 Mathematical Modeling of Hydraulic System

The hydraulic model of the network involves static nonlinear flow equations; the Hazen-Williams model is used for all pipe sections. A number of specific constraints define, for the purpose of optimization, the operation of PRVs and other active system components. In fact, modeling of such components as PRVs has to be, for the purpose of optimization, different from description of those components as used for simulation; optimization solvers require them to be represented by a set of constraints.

The most important elements of APSOP formulation are the following.

For each i-th network node flow continuity law must be fulfilled:

$$\sum_{j \in L : \Lambda^c_{ij} \neq 0} \Lambda^c_{ij} \cdot Q_{jk} = d_{ik}, \tag{1}$$

where Λ^c is connectivity (incidence) matrix for nodes, d_{ik} is i-th node demand at k-th hour (nonzero for demand node, zero for connecting node), and L is set of network links.

A pipe segment, with heads h_1 and h_2 at bordering nodes 1, 2, and flow Q considered positive when directed from node 1 to node 2, is described by the Hazen-Williams (HW) empirical head-loss formula:

$$h_1 - h_2 = A \cdot \mathrm{sgn} Q \cdot |Q|^{\alpha}, \tag{2}$$

where A is the resistance coefficient for that pipe and α is the flow exponent (α=1.852) (see [6]). Because of numerical difficulty with absolute value term in the HW formula (non-differentiability at 0 – no second derivative) we use its smooth approximation on interval $[-\delta, \delta]$ ($\delta = 0.1$) [1]:

$$\begin{aligned} h_1 - h_2 = &\left(\frac{3\delta^{\alpha-5}}{8} + \frac{1}{8}(\alpha-1)\alpha\delta^{\alpha-5} - \frac{3}{8}\alpha\delta^{\alpha-5}\right) Q^5 \\ &+ \left(-\frac{5\delta^{\alpha-3}}{4} - \frac{1}{4}(\alpha-1)\alpha\delta^{\alpha-3} + \frac{5}{4}\alpha\delta^{\alpha-3}\right) Q^3 \\ &+ \left(\frac{15\delta^{\alpha-1}}{8} + \frac{1}{8}(\alpha-1)\alpha\delta^{\alpha-1} - \frac{7}{8}\alpha\delta^{\alpha-1}\right) Q \end{aligned} \tag{3}$$

Outside of the interval we use original HW formula.

The throttle valves (TCV) are modeled in a similar way as a pipe segment with α=2. The pressure reducing valves (PRV) are modeled by set of nonlinear constraints involving multiplication of two or three variables.

For each i-th reservoir (tank) we have a mass-balance state equation:

$$V_{i,k+1} = V_{ik} + \sum_{j \in L : \Lambda^r_{ij} \neq 0} \Lambda^r_{ij} \cdot Q_{jk} \cdot \Delta t, \tag{4}$$

where V_{ik} is i-th reservoir volume at k-th hour, Λ^r is connectivity matrix for reservoirs, Q_{jk} is flow through j-th link at k-th hour, and Δt is the hydraulic time step.

For the k-th hour and for each reservoir (tank), average head $\bar{H}_{ik}$ required for flow modeling is computed as:

$$\bar{H}_{ik} = E_i + \frac{1}{12}\left(-x_{i,k-1} + 8 \cdot x_{ik} + 5 \cdot x_{i,k+1}\right), \tag{5}$$

where E_i is tank elevation, x_{ik} is tank level, and $x_{i,k-1} = f^{-1}(V_{i,k-1})$, $x_{ik} = f^{-1}(V_{ik})$, $x_{i,k+1} = f^{-1}(V_{i,k+1})$, and where $V_{i,k-1}$, V_{ik} and $V_{i,k+1}$ are, respectively, reservoir volumes for previous, current, and next hour, and $f(.)$ is the level-capacity curve, i.e., at each time $V = f(x)$. In equation (5) we used two-interval extended Simpson's rule because it was more numerically stable for the resulting nonlinear optimization problem.

At most of the nodes the bounds on admissible pressure values are specified together with the demand patterns.

The decisions regarding j-th logical pumping station operation during each hour are specified as average head gain ΔH_{jk} (in meters) and average aggregate flow Q_{jk} (in megalitres); they are related to power consumption (in MWh) at this pump station given by

$$P_{jk} = \beta \frac{\Delta H_{jk} \cdot Q_{jk}}{\eta_j(\Delta H_{jk}, Q_{jk})}, \tag{6}$$

where β is the unit conversion coefficient, and $\eta_j(\Delta H_{jk}, Q_{jk})$ is the aggregated pumping station efficiency. Constraints on admissible values of ΔH_{jk} and Q_{jk}, with bounds on Q_{jk} dependent on ΔH_{jk}, are computed from the individual pump curves, while aggregated efficiency is computed from pump efficiency curves assuming best configuration of active pumps for given head gain and aggregate flow.

The performance index is computed as

$$J = \sum_{jk} c_{jk} \cdot P_{jk} + \sum_{l} P_l^{\max}, \tag{7}$$

where the unit energy price:

$$c_{jk} = 0.22 \cdot c_{j,\mathrm{con}} + 0.78 \cdot c_{j,\mathrm{spot},k}; \tag{8}$$

the first unit price component is fixed according to long term contract with electrical power supplier and the second term is related to local energy market spot price $c_{j,\mathrm{spot},k}$ at given location and hour k. This price is unknown for a decision maker before actual real time occurrence and so has to be forecasted prior to performing the optimization. The second sum term of the performance index is related to peak power consumption at each of physical pumping stations; for a given l-th station:

$$P_l^{\max} = c^p \max_k \left(\sum_{j \in \bar{l}} P_{jk} \right), \tag{9}$$

where c^p is the peak energy cost and $\bar{l}$ is the set of logical pump stations within physical pump station l. Th minimax objective (9) can be easily converted into a conventional nonlinear programming form by introducing an auxiliary variables and additional constraints to represent the above objective [11].

The decision variables in resulting optimization problem also include:

– flows and head losses for every pipe and valve
– heads at every junction and demand node
– heads, volumes and water levels for every reservoir and elevated tank

For all variables there are simple bounds constraints.

4 The NLP Problem

The resulting nonlinear optimization problem (NLP) is a truly large scale nonlinear optimization problem. The basic, 24-hour period, version involves above 250000 variables and nearly 600000 equality and inequality constraints. In fact, the authors have found so far only one truly comparable problem, in size and complexity, reported in the literature and concerned with operation of the Berlin Water Works (Berlin Wasserbetriebe), presented in [4,3]. Yet, TWS is about twice as big in size, i.e., in the number of components, than the Berlin system. Also, in APSOP there was a need to take into account, properly modeled, pressure reducing valves, and, as mentioned above, to use specific, complicated, energy tariffs. These components were not present in water distribution system as considered by Burgschweiger et. al. [4,3], while the inclusion of such elements increases considerably both the modeling effort and complexity of the optimization problem.

5 Solution Method: The IPOPT Solver

For the solution of APSOP we use a primal-dual interior-point algorithm with line-search minimization based on the filter method, used in the implementation of the IPOPT solver [10]. An open-source C++ version of IPOPT is available at `http://projects.coin-or.org/Ipopt`. A formal description and analysis of the filter line-search procedure implemented in the IPOPT solver can be found in [9]. In comparison with traditional line-search algorithms, such as a single merit function technique, the filter method is usually less conservative and makes it possible to take larger stepsizes. Moreover, the protection in the form of a restoration phase makes the filter algorithm resistant to unnecessary errors, such as those presented in [8].

The computationally most expensive part of the optimization algorithm implemented in the IPOPT solver (not including computations of the objective function, constraints and their derivatives) is the solution of the linear system of equations, which is most often of high order and has a sparse structure. For its factorization and solution, IPOPT uses external sparse direct linear solvers, such as MA27 (default option), MA57, WSMP, PARDISO and MUMPS.

6 Numerical Results

The IPOPT solver calculated the least-cost operational aggregated flows and head gains at each logical pumping station for 24-hours time horizon (with 1-hour discretization). The hydraulic model of TWS in EPANET format is converted to the NLP problem which contains all the network links and nodes, without use of any simplification or skeletonization algorithms. The unit energy price c_{jk} was time-dependent but the same for all PSs, and did not contain the spot market term. The reservoir and elevated tank volumes were kept between 100% and 10% of the maximum volume during the 24-hour horizon and the final reservoir volume was made equal to that from historical simulation. These constraints were satisfied and the results for the two reservoirs and two elevated tanks are presented, respectively, in figures 1 and 2. The original and optimal aggregated pump flows from the two major sources in the network and the corresponding electricity tariffs are shown in figure 3. The optimal aggregated pumping station flows have a direct correspondence with the electricity tariffs. There are higher flows during lower tariff periods and vice versa. The pumps flows before optimization did not follow the tariffs. The obtained energy savings compared to original controls was approximately 11%. The reduction of pumping energy costs at water distribution systems by optimizing pumping schedules is typically up to 10%.

IPOPT was executed on an Intel Q6600 PC with a clock speed of 2.4 GHz. The total calculation time for 24-hour problem was approximately 4 hours and required almost 2000 IPOPT iterations. The starting point for IPOPT optimizer was generated from EPANET simulation of TWS hydraulic model with historical pumping schedules. IPOPT was configured to use MA57 matrix solver which had been found the most stable and reliable for our NLP problem. The solver convergence was very slow because of high nonlinearity of the NLP problem.

7 Conclusions and Future Work

One possibility to make the computations faster is to use parallel matrix solvers (like PARDISO or WSMP) for matrix computations within the IPOPT solver. Yet, it appears that the structure of APSOP, due to underlying tightly connected hydraulic network, does not create enough flexibility to achieve a significant speedup through the use of parallel computing. The other possibility for improving convergence and solution time for IPOPT solver is to use proper NLP problem scaling (own scaling of variables and constraints). The third, very promising technique, is based on model size reduction algorithms, such as skeletonization and simplification of hydraulic network structure: elimination of short pipes, reduction of parallel pipes and node merging.

Nevertheless, being able to solve APSOP in a reasonable time is, by itself, an important, in fact crucial, step in development of Transmission Operation Optimizer for the Toronto Water System.

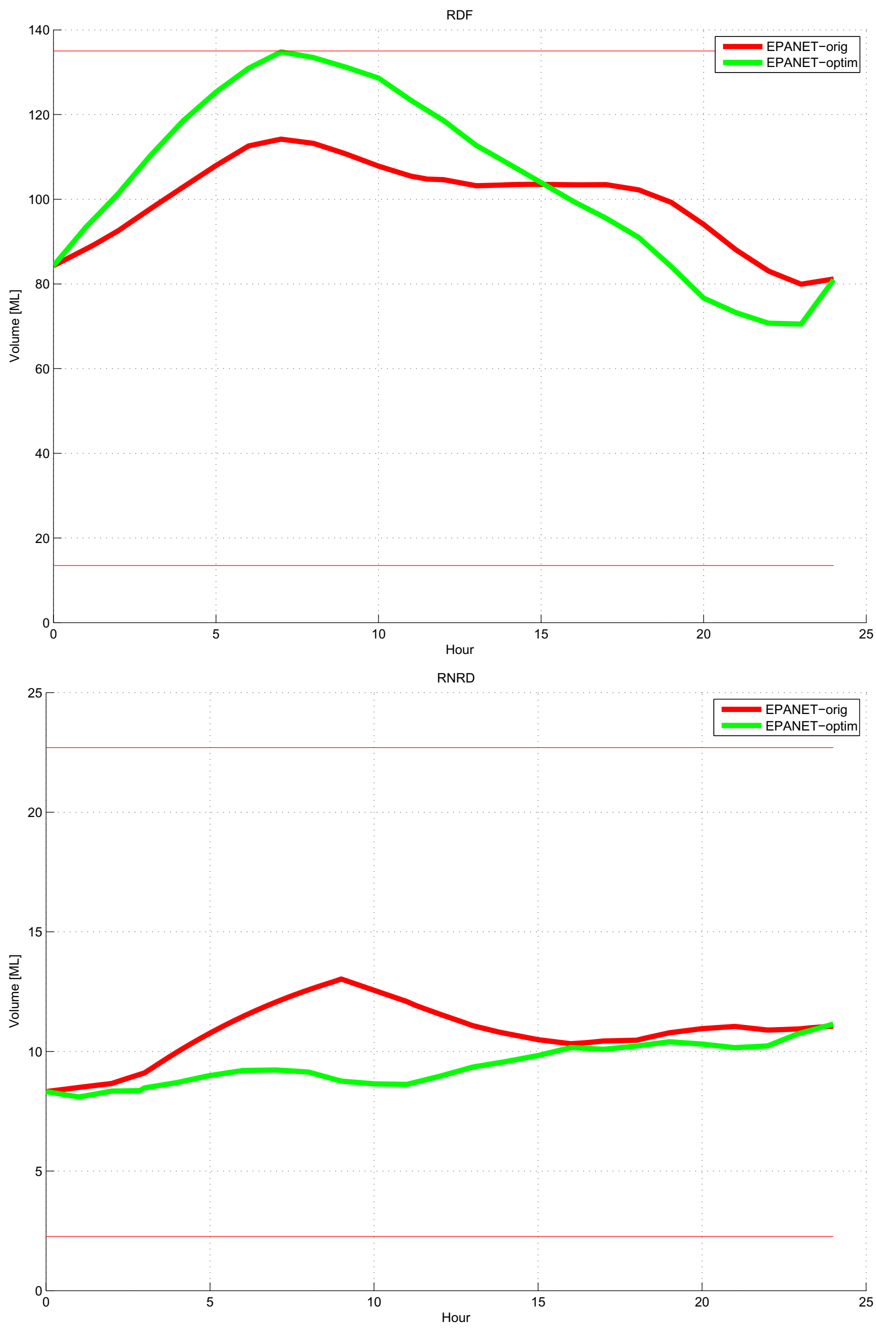


Fig. 1. Reservoir volume profiles for historical and optimized aggregated flows at logical PSs obtained from EPANET simulation

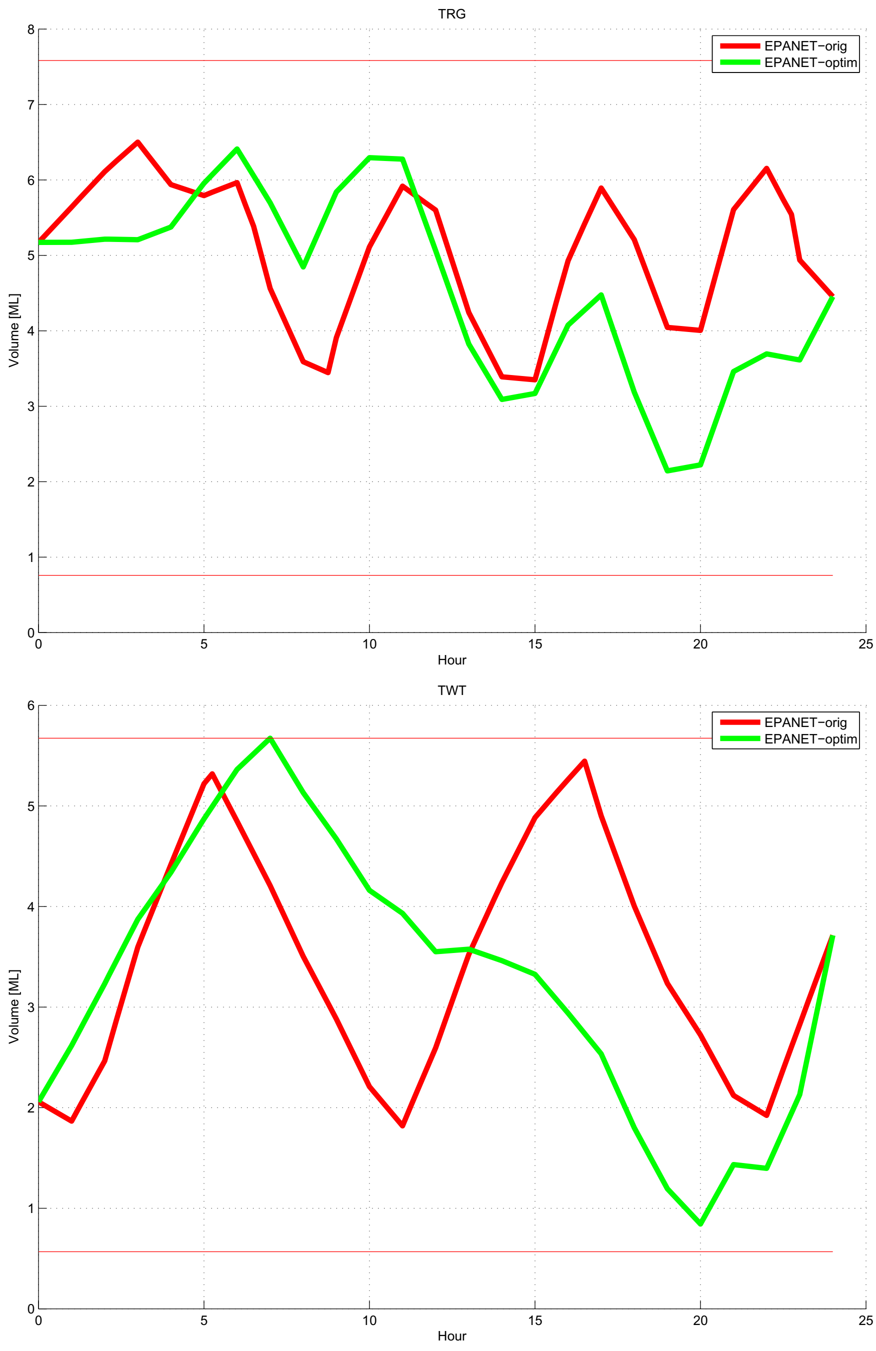


Fig. 2. Elevated tank volume profiles for historical and optimized aggregated flows at logical PSs obtained from EPANET simulation

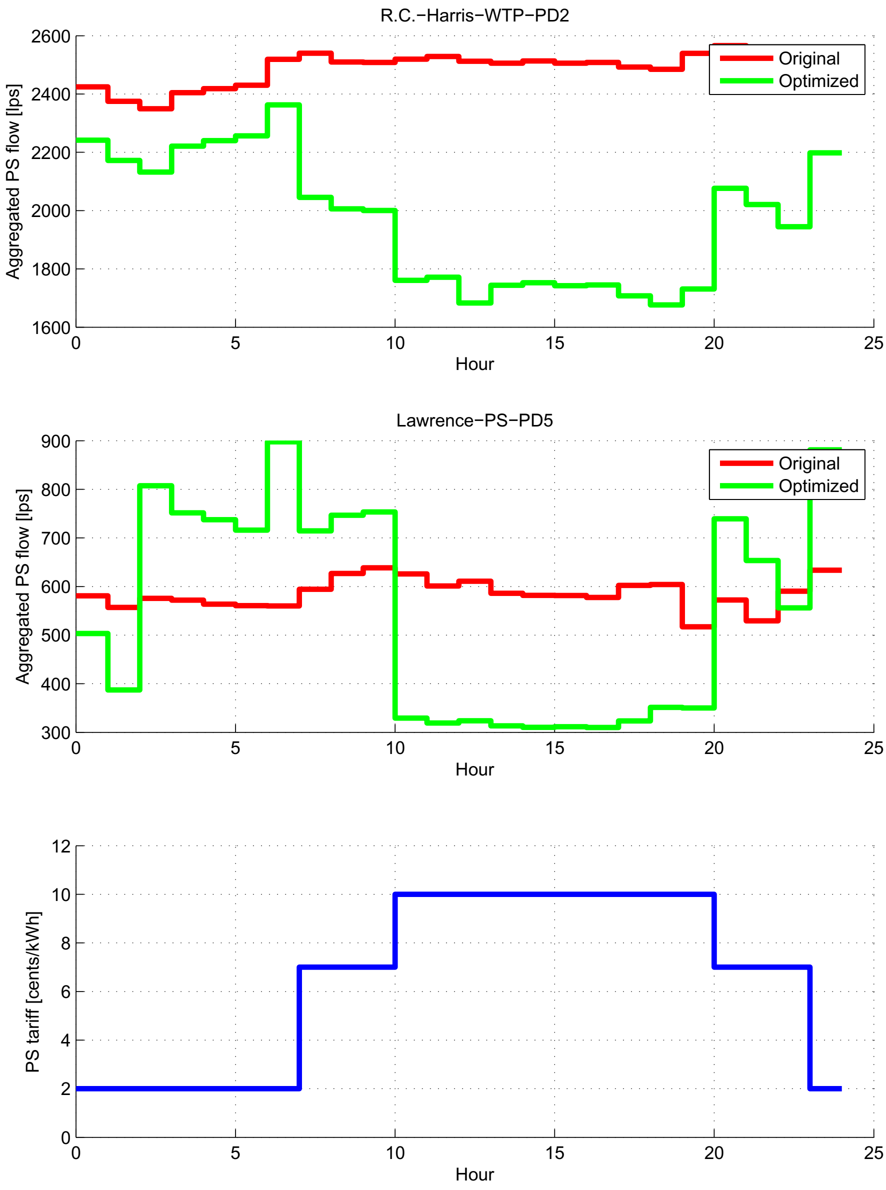


Fig. 3. Aggregated flows for two pumping stations of TWS, plus daily electricity tariff

References

1. Bragalli, C., D'Ambrosio, C., Lee, J., Lodi, A., Toth, P.: Water network design by MINLP. Tech. Rep. IBM Research Report RC 24495, IBM T. J. Watson Research Center, Yorktown Heights, USA (February 2008), http://www.optimization-online.org/DB_FILE/2008/03/1924.pdf
2. Brdys, M.A., Ulanicki, B.: Operational Control of Water Systems: Structures, algorithms and applications. Prentice Hall, New York (1994)
3. Burgschweiger, J., Gnädig, B., Steinbach, M.C.: Nonlinear programming techniques for operative planning in large drinking water networks. The Open Applied Mathematics Journal 3, 14–28 (2009), http://www.zib.de/Publications/Reports/ZR-05-31.pdf
4. Burgschweiger, J., Gnädig, B., Steinbach, M.C.: Optimization models for operative planning in drinking water networks. Optimization and Engineering 10(1), 43–73 (2009), http://www.zib.de/Publications/Reports/ZR-04-48.pdf
5. Mays, L.W.: Optimal Control of Hydrosystems, pierwsze edn. Marcel Dekker, New York (1997)
6. Methods, H.: Advanced Water Distribution Modeling and Management, 1st edn. Haestad Press, Waterbury (2003), http://www.haestad.com/books/pdf/awdm.pdf
7. Rossman, L.A.: EPANET 2 users manual. Tech. Rep. EPA/600/R-00/057, U.S. States Environmental Protection Agency, National Risk Management Research Laboratory, Office of Research and Development, Cincinnati, Ohio, USA (2000), http://www.epa.gov/nrmrl/wswrd/dw/epanet/EN2manual.PDF
8. Wächter, A., Biegler, L.T.: Failure of global convergence for a class of interior point methods for nonlinear programming. Mathematical Programming 88(3), 565–574 (2000)
9. Wächter, A., Biegler, L.T.: Line search filter methods for nonlinear programming: Motivation and global convergence. SIAM Journal on Optimization 16(1), 1–31 (2005)
10. Wächter, A., Biegler, L.T.: On the implementation of a primal-dual interior-point filter line-search algorithm for large-scale nonlinear programming. Mathematical Programming 106(1), 25–57 (2006), http://www.research.ibm.com/people/a/andreasw/papers/ipopt.pdf
11. Williams, H.P.: Model Building in Mathematical Programming, 4th edn. Wiley (1999)

PerPI: A Tool to Measure Instruction Level Parallelism

Bernard Goossens, Philippe Langlois, David Parello, and Eric Petit

DALI Research Team, University of Perpignan Via Domitia,
52 av. P. Alduy, F-66860 Perpignan cedex, France
`first_name.name@univ-perp.fr`
`http://webdali.univ-perp.fr`

Abstract. We introduce and describe PerPI, a software tool analyzing the instruction level parallelism (ILP) of a program. ILP measures the best potential of a program to run in parallel on an ideal machine – a machine with infinite resources. PerPI is a programmer-oriented tool the function of which is to improve the understanding of how the algorithm and the (micro-) architecture will interact. PerPI fills the gap between the manual analysis of an abstract algorithm and implementation-dependent profiling tools. The current version provides reproducible measures of the average number of instructions per cycle executed on an ideal machine, histograms of these instructions and associated data-flow graphs for any x86 binary file. We illustrate how these measures explain the actual performance of core numerical subroutines when measured run times cannot be correlated with the classical flop count analysis.

Keywords: Run time performance, instruction level parallelism, ideal processor, BLAS, polynomial evaluation, mixed precision.

1 Introduction

1.1 Motivation

We introduce PerPI, a programmer-oriented tool focusing the instruction level parallelism of numerical algorithms. This tool is motivated by results like those

Table 1. Flop counts and run times are not proportional

Measure	Eval	AccurateEval1	AccurateEval2
Flop count	2n	$22n + 5$	$28n + 4$
Flop count ratio (/Eval)	1	≈ 11	≈ 14
Measured #cycles ratio (/Eval)	1	2.8 – 3.2	8.7 – 9.7

presented in Table 1 where two algorithms, AccurateEval1 and AccurateEval2, are respectively compared to a third one, Eval [3]. The three algorithms evaluate a polynomial of degree n. Eval is the classical Horner algorithm, AccurateEval1 and AccurateEval2 are two competing evaluations which are both twice more accurate. These two algorithms solve the same problem: how can we double the accuracy of a core numerical subroutine? Such a need appears also, for example, in numerical linear algebra where an accurate iterative refinement relies on a dot product performed with twice the current computing precision [2].

Table 1 presents the flop counts and the ratios of flop counts and run times for the two accurate algorithms over Eval ones. The first two lines are significant for the algorithm's complexity, while the last one presents the range of run times measured on several desktop computers. Such measures are quite familiar when publishing new core numerical algorithms, *e.g.*, floating-point summation, dot product, polynomial evaluation — see entries in [8] for instance. Here AccurateEval1 appears to run about three times faster than AccurateEval2, whereas their flop counts are similar. Such a speedup is important for basic numerical subroutines that are used at any parallelism level. This gap between the classical manual analysis of the abstract algorithms (flop counts) and the measures provided by automatic profiling tools (cycle counts, with [6] for instance) has to be justified.

Of course, merely counting the number of flop within an algorithm does not fully explain the actual performance of its implementation, which depends on other factors such as parallelism and memory access. Moreover, measuring actual run times is hard to reproduce and yields results with a very short life-time since computing environments evolve fast. This process is very sensitive to numerous implementation parameters such as architectural and microarchitectural characteristics, OS versions, compilers and options, programming language, etc. Even if the same data test is used in the same execution environment, measured results suffer from numerous uncertainties: spoiling events (*e.g.*, OS process scheduling, interrupts), non-deterministic execution and accuracy of the timings [11].

> Measuring the computing time of summation algorithms in a high-level language on today's architectures is more of a hazard than scientific research [8].

We believe that this recent quotation is significant for (a call for) a change of practice in the numerical algorithm community. Indeed, uncertainty increases as the computer system complexity does, *e.g.*, multicore or hybrid architectures. Even in the community of program and compiling optimization, it is not always easy to trust this experimental process.

> If we combine all the published speedups (accelerations) on the well-known public benchmarks for four decades, why don't we observe execution times approaching to zero? [10]

A last difficulty comes from the gap between the algorithm design step and the profiling one. The algorithmic step benefits from the abstraction of high level programming languages and, more and more, from the interactivity of integrated developing frameworks such as Matlab. Run time performance analysis is a later step process, and it takes place in a technically more complex and change-prone environment. The programmer suffers from the lack of performance indicators, and associated tools, being independent of the targeted computing architecture that would help at the algorithmic level to choose the most efficient and long-lasting solutions.

2 Analysis: Principles and Pen-and-Paper Example

2.1 Principles

We propose to analyze the instruction level parallelism (ILP) of a program by simulating its run with a Hennessy-Patterson ideal machine [1]. An ideal machine has infinite resources: renaming registers, perfect branch predictor, perfect memory disambiguation. As a result, running a code on an ideal machine is like having at hand the full trace and picking up from this trace instructions as soon as their sources are available. In such a way, the run is ordered according to the only producer/consumer dependencies.

ILP represents the best potential of the instructions of a program that can be executed simultaneously. Every current processor exploits program's ILP thanks to well-known techniques such as pipelining, superscalar execution, out-of-order execution, dynamic branch prediction or address speculation, etc. The ideal machine removes all artificial constraints on ILP (registers, memory, control flow), so it runs the program in such a way that every instruction is scheduled immediately after the execution of the latest producer on which it depends.

The following example illustrates how to quantify this ILP and what kind of information is useful to understand and improve the potential performance of an algorithm.

2.2 A first Pen-and-Paper Analysis

The algorithms presented in Table 1 consist of one loop n times iterated. Figure 1 represents the data-flow graphs of the two accurate algorithms: (a) one iteration, (b) one iteration depending on its predecessor, and (c) the shape of the n iterations (or part of it) [3]. In each of the three parts of the figure, two consecutive horizontal layers represent two consecutive execution cycles within the ideal machine. To be performed manually, the data dependency analysis has been restricted to the floating-point operations, *i.e.*, to the algorithmic level description.

Table 2. Floating-point ILP as in Table 1

Measure	Eval	AccurateEval1	AccurateEval2
FP ILP	1	≈ 11	≈ 1.65

From these graphs, we count the number of floating-point operations and the number of cycles to run them, *i.e.*, the total number of nodes and the depth of the (c) graph. The ratio of these values measures the (floating-point) ILP,

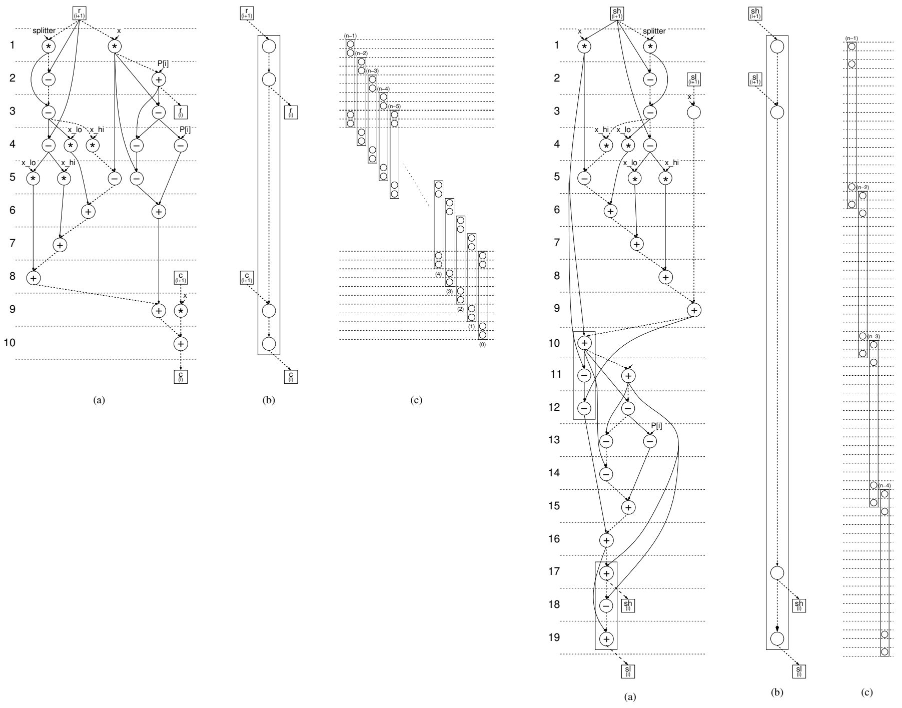


Fig. 1. Data-flow graphs of the main loop in AccurateEval1 (left) and AccurateEval2 (right). (a) one iteration, (b) its dependencies and (c) n iterations (part of it for the right one)

i.e., the average width of the full loop data-flow graph. These values are reported in Table 2. AccurateEval1 benefits from about 6.66 times more ILP than AccurateEval2. This certainly justifies that AccurateEval1 runs faster than AccurateEval2 on modern processors that are designed for exploiting this ILP. Of course no quantitative correlation with the measured cycles ratios can be done: current processors have limited resources compared to the ideal machine, and this pen-and-paper analysis only considers floating-point operations. In the next Section we present the PerPI tool which builds the full data dependency graph, including all the instructions in the trace, being floating point computations or integer control. Nevertheless, comparing this floating point ILP (Table 2) and the floating point count ratios (Table 1) of this manual analysis, we deduce that the accurate evaluation AccurateEval1 will run as fast as Eval on a processor that will exploit the whole ILP of this algorithm.

The graph analysis also exhibits the origin of such ILP differences. The two algorithms use almost the same groups of operations, but AccurateEval2 suffers from two bottle-necks identified as vertical rectangles on the (a) graph.

A detailed analysis is presented in [3]. In this perspective, this property will be useful to design other accurate algorithms more inspired by AccurateEval1 than by AccurateEval2.

3 The PerPI Tool

We now present the PerPI tool that automates this ILP analysis. PerPI currently includes the following facilities: ILP computation, ILP histogram and data flow graph displays.

3.1 Computing ILP

The measuring part of PerPI is a Pin tool [7]. Pin [4] is an Intel (R) free programmable tool. Pin consists of an engine which instruments any code at run time with user-defined measurement routines. PerPI is a set of routines aiming at computing the run code ILP. The ILP is computed while the real code is run. The examining routine gives the control of the examined code for a single instruction run and recovers control to update its examination statistics. This back and forth execution is continued until the examined code has been fully scanned. At each step of the examination, PerPI computes the run cycle of the examined instruction, increments the number of instructions run so far and possibly updates the highest run cycle. In such a way, PerPI computes ILP $= I/C$, where I is the number of machine instructions run, and C is the number of *steps* needed to complete the run. The higher the ILP, the more parallel the piece of code.

A *step* is defined as the following sequence of operations: for every *runnable* instruction, its source registers are read, its memory read references are loaded, its operations are computed, its destination registers are written, and eventually its memory write references are stored.

For example, `addl %eax,4(%ebp)` reads registers *eax* and *ebp*, computes $a = ebp + 4$, loads memory referenced by a (assume value v is loaded), computes $r = eax + v$, and stores r to memory referenced by a (the `addl` instruction could be the translation of a C source code instruction such as `x=x+y`, where x is in the function frame on the stack at address a and y is in register *eax*).

A step is performed in many cycles in a real machine. However in our tool, a step is considered as atomic to match the ideal machine. As in the example, ILP is the average number of machine instructions run per step. This definition of the ILP removes any micro-architectural details such as latency and throughput. We assume the piece of code is run on the best possible processor, with infinite resources and single cycle latency operators (including memory access and conditional and indirect branch resolution).

An instruction is *runnable* when all the source registers and all the memory read references are ready, *i.e.*, have been written by preceding instructions.

The Pin tool computes ILP as follows. For each instruction of the run in turn, apply the following procedure (*i.e.*, the procedure is applied to the full trace, in order).

1. For each source register, get the step at which it is updated
2. For each memory read reference, get the step at which it is updated
3. Let R be the latest of all the source register update steps
4. Let M be the latest of all the memory read reference update steps
5. The instruction is run at step $c = max(R, M) + 1$
6. For each destination register, mark it as being updated at step c
7. For each memory write reference, mark it as being updated at step c

While we compute the steps c, we adjust the step that has been last computed, giving the number of cycles of the run $C = \max(c)$.

For reproducibility, the system calls involved in the measured piece of code are not considered.

Table 3 illustrates how this algorithm computes the ILP of the expression `(a+b)+(c+d)`. The first instruction (eax=a, line 2) has its source ebp available at cycle 0 (line 1, column 7). It is run and it updates its destination register eax, making it available for later instructions at cycle 1 (line 2, column 3). Column 8 is the instruction number (I). Column 9 is its run cycle (c) and column 10 is the greatest run cycle (C). The last instruction (edx+=ebx) reads its sources ebx and edx at cycle 2 (look at the preceding line) and so is run and updates edx at cycle 3 (last line, column 6). There are 6 instructions ($I = 6$, last line column 8) run in 3 cycles ($C = 3$, last line column 10) on an ideal machine, which gives an ILP of $6/3 = 2$. The ideal machine runs this fragment of code at an average rate of two instructions per cycle.

Table 3. ILP computation yields ILP $= I/C = 2$, when evaluating $(a + b) + (c + d)$

		Availability as source register							
Instruction	Semantic	eax	ebx	ecx	edx	ebp	I	c	C
		0	0	0	0	0	0	0	0
`mov eax,DWP[ebp-16]`	eax=a	1	0	0	0	0	1	1	1
`mov edx,DWP[ebp-20]`	edx=b	1	0	0	1	0	2	1	1
`add edx,eax`	edx+=eax	1	0	0	2	0	3	2	2
`mov ebx,DWP[ebp-8]`	ebx=c	1	1	0	2	0	4	1	2
`add ebx,DWP[ebp-12]`	ebx+=d	1	2	0	2	0	5	2	2
`add edx,ebx`	edx+=ebx	1	2	0	3	0	6	3	3

3.2 Analyzing Facilities

The analysis part of the tool consists in histogram and graph displaying functions. These functions allow the user to zoom in and out of the trace. As in the example, the graph represents the instruction dependencies where an instruction j depends on an instruction i iff j has a source provided by i (j reads a register or a memory word x written by i and no instruction between i and j writes to x). The histogram represents the variation of the ILP along the steps.

The histogram tool is useful to locate the good (high ILP) and bad (low ILP) portions of the code run. The graph tool is useful to analyze why a code has a high or low ILP as illustrated in Section 4.

Table 4. ILP of `a+=b;` equals 4/3 for the RISC translation and 1 for the CISC one

c	RISC		CISC	
	Instruction	Semantic	Instruct.	Semantic
1	`load a,ra`	load memory a into register ra	`mov b,rb`	load mem. b into reg. rb
1	`load b,rb`	load memory b into register rb		
2	`add ra,rb,ra`	ra:=ra+rb	`add rb,a`	add reg. rb to mem. a
3	`store ra,a`	store register ra into memory a		

3.3 How to Interpret the Measured ILP?

ILP is defined as the number of machine instructions run divided by the number of ideal machine cycles needed to run them all on the ideal machine.

This definition shows that ILP is architecture-dependent. The number of instructions depends both on the machine language employed and the compiler.

For example, it is easy to exhibit examples where a RISC-like (*e.g.*, PowerPC, ARM, MIPS, SPARC) translation of a high-level code sequence shows a higher ILP than its CISC-like (*e.g.*, x86) equivalent. Table 4 illustrates this for the C code instruction `a += b;`. The *c* column depicts the cycle during which the instruction is run (starting from cycle 1).

In this example, we have ILP(RISC) $>$ ILP(CISC). This comes from the load/store model inherent to the RISC-like machine languages in which an instruction is either a memory access (load or store) or a computation involving registers only. In a CISC-like machine language, an instruction may involve both a memory access and a computation. This difference results in a RISC translation having more instructions than its CISC equivalent, leading to a possibly higher ILP (if more instructions are run in the same number of cycles).

However, we may notice that #cycles(CISC) $<$ #cycles(RISC), meaning that the CISC code can be run faster than the RISC one. We may also notice that #instructions(CISC) $<$ #instructions(RISC), meaning that the CISC code needs less resources than the RISC one.

Another difference between RISC and CISC leads to the opposite ILP ranking. Any x86 machine language computing instruction has an accumulating destination whereas in any RISC machine language, the destination may be distinct from the sources. As a consequence, a succession of computations may be translated in less instructions in a RISC language than in a CISC language.

A second example illustrating the preceding remark is given with the translation of $x = |a - b|$ from the C code sequence `x=(a-b>=0)?(a-b):(b-a);`. Corresponding RISC and CISC language translations are presented in Table 5. In this second example, we have ILP(RISC) $<$ ILP(CISC). We also have #cycles(RISC) $<$ #cycles(CISC) and #instructions(RISC) $<$ #instructions(CISC).

These two examples show that ILP should not be taken as the ultimate code quality factor. ILP is dependent on the architecture style (RISC vs CISC, 2-operands vs 3-operands instructions). A high ILP is not synonymous with a fast run but rather with a run which can fill the processor parallel units.

Table 5. ILP of $x = |a - b|$ equals 1.25 for RISC and 1.4 for CISC

c	RISC		CISC	
	Instruction	Semantic	Instruct.	Semantic
1	`load a,ra`	load mem. a into reg. ra	`mov a,ra`	load mem. a into reg. ra
1	`load b,rb`	load mem. b into reg. rb	`mov b,rb`	load mem. b into reg. rb
2	`sub ra,rb,rx`	rx:=ra-rb	`mov rb,rc`	rc:=rb
2			`sub ra,rb`	rb:=ra-rb
3	`csub (rx<0), 0,rx,rx`	rx:=(rx<0)?(0-rx):rx	`sub rc,ra`	ra:=rc-ra
4	`store rx,x`	store reg. rx into mem. x	`cmovge ra,rb`	rb:=(b-a≥0)?ra:rb
5			`mov rb,x`	x:=rb

4 Examples of Results

We present PerPI results for some accurate summation algorithms introduced in [5,9,8] and the previous polynomial evaluation algorithms. Sum2 and SumXBLAS are in that sense similar to AccurateEval1 and AccurateEval2. These algorithms are implemented as C functions and are called in a main part. From a practical point of view, binary files are submitted to PerPI through a graphical interface, and then some menu items generate the following outputs.

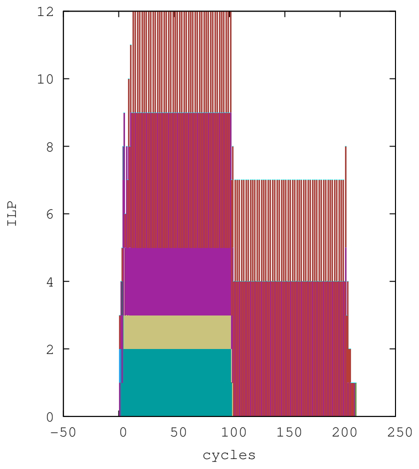
We first illustrate the ILP measure with Figure 2. Every called subroutine is analyzed, *i.e.*, PerPI returns the number of machine instructions I, the number of steps C, and the corresponding ILP. One run is enough since these values are reproducible.

```
Sum      ::I[511] ::C[105]::ILP[4.86]
Sum2     ::I[1617]::C[214]::ILP[7.55]
SumXBLAS ::I[2097]::C[898]::ILP[2.33]
```

Fig. 2. ILP measure for the three summation algorithms from [5] (100 summands)

Corresponding histograms are presented in Figure 3 and can be zoomed in as in Figure 4. This latter exhibits the color significance. In this case the red bars correspond to floating point operations, while purple ones are data transfers. These histograms exhibit the regularity of the ILP of the two algorithms and the better efficiency of Sum2.

The shape of the histograms starts with a high ILP part which comes from the control flow instructions. They usually are independent of the data flow computation (they control it) and so can all of them start simultaneously on the ideal machine. Branch, increment and compare (loop control) instructions are located only in this initial part of the histogram. This part serves also as a data flow ILP increasing period as the precomputed control flow instructions launch the data flow instructions which depend on them. After this, we find a data flow plateau (more uniformly colored) and an ILP decreasing end part.



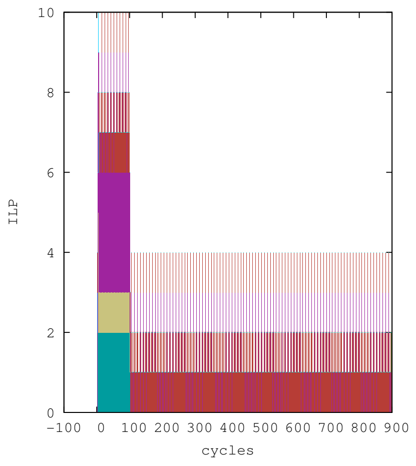


Fig. 3. ILP histograms for Sum2 and SumXBLAS and 100 summands

The last outputs are the data-flow graphs presented in Figure 5. This output automates the pen-and-paper results of Figure 1; cycles are on the Y-axis. After zooming into interesting parts of this graph, corresponding program instructions are displayed in such a way that the programmer can analyze the code.

In Figure 6 we display a zoom into the innermost loop for two other accurate summation algorithms introduced in [9,8], resp. AccSum and FastAccSum. The count of the floating-point operations suggests that FastAccSum should run about 30% faster than AccSum. Performance counter timing of some implementation confirms this speed-up – as in [8] we measure it for instance using gcc -O3 on an Intel Core 2. Nevertheless PerPI yields measures and data flow graphs that exhibit a higher degree of parallelism in one of the innermost loops of AccSum. This parallelism was not automatically detected by the previously mentioned implementation (as the assembly code reveals). A classical way to transform ILP into data parallelism is vectorization, here using SSE instructions. Such an improved implementation of AccSum now exhibits an average speed-up of 1.7 compared to FastAccSum[1] – while no cache size limitation applies. The AccSum ILP potential could be caught by vectorization while the lack of ILP in FastAccSum did not give any advantage compared to a vectorized implementation. This analysis also tells us that this algorithm may even benefit from larger vector microarchitectures as the ones present today in GPUs or tomorrow in AVX units in Intel corei7 2011 releases. It still has some ILP left.

[1] A speed-up of 1.3 has also been identified later with the newest version of the icc compiler without having to modify the source code. This again illustrates how the measured run times are dependent of the compiler and its versions.

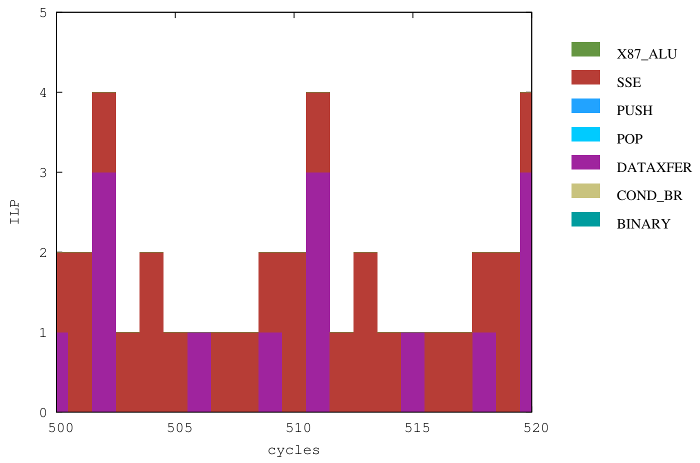


Fig. 4. Zoom of Figure 3 (SumXBLAS) with corresponding instruction type

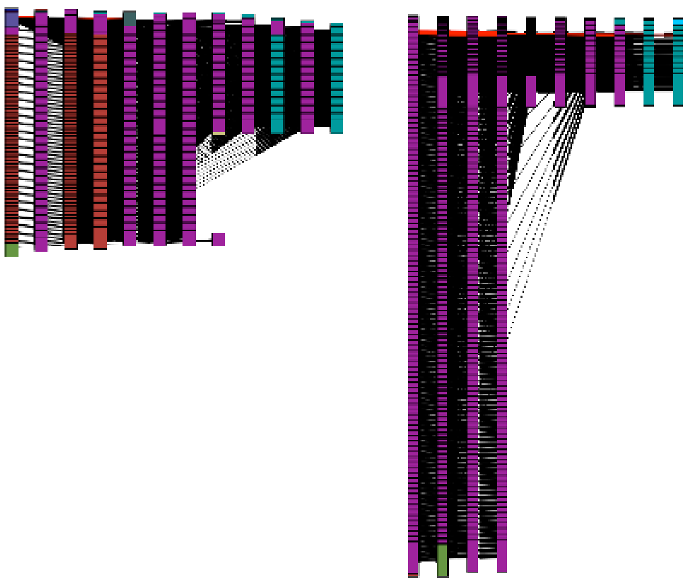


Fig. 5. Sum2 and SumXBLAS data flow graphs for 100 summands (Y-axis: cycles, vertical scales reduce the height of the right one)

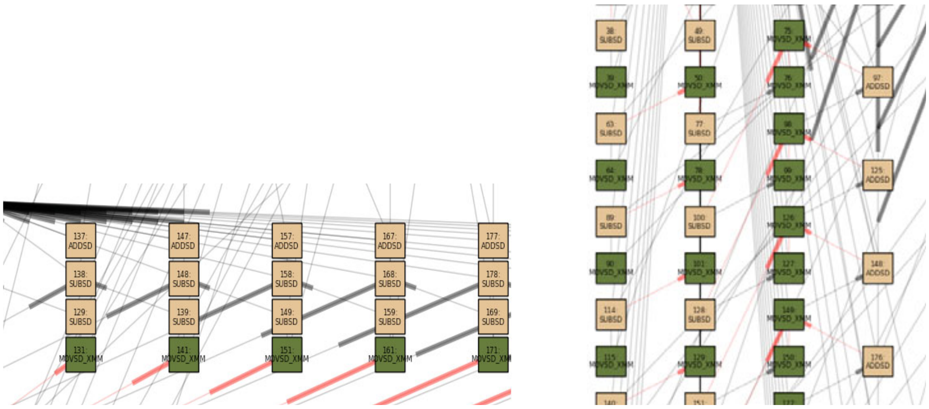


Fig. 6. Zoom into the dataflow graph of corresponding instructions in an innermost loop for two accurate summation algorithms from [9,8] (Y-axis: cycles). AccSum (left) exhibits more ILP than FastAccSum (right). The added ILP can be exploited thanks to vector units (SSE). The left zoom is actually only partially displayed (it is to be continued on each side with the same 4-steps shape), while the right zoom displays the full (but more limited) width of the loop for FastAccSum.

5 Conclusions and Current Work

The presented performance analysis and its PerPI tool aim to fill the gap between high level algorithm analysis and machine-dependent profiling tools. We illustrate on some core numerical algorithms that the first results are interesting and validate the proposed approach. These results are reproducible and help the programmer both to justify the measured performances and to improve the algorithm. As PerPI is based on Pin, it handles x86 machine code only. We have commented on how the machine language has an impact on the ILP measure. The presented version of PerPI will be publicly available soon. Work is in progress to extend the analysis facilities implemented in PerPI, as for example identifying longest dependency instruction chains or introducing constraints within the ideal machine.

References

1. Hennessy, J.L., Patterson, D.A.: Computer Architecture – A Quantitative Approach, 4th edn. Morgan Kaufmann (2007)
2. Higham, N.J.: Accuracy and Stability of Numerical Algorithms. SIAM (2002)
3. Langlois, P., Louvet, N.: More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms. Technical Report, DALI Research Team (2007), http://hal.archives-ouvertes.fr/hal-00165020

4. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: PLDI 2005: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200 (2005)
5. Ogita, T., Rump, S.M., Oishi, S.: Accurate Sum and Dot Product. SIAM J. Sci. Comput. 26(6), 1955–1988 (2005)
6. PAPI, http://icl.cs.utk.edu/papi
7. Pin, http://www.pintool.org
8. Rump, S.M.: Ultimately fast accurate summation. SIAM J. Sci. Comput. 31(5), 3466–3502 (2009)
9. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation – part I: Faithful rounding. SIAM J. Sci. Comput. 31(1), 189–224 (2008)
10. Touati, S.: Towards a Statistical Methodology to Evaluate Program Speedups and their Optimisation Techniques. Technical Report, PRISM, UVSQ (2009), http://hal.archives-ouvertes.fr/hal-00356529/en/
11. Weaver, V., Dongarra, J.: Can Hardware Performance Counters Produce Expected, Deterministic Results? In: 3rd Workshop on Functionality of Hardware Performance Monitoring, Atlanta, GA (December 4, 2010)

InterCell: A Software Suite for Rapid Prototyping and Parallel Execution of Fine Grained Applications

Jens Gustedt[1,3], Stéphane Vialle[2,3], Hervé Frezza-Buet[2],
D'havh Boumba Sitou[4], Nicolas Fressengeas[4], and Jeremy Fix[2]

[1] INRIA Nancy – Grand Est, France
[2] SUPELEC – UMI GT-CNRS 2958, France
[3] AlGorille INRIA Project Team, France
[4] LMOPS laboratory, Metz University and SUPELEC, France

Abstract. InterCell is an open and operational software suite for implementation, code generation and interactive simulation of fine grained parallel computational models. This article describes the software architecture, some use cases from physics and cortical networks as well as first performance measurements.

Keywords: fine grained parallel models, interactivity, rapid prototyping.

1 Introduction

The goal of the InterCell project is to help non-experts in parallel computing to use large scale parallel computers when developing models for physical phenomena, especially when these models have to be evaluated at large scale. To achieve this goal a software suite has been developed in order to allow rapid design and implementation of fine grained parallel computing models on coarse grained parallel architectures, *e.g* clusters or mainframes. The InterCell development cycle typically has several stages:

(1) rapid design of a mathematical model,
(2) automatic implementation of a fine grained parallel simulator,
(3) parallel execution of large scale interactive simulations, and
(4) large scale prototyping from the very beginning of model design.

Fine grained models of computation are widely adapted in different application domains. For our project this concerns two of these domains, namely the modeling of physical phenomena that have some notion of 'locality' (spatial or timely) and the modeling and development of neuromimetic networks [7]. But most likely InterCell could be useful for other domains as well.

Fig. 1 introduces the InterCell software architecture. At top level users describe their problems with application domain tools, such as a PDE solver or a cortical

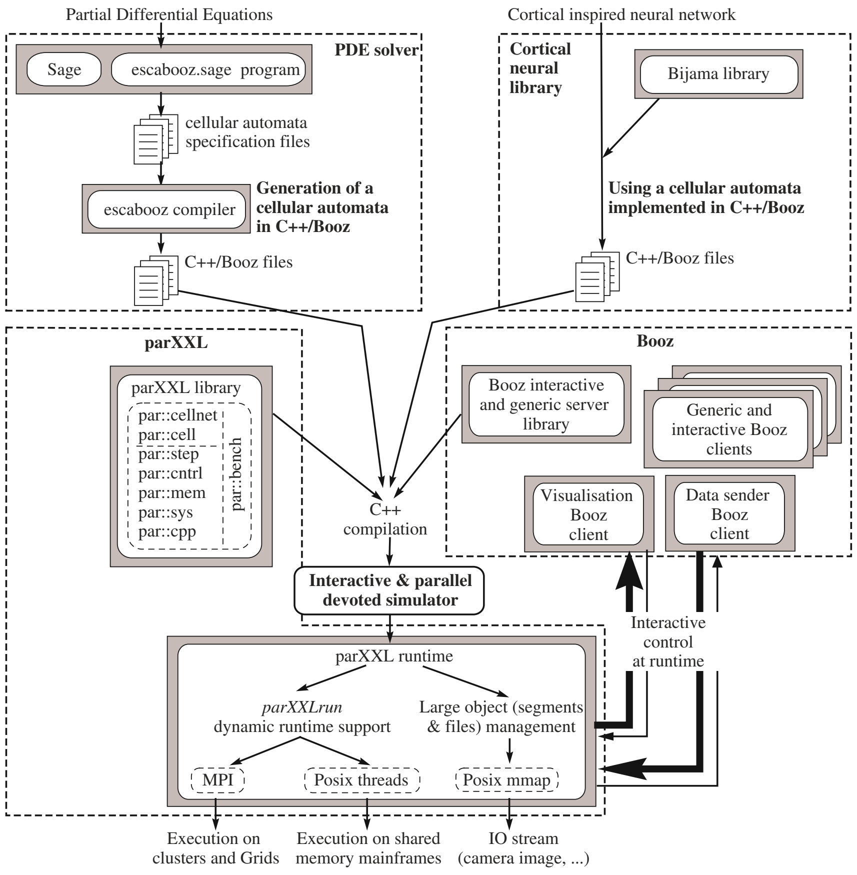

Partial Differential Equations
PDE solver
Sage
escabooz.sage program
cellular automata specification files
escabooz compiler
Generation of a cellular automata in C++/Booz
C++/Booz files
Cortical inspired neural network
Cortical neural library
Bijama library
Using a cellular automata implemented in C++/Booz
C++/Booz files
parXXL
parXXL library
par::cellnet
par::cell
par::step
par::cntrl
par::mem
par::sys
par::cpp
par::bench
C++ compilation
Booz
Booz interactive and generic server library
Generic and interactive Booz clients
Visualisation Booz client
Data sender Booz client
Interactive & parallel devoted simulator
Interactive control at runtime
parXXL runtime
parXXLrun dynamic runtime support
Large object (segments & files) management
MPI
Posix threads
Posix mmap
Execution on clusters and Grids
Execution on shared memory mainframes
IO stream (camera image, ...)

Fig. 1. Global architecture of our interactive problem modeler and PDE solver on large parallel and distributed computers

```
#Poisson's equation for semiconductor devices
eq1=(lmbda*nlap(phi(x,y),r,dr) ==
      ni*( exp(phi(x,y))-exp(-phi(x,y)))-dop(x,y)).substitute(x=0,y=0)

#Newman contidions on one border
anp=(lmbda*nd2(phi(x,y),y,dy) ==
      ni*( exp(phi(x,y))-exp(-phi(x,y)))-dop(x,y)).substitute(x=0,y=0)
```

Fig. 2. SAGE file (extract) specifying the electrostatic potential of a 2D P-N junction

inspired neural network simulator. These high level tools generate fine grained parallel simulators, using a C++ library named Booz[1]. This library ensures the interactive control of the simulations and in turn uses the parXXL library[2]. With that, it efficiently maps the fine grained computations on coarse grained parallel architectures. The parXXL runtime hides the underlying parallel or distributed hardware. The final software has two parts: a parallel and interactive server that handles the actual computations, and a set of easy-to-use control and visualization clients.

2 Fine Grained Parallel Computations

Fine grained computations that act on statically structured data (generally matrices) are nowadays well mastered and can be parallelized on coarse grained architectures (typically multicore clusters) with good results.

The case focused here is the computation on unstructured data for which the structure may even change occasionally and where the compute function that has to be executed may differ for each data point. Here an efficient mapping of computations to processors is not straightforward and good efficiency is generally difficult to achieve. parXXL provides a framework that facilitates programming under such constraints and draws good performances out of nowadays platforms.

The parXXL framework, see [5], includes several software layers, as shown in Fig. 1. Important for this project here are the following.

`par::cell`: a set of functionalities and a programming model to design and implement fine grained computations in the paradigm of so-called cellular computation. This layer allows to dynamically create and connect *cells* to establish cellular networks that are *executed cyclically.* When created, each cell is associated to four *cell behavior functions*: a function that is executed in each compute cycle, a query function that can be used to capture the state of the cell, a constructor and a destructor. A network of cells can easily be controlled by a sequential program, using *missions*, to create cells, execute one compute cycle, or extract data from the cells.

`par::mem`: an abstraction layer for handling large `chunks` of data. These allow for an efficient handling of large tables that are allocated on the heap or inside files and that can be resized dynamically. Technically, such a chunk may refer to memory on the heap (allocated with `malloc`), in shared segments (allocated through `shm_open`) or in files.
For the `par::cell` layer, it allows to group the cell data and output and access them in order or through hashed indices.

`par::cntrl`: handles the basic communication functionalities. It abstracts from the underlying runtime, currently MPI or POSIX threads. In particular important for this project has been the `transfer` family of functions that implements a `all_to_all_v` communication (used when each parXXL process

[1] `http://ims.metz.supelec.fr/spip.php?rubrique27`
[2] `http://parxxl.gforge.inria.fr/doxymentation/`

needs to exchange different data with all other processes). In combination with the resizability of the `mem::chunk`, `transfer` dispenses to specify communication sizes and to allocate buffers beforehand.
For the `par::cell` layer, these functions are mainly useful to implement cell communications, cell network creation and update.

`par::bench`: is used to instrument the library and to collect various performance data. In particular it registers the number of communications and their size, wall-clock and CPU times.

All these features are the foundations of parXXL, and have already been introduced in [5] and [4]. However, we have improved parXXL to support more asynchronous execution of cell networks, to easily collect and save results of large cell networks, and to increase its portability.

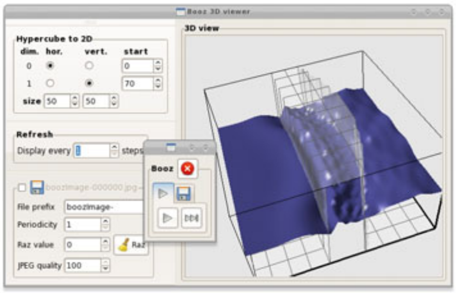
Cells can communicate the data from output to input channels in a synchronous or quasi-asynchronous mode. If in synchronous mode, cell input is updated at the end of each computation cycle. In quasi-asynchronous mode, cells are grouped in subsets and the output channels of the different groups are routed at different communication sub-cycles in the middle of the cell computations. So different cells reading a same output channel can read different values, depending on the concrete time of execution of their behavior functions in the computation cycle. The number of communication sub-cycles can be tuned at execution time. More sub-cycles lead to more asynchronism but to longer cell execution cycles.

In order to ease the extraction of results from large cell networks, we have designed some *collector* generic classes, and we have defined some new *missions* to easily run these data collection from a controlling sequential program. All these collections of large data are stored in chunks. These have been improved to map into files and avoid memory size limitations. Some optimized functionalities to read and write large data files storing N-dimension arrays of output channel values, have been added to parXXL. For example, they ease the initialization of large cell networks from data large files containing initial values for the output channels.

Finally, three parXXL runtimes exist: a first on top of MPI, a second on top of POSIX threads for multicore shared memory architectures, and a third on top of shared memory segments. All runtimes are available on 32 and 64 bits architectures. Moreover, great efforts have been made to improve the portability of our C++ template classes and functions, and our C++ meta-computing code. All these improvements make parXXL available and efficient on a larger set of parallel architectures.

3 Interactive Parallel Computations

One original property of the InterCell software suite is that cellular computation can be performed interactively. It allows visualization, writing and loading of snapshots, setting of cell values. This are all performed while the cellular automaton is briefly suspended. In addition it allows to step through the execution of the application. In our context that means that compute cycle after compute cycle may be observed individually. These features are provided by the Booz library that includes a visualization client, see Fig. 3.



Hypercube to 2D
dim. hor. vert. start
0 ● ○ 0
1 ○ ● 70
size 50 50
Refresh
Display every 1 steps
boozimage-000000.jpg
File prefix boozimage-
Periodicity 1
Raz value 0 Raz
JPEG quality 100

Fig. 3. Example of interactive Booz client

This interactivity allows to use a cluster for situated systems, like robots, where cellular computation models the inclusion of an artificial brain in some real robot perceptivo-motor loop. It also provides a real-time view of the running process, that allows to detect convergence problems of the cellular models. Such an on-line availability allows programmers of cellular automata to prototype their model at a large scale, from the very first design stage. This is of primary importance since properties of large scale discrete dynamical systems are not easily predictable from small prototypes.

4 Examples of InterCell Usage

Fig. 4 to 7 show examples of InterCell simulations. Fig. 4 is a classic 2D-Jacobi relaxation, where each *cell* simply computes the average value of its four neighbors. It has been implemented to test the functionality of our software suite. The application of Fig. 5 is modeled as a 2D grid of cells, each connected to its 8 neighbors. Here, each cell represents the elongation of a coil spring that is coupled to neighboring springs, in order to create 2D waves along the grid surface. The springs have different elasticity, the "lens" that is visible in light yellow shows the distribution of the elasticity among the springs.

The next sections, give the details of two simulations are real use cases of the InterCell suite, and that correspond to applicative research that was achieved in our laboratories.

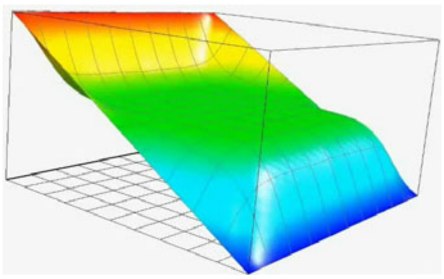


Fig. 4. InterCell simulation of a Jacobi relaxation

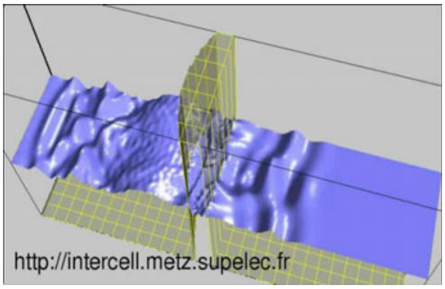


Fig. 5. InterCell simulation of a wave propagation

4.1 High Level Application Codes

As illustrated in Fig. 1, applications are developed using high level programming environments and not the parXXL or Booz layers directly. The *semiconductor simulation* detailed in Section 4.2 has been implemented using only the SAGE programming environment. This allows to implement our PDE within a mathematical paradigm, easily. Our PDE solver module then automatically generates C++ source files that wrap Booz and parXXL functionalities. A final C++ compilation produces a parallel application running on any parXXL runtime.

The cortical networks detailed in Section 4.3 have been developed in C++, using our Bijama library (see Fig. 1). Again, the application developer focusses on the expression of his scientific models. For himself, he does not implement process creation, internode cluster communication, or process synchronization. However, the distribution of the neural network on the different cluster nodes is not yet fully automatic and requires some directives of the application developer. This issue in currently under investigation.

4.2 Modeling and Simulating a Semiconductor

Fig. 6 is a more complex simulation from semiconductor physics. It shows the result of a simulation of the electrostatic potential in a 2D P-N junction whose N-doped side is the square upper part while the P-doped part is the rest. This computation is done through the `sage/escabooz` part as described in Fig. 1.

The numerical method is based on a modified version of the Least Squares Finite Element Method (LSFEM), see [6]. From LSFEM, we have derived a "*local only*" recursive rule. It allows for each point in a mesh to be considered as an independent automaton. This is particularly well suited for fine grained parallel computing. The initial problem is a partial differential system of equations involving a Poisson equation and the field expressions from the doping of the material. Added to this system is a set of boundary conditions: Dirichlet type where ohmic contacts are present, and Neumann type elsewhere.

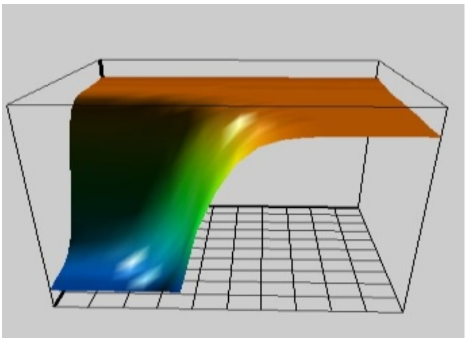


Fig. 6. InterCell simulation of a 2D P-N junction

The complete modeling and development process is thus as follows: the physicist (non-expert in parallelism) programs his equations in the SAGE[8] language, see Fig. 2, focusing entirely on physical and mathematical issues. Then, the `escabooz.sage` software suite, formally derives an update rule for each point of a given discretization mesh. Thereby it describes a complete cellular automaton. The SAGE program applies Newton's minimization method to a global error term. This error results from a discretized form of the initial partial differential problem. Following Newton's method, an approximate solution is fed to the

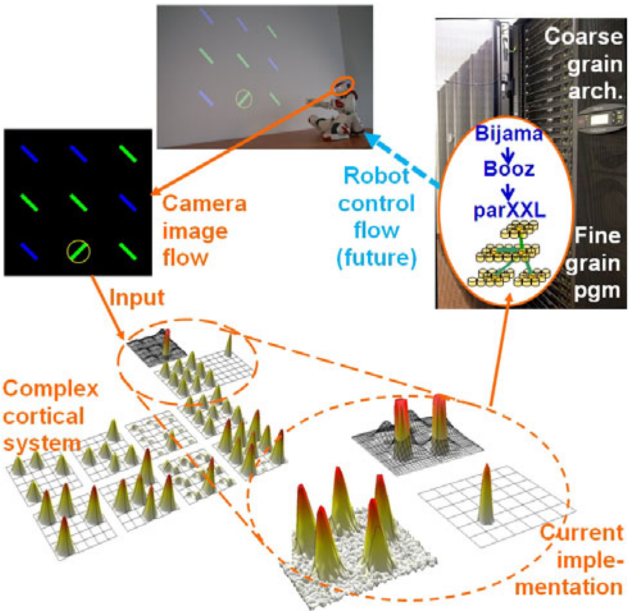


Fig. 7. InterCell implementation and run of a biologically-inspired neural network, for environment perception and robot control

automaton. Once run in asynchronous mode, the automaton eventually stabilizes around a fixed point. This is the nearest minimum of the error term and corresponds to the solution to the discretized problem.

4.3 Modeling and Simulating Cortical Networks

Fig. 7 illustrates a second kind of application of InterCell simulations. In this simulation, we aim at studying the emergent properties of dynamic neural fields, a model of cortical neural tissue [1], in particular focusing on sensorimotor control of embedded physical agents (e.g. a robot). The interaction of perceptive, motor and motivational flows of information within the neural network allows the agent to interact on a physical world. The bio-inspired nature of this work requires to simulate large population of neurons that are permanently feeded by a perceptive input and that produce motor actions. To simulate the dynamic neural fields, the continuous equation (1) is discretized using the Euler scheme.

$$\tau \frac{du}{dt}(x,t) = -u(x,t) + h + \sum_y w(x,y) f(u(y,t)) + s(x,t) \tag{1}$$

where $u(x,t)$ is the membrane potential of the neuron at position x and time t, h is a constant baseline, $w(x,y)$ is the kernel defining the interactions within the neural field and $s(x,t)$ is the input provided at position x and time t. The membrane potential $u(x,t)$ of all the cells evolves according to the same equation and therefore the computations are homogeneous across the cells. Simulating dynamic neural fields is, in this regard, particularly well suited for parallel implementations since a simulation usually involves large populations of fine-grained units. InterCell provides essential tools for scaling up models for realistic situations.

The simulation shown in Fig. 7 is a visual search task involving 11 2D dynamic neural fields, each made of 60×60 neurons (see [3] for details). The perceptive input is pre-processed along several dimensions (two colors, two orientations) and feeds a perceptive neural field. Specific connectivities within and between the fields lead to different emergent properties at the level of a neural field such as a competition between potential candidate targets, a working memory of targets that have been analysed or anticipatory mechanisms when camera movements are involved. On the bottom left of Fig. 7 is represented a visualization of the simulation with InterCell tools. The opportunity to visualize the whole network or a part of it is essential for tuning the parameters of the neural fields. In addition, InterCell allows to interact with the simulation on-line, constantly perturbating the network with a new perceptive input which is critical in the study of sensorimotor control.

The performance measurements provided in Section 5.1 were evaluated on a subpart of the model. This subpart involves three neural fields consisting of an input feeded by five stimuli, a competition neural field and a working memory. The connectivity within this model is rather dense. It contains 10800 neurons with a total of around 30 million connections. InterCell easily handles large networks with dense connections while such a scale-up is hardly handled by a single computer.

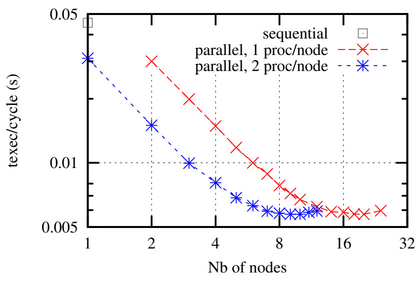
The modeling and development process is as follows: the computational neuroscientist (not expert in parallelism) writes done the differential equations governing the evolution of the state of the neurons. This definition is written in `C++` with the `Bijama` library. It involves defining a step method for the units, the connectivity kernel, the parameters of the equation (e.g. h in eq. 1) as well as communication methods that allow to embed the model within the physical environment. Once these methods have been defined, InterCell automatically handles the parallel computations and communications and the situated agent can be controlled by a simulation running on a cluster without taking care of where and how the simulation actually runs.

5 Experimental Performances

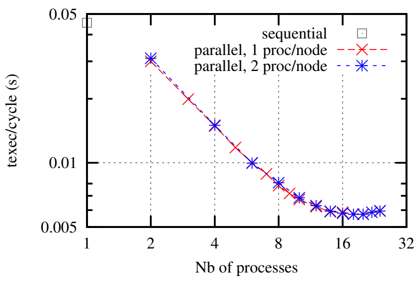
5.1 Performances of a Wave Propagation Simulation

In Fig. 8 we show the results of a first performance evaluation of the second application of Section 4 on the InterCell cluster, see Fig. 5. This cluster consists of 256 2.66 GHz Xeon bi-core nodes that are connected via standard Gbit Ethernet.

The example application consists of a 100×300 grid of cells that is split evenly among the parXXL processes. We experimented several different splitting strategies to find out that (for this example) the difference in performance is negligible. Thus, here we only give the values for a split along the long side



(a) As a function of the number of used computing nodes



(b) As a function of the number of run parXXL processes

Fig. 8. Execution time per compute cycle of the wave propagation simulation

(300) of the grid. Thus the grid is divided into 1×2 parts, 1×3 parts etc, where each part is treated by a separate parXXL process. In one series of experiments we placed one parXXL process per compute *node* and in a second series we placed two, *i.e* one parXXL process per compute *core*.

Each data point in the figures represents an average over several runs of batches of 1000 compute cycles. We have chosen 1000 compute cycles per run, because it leads to execution times of our benchmarks from $5.7s$ to $45.5s$ in function of the number of parXXL processes used. These times are all several orders of magnitude greater than the precision of our time measurement tool. We did not observe significant variances of our measurements other than changing the number of process per core or node. So we concluded that the variation that was introduced by the OS or the interconnexion network was negligible. As a consequence, we only performed 3 runs per parameter set to do the averaging.

Plotted are the run times broken down to the time for one compute cycle for the network. Fig. 8(a) shows the time against the number of nodes, Fig. 8(b) the time against the number of parXXL processes. We see that both series show an optimal speedup in the range of 2–8 processes, and up to 16 processes the speedup is still reasonable. From thereon the addition of additional processes / nodes doesn't accelerate the computation. So for the given problem, a number of about 2000 cells per parXXL process is a reasonable minimal requirement. Fig. 8(a) also demonstrates that this setting is well suited to take advantage of the two cores in each node.

5.2 Performances of a Biologically-Inspired Neural Network

In Fig. 9 we show the results of a first performance evaluation of the biologically-inspired neural network application introduced in Section 4.3, see the bottom of Fig. 7. Again we used the dual-core nodes InterCell cluster to run our benchmarks, and a similar performance measurement approach.

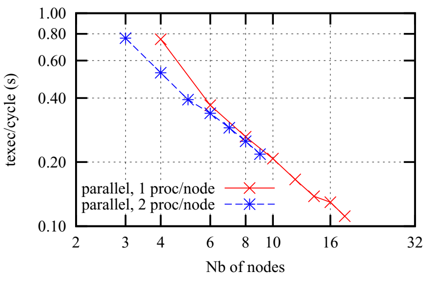
This neural network system is composed of three cortical maps: a medium one and two large ones. Execution attempts on 1 or 2 nodes failed, because of lack of memory. We implemented the medium map on 2 parXXL processes running on 2 CPU cores located on 1 or 2 nodes (dual-core nodes), and we distributed each large map on 1 to 8 processes running on other cores and nodes. We realized two series of benchmarks with 1 and 2 parXXL processes per node. To fulfill the minimum memory requirements of the application, we had to launch at least 4 parXXL processes running on 4 nodes and 6 parXXL process running on 3 nodes, respectively.

We achieved a good scalability distributing the large cortical maps, and a speed up close to 6.8 using $18 = 2 + 2 \times 8$ nodes in place of $4 = 2 + 2 \times 1$ with one process per node, see Fig. 9(a). However, at the opposite of the wave propagation simulation, execution times are approximately 2 times longer when running two parXXL processes per dual-core node, see Fig. 9(b). More investigations are required to identify the contention: this might be due to a memory contention during computations steps, or a network contention during communication steps, or both.

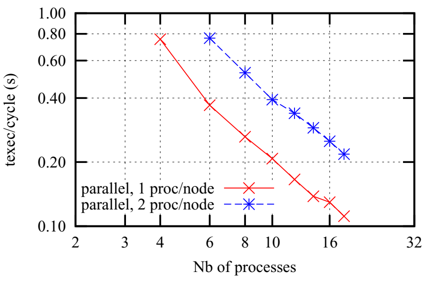
6 Conclusion and Perspectives

In this paper we presented InterCell, an open, operational software suite published under the GPL, see `http://ims.metz.supelec.fr`. This *development tool* is currently used by researchers in optics, photonics, and cortically-inspired neural networks. The later models are generally large and require interactive execution on large parallel systems. To these researchers, InterCell offers an easy-to-use tool to model and implement on a large, realistic scale. It provides automatic code generation and permits the parallel and interactive control of the simulation. First performance measurements are satisfying and show a good potential to address problems on a larger scale.

The next step in the development of InterCell will thus be to tackle applications of a larger scale: complex models are under investigations and large scale simulations are being implemented. Therefore, the Sage program that is currently used to specify the cellular automaton (which is still sequential) has to



(a) As a function of the number of used computing nodes



(b) As a function of the number of run parXXL processes

Fig. 9. Execution time per compute cycle of the biological inspired neural network

be parallelized to be able to process large problems rapidly. Also, some serial optimizations remain possible in the parXXL cell management.

Acknowledgments. The authors wish to thank Region Lorraine for its support to the InterCell project in the framework of the 2007-2013 CPER "Modélisations, informations et systèmes numériques (MIS)". Also we want to specially thank Patrick Mercier for his continuous support on the InterCell cluster.

References

1. Amari, S.: Dynamics of pattern formation in lateral-inhibition type neural fields. Biol. Cybern. 27(2), 77–87 (1977)
2. Boumba Sitou, D., Ould Saad Hamady, S., Fressengeas, N., Frezza-Buet, H., Vialle, S., Gustedt, J., Mercier, P.: Cellular based simulation of semiconductors thin films. In: Innovations in Thin Film Processing and Characterization - ITFPC 2009, France, Nancy (2009), http://hal.archives-ouvertes.fr/hal-00433062/en/
3. Fix, J., Rougier, N., Alexandre, F.: From physiological principles to computational models of the cortex. J. Physiol. Paris 101(1-3), 32–39 (2007)
4. Fressengeas, N., Frezza-Buet, H., Gustedt, J., Vialle, S.: An interactive problem modeller and pde solver, distributed on large scale architectures. In: Third International Workshop on Distributed Frameworks for Multimedia Applications - DFMA 2007, IEEE, France (2007), http://hal.inria.fr/inria-00139660/en/
5. Gustedt, J., Vialle, S., De Vivo, A.: The parXXL Environment: Scalable Fine Grained Development for Large Coarse Grained Platforms. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 1094–1104. Springer, Heidelberg (2007),
http://hal-supelec.archives-ouvertes.fr/hal-00280094/en/
6. Jiang, B.N.: The Least-squares Finite Element Method: Theory and Applications in Computational Fluid Dynamics and Electromagnetics. Springer, Heidelberg (1998)
7. Ménard, O., Frezza-Buet, H.: Model of multi-modal cortical processing: Coherent learning in self-organizing modules. Neural Networks 18(5-6), 646–655 (2005)
8. Stein, W.A., et al.: Sage Mathematics Software (Version 4.3). The Sage Development Team (2009), http://www.sagemath.org

PAS2P Tool, Parallel Application Signature for Performance Prediction

Alvaro Wong*, Dolores Rexachs, and Emilio Luque

Computer Architecture and Operating System Department,
Universidad Autónoma de Barcelona
Barcelona, Spain
alvaro@caos.uab.es,
{dolores.rexachs,emilio.luque}@uab.es

Abstract. Accurate prediction of parallel applications' performance is becoming increasingly complex. We seek to characterize the behavior of message-passing applications by extracting a signature to predict the performance in different target systems. We have developed a tool we called Parallel Application Signature for Performance Prediction (PAS2P) that strives to describe an application based on its behavior. Based on the application's message-passing activity, we have been able to identify and extract representative phases, with which we created a signature. We have experimented using scientific applications and we predicted the execution times on multicore architectures with an average accuracy of over 97%.

Keywords: Performance Prediction, Parallel Application Signature.

1 Introduction

Scientific programmers using any of the existing parallel programming models often must rely on performance analysis tools to help them optimize the performance of their programs. We propose a tool (PAS2P) that makes an analysis of the trace obtained by the execution of an application, with which it is extracted a signature (machine-independent application model) represented by a set of phases and weights, with their respective checkpoints, that predicts the execution time of the application on other target machine by running the signature and the processing times obtained. That helps to identify performance issues with minimal effort by providing information on the phases behavior such as communication, computational or waiting times.

To know the performance of a parallel application, our tool can helps to compared by means of different algorithms that solves the same problem, quickly and precisely, because the signature execution time is much less than the execution time of application, in contrast with the whole application the time required to run it thoroughly is an onerous requirement.

* This research has been supported by the MICINN Spain under contract TIN2007-64974, Appreciation to CONACYT-MEXICO for the support provided to Alvaro Wong.

We propose identifying these repetitive portions of an application creating a tool, which we dubbed Parallel Application Signature for Performance Prediction (PAS2P), to characterize message-passing parallel applications. PAS2P instruments and executes applications in a parallel machine, and produces a trace log. The data collected is used to characterize computation and communication behavior. To obtain the machine-independent application model, the trace is assigned a logical global clock according to causality relations between communication events, through an algorithm inspired by Lamport [8]. Then, we identify and extract the most relevant event sequences (phases) and assign them a weight from the number of times they occur. Finally, we create a signature defined by a set of phases selected depending on the value obtained by multiplying the weight of each phase by its execution time. This is the signature through whose execution in different target systems allows us to measure the execution time of each phase, and hence to estimate the entire application's run time in each of those systems. We do this by extrapolating of each phase's execution time using the weights we have obtained.

In order to evaluate the quality of the proposed tool, we have conducted a series experiments extracting signatures from applications such as CG and BT from NPB [1], Sweep3D [6], Parallel Ocean Model (POP) [12] and SMG2000 [2].

2 Related Work

There are other works which are more focused on the creation of an application signature. Snavely et al [10], extract the signature of an application using tools that allow them to capture its profile with emphasis on memory access patterns. To generate a machine signature, they use a probe developed for determining the "feeds and speeds" at single-processor or SMP nodes. Finally, they run these results on a network simulator to predict the performance of the parallel application. The main difference with our approach lies in the fact that, as our signature is the "core" of the parallel application itself, when we execute it on different parallel computers, real memory access patterns and the real computation resources requirements are used to evaluate the performance.

John Gustafson et al [4] propose a method that involve the creation of two profiles, a hardware signature and an application signature. In our proposed method, we do not need hardware characteristics as we create one single machine-independent signature because we execute in real systems. When we execute the signature on target machines, real memory access patterns and the real computation resources requirements are used to evaluate the performance.

S. Sodhi et al [11], claim that it is possible to obtain what they called a "performance skeleton" of an application by means of execution traces, and then generate code. We use the trace to create a signature using checkpoints. In short, we're using relevant application segments to predict the application's performance, instead of creating mock-ups.

Dimemas tool [3] is a performance prediction simulator for message-passing applications. In our approach we create a signature that represents the application, with the advantage that we can execute this signature on real systems quickly without simulation.

3 Methodology

Applications typically possess highly repetitive behavior, and parallel applications are no exception [9]. To characterize the computational and comunications-related behaviour of parallel applications, we propose identifying these repetitive portions of an application. We use this information to create a signature that, when executed, will allow the prediction of the full execution time in the particular machine where the signature was run.

As shown in Figure 1, there is a sequence of stages that are necessary to obtain the relevant portions (phases) and their weights. With this information, we can proceed to create a completely machine-independent signature for each application, that we can then execute in other systems in a shorter amount of time, since the execution time of the signature will always be a small fraction of the whole application's runtime. Finally, in the last stage, we predict the full execution time of the parallel application by adding the execution time of all the phases multiplied by their weights.

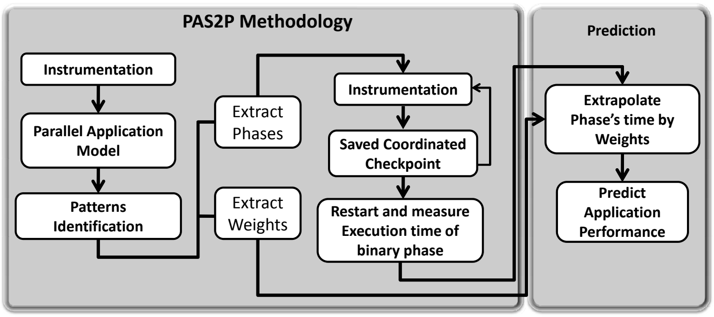

PAS2P Methodology
Instrumentation
Parallel Application Model
Patterns Identification
Extract Phases
Extract Weights
Instrumentation
Saved Coordinated Checkpoint
Restart and measure Execution time of binary phase
Prediction
Extrapolate Phase's time by Weights
Predict Application Performance

Fig. 1. PAS2P Methodology

As shown in Figure 1, there is a sequence of stages that are necessary to obtain the relevant portions (phases) and their weights. With this information, we can proceed to create a completely machine-independent signature for each application. Finally, to predict the full execution time of the application by adding the execution time of all the phases multiplied by their weights.

3.1 Data Collection

To instrument the applications, we need to collect communication and computation time. We instrument the communication calls to produce a log trace, adding specific code to extract the computational time between each pair of MPI primitives.

Starting from the concept of "Basic Block" (BB) [9] we can define a similar concept for parallel applications, and other similar helpful ones:

Event: The action of sending or receiving a message.

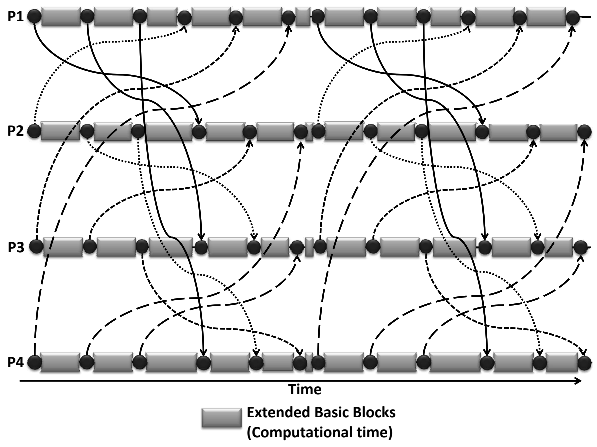


Fig. 2. Extended Basic Blocks

Extended Basic Block (EBB): We define it as a segment of a process whose beginning and end are defined by occurrences of messages, either sent or received. We may also say that it is a "computational time" segment bounded by communication actions, illustrated in Fig. 2.

3.2 Parallel Application Model

Synchronization between computing nodes, which is absent in sequential applications, becomes necessary.

In a previous work [13], we showed a logical clock based on the order of precedence of events across processes as defined by Lamport [8].

When we increase the number of processes, we found that the quality of prediction falls, due to processes becoming more independent and because there is a non-deterministic ordering of receives.

To solve the non-deterministic events (receives) problem, we have decided to introduce a new algorithm [14] inspired by Lamport's. Through this algorithm, we define a new logical ordering, in which, if one process sends a message in a logical time (LT), its receive will be modeled to arrive at LT + 1 and never afterwards. We show how we assign a Logical Time to events in Figure 3.

Once all events have been assigned an LT, we create a logical trace where Logical Times will be given by LT for the Send events (LTSend) and LT for the reception events (LTRecv), and so on as shown in Figure 3. We know that the message reception ordering may be random in the execution due to variable delays in the interconnection network generated by message collision; therefore

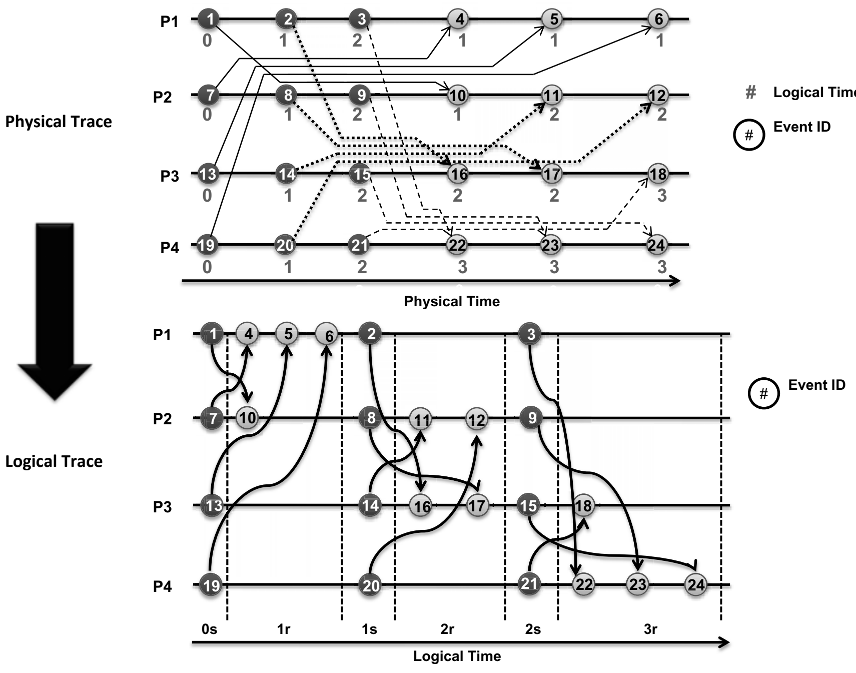


Fig. 3. Physical trace to Logical trace

we perform a permutation only inside the LTRecvs of the logical trace so the reception events will be ordered ascendant. Finally, once we have located each event, we divide the logical trace into more logical times, that is, there can only be one event for each process in a Logical Time.

Now, we are able to introduce two new concepts:

Tick: Logical time unit.

Parallel Basic Block (PBB): The set of Extended Basic Blocks delimited by two ticks. The first tick defined as Entry Point has at least one event, and the second tick defined as Exit Point also has at least one event.

3.3 Pattern Identification

To find the repetitive behavior of an application, we need to compare the behavior of each PBB and see if there are any similarities. We search for similarity between two PBBs based on the three main components of its structure as shown in Figure 4:

1. Communication Type: each of the assigned values of the entry points and each of the assigned values of the exit points should be the same; the tool compares the communication type. A positive value (source process) when an event is a Send and a negative value (destination process) when it's a Receive. When there are no events in a PBB, the value is zero.

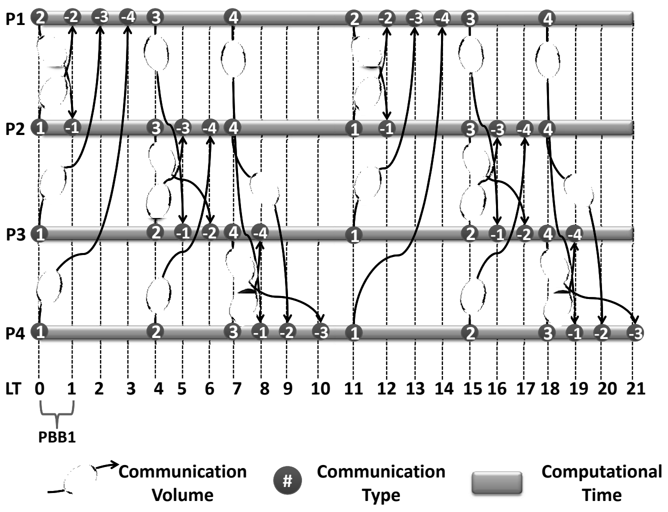


Fig. 4. Values of PBBs

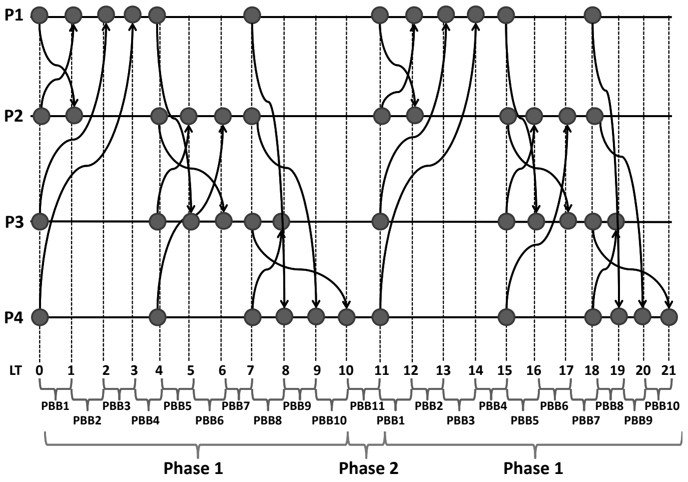

P1
P2
P3
P4
LT 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
PBB1 PBB2 PBB3 PBB4 PBB5 PBB6 PBB7 PBB8 PBB9 PBB10 PBB11 PBB1 PBB2 PBB3 PBB4 PBB5 PBB6 PBB7 PBB8 PBB9 PBB10
Phase 1
Phase 2
Phase 1

Fig. 5. Discovered phases

2. Communication Volume: each of the values of the entry and exit points must be similar, and can accept a difference of 5%.
3. Computation Time: each computational time allows for a difference of 5%.

Now we seek the PBBs with the similar behavior to “PBB1”, we find that “PBB12” is similar to the behavior of “PBB1”. So we rename it “PBB1” as it is essentially the same PBB. We apply this method to each of the PBBs, in which case the PBB list, obtained, including eleven different PBBs is shown in Figure 5. To identify and create the phases, a Phase is defined as sub-chains of grouped

PBBs that repeat along the execution. If we look at this simple example, we can identify two phases, see Figure 5.

Once we have identified phases, we proceed to create its weight vector and define the relevant phases.

Weight Vector: Will be given by the frequency in which each phase repeats.

Relevant Phase: A phase is relevant when the weight vector multiplied by the execution time is representative of the total runtime of the application.

3.4 Parallel Application Signature and Execution Prediction

Construction of the Signature. In order to build the signature, we have to built it in the same machine (machine base), to do this, we re-run the application to make the coordinated checkpoints [7] before each relevant phase happens. The checkpoint operation is taken before the starting point of the specific phase, in order to guarantee a correct warm-up time for the machine's components (cache, TLBs, etc) [5].

Signature Execution and Performance Prediction. Now, we can run the signature on target machines. This is done restarting from the saved state and start measuring from the point a phase begins until it ends. We repeat this method and proceed to execute all constituent phases. Finally, we obtained a unique signature for each program to run in different clusters. Once with the execution time of each phase and the weights of each phase, to predict the execution time of the whole application, we multiply the execution time of each phase by its weight.

4 Results

In this section, we show how we apply the PAS2P tool for the extraction of phases from scientific applications. We show the execution of the signature of each application on two clusters varying the number of cores. We predicted their execution time and demonstrated the prediction quality of each signature.

To evaluate the quality of the prediction and validate the proposed methodology, we performed the experimental evaluation in two target machines, labeled

Table 1. Cluster characteristics.

Cluster	Characteristics
Cluster A	Dual-Core Intel(R) Xeon(R) CPU 5150 2.66GHz 4MB L2, 8 GB Fully Buffered DIMM 667 MHz, Network Gigabit Ethernet, 128 cores.
Cluster B	2 x Quad-Core Intel(R) Xeon(R) CPU E5430, 2.66GHz 2x6MB cache L2, 16 GB RAM Fully Buffered DIMMs 667MHz, Network Gigabit Ethernet, 64 cores.

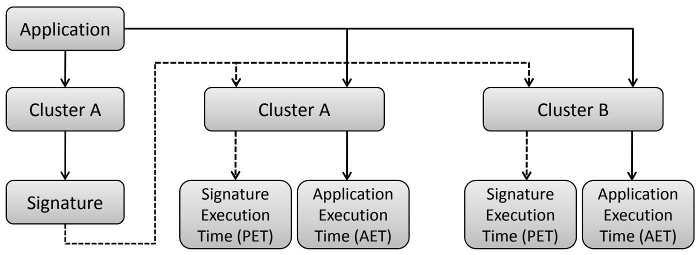

Application
Cluster A
Signature
Cluster A
Cluster B
Signature Execution Time (PET)
Application Execution Time (AET)
Signature Execution Time (PET)
Application Execution Time (AET)

Fig. 6. Experimental methodology

A and B. We present the results obtained for the following Parallel Program: CG, BT and SP from the NPB, Sweep3D, POP (Parallel Ocean Model) and SMG2000 and the characteristics of these machines are shown in Table 1. We show the Predicted Execution Time of each application and demonstrated the prediction quality of each signature.

The methodology of the results consists in executing each application, as shown in Figure 6, on Cluster A to extract its signature and afterwards execute it over Cluster A and B to obtain the Signature Execution Time (SET). Finally we execute the application over the two cluster to compare the Predicted Execution Time (PET) with the Application Execution Time (AET) to show the Prediction Execution Time Error (PETE).

To obtain the following results, we applied the proposed methodology in the above mentioned applications to extract phases and obtain the applications' signatures. Once we have run the signatures from all applications, we know the execution time of each phase, and therefore the Signature Execution Time (SET) which is the sum of all constituent phases. Now, to obtain the Predicted Execution Time (PET), we multiply the execution time of each phase by the weight vector given by the PAS2P, then sum up all times obtained.

The results from cluster A and B are shown in Tables 2. When we compare columns 3 (SET) and 7 (AET), we observe we have notably shortened the SET compared with the AET. In column 4 we put the percentage value from the division of SET by AET to demonstrate the reduction versus the Application Execution Time. In column 5, we can see the Predicted Execution Time (PET). Finally, in column 7, Prediction Execution Time Error is presented.

We have executed on two clusters to verify that the signature behavior is machine-independent. That is, we have the same number of phases and weights and the only part that varies is the execution time of the signature, which allows us to know (to predict) the application performance on each cluster we use.

With these results, we can show that the Signature Execution Time is lower in 98.37% than the Application Execution Time and the quality of prediction with an average accuracy of over 97.55%.

Table 2. Predictions on cluster A.

Program	Processes / Cores	SET (Sec)	SET vs. AET(%)	PET (Sec)	PETE (%)	AET (Sec)
Predictions on cluster A						
CG	64/32	5.39	0.22	2413.01	0.01	2412.70
CG	64/64	3.18	0.34	1165.24	2.85	1199.39
BT	64/32	6.24	0.58	1055.04	1.02	1066.00
BT	64/64	5.02	0.83	597.96	1.08	604.47
SP	64/32	7.76	0.76	1004.12	0.45	1008.74
SP	64/64	3.50	0.78	441.429	1.38	447.61
SMG2000	64/32	11.58	1.94	581.29	2.40	595.55
SMG2000	64/64	6.15	3.23	187.20	1.54	190.12
Sweep3D	32/16	2.44	0.10	2235.15	1.34	2265.34
Sweep3D	32/32	1.94	0.15	1257.03	0.27	1260.32
POP	64/32	19.92	1.48	1324.32	1.44	1343.55
POP	64/64	15.48	1.95	758.31	4.33	792.56
Predictions on cluster B						
CG	64/32	8.42	0.29	2793.42	1.90	2847.42
CG	64/64	4.87	0.32	1504.66	0.48	1511.91
BT	64/32	13.47	0.80	1652.65	0.9	1667.64
BT	64/64	10.19	0.77	1302.76	0.55	1309.91
SP	64/32	2.04	0.24	808.76	1.28	819.17
SP	64/64	2.08	0.51	388.367	3.05	400.55
SMG2000	64/32	16.75	2.63	633.23	0.38	635.61
SMG2000	64/64	8.37	10.15	162.87	2.32	166.74
Sweep3D	32/16	4.32	0.17	2494.36	0.06	2492.74
Sweep3D	32/32	3.01	0.22	1328.04	0.40	1322.62
POP	64/32	22.79	1.41	1608.85	0.17	1611.59
POP	64/64	18.36	1.79	1016.01	0.61	1022.28
SET: Signature Execution Time SET vs. AET: 100(SET/AET) PET: Predicted Execution Time AET: Application Execution Time PETE: Prediction Execution Time Error						

5 Conclusions and Future Work

PAS2P methodology allows us to generate a model of a parallel application, and subsequently, extract its most significant behavior (phases) automatically in order to create a signature, that by its execution, lets us predict the application's performance on different parallel computers. We have tested our methodology with a set of scientific applications varying numbers of cores obtaining a 97% prediction quality.

We propose the PAS2P tool to help programmers to find scientific computation which are performance issues when designing the scientific algorithms. This tool allows us to generate a model of the application, and subsequently, extract its most significant phases automatically in order to create a signature, that by its execution, lets us predict the application's performance on target machines.

We are working in the analysis the impact of the workload. We know workload is an important characteristic of the programs; it may be that different workloads alter the behavior of the phases or cause them to increase the weights.

Acknowledgments. This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974, Appreciation to CONACYT-MEXICO for the support provided to Alvaro Wong.

References

1. Bailey, D., Barszcz, E., Barton, J., Browning, D.: The NAS Parallel Benchmarks. International Journal of High Performance Computing (January 1991)
2. Brown, P.N., Falgout, R.D., Jones, J.E.: Semicoarsening multigrid on distributed memory machines. SIAM Journal on Scientific Computing 21, 1823–1834 (2000)
3. Girona, S., Labarta, J., Badía, R.M.: Validation of Dimemas Communication Model for MPI Collective Operations. In: Dongarra, J., Kacsuk, P., Podhorszki, N. (eds.) PVM/MPI 2000. LNCS, vol. 1908, p. 39. Springer, Heidelberg (2000)
4. Gustafson, J., Snell, Q.: Hint: A new way to measure computer performance. In: Hawaii International Conference on System Sciences, p. 392 (1995)
5. Hamerly, G., Perelman, E., Calder, B.: How to use simpoint to pick simulation points. ACM SIGMETRICS Performance Evaluation Review 31(4), 25–30 (2004)
6. Hoisie, A., Lubeck, O., Wasserman, H.: Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional. Journal of High Performance Computing Applications (January 2000)
7. Hursey, J., Squyres, J.M., Lumsdaine, A.: A checkpoint and restart service specification for open mpi. Technical Report TR635, Indiana University, Bloomington, Indiana, USA (July 2006)
8. Lamport, L., Time, C.: The Ordering of Events in a Distributed System. Communications of the ACM 21(7), 558–565 (1978)
9. Sherwood, T., Perelman, E., Calder, B.: Basic block distribution analysis to find periodic behavior and simulation points in applications. In: International Conference on Parallel Architectures and ... (January 1991)
10. Snavely, A., Carrington, L., Wolter, N., Labarta, J.: A framework for performance modeling and prediction. Supercomputing (January 2002)
11. Sodhi, S., Subhlok, J., Xu, Q.: Performance prediction with skeletons. Cluster Computing 11(2), 151–165 (2008)
12. Vetter, J.: Performance analysis of distributed applications using automatic classification of communication inefficiencies. In: ICS 2000: Proceedings of the 14th International Conference on Supercomputing, New York, NY, USA, pp. 245–254 (2000)
13. Wong, A., Rexachs, D., Luque, E.: Parallel application signature. In: IEEE International Conference on Cluster Computing and Workshops, CLUSTER 2009, August 31-September 4, pp. 1–4 (2009)
14. Wong, A., Rexachs, D., Luque, E.: Parallel application signature for performance prediction. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2010), vol. 2. CSREA Press (2010)

A Software Tool for Federated Simulation of Wireless Sensor Networks and Mobile Ad Hoc Networks

Ewa Niewiadomska-Szynkiewicz[1,2] and Andrzej Sikora[1]

[1] Research and Academic Computer Network (NASK),
Wawozowa 18, 02-796 Warsaw, Poland
{ewan,andrzejs}@nask.pl

[2] Institute of Control and Computation Engineering,
Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland

Abstract. The applicability of parallel discrete event simulation to design and evaluation of ad hoc networks is discussed. In this paper we describe the design, functionality, implementation and performance of our software system named MobASim. It is a Java-based integrated framework for wireless sensor and mobile ad hoc networks simulation performed on parallel computers and computer clusters that utilizes the paradigm of federating simulators and asynchronous distributed simulation technology. The computational results presented in the final part of the paper show the efficiency of parallel simulation performed upon MobASim and the application of our tool to design self-organizing communication network.

Keywords: Ad hoc network, MANET, WSN, discrete event system (DEVS), parallel discrete event simulation (PDES).

1 Introduction

The ad hoc networking is a relatively new area of research that has become extremely popular over the last decade and is rapidly increasing its advance into different areas of technology. Two types of such networks can be distinguished: Wireless Sensor Networks (WSNs) and Mobile Ad hoc Networks (MANETs). Typical WSN consists of a large number of homogenous, stationary nodes, i.e., densely deployed sensor devices. Nodes networked through wireless must gather local data and communicate with other nodes. MANET is formed through the cooperation of an arbitrary set of independent nodes, i.e., mobile, heterogeneous wireless devices. The nodes are free to move randomly and organize themselves. The network's wireless topology may change rapidly and unpredictably. There is no prearrangement assumption about specific role each node should perform. It makes its decision independently, based on the situation in the domain and its knowledge about the network.

Ad hoc architecture has many benefits, however its flexibility come at a price. Currently research effort is directed toward the specifics and constraints in ad hoc networking, such as: limited transmission range, limited link bandwidth and quality of transmission, constrained resources, mobility and multihop nature of the network [1,3,17]. Design, development and evaluation of such networks are non-trivial task. The complexity and scale of modern networks limit the applicability of purely analytic approaches. In this paper, we discuss some guidelines related to wireless ad hoc networks modeling and simulation. We model WSN and MANET simulators using discrete event systems methodology (DEVS) [2,18], and address the challenges to design high-performance simulation of these systems. Next, we describe the integrated software framework MobASim for simulation of ad hoc networks.

2 Simulation of Ad Hoc Networks

2.1 Wireless Network Simulation Tools

Recently a number of software systems for cable and wireless networks simulation have been developed to aid programmers. A survey of open source and commercial platforms for simulation of wireless networks is presented in [4,7,8,9,15]. Some of them are dedicated systems that are focused on a specific attribute of the behavior of a given network simulation, the others are general purpose tools that can be used for development and testing various types of networks. The high-level functionality, rich feature sets, comprehensive documentation and support options are available in commercial simulators: OPNET Modeler (www.opnet.com), QualNet (www.scalable-networks.com), which is the commercial version of GloMoSim (pcl.cs.ucla.edu/projects/glomosim). The commercial simulators have many advantages and offer scalable solutions but they are costly and typically do not offer the customization options of an open source simulator. Hence, they may not be the best choice for new technology design. Ns-2 (www.isi.edu/nsnam/ns), OMNeT++ (www.omnetpp.org) and GloMoSim are very popular open source simulators primarily used in network planing, research and education. However, all these general purpose network simulators are often not very suitable for large scale WSNs and MANETs. They do not address with the expected accuracy the complex interactions among nodes and environment. The other solutions are tools dedicated to simulate WSNs and MANETs. The TinyOS operating system is a common framework for sensor applications. TOSSIM (docs.tinyos.net/index.php/TOSSIM) is a discrete-events simulator for TinyOS wireless sensor networks. By exploiting the sensor network domain and TinyOS's design, TOSSIM can capture network behavior at a high fidelity while scaling to thousands of nodes. The same code can be used for simulation and real test-bed operating in TinyOS. A powerful product – NCTUns (nsl10.csie.nctu.edu.tw) for ad hoc network simulation and emulation was developed by SimReal Inc. The other system – SWANS is a scalable platform for large scale WSN simulation. SWANS is built atop the JiST (jist.ece.cornell.edu) – a high-performance discrete event simulation engine that runs over a standard

Java virtual machine. It should be pointed that most of presented open source simulators are now professional tools that compete with commercial software.

2.2 Parallel Simulation of WSN and MANET

The main difficulty in ad hoc networks simulation is the enormous computation power, i.e., speed and memory requirements needed to execute all events involved by internodes communication, and time varying topology. Another problem is scalability, i.e. how a given simulator scales for large topologies, high speed channels, and nodes mobility. As a consequence, the developments of methods to speed up calculations has recently received a great deal of interest. Parallel discrete event simulation (PDES) is a promising technique when performing the analysis of complex network systems [3,12]. Parallel execution of simulation can improve the scalability of a given simulator both in term of network size and execution speed, enabling large scale networks to be simulated in real time.

When DEVS simulation runs, it pulls events of the event queue (sorted by time) and executes them. In sequential discrete event simulation all computational processes simulating the physical ones access the same event list. In parallel discrete event simulation the shared data objects, i.e. the global clock and global event list are discarded – each computational process maintains its own local clock and local event list. Hence, all processes require explicit schemes for synchronization. Since events are characterized by their time stamps, the obvious mechanism for achieving PDES is to simultaneously execute the set of events with the same time stamp (i.e., occurring at the same time) on multiple processors or machines. Such mechanism is labeled synchronous parallel simulation. This approach is extensively restrictive. In asynchronous approach, the times of events do not dictate the order of execution of the events. To obtain the correct simulation results the calculations have to be performed with respect to the schemes described in [10,11,18]. Asynchronous simulation is much more effective due to its potentially high performance on a parallel platform [14]. In the last decades numerous integrated software platforms for parallel and distributed processing have been developed. OPNET, OMNeT++, GloMoSim, QualNet, NTUns provide tools for parallel and distributed calculations.

3 The MobASim System

There is a variety of simulators that ad hoc networks researchers and engineers are using to cover their needs. The common ones were described in the section 2. However, there were some reasons we decided to build our own simulator. We have found that most existing tools focus on the various IEEE standards for wireless communication with the lack of the reliable radio channel modeling. NS-2 and OMNeT++ provide only simplified wireless transmission models, TOSSIM does not model radio propagation. The more advanced signal propagation channel models are supported by NCTUns. Moreover, TOSSIM, GloMoSim, OMNeT++, OPNET do not support gathering power measurements, and do not consider signal strength. Many commonly used simulators do not model dislocations of nodes (TOSSIM) or provide simple mobility models (ns-2, OPNET, GloMoSim, QualNet, Castalia project in OMNeT++ : castalia.npc.nicta.com.au). Usually they have only implemented a mobility pattern model – the user just describes the destination point of a line segment (Castalia), and simple random mobility models (NCTUns). The other reason for developing a new simulator was the complicated architecture of available tools and limitations in results visualization and user-system interaction. In case of OPNET, OMNeT++ or ns-2 systems a user must read a large number of manuals to learn how to use the tool. The source coding is usually specialized for a given simulator and it is not easy to implement a given application and extend the system with the modules developed by the user (TOSSIM, OPNET, ns-2). Hence, we have decided to build an open expandable, flexible and scalable simulator that can be used for real-time simulation of self-organizing MANET and WSN. The focus was on reliable signal propagation, nodes' motion modeling and advanced programming interface. We plan to use our tool to ad hoc networks design and on-line decision support in operational management in real life applications (i.e., self-organizing network for monitoring, design and support in rescue actions, etc.).

3.1 MobASim Overview

MobASim is a general purpose software system for ad hoc networks simulation. The DEVS methodology is applied to model ad hoc network operation, i.e., the process being modeled is understood to advance through events. In MobASim simulation events can represent hardware interrupts (such as switch off the node) or high-level system events (such as packet reception, dislocation of nodes, etc.). In our application a model of each node of ad hoc network is composed of three types of components responsible for different functionalities: *Device* – a static or mobile radio device; *Communication Manager* – an object that models the wireless communication with other nodes in the network; *Mobility Manager* – an object responsible for tracking the node on the map and collision avoidance. Hence, the logical process (LP) simulating each node of given WSN or MANET simulator developed upon MobASim consists of three sets of classes (see Fig. 1): D – implementing tasks to be performed by static or mobile nodes; CM – implementing internode wireless communication and updating of the network communication topology; MM – implementing mobile nodes movement and providing the access to the information about environment and other nodes' positions in the network.

Three techniques for estimation of the signal degradation with a distance (path loss) are extensively used in practice [6]: long-distance, log-normal shadowing, and fading model. The MobASim simulator provides two models: long-distance and log-normal shadowing, hence each process CM implements one of these two models. The long-distance models combine analytical and empirical methods. They were developed to predict variations of the signal intensity over large distances. The log-normal shadowing models consider the fact that the transmission area of a transmitter may be different at two different locations,

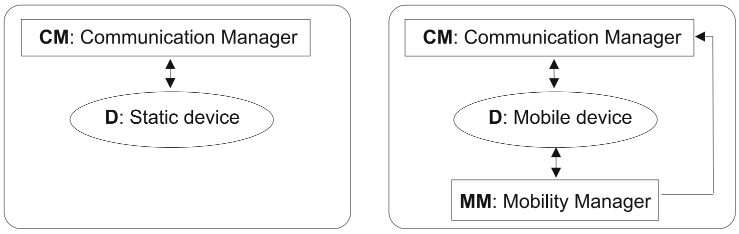


Fig. 1. Architecture of logical processes – WSN and MANET nodes

which leads to measure signals that are different than the average value calculated by the long-distance model. In these models path loss is modeled as a random variable with log-normal distribution.

The MobASim system provides the library of classes implementing the IEEE 802.15.4 standard for wireless communication. We have implemented physical and MAC (*Medium Access Control*) layer protocols according to the specification drawn up by the IEEE Computer Society, network and application layers protocols according to the specification drawn up by ZigBee Standard Organization. The user can select one of three types of MAC protocols.

Modeling of node mobility plays the crucial role in MobASim. The real-life movement patterns are very difficult to obtain, realistic models are usually very complicated. Many mobility models have been introduced. The detailed survey of these models is presented in [13]. In current version of MobASim each process from the group MM implements one of the following models for mobility pattern generation:

– The user describes the destination point of a line segment.
– The random waypoint model (two variants), i.e., random mobility model with randomly generated destination point and velocity.
– The map-based model that is used for applications in which nodes are constrained to move within defined paths.
– Our novel model named PFM (*Potential Function Model*) that combines the idea of potential function and particle-based mobility modeling. It calculates a collision-free movement at a group of mobile devices, and is dedicated to self-organizing network modeling.

In all mobility provided in MobASim the obstacles are included. The obstacles are generated by the user or are loaded from a real map. It is assumed that wireless signal is obstructed by the obstacles, too. All models utilize DEVS methodology. The state of each mobile node is described by four variables: location within the deployment region, orientation, velocity, and energy stored in the node. It is possible to combine various mobility models in one simulator, i.e., the model of mobility can switch w.r.t. the current state of the node.

3.2 Components of the System

The system consists of two sets of components: MobASim Simulator and Modeler, Fig. 2. The MobASim Modeler provides tools that allow to create the model of the system to be simulated. The model can be generated using the MobASim graphical editor or can be read from an XML file. The bidirectional interface to XML file that uses ASimML language – XML Schema specification for building XML file with description of parameterized system model is provided. A user can perform various operations during the experiment, i.e., configure the network system and manage the simulation. Hence, the MobASim Modeler consists of the following set of components implementing models and standards presented in section 3.1: *communication models library* – a collection of classes implementing models of wireless transmission, *wireless communication standards library* – a collection of classes implementing wireless communication standards, *mobility models library* – a collection of classes implementing mobility models, *GUI* – the graphical interface responsible for user-system interaction, *MobASim database* that stores all geographical information – a map of a deployment area, and all network nodes' positions.

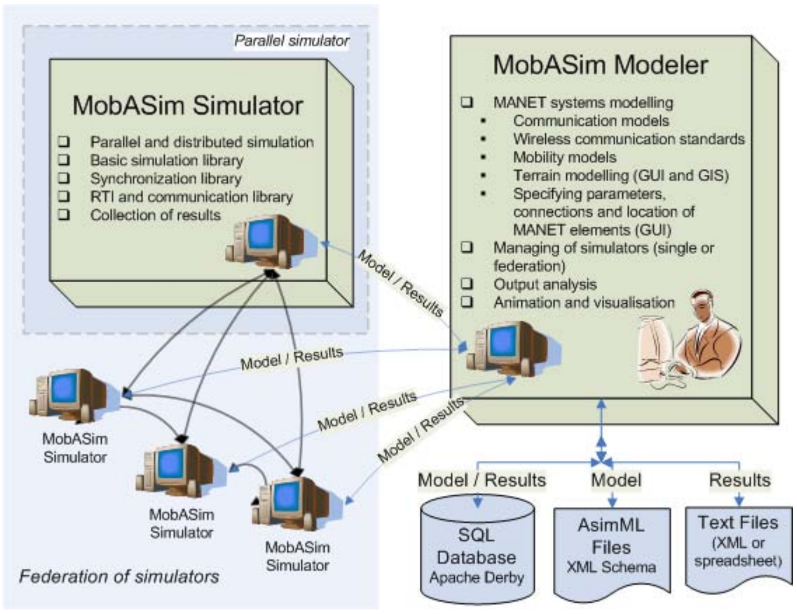


Fig. 2. Components of the MobASim system

The second component – MobASim Simulator is responsible for the DEVS simulator implementation and performance. It provides: *basic library* – a collection of classes implementing basic elements of each DEVS simulator, such as:

logical processes, events, event lists, messages passing, etc., *runtime infrastructure* – the library of classes that provide communication between the processes and machines, *synchronization protocols library* – a collection of classes implementing synchronization protocols reported in the next section.

3.3 Parallel Implementation of MobASim

MobASim is an integrated framework for parallel simulation. The model of an application to be simulated is decomposed into a number of entities with respect to their functionality and data requirements. A fine grained decomposition (a set of nodes) or coarse grained (a set of networks) are available. All network nodes are implemented as logical processes (LPs) utilizing classes from MobASim libraries. LPs communicate with each other through message-passing. Each simulator created upon MobASim has a hierarchical structure as depicted in Fi*Logical Process* (LP) – simulating the network nodes operation, *Domain* – a set of LPs, *Simulator* – a collection of domains. Computational processes that belong to the same level of hierarchy are synchronized. To provide high performance and

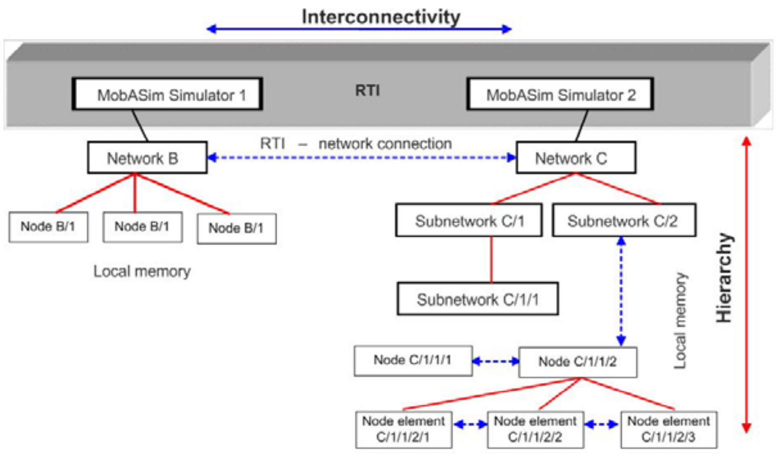


Fig. 3. Hierarchical structure of MobASim application

scalability we utilized the paradigm of federating disparate simulators [5] and asynchronous parallel simulation technology [18]. Such implementation distinguishes our software from the commonly used tools. Federated simulation is a way to interconnect separate simulators to serve as a *simulation-of-simulations*. In our application a model of given application is designed as a federation of disparate simulators of network nodes (Fig. 4) or networks (Fig. 5). By linking each of these simulators together through runtime infrastructure (RTI) that enables information exchange between them, they form a federated simulation that can be handled by a set of processors or computers. MobASim enables to

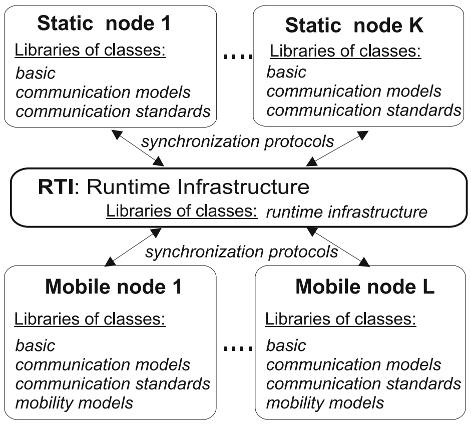


Fig. 4. A federation of simulators of wireless devices: fine grained parallelization

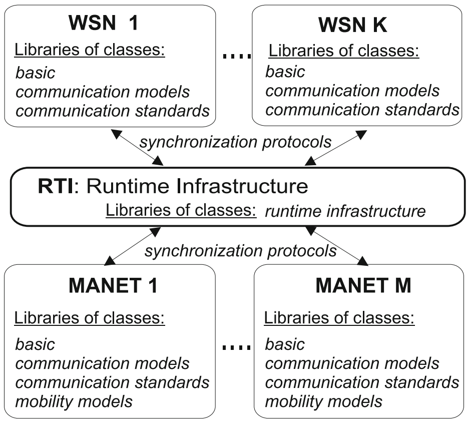


Fig. 5. A federation of simulators of networks: coarse grained parallelization

do synchronous and asynchronous simulation. Four algorithms for asynchronous simulator implementation are supplied: conservative protocol with null messages (CMB) [10], window conservative protocol (WCP) [11], optimistic Time Warp (TW) [11] and hybrid Moving Time Window (MTW) [16]. Parallel and distributed versions of MobASim were implemented; one simulator can utilize both of them.

The MobASim system is completely based on Java. It is built atop the ASimJava library described in [14], and developed by the authors. The Java messaging service (JMS) API provided by Sun Microsystems is used for interprocess communication.

3.4 Simulation Under MobASim

Three main phases can be distinguished during a simulation experiment: preparatory phase, configuration phase and experimental phase. At the first one the model of MANET or WSN to be simulated is investigated, developed and implemented. Within the configuration phase the user is asked to enter the parameters concerned with the whole simulated network (number of nodes, transmission model, etc.), the parameters concerned with each node (radio range, minimal and maximal speed, mobility model, routing protocol, energy reserve, etc.), and set up the simulation experiment (simulation time, number of processes, number of machines, etc.). The setting windows (MobASim GUI) are used to facilitate the configuration phase. The network can be constructed graphically. Finally, the application is saved in XML configuration files and can be used in many future simulations. After completing all initial settings and implementing all expected modules, the experimental phase begins. All processes are executed. The results of calculations are displayed (wireless transmission, time varying topology, etc.). The user employs the monitoring and analysis of the current situation (see Fig. 6). All results may be recorded into the disc file during the experiment.

4 Simulation Results

MobASim software was used to perform simulation of several network systems. It allows for setting up simulation experiments on parallel machines or computer clusters, and the analysis for different types of ad hoc networks. It does not involve any restrictions regarding the size of the simulation. The calculation speedup depends on the application and its decomposition. In this paper we present the results of two case studies. The first is concerned with the design of MANET to support the rescue action, the second presents the efficiency of parallel simulation of WSN upon MobASim software.

Example 1. Consider a situation where the fixed network infrastructure in a disaster area is damaged due to human activity. A rescue mission requires that new communication channels be quickly established in order to coordinate the

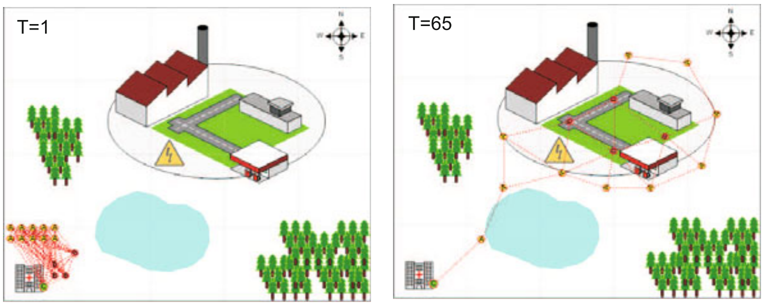


Fig. 6. MobASim: Design of communication network for rescue mission

actions. MANET can be successfully used here. It will enable communications with an adequate quality and will adapt to changing conditions and requirements in the danger zone. Computer simulation performed upon MobASim can be used to design an ad hoc network that will provide the continuous communication with all rescuers. During the tests the bandwidth of all links and current traffic are calculated, and the critical paths are pointed. The animation of time varying network topology – all nodes moving from the initial position to the destination, avoiding the obstacles – are displayed. The user can keep track how the communication network created by a set of mobile nodes adopts to the new positions of rescue teams. The current network connectivity is marked by lines connecting the nodes. Figure 6 shows the dynamically changing network topology during the entire network operational lifetime. The snapshots of initial and final topologies calculated for 1 and 65 time stamps are presented.

Example 2. The goal of experiments was to simulate 10 hours of WSN operation. The network was created by 60 sensor nodes. In case of battery equipped typical WSN, the primary design goal is to optimize the amount of energy used for transmission. The popular power save protocols attempt to save nodes energy by putting its radio transceiver in the sleep state. In our experiment it was assumed that each node was 2 hours in active mode and 8 hours in sleep mode. The objective of the test was to compare the efficiency of parallel simulation with the sequential realization. Two variants of implementation – **S** (sequential) and **P** (parallel) performed on the following hardware platforms were compared: **S** – AMD Sempron 1,67 GHz; **P** – AMD Sempron 1,67 GHz, AMD Athlon 1,2 GHz, Intel Core2 Duo 2,2 GHz. The execution times of each experiment are given in Table 1. We can observe that parallel simulation developed based on MobASim software can seriously speed up simulations of WSN operation w.r.t. sequential implementation. The calculation speedup depends on the size of the network model and assumed degree of parallelism.

Table 1. Results of WSN simulation – two implementations

	S (1 processor)	**P**(4 processors)
simulation time [s]	155	44
speedup w.r.t. sequential simulation	-	3.52

5 Conclusions

We described the MobASim software platform for ad hoc networks modeling and simulation. MobASim can support researches and engineers in network design, research, and network education. The system offers many unique advantages that can not be easily achieved by commercial and common open source simulators. MobASim is easy to use, and is especially useful in large scale applications in which the speed of simulation is of essence, such as real time mobility ad hoc network simulation. The federated approach to parallel simulation of networks, provided functionality, easy usage and its extensibility to include other open source modules or modules developed by a user, make our tool different from the popular software systems for simulation. We plan to release our system for research and education.

References

1. Aggelou, G.: Mobile Ad Hoc Networs. From Wireless LANs to 4G Networks. McGraw-Hill, USA (2005)
2. Banks, J., Carson, J.S., Nelson, B.L., Nicol, D.M.: Discrete-Event System Simulation, 5th edn. Pearson Education (2010)
3. Basagni, S., Conti, M., Giordano, S., Stojmenovic, I.: Mobile Ad Hoc Networking. Wiley-Interscience, IEEE Press (2004)
4. Caro, G.D.: Analysis of Simulation Environments for Mobile Ad Hoc Networks. Technical Report No IDSIA-24-03, IDSIA, Switzerland (2003)
5. Ferenci, S.L., Perumalla, K.S., Fujimoto, R.M.: An approach for federating parallel simulators. In: Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS 2000), pp. 63–70 (2000)
6. Forouzan, B.: Data Communications and Networking. McGraw-Hill (2004)
7. Hogie, L., Bouvry, P., Guinand, F.: An Overview of MANETs Simulation. In: Proc. of 1st Inter. Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, pp. 81–101 (2005)
8. Kasch, W.T., Ward, J.R., Andrusenko, J.: Wireless Network Modeling and Simulation Tools for Designers and Developers. IEEE Communications Magazine 46, 120–127 (2008)
9. Mehta, S., Ullah, N., Kabir, H., Sultana, N., Kwak, K.S.: A Case Study of Networks Simulation Tools for Wireless Networks. In: Proc. of the 3rd Asia International Conference on Modelling & Simulation, pp. 661–666 (2009)
10. Misra, J.: Distributed Discrete-Event Simulation. Computing Surveys 18 (1986)
11. Nicol, D.M., Fujimoto, R.: Parallel Simulation Today. Annals of Operations Research 53, 249–285 (1994)
12. Perrone, L.F., Yuan, Y., Nicol, D.M.: Simulation of large scale networks: Modeling and simulation best practises for wireless ad hoc networks. In: Proceedings of the 35th Winter Simulation Conference, pp. 685–693 (2003)
13. Roy, R.R.: Hanbook of Mobile Ad Hoc Networks for Mobility Models. Springer, USA (2010)
14. Sikora, A., Niewiadomska-Szynkiewicz, E.: A Federated Approach to Parallel and Distributed Simulation of Complex Systems. International Journal of Applied Mathematics and Computer Science 17, 99–106 (2007)
15. Singh, C.P., Vyas, O.P., Tiwari, M.K.: A Survey of Simulation in Sensor Networks. In: Proceedings of the Inter. Conf. Computational Intelligence for Modelling Control and Automation, pp. 867–872 (2009)
16. Socol, L.M., Briscoe, D.P., Wieland, A.P.: Mtw: a strategy for scheduling discrete simulation events for concurrent execution. In: Proceedings of the SCS Multiconference on Distributed Simulation, pp. 34–42 (1988)
17. Verdone, R., Dardari, D., Mazzini, G., Conti, A.: Wireless Sensors and Actuator Networks. Technologies, Analisis and Design. Elsevier (2008)
18. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation. Academic Press (2000)

Performance Engineering of GemsFDTD Computational Electromagnetics Solver

Ulf Andersson[1] and Brian J.N. Wylie[2]

[1] PDC Centre for High Performance Computing, KTH Royal Institute of Technology, Stockholm, Sweden
ulfa@pdc.kth.se
[2] Jülich Supercomputing Centre, Forschungszentrum Jülich, Jülich, Germany
b.wylie@fz-juelich.de

Abstract. Since modern high-performance computer systems consist of many hardware components and software layers, they present severe challenges for application developers who are primarily domain scientists and not experts with continually evolving hardware and system software. Effective tools for performance analysis are therefore decisive when developing performant scalable parallel applications. Such tools must be convenient to employ in the application development process and analysis must be both clear to interpret and yet comprehensive in the level of detail provided. We describe how the Scalasca toolset was applied in engineering the GemsFDTD computational electromagnetics solver, and the dramatic performance and scalability gains thereby achieved.

Keywords: performance engineering, parallel execution tuning, scalability, MPI, computational electromagnetics.

1 Introduction

Parallel applications develop from initial barely-functional implementations, via a series of debugging and performance improvement stages, into efficient and scalable versions via disciplined performance engineering practices. This process remains on-going throughout the productive lifetime of the application, as larger and more complex computational problem are addressed and upgraded hardware and system software become available. Although codes which become accepted benchmarks are more rigorously investigated than typical applications, they are not immune from the need for continual performance evaluation and re-engineering.

This paper considers the GemsFDTD code from the SPEC MPI2007 benchmark suite [6] which was found to perform particularly poorly at larger scales on distributed-memory computer systems. Initial analysis with the Scalasca toolset and then other tools pinpointed aspects in the application's initialization phase that severely limited scalability. Scalasca is an open-source toolset for analysing the execution behaviour of applications based on the MPI and OpenMP parallel programming interfaces supporting a wide range of current HPC platforms [2,3].

It combines compact runtime summaries, that are particularly suited to obtaining an overview of execution performance, with in-depth analyses of concurrency inefficiencies via event tracing and parallel replay. With its highly scalable design, Scalasca has facilitated performance analysis and tuning of applications consisting of unprecedented numbers of processes [10,11].

Based on these performance analyses, the developers of the GemsFDTD code could rework the initialization and further optimize the time-stepping loop, to realize substantial application execution performance and overall scalability improvements, ultimately leading to an entirely updated version of the benchmark. We review both initial and revised versions of the code, and their performance analysis with the Scalasca toolset which revealed now resolved and still remaining performance optimization opportunities.

2 GemsFDTD Code Versions

2.1 *113.GemsFDTD* — SPEC MPI2007 v1.1

The SPEC MPI2007 code *113.GemsFDTD* solves the Maxwell equations using the finite-difference time-domain (FDTD) method [8]. The radar cross-section of a perfectly conducting object is computed. *113.GemsFDTD* is written in Fortran 90 and is a parallel version of the SPEC CFP2006 (Floating Point Component of SPEC CPU2006) code *459.GemsFDTD*. *113.GemsFDTD* is a subset of a general purpose time-domain code for the Maxwell equations developed within the General ElectroMagnetic Solvers (GEMS) project at PSCI [5].

The core of the FDTD method is second-order accurate central-difference approximations of Faraday's and Ampere's laws. These central-differences are employed on a staggered Cartesian grid resulting in an explicit finite-difference method. These updates are performed in the module `material_class`. The FDTD method is also referred to as the Yee scheme. It is the standard time-domain method within computational electromagnetics [8].

An incident plane wave is generated using so-called Huygens' surfaces. This means that the computational domain is split into a total-field part and a scattered-field part, where the scattered-field part surrounds the total-field part. It uses the `excite_mod` module to compute the shape of the incident fields.

The computational domain is truncated by an absorbing layer in order to minimize the artificial reflections at the boundary. The uni-axial perfectly matched layer (UPML) by Gedney [1] is used. A time-domain near-to-far-field transformation computes the radar cross-section according to Martin and Pettersson [4], handled by the module `NFT_class`.

The execution time during the timestepping is concentrated in five subroutines, two update routines, two UPML routines, and the routine `NFT_store`.

The problem size in *113.GemsFDTD* is $580 \times 580 \times 580$ FDTD cells ($N_x = N_y = N_z = 580$) surrounded by a twelve cell UPML layer.

2.2 *145.lGemsFDTD* — SPEC MPI2007 v2.0

113.GemsFDTD was designed to be scalable up to 256 processes according to the aim of SPEC MPI 2007. Version 2.0 of SPEC MPI demanded that the codes were scalable up to 2048 processes. *113.GemsFDTD* failed miserably at this [7], thus an extensive rewrite was needed. The end result of this rewrite was *145.lGemsFDTD* ('l' signifies large) which was accepted into version 2.0 of SPEC MPI. (The original *113.GemsFDTD* is retained in SPEC MPI 2007 v2.0 medium-sized benchmark suite for compatibility with earlier releases.)

The problem size in *145.lGemsFDTD* is $960 \times 960 \times 960$ FDTD cells ($N_x = N_y = N_z = 960$) surrounded by a twelve cell UPML layer. This was selected in order to meet the SPEC request that the memory footprint should be slightly less than 64 GiB.

Initial performance analysis with Scalasca showed clearly that *113.GemsFDTD* performed a lot of 4-byte broadcasts during initialization. Time measurements inside the code itself showed that the `multiblock_partition` routine on the master rank was a serial bottleneck. In fact, the execution time increased with the number of blocks, which increases linearly with the number of MPI processes. This performance analysis made it clear where the hotspots in the code were and was very useful to the programmer.

2.3 Domain Decomposition of GemsFDTD

The original Gems code (*MBfrida*) lets the user select the number of processes (p) and the number of FDTD blocks in all three dimensions (Nbx, Nby, Nbz), where the total number of FDTD blocks are $NbF = Nbx \times Nby \times Nbz$. If the user selects $Nbx = Nby = Nbz = 0$, then the code sets $NbF = p$ and uses `MPI_Dims_create` to set (Nbx, Nby, Nbz). The size of the FDTD blocks are ($Nbx/N_x, Nby/N_y, Nbz/N_z$) (or slightly smaller). If a layer of UPML is added, then the total number of blocks become $NbT = (Nbx+2) \times (Nby+2) \times (Nbz+2)$.

In *113.GemsFDTD* it was chosen to always use `MPI_Dims_create`, but setting NbF to $p-2$ or $p-3$ instead of p so that NbF is an even number. This was done in order to get at least two processes that only have UPML blocks. NbF is then sometimes further decreased in order to avoid getting elongated FDTD-blocks which is undesirable since that will lead to more UPML-blocks and more data to communicate between the blocks. A precalculated table decides which NbF values are accepted. In the case of $p = 256$, 254 and 253 are not accepted values, while 252 ($=6\times6\times7$) is. Due to a bug in the code only 252+2=254 processes are used when computing a distribution of the 576 blocks (252 FDTD blocks and 324 UPML blocks).

When the number of blocks and their sizes are decided, the master computes the workload of each block and then, in the routine `multiblock_partition`, computes a distribution of the blocks onto the MPI ranks.

For *145.lGemsFDTD*, NbF is the largest approved value that is less than or equal to the number of processes p. For $p <= 256$, the same precalculated table as for *113.GemsFDTD* is used to decide whether a NbF value is approved, whereas for $p > 256$ we demand that $(NbT/NbF)/(1 + 6/\sqrt[3]{NbF}) < 1.3$ in order to approve NbF. This makes sure that NbT/NbF is bounded.

After a block distribution is computed, the master loops through the blocks and broadcasts information about each block. In *113.GemsFDTD* this part had $65\times NbT+25$ four-byte broadcasts. In *145.lGemsFDTD* this has been reduced to $7\times NbT+11$ small broadcasts by merging adjacent broadcasts. Further reductions were possible but would have meant extensive rewrites of the code.

2.4 Summary of the Main Changes

The major improvements in *145.lGemsFDTD* compared to *113.GemsFDTD* for the phases where they apply:

Initialization

1. Develop a new `multiblock_partition` routine designed for larger numbers of processes, which produces a completely different domain decomposition.
2. Use of fewer `MPI_Bcast` operations within `multiblock_distribute` and associated routines.
3. Separation of broadcasts and allocations in the block distribution phase, such that the rank that owns the block delays allocations for the block until after block information has been broadcasted for all blocks.

Time-stepping iterations

1. Removal of expensive recomputation of the communication pattern used to exchange blocks in `multiblock_communicate`, since the same communication pattern is used in each timestep.
2. Replacement of `MPI_Sendrecv` with non-blocking `MPI_Isend` and `MPI_Irecv` communication.
3. Interchanging loops in the near-to-far-field transformation computations. (This improvement was found analyzing the serial code *459.GemsFDTD*.)

3 Performance Measurements and Analyses

During the course of development of GemsFDTD, performance measurements of each version were done on a variety of platforms and with different compilers to verify the portability and effectiveness of each of the modifications. Although benefits depend on the respective processor, network and compiler/optimizer capabilities, execution performance analysis with Scalasca and the improvement achieved for a representative example system are studied in detail.

3.1 Execution Scalability

The original version of the GemsFDTD benchmark code (*113.GemsFDTD* SPEC in MPI2007 v1.1) and revised version (*145.lGemsFDTD* in SPEC MPI2007 v2.0) were executed on the *HECToR* Cray XT4 [9] with varying numbers of processes. The XT4 has quad-core 2.3 GHz AMD Opteron processors in four-socket compute blades sharing 8 GiB of memory, allowing the benchmark code to run with 32 or more MPI processes. The codes were built with PGI 9.0.4 compilers using typical optimization flags "-fastsse -O3 -Mipa=fast,inline" as well as processor-specific optimization. Although *113.GemsFDTD* would normally be run with a 'medium-sized' input dataset (`sphere.pec`), to allow comparison of the two versions we ran both with the 'large-sized' training (i.e., `ltrain`) dataset using `RAk_funnel.pec`. It was convenient to use only 50 timesteps rather than the `lref` benchmark reference number of 1500 timesteps, since previous analysis of GemsFDTD [7] determined that there was no significant variation in execution performance for each timestep. Full benchmark execution time can be estimated by multiplying the time for 50 iterations by a factor of 30 and adding the initialization/finalization time.

Execution times reported for the entire code and only for the 50 timestep loop iterations of both versions are shown in Figure 1. While the iterations are seen to scale well in both versions, the initialization phases only scale to 128 and 512 processes, respectively. For the original code, the initialization phase becomes particularly dominant, even with only 256 processes, and makes it prohibitive to run at larger scales.

3.2 Scalasca Performance Analyses

Both versions of GemsFDTD were prepared for measurement with the Scalasca instrumenter (1.3 release), which configured the Cray/PGI compiler to automatically instrument each user-level source routine entry and exits, and linked with its measurement library which include instrumented wrappers for MPI library routines. The instrumented executables were then run under the control of the Scalasca measurement and analysis nexus within single batch jobs (to avoid impact of acquiring different partitions of compute nodes in separate jobs).

By default, a Scalasca runtime summarization experiment consisting of a full call-path execution profile with auxilliary MPI statistics for each process is collected and stored in a unique archive directory. With all of the user-level source routines instrumented, there can be undesirable measurement overheads for small frequently-executed routines. An initial summary experiment is therefore scored to identify which routines should be filtered: with GemsFDTD, the names of nine routines were placed in a text file and specified to be filtered from subsequent measurements in Scalasca summary and trace experiments.

Scalasca summary analysis reports contain a breakdown of the total run time into pure (local) computation time and MPI time, the latter split into time for collective synchronization (i.e., barriers), collective communication and point-to-point communication, as detailed in Figure 2. Both code versions show good scaling of the computation time, with the new version demonstrating better absolute performance (at any particular scale) and better scalability overall. The extremely poor scalability of the original code version is due to the rapidly increasing time for collective communication, isolated to the numerous `MPI_Bcast`

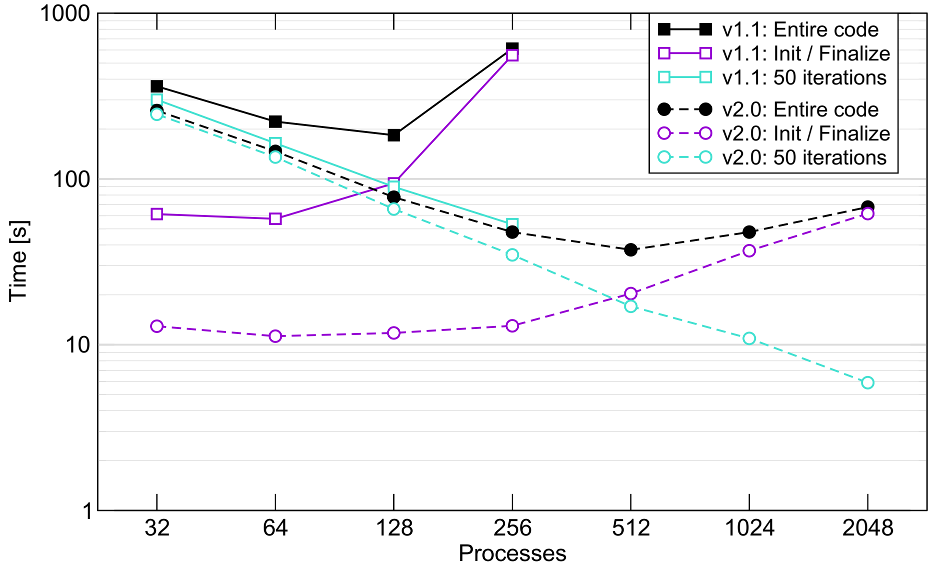


Fig. 1. Execution times of original and revised GemsFDTD versions with 'ltrain' dataset for a range of configuration sizes on Cray XT4

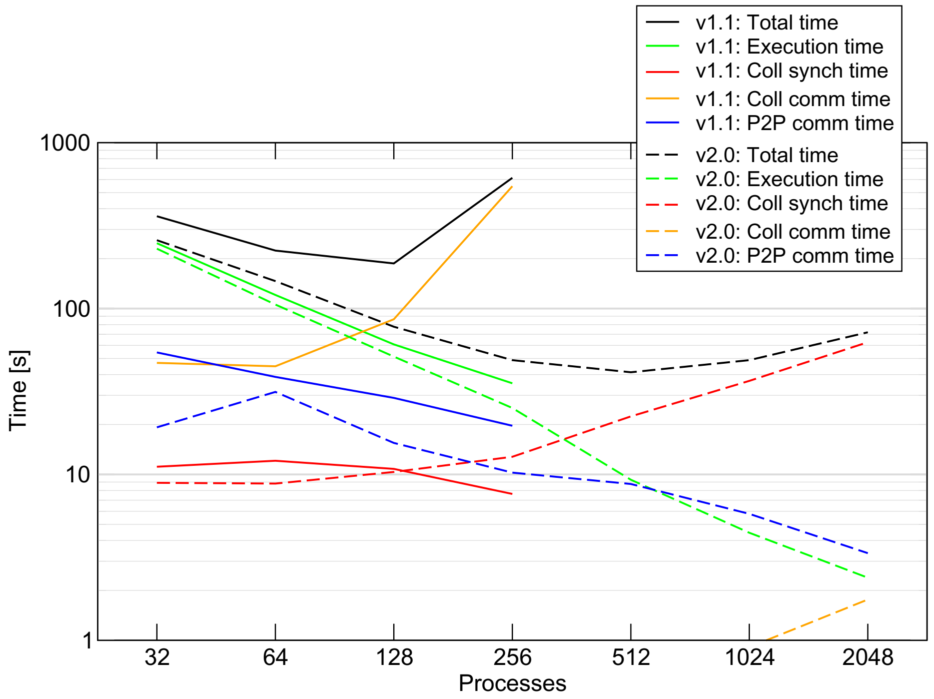


Fig. 2. Scalasca summary analysis breakdown of executions of original and revised GemsFDTD versions with 'ltrain' dataset on Cray XT4 (averages of all processes)

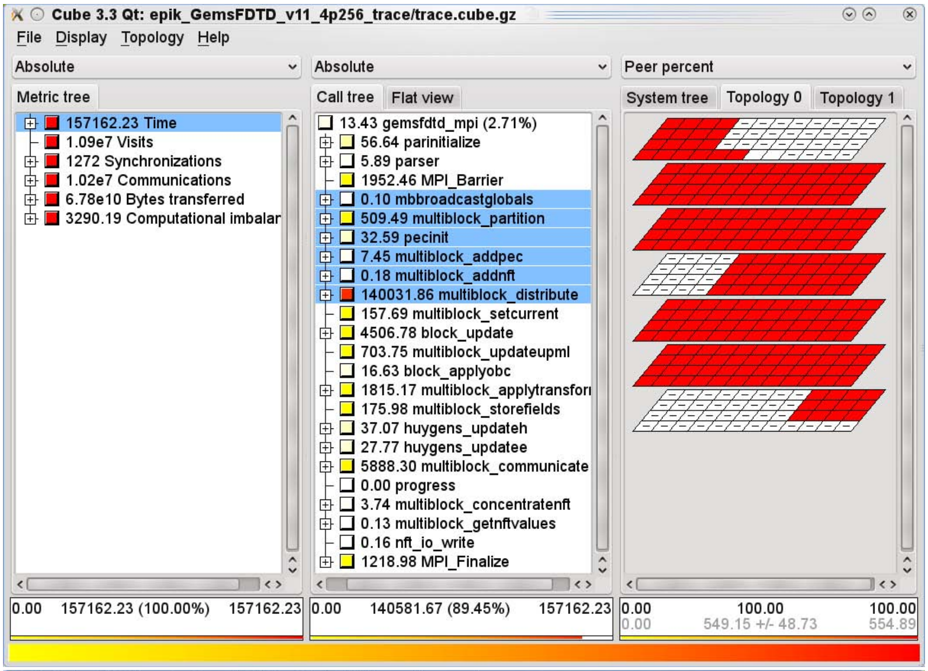
calls during initialization. Collective communication in the revised version is seen growing significantly at the largest scale, however, the primary scalability impediment is the increasing time for explicit barrier synchronization (which is not a factor in the original version). Point-to-point communication time is notably reduced in the revised version, but also scales less well than the local computation, which it exceeds for 1024 or more processes.

Scalasca automatic trace analysis profiles are similar to those produced by runtime summarization, however, they include assessments of waiting times inherent in MPI collective and point-to-point operations, such as *Late Sender* time when an early receiver process must block until the associated message transfer is initiated. Trace analysis profiles from 256-process experiments as presented by the Scalasca analysis report explorer GUI are shown in Figures 3 and 4 comparing the original and improved GemsFDTD execution performance. In Figure 3, with the metric for total time selected from the metric trees in the leftmost panes, and the routines that constitute the initialization phase of GemsFDTD selected from the central call-tree panes, the distribution of times per process is shown with the automatically acquired Cray XT4 machine topology in the right panes. The MPI process with rank 0 is selected in the v2.0 display, and the two processes in the uppermost row of compute nodes that idled throughout the v1.1 execution can be distinguished. (Only the subset of the HECToR Cray XT4 associated with the measured execution is shown, with non-allocated compute nodes grayed or dashed.) In contrast, Figure 4 features *Point-to-point communication time* selected from the metric tree and the associated MPI routines within `multiblock_communicate` of the solver iteration phase.

The Scalasca analysis reports from GemsFDTD v1.1 and v2.0 trace experiments with 256 processes on the Cray XT4 are compared in Table 1 to examine the 12-fold speedup in total execution time of the revised version. Dilation of the application execution time with instrumentation and measurement processing was under 2% compared to the uninstrumented reference version.

Initialization is more than 40 times faster through the combination of reworking the `multiblock_partition` calculation and using almost 9 times fewer broadcasts in `multiblock_distribute` (even though 3% more bytes are broadcast). Note that in v1.1 the majority of broadcast time (measured in `MPI_Bcast`) is actually *Late Broadcast* time on the processes waiting for rank 0 to complete `multiblock_partition`, however, the non-waiting time for broadcasts is also reduced 20-fold for v2.0. Improvement in the solver iterations is a more modest 33%, however, still with speedup in both calculation and communication times. Significant gains were therefore realized through use of non-blocking communication and improved load balance (including exploiting the two previously unused processes), despite 5% more data being transfered in `multiblock_communicate`.

The Scalasca traces are less than one quarter of the size for the v2.0 version due to the reduced number of broadcasts, requiring less than half a second to write the traces to disk and unify the associated definitions. Replay of recorded events requires correspondingly less time, however, the time processing event timestamps to correct logical inconsistencies arising from the unsynchronized



Cube 3.3 Qt: epik_GemsFDTD_v11_4p256_trace/trace.cube.gz

File Display Topology Help

Absolute | Absolute | Peer percent

Metric tree:

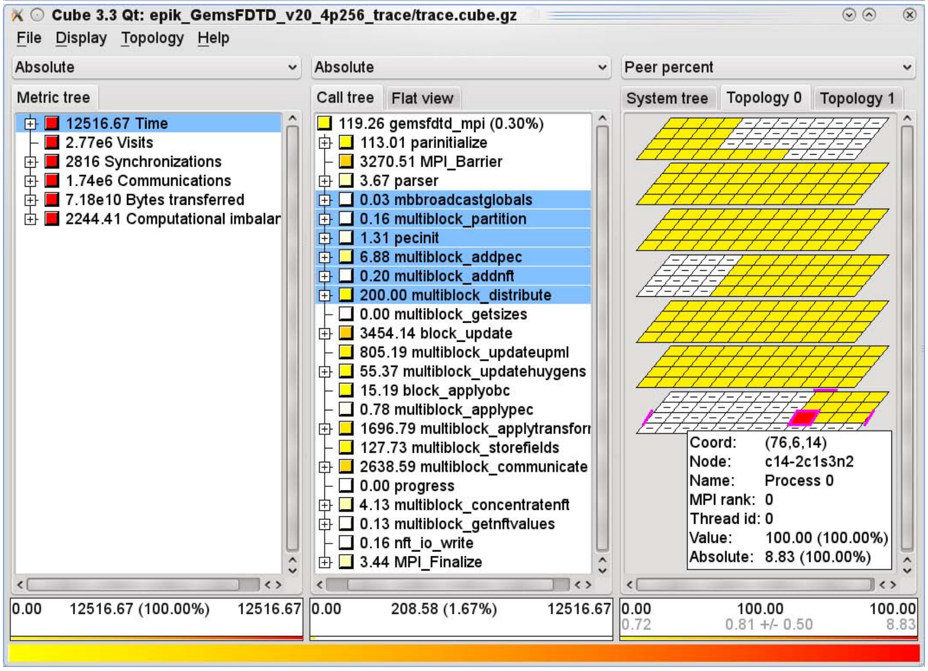
```
157162.23 Time
1.09e7 Visits
1272 Synchronizations
1.02e7 Communications
6.78e10 Bytes transferred
3290.19 Computational imbalar
```

Call tree | Flat view:

```
13.43 gemsfdtd_mpi (2.71%)
56.64 parinitialize
5.89 parser
1952.46 MPI_Barrier
0.10 mbbroadcastglobals
509.49 multiblock_partition
32.59 pecinit
7.45 multiblock_addpec
0.18 multiblock_addnft
140031.86 multiblock_distribute
157.69 multiblock_setcurrent
4506.78 block_update
703.75 multiblock_updateupml
16.63 block_applyobc
1815.17 multiblock_applytransfori
175.98 multiblock_storefields
37.07 huygens_updateh
27.77 huygens_updatee
5888.30 multiblock_communicate
0.00 progress
3.74 multiblock_concentratenft
0.13 multiblock_getnftvalues
0.16 nft_io_write
1218.98 MPI_Finalize
```

System tree | Topology 0 | Topology 1

```
0.00   157162.23 (100.00%)   157162.23
0.00   140581.67 (89.45%)    157162.23
0.00   100.00                100.00
0.00   549.15 +/- 48.73      554.89
```



Cube 3.3 Qt: epik_GemsFDTD_v20_4p256_trace/trace.cube.gz

File Display Topology Help

Absolute | Absolute | Peer percent

Metric tree:

```
12516.67 Time
2.77e6 Visits
2816 Synchronizations
1.74e6 Communications
7.18e10 Bytes transferred
2244.41 Computational imbalar
```

Call tree | Flat view:

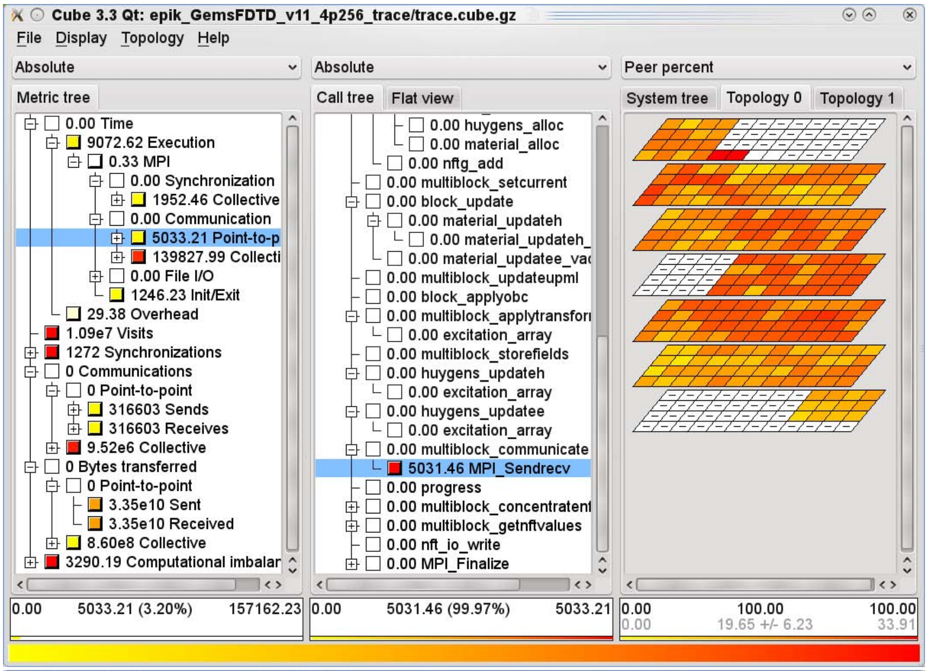
```
119.26 gemsfdtd_mpi (0.30%)
113.01 parinitialize
3270.51 MPI_Barrier
3.67 parser
0.03 mbbroadcastglobals
0.16 multiblock_partition
1.31 pecinit
6.88 multiblock_addpec
0.20 multiblock_addnft
200.00 multiblock_distribute
0.00 multiblock_getsizes
3454.14 block_update
805.19 multiblock_updateupml
55.37 multiblock_updatehuygens
15.19 block_applyobc
0.78 multiblock_applypec
1696.79 multiblock_applytransfori
127.73 multiblock_storefields
2638.59 multiblock_communicate
0.00 progress
4.13 multiblock_concentratenft
0.13 multiblock_getnftvalues
0.16 nft_io_write
3.44 MPI_Finalize
```

System tree | Topology 0 | Topology 1

```
Coord:     (76,6,14)
Node:      c14-2c1s3n2
Name:      Process 0
MPI rank:  0
Thread id: 0
Value:     100.00 (100.00%)
Absolute:  8.83 (100.00%)
```

```
0.00   12516.67 (100.00%)   12516.67
0.00   208.58 (1.67%)       12516.67
0.00   100.00               100.00
0.72   0.81 +/- 0.50        8.83
```

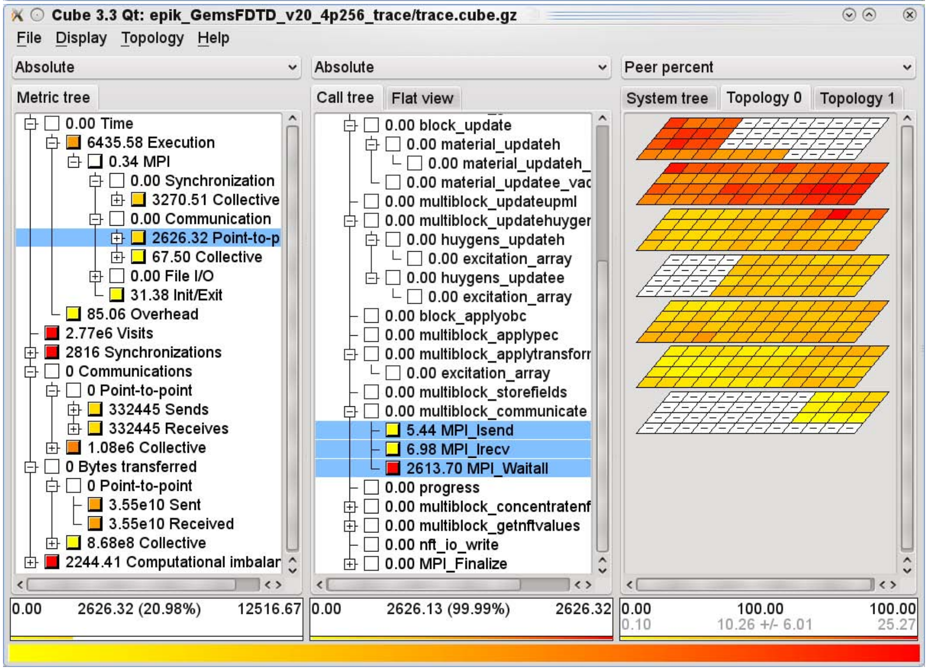
Fig. 3. Scalasca analysis report explorer presentations of GemsFDTD trace experiments with 'ltrain' dataset for 256 processes on Cray XT4 (v1.1 above and v2.0 below) highlighting 675-fold improvement of total time for the initialization phase



```
Cube 3.3 Qt: epik_GemsFDTD_v11_4p256_trace/trace.cube.gz
File  Display  Topology  Help
Absolute | Absolute | Peer percent
Metric tree | Call tree  Flat view | System tree  Topology 0  Topology 1

0.00 Time                              0.00 huygens_alloc
  9072.62 Execution                    0.00 material_alloc
    0.33 MPI                         0.00 nftg_add
      0.00 Synchronization         0.00 multiblock_setcurrent
        1952.46 Collective         0.00 block_update
      0.00 Communication             0.00 material_updateh
        5033.21 Point-to-p             0.00 material_updateh_
        139827.99 Collecti           0.00 material_updatee_vac
      0.00 File I/O                0.00 multiblock_updateupml
      1246.23 Init/Exit            0.00 block_applyobc
  29.38 Overhead                   0.00 multiblock_applytransfor
1.09e7 Visits                        0.00 excitation_array
1272 Synchronizations              0.00 multiblock_storefields
0 Communications                   0.00 huygens_updateh
  0 Point-to-point                   0.00 excitation_array
    316603 Sends                   0.00 huygens_updatee
    316603 Receives                  0.00 excitation_array
  9.52e6 Collective                0.00 multiblock_communicate
0 Bytes transferred                  5031.46 MPI_Sendrecv
  0 Point-to-point                 0.00 progress
    3.35e10 Sent                   0.00 multiblock_concentratent
    3.35e10 Received               0.00 multiblock_getnftvalues
  8.60e8 Collective                0.00 nft_io_write
3290.19 Computational imbalar      0.00 MPI_Finalize

0.00  5033.21 (3.20%)  157162.23 | 0.00  5031.46 (99.97%)  5033.21 | 0.00  100.00  100.00
                                                                  | 0.00  19.65 +/- 6.23  33.91
```



```
Cube 3.3 Qt: epik_GemsFDTD_v20_4p256_trace/trace.cube.gz
File  Display  Topology  Help
Absolute | Absolute | Peer percent
Metric tree | Call tree  Flat view | System tree  Topology 0  Topology 1

0.00 Time                          0.00 block_update
  6435.58 Execution                  0.00 material_updateh
    0.34 MPI                           0.00 material_updateh_
      0.00 Synchronization           0.00 material_updatee_vac
        3270.51 Collective         0.00 multiblock_updateupml
      0.00 Communication           0.00 multiblock_updatehuyger
        2626.32 Point-to-p           0.00 huygens_updateh
        67.50 Collective               0.00 excitation_array
      0.00 File I/O                  0.00 huygens_updatee
      31.38 Init/Exit                  0.00 excitation_array
  85.06 Overhead                   0.00 block_applyobc
2.77e6 Visits                      0.00 multiblock_applypec
2816 Synchronizations              0.00 multiblock_applytransforr
0 Communications                     0.00 excitation_array
  0 Point-to-point                 0.00 multiblock_storefields
    332445 Sends                   0.00 multiblock_communicate
    332445 Receives                  5.44 MPI_Isend
  1.08e6 Collective                  6.98 MPI_Irecv
0 Bytes transferred                  2613.70 MPI_Waitall
  0 Point-to-point                 0.00 progress
    3.55e10 Sent                   0.00 multiblock_concentratenf
    3.55e10 Received               0.00 multiblock_getnftvalues
  8.68e8 Collective                0.00 nft_io_write
2244.41 Computational imbalar      0.00 MPI_Finalize

0.00  2626.32 (20.98%)  12516.67 | 0.00  2626.13 (99.99%)  2626.32 | 0.00  100.00  100.00
                                                                  | 0.10  10.26 +/- 6.01  25.27
```

Fig. 4. Scalasca analysis report explorer presentations of GemsFDTD trace experiments with 'ltrain' dataset for 256 processes on Cray XT4 (v1.1 above and v2.0 below) highlighting 2-fold improvement of point-to-point communication time in solver phase

Table 1. Selected performance metrics and statistics for Scalasca trace experiments for GemsFDTD versions with 'ltrain' dataset for 256 MPI processes on Cray XT4. (Maximum of any individual measured process where qualified.)

GemsFDTD		v1.1	v2.0
Reference execution time [s]		611.3	47.8
Measured execution time [s]		613.8	48.6
Initialization		561.1	13.1
– `multiblock_partition` time [s]	(max)	509.5	0.2
– `multiblock_distribute` time [s]	(max)	554.9	0.9
– broadcast time [s]	(max)	554.1	0.3
– *Late Broadcast* time [s]	(max)	552.0	0.2
– number of broadcasts	(max)	37465	4214
– bytes incoming by broadcast [kiB]	(max)	146.3	150.1
Time-stepping iterations		52.6	35.0
– Calculation time [s]	(max)	47.6	34.4
– Communication time [s]	(max)	33.9	25.3
– number of sends/receives	(max)	2900	2200
– bytes sent [MiB]	(max)	134.5	140.7
– *Late Sender* time [s]	(max)	33.7	25.2
– number of *Late Senders*	(max)	1455	50
Scalasca tracing			
– Trace total size [MiB]		410	96
– Trace collection time [s]		0.5	0.4
– Trace analysis time [s]		227.6	2.3
– event replay analysis [s]		2.2	0.5
– event timestamp correction [s]		223.8	1.3
– number of violations corrected		2529	762

clocks on Cray XT compute nodes is reduced by over 170-fold, since correcting timestamps of collective operations is particularly expensive. For the v2.0 version of GemsFDTD at this scale, Scalasca trace collection and automatic analysis require only 6% additional job runtime compared to the usual execution time.

4 Conclusions

The comprehensive analyses provided by the Scalasca toolset were instrumental in directing the developer's performance engineering of a much more efficient and highly scalable GemsFDTD code. Since these analyses show that there are still significant optimization opportunities at larger scales, further engineering improvements can be expected in future.

Acknowledgements. John Baron of SGI helped identify some bottlenecks in the original benchmark using the MPInside profiling tool, which led the developers to replace `MPI_Sendrecv` with non-blocking communication. EPCC service staff provided valuable assistance to use their Cray XT4 system for the performance measurements. Part of this work has been performed under the HPC-Europa2 project (number 228398) with the support of the European Commission – Capacities Area – Research Infrastructures. Part of this work has been supported by the computing resources of the PDC, KTH, and the Swedish National Infrastructure for Computing (SNIC).

References

1. Gedney, S.D.: An anisotropic perfectly matched layer-absorbing medium for the truncation of FDTD lattices. IEEE Transactions on Antennas and Propagation 44(12), 1630–1639 (1996)
2. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
3. Jülich Supercomputing Centre, Germany: Scalasca toolset for scalable performance analysis of large-scale parallel applications (2010), http://www.scalasca.org/
4. Martin, T., Pettersson, L.: Dispersion compensation for Huygens' sources and far-zone transformation in FDTD. IEEE Transactions on Antennas and Propagation 48(4), 494–501 (2000)
5. Parallel & Scientific Computing Institute (PSCI), Sweden: GEMS: General ElectroMagnetic Solvers project (2005), http://www.psci.kth.se/Programs/GEMS/
6. Standard Performance Evaluation Corporation (SPEC), USA: SPEC MPI2007 benchmark suite, version 2.0 (2010), http://www.spec.org/mpi2007/
7. Szebenyi, Z., Wylie, B.J.N., Wolf, F.: SCALASCA Parallel Performance Analyses of SPEC MPI2007 Applications. In: Kounev, S., Gorton, I., Sachs, K. (eds.) SIPEW 2008. LNCS, vol. 5119, pp. 99–123. Springer, Heidelberg (2008)
8. Taflove, A., Hagness, S.C.: Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd edn. Artech House (2005)
9. University of Edinburgh HPCx, UK: HECToR: United Kingdom National Supercomputing Service (2010), http://www.hector.ac.uk/service/
10. Wylie, B.J.N., Böhme, D., Frings, W., Geimer, M., Mohr, B., Szebenyi, Z., Becker, D., Hermanns, M.-A., Wolf, F.: Scalable performance analysis of large-scale parallel applications on Cray XT systems with Scalasca. In: Proc. 52nd CUG Meeting, Edinburgh, Scotland. Cray User Group (May 2010)
11. Wylie, B.J.N., Böhme, D., Mohr, B., Szebenyi, Z., Wolf, F.: Performance analysis of Sweep3D on Blue Gene/P with the Scalasca toolset. In: Proc. 24th Int'l Parallel & Distributed Processing Symposium, Workshop on Large-Scale Parallel Processing, IPDPS–LSPP, Atlanta, GA, USA. IEEE Computer Society (April 2010)

Scheduling Architecture–Supported Regions in Parallel Programs

Marek Tudruj[1,2] and Łukasz Maśko[1]

[1] Institute of Computer Science of the Polish Academy of Sciences
ul. Ordona 21, 01–237 Warsaw, Poland
`{tudruj,masko}@ipipan.waw.pl`
[2] Polish–Japanese Institute of Information Technology
ul. Koszykowa 86, 02–008 Warsaw, Poland

Abstract. Current multicore system technology enables implementation of particular program functions like library operations, special functions generation, optimized data search etc. using dedicated computing units to increase overall program performance. A parallel system can be equipped with a set of such units to speed up execution of applications, which use such functionality. To properly model and schedule programs using such functions running on a dedicated hardware, a proper program representation must be introduced. The paper presents special scheduling algorithm for programs represented as graphs, based on a modified ETF heuristics. The algorithm is meant for a modular architecture composed of many CMP modules interconnected by a global data communication network. The assumed architecture of dedicated CMP modules enables personalized fully synchronous program execution, which uses communication on the fly to strongly reduce inter–core communication overheads.

Keywords: CMP architectures; program execution control, program scheduling, data communication optimization.

1 Introduction

Cluster–based systems, supported by adequate communication solutions, can strongly increase execution efficiency of parallel programs and scalability. Development of efficient architectural paradigms and new parallel programming styles for cluster computing are currently strongly stimulated by recent advances in the multicore processor technology. This is strongly supported by the interconnect–centric style in the design of Chip Multiprocessors (CMP) systems [1,2]. With interconnect–centric design style in multiple CMP systems both intra–cluster and inter–cluster network structures should be targeted. Due to technology limitations large monolithic systems should be replaced by systems in which a single layer network fabric connecting all processor cores can be replaced by a hierarchical structure of many CMP modules connected by a central global network as in the RoadRunner system [3] (currently number 7 on the TOP 500 list [12]). With such modular approach to the design of CMP systems both inter–module communication and intra module network can be optimized to match different specific technical requirements at local and global communication levels.

Optimal design of program code for such dedicated units can impose particular requirements on the structure and behavior of the involved embedded parts of programs. Parallel or distributed programs built for such executive architectural assumptions can be considered as built of well defined regions dedicated to execution in specialized parallel units bound with the rest of the program code by scripts (glue code) which define the necessary program execution interfaces.

In this paper, we propose a scheduling algorithm for parallel programs which are to be executed in a modular CMP system built of different kinds of constituent CMP modules. The system has a hierarchical modular structure of many smaller CMP modules with efficient local data exchange, interconnected by a central global network. Architecture of the CMP modules allows dynamic creation of temporary shared memory processor (SMP) core clusters [6,7,8], which provide efficient means for direct and fast transmission of shared data between data caches of computing cores by means of reads on the fly. The internal structures of a CMP module dynamically adjusts to communication requirements of a fragment of an application program part, which is mapped to it for execution. The presented algorithm is based on identification of code regions inside the program graph, which can be efficiently executed by dedicated architecturally supported CMP modules.

Scheduling programs to such defined executive systems requires special program representation and algorithms. The macro data flow graph representation of programs is used for the approach proposed in this paper. The scheduling algorithm is based on a modified Earliest Task First (ETF) [10] heuristics with extended system of task priorities. As a result, a scheduled program task graph is produced in which computation and internal communication are assigned to resources inside CMP modules and global communication between tasks is assigned to links of the global network. Performance of the scheduled programs is verified by experiments accomplished using a simulator, which executes structured application program graphs and evaluates their parallel efficiency.

The paper is composed of 4 parts. In the first part, the proposed architecture of the executive system is described. In the second part, the graph representation of programs applied to design the programs for the assumed architecture is explained. In the third part, a scheduling algorithm which set the necessary structure in the application programs is described. The fourth part presents a scheduling example, which illustrates the proposed approach.

2 General System Architecture

The general structure of the proposed parallel multi–CMP system is presented in Fig. 1. Basic system elements are CMP modules interconnected by a global network. A single CMP module consists of a number of processor cores, each with its local L1 data cache and a number of shared L2 data cache banks interconnected through a local data communication network, Fig. 2. Dynamic core

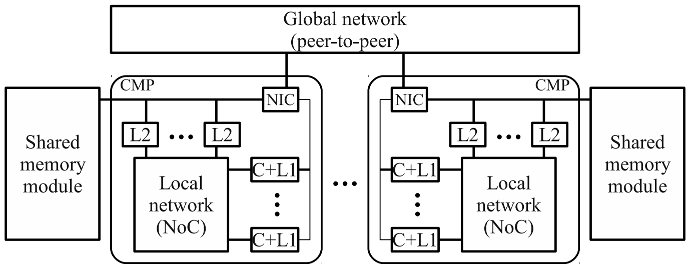


Fig. 1. General system structure

clusters can be created inside CMP modules using local data exchange networks (L2 buses) by connecting to them L1 banks. Dynamic core clusters enable very advanced group data communication involving a core or a cluster of cores, especially useful for shared data. Multiple parallel reads of data by many cores to their data caches can take place while a core writes data from its L1 cache to the cluster L2 memory (reads on the fly, similar to cache injection [5]). It is done by snooping L2 buses by L1 banks controllers and capturing the desired data. A new method for data exchange has been used among core clusters. It consists in dynamic switching of cores with their L1 data cache contents between L2 buses and provides a very fast way of data exchange between core clusters. It converts data transmissions through shared memory and/or some global network, into dynamic cluster reconfiguration with data transfers performed directly between L1 data caches. All L2 cache banks of a CMP module are connected to the local fragment of the distributed memory shared by all CMP modules in the system.

Tasks in programs are built according to a cache–controlled macro data–flow paradigm, so, all data have to be pre–fetched to core's L1 data cache before a task begins execution and L1 cache reloading is disabled. Current task results are sent to the L2 cache module only after task completes.

New features of the data cache organization consist in multi–ported structure of L1 data caches (multiple banks are used, which can be connected to many L2 buses). It enables parallel loading of arguments of subsequent numerical operations and many communications (or reads) on the fly performed at a time for a core. More details on the proposed architecture of the CMP modules and system execution control, however, for single level data caches can be found in [6,7,8].

The global interconnection network allows to exchange data between any shared memory fragments (modules) present in the system on request coming from a core. The global network can constitute a crossbar switch, a multistage network or a multibus structure. The global network thus provides standard data transfers between CMP modules, but at the cost of high latency. However, when data brought by global network are written to the CMP module shared memory, data transfers on the fly can be done to L2 data banks inside this module. This strongly accelerates data pre–fetching for computations performed by given CMP module cores and core clusters.

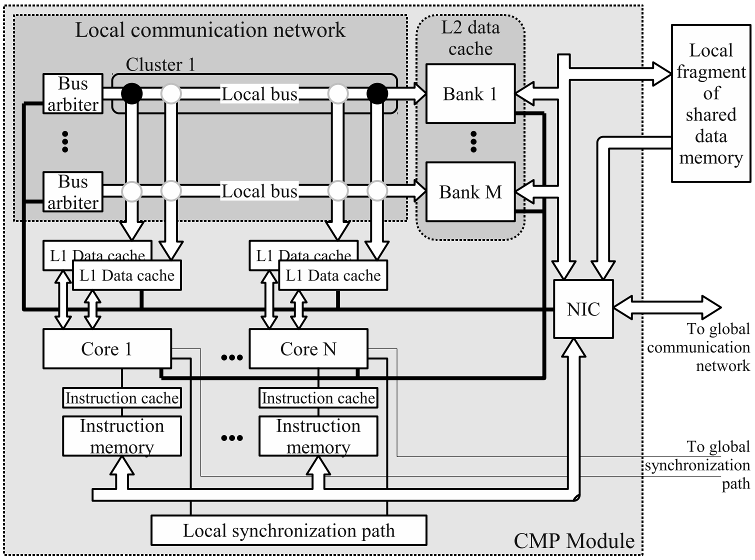


Fig. 2. The structure of a CMP module

The executive system may contain CMP modules as described above, and, additionally, a set of standard, general purpose CMP modules, which may be used for execution of parts of the graph, whose execution in architecturally supported modules is not profitable (because they are irregular or have small level of data sharing).

3 Graph Representation Used in Program Design

Programs for the proposed architecture are designed according to their macro data flow graph (MDFG) representation, which may be created automatically at compile time. Fig. 3a presents an simple MDFG consisting of 6 nodes (T0...T5) assigned to 5 cores (C1...C5) located in 2 CMP modules (CMP1, CMP2). To represent all actions which take place in a program executed in the proposed architecture we introduce an extended macro data flow graph (EMDFG) representation in which some new types of nodes are introduced in Fig. 3b: W2 – write of data from core's data cache L1 to L2, WM – write of some data from L2 to the shared memory of a CMP module, MMW – write from the shared memory of a CMP module to the memory of another CMP module, RqL1 – deposing read on the fly request in a BRC of a L1 bank, RL1 – read on the fly from the L1 bus, RqL2 – deposing a read on the fly request to a BRC of a L2 bank, RL2 – read on the fly from the L1–L2 bus to L2, B – a barrier, SW – switching a core's data cache bank from one cluster to another – i.e. from one L2 data cache bus to another. Results of task T0 are transmitted to L1 banks of cores C1 and C2 and L2 banks of cores C4 and C5 using data transfers on the fly. Task T0 writes its results from L1 to the L2 bank contained in the CMP1 module after fulfillment

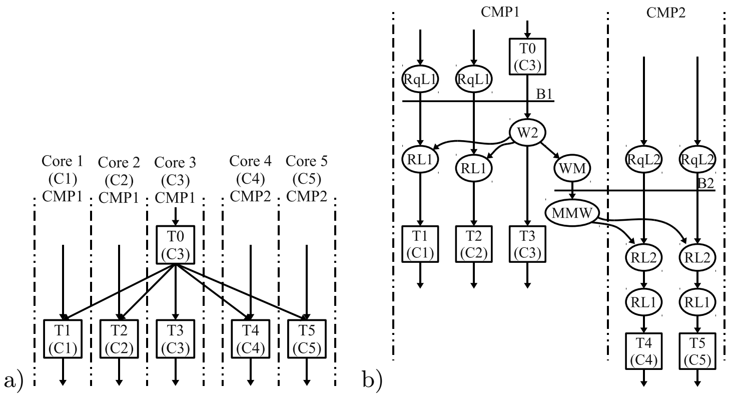


Fig. 3. A simple MDFG (a) and its extended notation (b)

of barrier B1 by read on the fly requests issued by tasks T1, T2. These tasks read on the fly data written by T0 to L2. T0 passes data to T3 in L1 cache. Some data produced by T0 are next sent back to the shared memory of CMP1 (WM). Next these data are sent to the CMP2 memory via the X–bar switch (MMW). When the data are written to the memory over the "L2–memory" bus in CMP2, they are read on the fly to two CMP2 L2 cache banks (RL2). Here barrier B2 works for cores belonging to two CMPs. After data are read to L2 banks they are pre–fetched to L1 caches of two CMP2 cores (RL1).

The read on the fly to L2 in CMP2, as in Fig. 3b, can be organized efficiently (without blocking the CMP1 NIC for a long time) if the barrier B2 can be fulfilled quickly from the side of CMP2. If however, the tasks in CMP2 become ready to receive data with a large delay, the pre–fetching to CMP2 L2 banks can be organized on the initiative of cores in CMP2. It is done by an independent read of data from memory to one of L2 banks and next a pre–fetch to L1 banks with the use of read on the fly for one L1 bank.

4 Regions in a Program Graph

Program graph can be logically divided into subgraphs of two forms: subgraphs, which show structural or functional features which qualify the respective program parts to be executed in system hardware modules with special architecture which can accelerate program execution, and subgraphs without promising features for special hardware acceleration.

In the case of our architecture, subgraphs of the first type are intended to be implemented inside CMP modules with architecture described in previous sections, called the architectural CMP modules – ACMPs. Consequently, program subgraphs assigned to ACMPs will be called Architecturally–Supported Regions (ASR). For the requirements of our architecture, such ASRs should show phase–like regular structure or high level of data sharing, which makes data transfers on the fly and dynamic core switching profitable.

The second kind of subgraphs contains nodes which constitute "a glue", which fills in the gaps between ASRs and can be executed using a set of general purpose hardware modules in a system with standard architecture. Such standard CMPs will be called general–purpose CMPs – GCMPs (in our case, standard multicore CMP modules). Regarding features of our architecture, the subgraphs for the standard CMP modules have either very irregular (un–phased) structure or they are very loosely data related with low level of data sharing, in which case using core switching and data transfers on the fly will not give performance profits.

Formally, a program is described by a macro data flow graph $G = (V, E)$, where V, E are the set of nodes and edges of the graph, respectively. G can be divided into two disjoint sets of subgraphs V_r and V_a, such that $V = V_s \cup V_a$, $V_s \cap V_a = \emptyset$ where V_s contains standard nodes, which constitute a glue, and V_a contains nodes belonging to ASRs. The division may be determined automatically by a compiler, or it can be done manually by a programmer. We assume, that incoming data communications to a given ASR may happen only at its beginning (before its computations are started) and outgoing data transfers are possible after the region execution is finished. No additional external transfers are allowed during execution of a region. Under such restrictions, ASRs can correspond to subroutines, which may be replaced by a single meta node in the program macro data flow graph. We will call the meta–nodes in such transformed program graph the "architectural nodes".

We assume, that each ASR program subgraph is optimally mapped to cores in an architectural CMP by separate application of the special scheduling algorithm such as described in [9,10]. We assume that only one ASR subgraph may be executed on a given architectural CMP at a time and that all the cores in this module are potentially enabled for the ASR execution. We also assume, that execution of any ASR may not be interrupted.

5 Scheduling of Programs with Architecturally Supported Regions

The assumed architecture shows heterogeneity of applied computing resources. The algorithm schedules standard nodes to GCMPs and architectural nodes to ACMPs. It aims at minimal program execution time by equal loads of all available resources. Regarding the selection of nodes for mapping to system resources, the proposed scheduling algorithm is based on list scheduling with the ETF heuristics [10], but the selection is extended by classification of nodes in the program graph to enable gradual load balancing of both general purpose and architectural CMPs. The classification consists in prioritization of nodes in such way, that at each point of the scheduling algorithm, only such glue nodes are considered, whose results are required for execution of the topologically nearest ASRs in the graph. Other standard nodes are scheduled within the remaining available processing resources preventing from delaying execution of other nodes.

Assignment of the 1^{st} level priority (denoted as $pr_1(v)$ for each node v) divides a set of architectural nodes into layers used to schedule the program nodes in the

Algorithm 1. Definition of 2^{nd} level priorities for a set U of nodes

```
1: Let $p = 0$ be the first value of priority for this set.
2: while U is not empty do
3:    Determine $X_u$ sets for all nodes from $U$.
4:    Determine a subset $V$ of architectural nodes from $U$ chosen to be assigned priority $p$ and $X_V$ as a sum of sets $X_v$ for all $v \in V$, using Algorithm 2.
5:    Assign $pr_2(u) = p$ for all nodes $u$ from $V \cup X_V$.
6:    Remove the architectural nodes, which belong to $V$ from the set $U$.
7:    Let $p = p + 1$
8: end while
```

breadth–first–way. Each such layer contains a subset of nodes, which are pair–wise independent, i.e. there is no data dependency between any two of them. The scheduling algorithm tries to schedule such architectural nodes layer–by–layer. The layers are based on topological properties of a graph and are created using graph paths analysis. To compute 1^{st} level priorities, an "architectural task graph" $G_a = (V_a, E')$ is defined. In this graph, nodes correspond to architectural nodes in an initial graph G. For two nodes $u, v \in V_a$, an edge $u \rightarrow v \in E'$ exists in G_a, if there is a directed path between these two nodes in original graph G containing only standard nodes. For each $u \in V_a$, the priority of a node u is equal to its depth in graph G_a (the number of nodes on the longest path leading to u from one of the nodes which have no predecessors in G_a). Priorities for standard nodes depend on priorities of architectural nodes. For each $v \in V_s$, we determine a set $X \subset V_a$ of nodes such that there exists a path from node v to each of these nodes. If X is not empty, the priority of a node v is equal to minimal priority over the nodes from X and is equal to $\max_{u \in V_a}(pr_1(u)) + 1$ otherwise. A set of nodes with the same 1^{st} level priority constitutes a layer of nodes.

The 2^{nd} level priority (denoted as $pr_2(v)$ for each node v) aims at division of nodes in layers determined by the 1^{st} level priority, into subsets in such way,

Algorithm 2. Selection of architectural nodes to be assigned the same priority in a loop in Algorithm 1

```
1: Let $V = \emptyset$ and $X_V = \emptyset$
2: while $|V|$ is smaller than the number of resources for execution of architectural nodes do
3:    if $V$ is empty then
4:       for all tasks $u$ from $U$ do
5:          Schedule a subgraph $X_u$ on available resources dedicated to execution of the standard nodes, using an ETF–based list scheduling.
6:       end for
7:       Select such node $u$ from $U$, for which its $X_u$ set gives the shortest schedule in the previous step and uses the smallest number of resources.
8:    else
9:       Select node $u \in U$ such, that $X_u \cap X_V$ is the biggest.
10:   end if
11:   Let $V = V \cup \{u\}$ and $X_V = X_V \cup X_u$
12: end while
```

that we can obtain equal, high load of all CMPs. Each subset of nodes contains no more architectural nodes, than the number of ACMPs in the system. We determine 2^{nd} level priorities, if there is a set U of architectural nodes, which have the same 1^{st} level priorities. For each node $u \in U$ we determine X_u as a set of standard nodes $v \in V$ such that there exists a directed path from v to u, $pr_1(v) = pr_1(u)$, and which have no 2^{nd} level priority assigned yet. The nodes in such sets are used to put architectural nodes in order. Priorities for nodes from the U–sets are assigned using the algorithm shown as Algorithm 1.

The final priority $pr(v)$ is defined as follows: for two nodes u and v (both must be either architectural or standard), $pr(u) < pr(v) \iff pr_1(u) < pr_1(v) \vee (pr_1(u) = pr_1(v) \wedge pr_2(u) < pr_2(v))$.

Algorithm 3. List scheduling algorithm with modified ETF heuristics

```
1: {Input: a program graph $G = (V, E)$}
2: Determine architectural task graph $G'$ based on graph $G$ and, based on it, compute
   priorities $pr_1(v)$ for all nodes $v \in G$.
3: **for** all nodes $v \in G$ determine $pr_2(v)$ using Algorithm 1 **do**
4:    Let $P$ be the set of ready nodes from graph $G$. Initially, insert all the nodes
      without predecessors from $G$ into $P$.
5:    **while** $P$ is not empty **do**
6:       Let $pr_{min} = \min_{u \in P}(pr(u))$
7:       **for** all nodes $u \in P$ such that $pr(u) = pr_{min}$ and all cores $p$ **do**
8:          Check the earliest possible execution start time of node $u$ on core $p$ and
            select such pair $(u, p)$, for which the selected node's execution is the earliest
            on the selected core.
9:          Schedule the selected node for execution on the chosen core.
10:         Remove this node from $P$.
11:         **for** all descendants of the selected node **do**
12:            **if** all their predecessors have already been scheduled **then**
13:               insert them into $P$
14:            **end if**
15:         **end for**
16:      **end for**
17:      **for** all nodes $v \in P$ with higher priorities **do**
18:         Check, if any such node may be executed on any core in such way, that its
            execution ends before execution of the node selected in the previous step.
19:         **if** there exists such node $v$ **then**
20:            Schedule $v$ for execution on the selected core.
21:            Remove $v$ from $P$
22:            **for** all descendants of node $v$ **do**
23:               **if** all their predecessors have already been scheduled **then**
24:                  insert them into $P$
25:               **end if**
26:            **end for**
27:         **end if**
28:      **end for**
29:   **end while**
30: **end for**
```

The final scheduling algorithm (Algorithm 3) is based on list scheduling. Standard (glue) nodes are scheduled for execution on general purpose cores, architectural nodes are scheduled for execution on architectural CMPs. Each time, when a node is to be selected, first, the nodes with the lowest priorities are examined and scheduled. Other nodes are scheduled only when their execution doesn't interfere with execution of those with currently the lowest priority. Such selection of nodes leads to a situation, where architectural nodes may be executed as soon as possible, using cores from architectural CMPs. In the same time, general purpose cores may be used to execute standard nodes, whose execution doesn't depend on architectural nodes and which are required for further computations.

6 A Scheduling Example

As an example, scheduling of a graph of Strassen parallel square matrix multiplication algorithm is presented. Fig. 4 presents the graph of this algorithm unrolled at 1^{st} recursion level. Each multiplication node Mi is further parallelized using standard multiplication method with decomposition of matrices into quarters. Each such node is a parallel task, which needs 8 cores for execution. The parallel multiplication nodes are additionally structured to use core switching and reads on the fly, so they might be efficiently executed on ACMPs as architectural nodes. The addition nodes in the Strassen algorithm (nodes labeled ”A i,1–4”) are also parallelized with decomposition into quarters to preserve the same data granularity at each computation level, but due to their simple internal structure, they are treated as ”glue nodes”. All the experiments were prepared with the software simulator of the proposed architecture.

Table 1 shows parallel speedup of the scheduled Strassen algorithm graph for 3 different recursion levels, different matrix sizes and different system configurations, comparing to standard serial multiplication. Two versions of the algorithm were studied: ”reg” – regions scheduling without priorities and ”reg+pri” – regions scheduling with priorities. The numbers of ACMPs and GCMPs (8 and

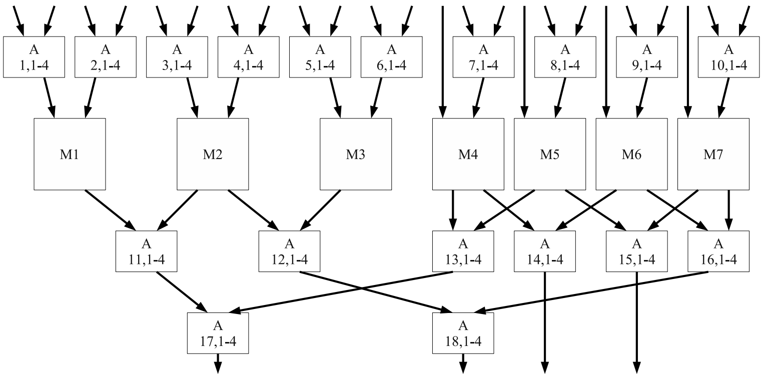


Fig. 4. The general view of the exemplary program graph

Table 1. Parallel speedup of the scheduled Strassen algorithm graph, comparing standard serial multiplication

Rec. level	System config.		Algorithm	Matrix size				
	ACMPs	GCMPs		32	64	128	256	512
1	7	2	reg, reg+pri	8.15	14.05	22.76	33.42	43.85
		4	reg, reg+pri	9.56	16.18	25.54	36.36	46.31
		8	reg, reg+pri	9.64	16.31	25.70	36.52	46.44
2	49	2	reg	6.70	13.11	27.52	53.24	96.21
			reg+pri	7.41	14.28	28.03	55.52	113.74
		16	reg, reg+pri	23.43	43.32	78.61	135.33	213.47
		32	reg, reg+pri	27.51	50.52	90.54	152.83	234.78
		64	reg, reg+pri	28.22	51.74	92.54	155.71	238.18
3	343	2	reg	3.22	6.22	12.85	24.92	51.43
			reg+pri	3.88	7.47	14.67	29.06	57.83
		16	reg	21.69	42.11	81.38	162.49	315.29
			reg+pri	23.90	46.01	90.41	178.85	355.98
		128	reg, reg+pri	77.64	147.08	278.98	518.01	916.26
		256	reg, reg+pri	82.74	156.54	296.30	548.04	963.16
		512	reg, reg+pri	84.32	159.48	301.65	557.27	977.45

4 cores, respectively) were so selected as to enable full parallelization of multiplication in ACMPs and to check the influence of the number of GCMPs on the overall program performance due to potential serialization of the execution of the "glue". Cores in ACMPs and GCMPs are assumed to be equally fast in terms of computation. Computations were assumed to be 3 times faster than local communication on the L1–L2 bus. Global communication was assumed to be 4 times slower than the local one. Execution at higher recursion level increased program parallelization level and also the "glue" part of the graph increased. It required more ACMPs and GCMPs to obtain maximal speedup. At the 1^{st} recursion level the speedup saturates already for 4 GCMPs. At the 2^{nd} and 3^{rd} recursion levels the speedup saturates above 32 and 256 GCMPs, respectively. The influence of the special priorities of tasks is visible when the execution of the "glue" is "squeezed" to a small number of GCMPs. The speedup strongly depends on the matrix size since it decides on parallelization grain. For considered matrix sizes 32–512, the parallelization grain is determined by the size of the matrix parts used for elementary serial multiplication after recursive decomposition at a given recursion level – for example at the 3^{rd} recursion level, the grain is the smallest: it is 2, 4, 8, 16, 32, respectively.

7 Conclusions

Special parallel program scheduling algorithms for modular CMP systems have been proposed in the paper. The target system is composed of two kinds of CMP modules interconnected by a global network: architectural CMP modules compatible with the described architecture, dedicated for execution of these parts of the graph (Architecturally Supported Regions), which can profit from execution based on core switching and data transfers on the fly, and general purpose shared memory CMP modules, built of standard cores with classic interconnections, used for execution of other (glue) nodes. Special graph program representation, required for designing programs for such system, has been proposed. Because of heterogeneity of the system, the proposed scheduling algorithm extends the standard list scheduling method with ETF heuristics by application of multi–level priorities of graph nodes, which enables efficient use of computing resources to achieve load balancing at the level of both general and architectural CMP modules. To illustrate the proposed scheduling algorithm, execution of program graphs of parallel Strassen matrix multiplication with data decomposition into quarters have been analyzed. To obtain maximal parallel speedup, the number of applied ACMPs and GCMPs should match the requirements of the program graph. Otherwise performance is reduced. The extension of the algorithm by special task priorities improves the quality of scheduling in the assumed architecture when execution of the "glue" is strongly serialized or the number of used ACMPs does not cover the width of the program graphs.

References

1. Owens, J.D., et al.: Research Challenges for On–Chip Interconnection Networks. IEEE MICRO, 96–108 (September-October 2007)
2. Kundu, S., Peh, L.S.: On–Chip Interconnects for Multicores. IEEE MICRO, 3–5 (September-October 2007)
3. Koch, K.: Roadrunner System Overview, Los Alamos National Laboratory, `http://www.lanl.gov/orgs/hpc/roadrunner/rrinfo/RR%20webPDFs/RRSystemOversm.pdf`
4. Lepère, R., Trystram, D., Woeginger, G.J.: Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 146–157. Springer, Heidelberg (2001)
5. Milenkovic, A., Milutinovic, V.: Cache Injection: A Novel Technique for Tolerating Memory Latency in Bus-Based SMPs. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 558–566. Springer, Heidelberg (2000)
6. Tudruj, M., Maśko, Ł.: Dynamic SMP Clusters with Communication on the Fly in NoC Technology for Very Fine Grain Computations. In: 3rd Int. Symp. on Parallel and Distributed Computing, ISPDC 2004, Cork, pp. 97–104 (July 2004)
7. Tudruj, M., Maśko, Ł.: Towards Massively Parallel Computations Based on Dynamic SMP Clusters wih Communication on the Fly. In: Proceedings of the 4th International Symposium on Parallel and Distributed Computing, ISPDC 2005, Lille, France, July 4-6, pp. 155–162. IEEE CS Press (2005)
8. Tudruj, M., Maśko, Ł.: Fast Matrix Multiplication in Dynamic SMP Clusters with Communication on the Fly in Systems on Chip Technology. In: Proc. of PARELEC 2006, pp. 77–82 (September 2006)

9. Masko, Ł., Dutot, P.–F., Mounié, G., Trystram, D., Tudruj, M.: Scheduling Moldable Tasks for Dynamic SMP Clusters in SoC Technology. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 879–887. Springer, Heidelberg (2006)
10. Masko, Ł., Tudruj, M.: Task Scheduling for SoC–Based Dynamic SMP Clusters with Communication on the Fly. In: 7th Int. Symp. on Parallel and Distributed Computing, ISPDC 2008, pp. 99–106 (2008)
11. Hwang, J.–J., Chow, Y.–C., Anger, F.D., Lee, C.–Y.: Scheduling precedence graphs in systems with interprocessor communication times. SIAM Journal on Computing 18(2) (1989)
12. http://www.top500.org/

Author Index