Pekka Manninen
Per Öster (Eds.)

LNCS 7782

Applied Parallel and Scientific Computing

11th International Conference, PARA 2012
Helsinki, Finland, June 2012
Revised Selected Papers

Springer

Pekka Manninen Per Öster (Eds.)

Applied Parallel and Scientific Computing

11th International Conference, PARA 2012
Helsinki, Finland, June 10-13, 2012
Revised Selected Papers

Springer

Volume Editors

Pekka Manninen
CSC - IT Center for Science Ltd
02101 Espoo, Finland
E-mail: pekka.manninen@csc.fi

Per Öster
CSC - IT Center for Science Ltd
02101 Espoo, Finland
E-mail: per.oster@csc.fi

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-36802-8 e-ISBN 978-3-642-36803-5
DOI 10.1007/978-3-642-36803-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013932191

CR Subject Classification (1998): G.1.0, G.1.3-6, F.2.1-2, C.1.4, F.1.2, C.4, G.4

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

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

Printed on acid-free paper

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

Preface

These proceedings contain a set of papers presented at the PARA 2012 conference held during June 10–13, 2012, in Helsinki, Finland. It was the 11th in the PARA conference series. The general theme of the PARA conference series is the "State of the Art in Scientific and Parallel Computing," and a special theme of the PARA 2012 conference was "Toward Exascale."

June in Helsinki means long days. The sun approaches the horizon not before 11 pm, which gives plenty of time for long and interesting discussions. For the PARA 2012 participants, many of the discussions were likely around this year's theme, exascale computing. Exascale supercomputers are predicted to emerge in 6 to 10 years' time. The path toward this regime of computing has been pretty clear for more than a decade and implies dramatically increased parallelism in all levels of the computing systems. We already know how to handle massive parallelism, but now we are talking about excessively massive parallelism: two or three orders of magnitude above today's leading petaflop/s systems. The real challenge is not just to build a computer system of theoretical exascale performance, but rather to be able to devise real-world scientific applications that harness the exaflop/s computing power. This fact was the background for the choice of the theme for the conference: Addressing the challenges due to exploiting the ever-increasing parallelism and to the changing balance between the computers' subsystems performance requires novel, more suitable algorithms, programming models, performance optimization techniques, and numerical libraries.

The conference itself consisted of a number of selected contributed talks divided into topical parallel sessions, and five thematic minisymposia. These were accompanied by keynote talks and interactive sessions. In these proceedings, we have, based on a peer-review process, selected 35 technical full articles for publication, categorized as follows:

- Advances in HPC Applications
- Parallel Algorithms
- Performance Analysis and Optimization
- Applications of Parallel Computing in Industry and Engineering
- HPC Interval Methods

In addition to these, three of the topical minisymposia are described by a corresponding overview article on the minisymposium topic. In order to cover the state of the art of the field, we have included at the end of the proceedings a set of extended abstracts that describe some of the conference talks not elaborated by a full article. These extended abstracts have not been peer-reviewed.

We would like to thank everyone who contributed to the PARA 2012 conference: keynote speakers Jack Dongarra, Björn Engquist, and Mark Parsons; Minisymposium Chairs; presenters of contributed talks; tutorial instructors; all attendees; and everyone involved in practical arrangements. Especially all the authors of these proceedings as well as the numerous reviewers are gratefully acknowledged.

December 2012

P. Manninen
P. Öster

Organization

Steering Committee

Jack Dongarra, Honorary Chair	University of Tennessee, USA
Jerzy Wasniewski, Chair	Technical University of Denmark
Petter Bjørstad	University of Bergen, Norway
Anne C. Elster	Norwegian University of Science and Technology, Norway
Björn Engquist	University of Texas at Austin, USA
Hjálmtýr Hafsteinsson	University of Iceland
Kristján Jónasson	University of Iceland
Kimmo Koski	CSC – IT Center for Science Ltd, Finland
Bo Kågström	University of Umeå, Sweden
Kaj Madsen	Technical University of Denmark
Per Öster	CSC – IT Center for Science Ltd, Finland

Organizing Committee

Per Öster, Chair	CSC – IT Center for Science Ltd, Finland
Pekka Manninen, Co-chair	CSC – IT Center for Science Ltd, Finland
Juha Fagerholm	CSC – IT Center for Science Ltd, Finland

Table of Contents

Part I: Overview Articles of Topical Minisymposia

Computational Physics on Graphics Processing Units 3
Ari Harju, Topi Siro, Filippo Federici Canova, Samuli Hakala, and Teemu Rantalaiho

Preparing Scientific Application Software for Exascale Computing 27
J.A. Åström, A. Carter, J. Hetherington, K. Ioakimidis, E. Lindahl, G. Mozdzynski, R.W. Nash, P. Schlatter, A. Signell, and J. Westerholm

PRACE DECI (Distributed European Computing Initiative) Minisymposium 43
Chris Johnson, Adam Carter, Iain Bethune, Kevin Statford, Mikko Alava, Vitor Cardoso, Muhammad Asif, Bernhard S.A. Schuberth, and Tobias Weinzierl

Part II: Advances in HPC Applications

Parallel Electronic Structure Calculations Using Multiple Graphics Processing Units (GPUs) 63
Samuli Hakala, Ville Havu, Jussi Enkovaara, and Risto Nieminen

An Environment for Service Composition, Execution and Resource Allocation 77
Jan Kwiatkowski, Krzysztof Juszczyszyn, and Grzegorz Kolaczek

Multicore and Accelerator Development for a Leadership-Class Stellar Astrophysics Code 92
O.E. Bronson Messer, J. Austin Harris, Suzanne Parete-Koon, and Merek A. Chertkow

Steering and In-situ Visualization for Simulation of Seismic Wave Propagation on Graphics Cards 107
David Michéa, Joachim Pouderoux, Fabrice Dupros, and Hideo Aochi

PCJ - New Approach for Parallel Computations in Java 115
Marek Nowicki and Piotr Bała

Porting Production Level Quantum Chromodynamics Code to Graphics Processing Units – A Case Study 126
Teemu Rantalaiho

Time Propagation of Many-Body Quantum States on Graphics Processing Units 141
Topi Siro and Ari Harju

Parallel Numerical Simulation of Seismic Waves Propagation with Intel Math Kernel Library 153
Mikhail A. Belonosov, Clement Kostov, Galina V. Reshetova, Sergey A. Soloviev, and Vladimir A. Tcheverda

Part III: Parallel Algorithms

Blocked Schur Algorithms for Computing the Matrix Square Root 171
Edvin Deadman, Nicholas J. Higham, and Rui Ralha

Distributed Evolutionary Computing System Based on Web Browsers with JavaScript 183
Jerzy Duda and Wojciech Dłubacz

Use of Direct Solvers in TFETI Massively Parallel Implementation 192
Václav Hapla, David Horák, and Michal Merta

Parallel Implementation of the Sherman-Morrison Matrix Inverse Algorithm 206
Xin He, Marcus Holm, and Maya Neytcheva

Multi-threaded Nested Filtering Factorization Preconditioner 220
Pawan Kumar, Karl Meerbergen, and Dirk Roose

An Approach of the QR Factorization for Tall-and-Skinny Matrices on Multicore Platforms 235
Sergey V. Kuznetsov

Approximate Incomplete Cyclic Reduction for Systems Which Are Tridiagonal and Strictly Diagonally Dominant by Rows 250
Carl Christian Kjelgaard Mikkelsen and Bo Kågström

Fast Poisson Solvers for Graphics Processing Units 265
Mirko Myllykoski, Tuomo Rossi, and Jari Toivanen

Parallel Feature Selection for Regularized Least-Squares 280
Sebastian Okser, Antti Airola, Tero Aittokallio, Tapio Salakoski, and Tapio Pahikkala

Part IV: Performance Analysis and Optimization

Topology Aware Process Mapping 297
Sebastian von Alfthan, Ilja Honkonen, and Minna Palmroth

Numprof: A Performance Analysis Framework for Numerical Libraries 309
Olli-Pekka Lehto

Multi-core Scalability Measurements: Issues and Solutions 319
Mattias Linde

AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications 328
Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, and François Bodin

Vectorized Higher Order Finite Difference Kernels 343
Gerhard Zumbusch

Part V: Applications of Parallel Computing in Industry and Engineering

Preconditioning for Large Scale Micro Finite Element Analyses of 3D Poroelasticity 361
Peter Arbenz and Erhan Turan

Parallel Solvers for Numerical Upscaling 375
R. Blaheta, O. Jakl, J. Starý, and Erhan Turan

Parallel Numerical Algorithms for Simulation of Multidimensional Ill-Posed Nonlinear Diffusion-Advection-Reaction Models 387
Raimondas Čiegis and Andrej Bugajev

Efficient Setup of Aggregation AMG for CFD on GPUs 398
Maximilian Emans and Manfred Liebmann

Accelerating 2D-to-3D Video Conversion on Multi-core Systems 410
Jiangbin Feng, Yurong Chen, Eric Li, Yangzhou Du, and Yimin Zhang

Solution of Multi-objective Competitive Facility Location Problems Using Parallel NSGA-II on Large Scale Computing Systems 422
Algirdas Lančinskas and Julius Žilinskas

Simulation Software for Flow of Fluid with Suspended Point Particles in Complex Domains: Application to Matrix Diffusion 434
Jukka I. Toivanen, Keijo Mattila, Jari Hyväluoma, Pekka Kekäläinen, Tuomas Puurtinen, and Jussi Timonen

Part VI: HPC Interval Methods

Introduction to the "HPC Interval Methods" Minisymposium 449
Bartłomiej Jacek Kubica

Parallel Interval Newton Method on CUDA 454
Philip-Daniel Beck and Marco Nehmeier

Heterogeneous Multi-agent Evolutionary System for Solving Parametric Interval Linear Systems 465
Jerzy Duda and Iwona Skalna

Interval Finite Difference Method for Solving the One-Dimensional Heat Conduction Problem with Heat Sources 473
Malgorzata A. Jankowska and Grazyna Sypniewska-Kaminska

Interval Arithmetic and Automatic Differentiation on GPU Using OpenCL 489
Grzegorz Kozikowski and Bartłomiej Jacek Kubica

Tuning the Interval Algorithm for Seeking Pareto Sets of Multi-criteria Problems 504
Bartłomiej Jacek Kubica and Adam Woźniak

A Central-Backward Difference Interval Method for Solving the Wave Equation 518
Andrzej Marciniak and Barbara Szyszka

Part VII: Collection of Extended Abstracts

Using State-of-the-Art Sparse Matrix Optimizations for Accelerating the Performance of Multiphysics Simulations 531
Vasileios Karakasis, Georgios Goumas, Konstantinos Nikas, Nectarios Koziris, Juha Ruokolainen, and Peter Råback

Schedule Optimisation for Interactive Parallel Structure Simulations 536
Jovana Knežević, Ralf-Peter Mundani, and Ernst Rank

A Shared Memory Parallel Implementation of the IRKA Algorithm for $\mathcal{H}_2$ Model Order Reduction (Extended Abstract) 541
Martin Köhler and Jens Saak

Parallel Block Preconditioning by Using the Solver of Elmer 545
Mika Malinen, Juha Ruokolainen, Peter Råback, Jonas Thies, and Thomas Zwinger

Distributed High-Performance Parallel Mesh Generation with ViennaMesh 548
Jorge Rodríguez, Josef Weinbub, Dieter Pahr, Karl Rupp, and Siegfried Selberherr

Computational Chemistry Studies of LIGNOLs 553
Thomas Sandberg, Patrik Eklund, and Matti Hotokka

Investigation of Load Balancing Scalability in Space Plasma Simulations 558
Ata Turk, Gunduz V. Demirci, Cevdet Aykanat, Sebastian von Alfthan, and Ilja Honkonen

A Lightweight Task Graph Scheduler for Distributed High-Performance Scientific Computing 563
Josef Weinbub, Karl Rupp, and Siegfried Selberherr

Author Index 567

Part I

Overview Articles of Topical Minisymposia

Computational Physics on Graphics Processing Units

Ari Harju[1,2], Topi Siro[1,2], Filippo Federici Canova[3], Samuli Hakala[1], and Teemu Rantalaiho[2,4]

[1] COMP Centre of Excellence, Department of Applied Physics, Aalto University School of Science, Helsinki, Finland
[2] Helsinki Institute of Physics, Helsinki, Finland
[3] Department of Physics, Tampere University of Technology, Tampere, Finland
[4] Department of Physics, University of Helsinki, Helsinki, Finland

Abstract. The use of graphics processing units for scientific computations is an emerging strategy that can significantly speed up various algorithms. In this review, we discuss advances made in the field of computational physics, focusing on classical molecular dynamics and quantum simulations for electronic structure calculations using the density functional theory, wave function techniques and quantum field theory.

Keywords: graphics processing units, computational physics.

1 Introduction

The graphics processing unit (GPU) has been an essential part of personal computers for decades. Their role became much more important in the 90s when the era of 3D graphics in gaming started. One of the hallmarks of this is the violent first-person shooting game DOOM by the id Software company, released in 1993. Wandering around the halls of slaughter, it was hard to imagine these games leading to any respectable science. However, twenty years after the release of DOOM, the gaming industry of today is enormous, and the continuous need for more realistic visualizations has led to a situation where modern GPUs have tremendous computational power. In terms of theoretical peak performance, they have far surpassed the central processing units (CPU).

The games started to have real 3D models and hardware acceleration in the mid 90s, but an important turning point for the scientific use of GPUs for computing was around the first years of this millennium [1], when the widespread programmability of GPUs was introduced. Combined with the continued increase in computational power as shown in Fig. 1, the GPUs are nowadays a serious platform for general purpose computing. Also, the memory bandwidth in GPUs is very impressive. The three main vendors for GPUs, Intel, NVIDIA, and ATI/AMD, are all actively developing computing on GPUs. At the moment, none of the technologies listed above dominate the field, but NVIDIA with its CUDA programming environment is perhaps the current market leader.

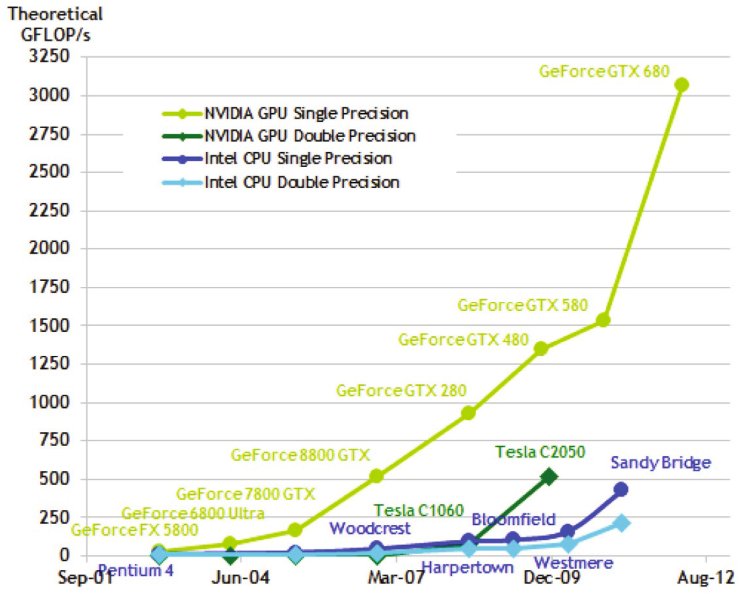


Fig. 1. Floating point operations (FLOPS) per second for GPUs and CPUs from NVIDIA and Intel Corporations, figure taken from [2]. The processing power of the currently best GPU hardware by the AMD Corporation is comparable to NVIDIA at around 2600 GFLOPS/s.

1.1 The GPU as a Computational Platform

At this point, we have hopefully convinced the reader that GPUs feature a powerful architecture also for general computing, but what makes GPUs different from the current multi-core CPUs? To understand this, we can start with traditional graphics processing, where hardware vendors have tried to maximize the speed at which the pixels on the screen are calculated. These pixels are independent primitives that can be processed in parallel, and the number of pixels on computer displays has increased over the years from the original DOOM resolution of 320×200, corresponding to 64000 pixels, to millions. The most efficient way to process these primitives is to have a very large number of arithmetic logical units (ALUs) that are able to perform a high number of operations for each video frame. The processing is very data-parallel, and one can view this as performing the same arithmetic operation in parallel for each primitive. Furthermore, as the operation is the same for each primitive, there is no need for very sophisticated flow control in the GPU and more transistors can be used for arithmetics, resulting in an enormously efficient hardware for performing parallel computing that can be classified as “single instruction, multiple data” (SIMD).

Now, for general computing on the GPU, the primitives are no longer the pixels on the video stream, but can range from matrix elements in linear algebra to physics related cases where the primitives can be particle coordinates in classical molecular dynamics or quantum field values. Traditional graphics processing teaches us that the computation would be efficient when we have a situation where the same calculation needs to be performed for each member of a large data set. It is clear that not all problems or algorithms have this structure, but there are luckily many cases where this applies, and the list of successful examples is long.

However, there are also limitations on GPU computing. First of all, when porting a CPU solution of a given problem to the GPU, one might need to change the algorithm to suit the SIMD approach. Secondly, the communication from the host part of the computer to the GPU part is limited by the speed of the PCIe bus coupling the GPU and the host. In practice, this means that one needs to perform a serious amount of computing on the GPU between the data transfers before the GPU can actually speed up the overall computation. Of course, there are also cases where the computation as a whole is done on GPU, but these cases suffer from the somewhat slower serial processing speed of the GPU.

Additional challenges in GPU computing include the often substantial programming effort to get a working and optimized code. While writing efficient GPU code has become easier due to libraries and programmer friendly hardware features, it still requires some specialized thinking. For example, the programmer has to be familiar with the different kinds of memory on the GPU to know how and when to use them. Further, things like occupancy of the multiprocessors (essentially, how full the GPU is) and memory access patterns of the threads are something one has to consider to reach optimal performance. Fortunately, each generation of GPUs has alleviated the trouble of utilizing their full potential. For example, a badly aligned memory access in the first CUDA capable GPUs from NVIDIA could cripple the performance by drastically reducing the memory bandwidth, while in the Fermi generation GPUs the requirements for memory access coalescing are much more forgiving.

2 Molecular Dynamics

Particle dynamics simulation, often simply called Molecular dynamics (MD), refers to the type of simulation where the behaviour of a complex system is calculated by integrating the equation of motion of its components within a given model, and its goal is to observe how some ensemble-averaged properties of the system originate from the detailed configuration of its constituent particles (Fig. 2).

In its classical formulation, the dynamics of a system of particles is described by their Newtonian equations:

$$m_i \frac{d^2 \boldsymbol{x}_i}{dt^2} = \sum_j \boldsymbol{F}_{ij} \qquad (1)$$

where m_i is the particle's mass, $\boldsymbol{x}_i$ its position, and $\boldsymbol{F}_{ij}$ is the interaction between the i-th and j-th particles as provided by the model chosen for the system under

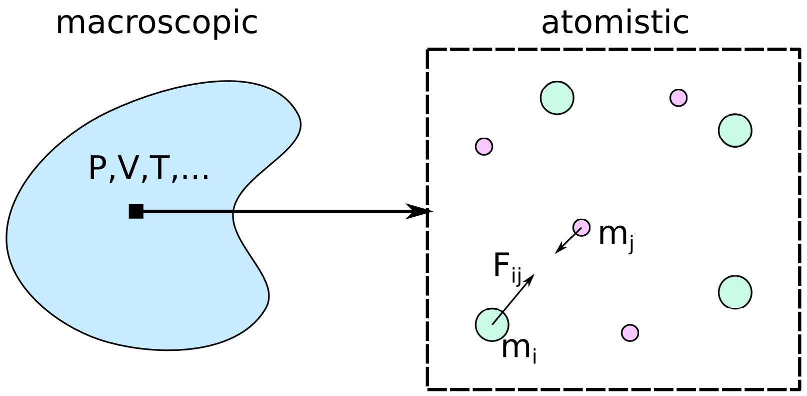


Fig. 2. Schematic presentation of the atomistic model of a macroscopic system

study. These second order differential equations are then discretised in the time domain, and integrated step by step until a convergence criterion is satisfied.

The principles behind MD are so simple and general that since its first appearance in the 70s, it has been applied to a wide range of systems, at very different scales. For example, MD is the dominant theoretical tool of investigation in the field of biophysics, where structural changes in proteins [3,4,5,6] and lipid bilayers [7,8] interacting with drugs can be studied, ultimately providing a better understanding of drug delivery mechanisms.

At larger scales, one of the most famous examples is known as the *Millenium Simulation*, where the dynamics of the mass distribution of the universe at the age of 380000 years was simulated up to the present day [9], giving an estimate of the age of cosmic objects such as galaxies, black holes and quasars, greatly improving our understanding of cosmological models and providing a theoretical comparison to satellite measurements.

Despite the simplicity and elegance of its formulation, MD is not a computationally easy task and often requires special infrastructure. The main issue is usually the evaluation of all the interactions $\boldsymbol{F}_{ij}$, which is the most time consuming procedure of any MD calculation for large systems. Moreover, the processes under study might have long characteristic time scales, requiring longer simulation time and larger data storage; classical dynamics is chaotic, i.e. the outcome is affected by the initial conditions, and since these are in principle unknown and chosen at random, some particular processes of interest might not occur just because of the specific choice, and the simulation should be repeated several times. For these reasons, it is important to optimise the evaluation of the forces as much as possible.

An early attempt to implement MD on the GPU was proposed in 2004 [10] and showed promising performance; at that time, general purpose GPU computing was not yet a well established framework and the N-body problem had to be formulated as a rendering task: a *shader* program computed each pair interaction $\boldsymbol{F}_{ij}$ and stored them as the pixel color values (RBG) in an $N \times N$ texture. Then, another shader would simply sum these values row-wise to obtain the total force on each particle and finally integrate their velocities and positions. The method is called all-pairs calculation, and as the name might suggest, it is quite expensive as it requires $\mathcal{O}(N^2)$ force evaluations. The proposed implementation was in no way optimal since the measured performance was about a tenth of the nominal value of the device, and it immediately revealed one of the main issues of the architecture that still persists nowadays: GPUs can have a processing power exceeding the teraflop, but, at the same time, they are extremely slow at handling the data to process since a memory read can require hundreds of clock cycles. The reason for the bad performance was in fact the large amount of memory read instructions compared to the amount of computation effectively performed on the fetched data, but despite this limitation, the code still outperformed a CPU by a factor of 8 because every interaction was computed concurrently. A wide overview of optimisation strategies to get around the memory latency issues can be found in Ref. [11], while, for the less eager to get their hands dirty, a review of available MD software packages is included in Ref. [12].

In the current GPU programming model, the computation is distributed in different threads, grouped together as blocks in a grid fashion, and they are allowed to share data and synchronise throughout the same block; the hardware also offers one or two levels of cache to enhance data reuse, thus reducing the amount of memory accesses, without harassing the programmer with manual pre-fetching. A more recent implementation of the all-pair calculation [13] exploiting the full power of the GPU can achieve a performance close to the nominal values, comparable to several CPU nodes.

The present and more mature GPGPU framework allows for more elaborate kernels to fit in the device, enabling the implementation of computational tricks developed during the early days of MD [14] that make it possible to integrate N-body dynamics accurately with much better scaling than $\mathcal{O}(N^2)$. For example, in many cases the inter-particle forces are short range, and it would be unnecessary to evaluate every single interaction $\boldsymbol{F}_{ij}$ since quite many of them would be close to zero and just be neglected. It is good practice to build lists of neighbours for each particle in order to speed up the calculation of forces: this also takes an $\mathcal{O}(N^2)$ operation, although the list is usually only recalculated every 100-1000 timesteps, depending on the average mobility of the particles. The optimal way to build neighbour lists is to divide the simulation box in voxels and search for a partcle's neighbours only within the adjacent voxels (Fig. 3a), as this procedure requires only $\mathcal{O}(N)$ instructions. Performance can be further improved by sorting particles depending on the index of the voxel they belong, making neighbouring particles in space, to a degree, close in memory, thus increasing coalescence and cache hit rate on GPU systems; such a task can be done with radix count sort [15,16,17] in $\mathcal{O}(N)$ with excellent performance, and it was shown to be the winning strategy [18].

Unfortunately, most often the inter-particle interactions are not exclusively short range and can be significant even at larger distances (electrostatic and gravitational forces). Therefore, introducing an interaction cut-off leads to the

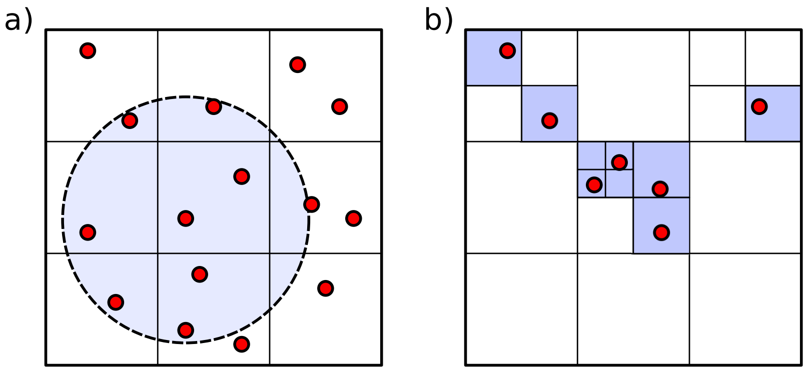


Fig. 3. Illustration of different space partition methods. In dense systems (a) a regular grid is preferred, and neighbouring particles can be searched in only a few adjacent voxels. Sparse systems (b) are better described by hierarchical trees, excluding empty regions from the computation.

wrong dynamics. For dense systems, such as bulk crystals or liquids, the electrostatic interaction length largely exceeds the size of the simulation space, and in principle one would have to include the contributions from several periodic images of the system, although their sum is not always convergent. The preferred approach consists of calculating the electrostatic potential $V(\boldsymbol{r})$ generated by the distribution of point charges $\rho(\boldsymbol{r})$ from Poisson's equation:

$$\nabla^2 V(\boldsymbol{r}) = \rho(\boldsymbol{r}) \tag{2}$$

The electrostatic potential can be calculated by discretising the charge distribution on a grid, and solving Eq. 2 with a fast Fourier transform (FFT), which has $\mathcal{O}(MlogM)$ complexity (where M is the amount of grid points): this approach is called particle-mesh Ewald (PME). Despite being heavily non-local, much work has been done to improve the FFT algorithm and make it cache efficient [19,20,21,22,23], so it is possible to achieve a 20-fold speed up over the standard CPU FFTW or a 5-fold speedup when compared to a highly optimised MKL implementation. The more recent multilevel summation method (MSM) [24] uses nested interpolations of progressive smoothing of the electrostatic potential on lattices with different resolutions, offering a good approximation of the electrostatic $\mathcal{O}(N^2)$ problem in just $\mathcal{O}(N)$ operations. The advantage of this approach is the simplicity of its parallel implementation, since it requires less memory communication among the nodes, which leads to a better scaling than the FFT calculation in PME. The GPU implementation of this method gave a 25-fold speedup over the single CPU [25]. Equation 2 can also be translated into a linear algebra problem using finite differences, and solved iteratively on multigrids [26,27] in theoretically $\mathcal{O}(M)$ operations. Even though the method initially requires several iterations to converge, the solution does not change much in one MD step and can be used as a starting point in the following step, which in turn will take much fewer iterations.

On the other hand, for sparse systems such as stars in cosmological simulations, decomposing the computational domain in regular boxes can be quite harmful because most of the voxels will be empty and some computing power and memory is wasted there. The optimal way to deal with such a situation is to subdivide the space hierarchically with an octree [28] (Fig. 3b), where only the subregions containing particles are further divided and stored. Finding neighbouring particles can be done via a traversal of the tree in $\mathcal{O}(NlogN)$ operations. Octrees are conventionally implemented on the CPU as dynamical data structures where every node contains reference pointers to its parent and children, and possibly information regarding its position and content. This method is not particularly GPU friendly since the data is scattered in memory as well as in the simulation space. In GPU implementations, the non-empty nodes are stored as consecutive elements in an array or texture, and they include the indices of the children nodes [29]. They were proved to give a good acceleration in solving the N-body problem [13,30,31]. Long range interactions are then calculated explicitly for the near neighbours, while the fast multipole method (FMM) [32,33] can be used to evaluate contributions from distant particles. The advantage of representing the system with an octree becomes now more evident: there exists a tree node containing a collection of distant particles, which can be treated as a single multipole leading to an overall complexity $\mathcal{O}(N)$. Although the mathematics required by FMM is quite intensive to evaluate, the algorithms involved have been developed and extensively optimised for the GPU architecture [34,35,36], achieving excellent parallel performance even on large clusters [37].

In all the examples shown here, the GPU implementation of the method outperformed its CPU counterpart: in many cases the speedup is only 4-5 fold when compared to a highly optimised CPU code, which seems, in a way, a discouraging result, because implementing an efficient GPU algorithm is quite a difficult task, requiring knowledge of the target hardware, and the programming model is not as intuitive as for a regular CPU. To a degree, the very same is true for CPU programming, where taking into account cache size, network layout, and details of shared/distributed memory of the target machine when designing a code leads to higher performance. These implementation difficulties could be eased by developing better compilers, that check how memory is effectively accessed and provide higher levels of GPU optimisation on older CPU codes automatically, hiding the complexity of the hardware specification from the programmer. In some cases, up to 100 fold speedups were measured, suggesting that the GPU is far superior. These cases might be unrealistic since the nominal peak performance of a GPU is around 5 times bigger than that of a CPU. Therefore, it is possible that the benchmark is done against a poorly optimised CPU code, and the speedup is exaggerated. On the other hand, GPUs were also proven to give good scaling in MPI parallel calculations, as shown in Refs. [31] and [37]. In particular, the AMBER code was extensively benchmarked in Ref. [38], and it was shown how just a few GPUs (and even just one) can outperform the same code running on 1024 CPU cores: the weight of the communication between nodes exceeds the benefit of having additional CPU cores, while the few GPUs do not suffer from this latency and can deliver better performance, although the size of the computable system becomes limited by the available GPU memory. It has to be noted how GPU solutions, even offering a modest 4-5 fold speedup, do so at a lower hardware and running cost than the equivalent in CPUs, and this will surely make them more appealing in the future. From the wide range of examples in computational physics, it is clear that the GPU architecture is well suited for a defined group of problems, such as certain procedures required in MD, while it fails for others. This point is quite similar to the everlasting dispute between raytracing and raster graphics: the former can explicitly calculate photorealistic images in complex scenes, taking its time (CPU), while the latter resorts to every trick in the book to get a visually "alright" result as fast as possible (GPU). It would be best to use both methods to calculate what they are good for, and this sets a clear view of the future hardware required for scientific computing, where both simple vector-like processors and larger CPU cores could access the same memory resources, avoiding data transfer.

3 Density-Functional Theory

Density functional theory (DFT) is a popular method for *ab-initio* electronic structure calculations in material physics and quantum chemistry. In the most commonly used DFT formulation by Kohn and Sham [39], the problem of N interacting electrons is mapped to one with N non-interacting electrons moving in an effective potential so that the total electron density is the same as in the original many-body case [40]. To be more specific, the single-particle Kohn-Sham orbitals $\psi_n(\mathbf{r})$ are solutions to the equation

$$H\psi_n(\mathbf{r}) = \epsilon_n \psi_n(\mathbf{r}), \tag{3}$$

where the effective Hamiltonian in atomic units is $H = -\frac{1}{2}\nabla^2 + v_H(\mathbf{r}) + v_{ext}(\mathbf{r}) + v_{xc}(\mathbf{r})$. The three last terms in the Hamiltonian define the effective potential, consisting of the Hartree potential v_H defined by the Poisson equation $\nabla^2 v_H(\mathbf{r}) = -4\pi\rho(\mathbf{r})$, the external ionic potential v_{ext}, and the exchange-correlation potential v_{xc} that contains all the complicated many-body physics the Kohn-Sham formulation partially hides. In practice, the v_{xc} part needs to be approximated. The electronic charge density $\rho(\mathbf{r})$ is determined by the Kohn-Sham orbitals as $\rho(\mathbf{r}) = \sum_i f_i |\psi_i(\mathbf{r})|^2$, where the f_i:s are the orbital occupation numbers.

There are several numerical approaches and approximations for solving the Kohn-Sham equations. They relate usually to the discretization of the equations and the treatment of the core electrons (pseudo-potential and all electron methods). The most common discretization methods in solid state physics are plane waves, localized orbitals, real space grids and finite elements. Normally, an iterative procedure called self-consistent field (SCF) calculation is used to find the solution to the eigenproblem starting from an initial guess for the charge density [41].

Porting an existing DFT code to GPUs generally includes profiling or discovering with some other method the computationally most expensive parts of the

SCF loop and reimplementing them with GPUs. Depending on the discretization methods, the known numerical bottlenecks are vector operations, matrix products, Fast Fourier Transforms (FFTs) and stencil operations. There are GPU versions of many of the standard computational libraries (like CUBLAS for BLAS and CUFFT for FFTW). However, porting a DFT application is not as simple as replacing the calls to standard libraries with GPU equivalents since the resulting intermediate data usually gets reused by non standard and less computationally intensive routines. Attaining high performance on a GPU and minimizing the slow transfers between the host and the device requires writing custom kernels and also porting a lot of the non-intensive routines to the GPU.

Gaussian basis functions are a popular choice in quantum chemistry to investigate electronic structures and their properties. They are used in both DFT and Hartree-Fock calculations. The known computational bottlenecks are the evaluation of the two-electron repulsion integrals (ERIs) and the calculation of the exchange-correlation potential. Yasuda was the first to use GPUs in the calculation of the exchange-correlation term [42] and in the evaluation of the Coulomb potential [43]. The most complete work in this area was done by Ufimtsev *et al.*. They have used GPUs in ERIs [44,45,46], in complete SCF calculations [47] and in energy gradients [48]. Compared to the mature GAMESS quantum chemistry package running on CPUs, they were able to achieve speedups of more than 100 using mixed precision arithmetic in HF SCF calculations. Asadchev *et al.*. have also done an ERI implementation on GPUs using the uncontracted Rys quadrature algorithm [49].

The first complete DFT code on GPUs for solid state physics was presented by Genovese *et al.*. [50]. They used double precision arithmetic and a Daubechies wavelet based code called BIGDFT [51]. The basic 3D operations for a wavelet based code are based on convolutions. They achieved speedups of factor 20 for some of these operations on a GPU, and a factor of 6 for the whole hybrid code using NVIDIA Tesla S1070 cards. These results were obtained on a 12-node hybrid machine.

For solid state physics, plane wave basis sets are the most common choice. The computational schemes rely heavily on linear algebra operations and fast Fourier transforms. The Vienna ab initio Simulation Package (VASP) [52] is a popular code combining plane waves with the projector augmented wave method. The most time consuming part of optimizing the wave functions given the trial wave functions and related routines have been ported to GPUs. Speedups of a factor between 3 and 8 for the blocked Davinson scheme [53] and for the RMM-DIIS algorithm [54] were achieved in real-world examples with Fermi C2070 cards. Parallel scalability with 16 GPUs was similar to 16 CPUs. Additionally, Hutchinson *et al.* have done an implementation of exact-exchange calculations on GPUs for VASP [55].

Quantum ESPRESSO [56] is a electronic structure code based on plane wave basis sets and pseudo-potentials (PP). For the GPU version [57], the most computationally expensive parts of the SCF cycle were gradually transferred to run on GPUs. FFTs were accelerated by CUFFT, LAPACK by MAGMA and other routines were replaced by CUDA kernels. GEMM operations were replaced by the parallel hybrid phiGEMM [58] library. For single node test systems, running with NVIDIA Tesla C2050, speedups between 5.5 and 7.8 were achieved and for a 32 node parallel system speedups between 2.5 and 3.5 were observed. Wand et al. [59] and Jia *et al.*. [60] have done an implementation for GPU clusters of a plane wave pseudo-potential code called PEtot. They were able to achieve speedups of 13 to 22 and parallel scalability up to 256 CPU-GPU computing units.

GPAW [61] is a density-functional theory (DFT) electronic structure program package based on the real space grid based projector augmented wave method. We have used GPUs to speed up most of the computationally intensive parts of the code: solving the Poisson equation, iterative refinement of the eigenvectors, subspace diagonalization and orthonormalization of the wave functions. Overall, we have achieved speedups of up to 15 on large systems and a good parallel scalability with up to 200 GPUs using NVIDIA Tesla M2070 cards [62].

Octopus [63,64] is a DFT code with an emphasis on the time-dependent density-functional theory (TDDFT) using real space grids and pseudo-potentials. Their GPU version uses blocks of Kohn-Sham orbitals as basic data units. Octopus uses GPUs to accelerate both time-propagation and ground state calculations. Finally, we would like to mention the linear response Tamm-Dancoff TDDFT implementation [65] done for the GPU-based TeraChem code.

4 Quantum Field Theory

Quantum field theories are currently our best models for fundamental interactions of the natural world (for a brief introduction to quantum field theories – or QFTs – see for example [66] or [67] and references therein). Common computational techniques include perturbation theory, which works well in quantum field theories as long as the couplings are small enough to be considered as perturbations to the free theory. Therefore, perturbation theory is the primary tool used in pure QED, weak nuclear force and high momentum-transfer QCD phenomena, but it breaks up when the coupling constant of the theory (the measure of the interaction strength) becomes large, such as in low-energy QCD.

Formulating the quantum field theory on a space-time lattice provides an opportunity to study the model non-perturbatively and use computer simulations to get results for a wide range of phenomena – it enables, for example, one to compute the hadronic spectrum of QCD (see [68] and references therein) from first principles and provides solutions for many vital gaps left by the perturbation theory, such as structure functions of composite particles [69], form-factors [70] and decay-constants [71]. It also enables one to study and test models for new physics, such as technicolor theories [72] and quantum field theories at finite temperature [73], [74] or [75]. For an introduction to Lattice QFT, see for example [76], [77] or [78].

Simulating quantum field theories using GPUs is not a completely new idea and early adopters even used OpenGL (graphics processing library) to program the GPUs to solve lattice QCD [79]. The early GPGPU programmers needed to set up a program that draws two triangles that fill the output texture of desired size by running a "shader program" that does the actual computation for each output pixel. In this program, the input data could be then accessed by fetching pixels' input texture(s) using the texture units of the GPU. In lattice QFT, where one typically needs to fetch the nearest neighbor lattice site values, this actually results in good performance as the texture caches and layouts of the GPUs have been optimized for local access patterns for filtering purposes.

4.1 Solving QFTs Numerically

The idea behind lattice QFT is based on the discretization of the path integral solution to expectation values of time-ordered operators in quantum field theories. First, one divides spacetime into discrete boxes, called the lattice, and places the fields onto the lattice sites and onto the links between the sites, as shown in Fig. 4. Then, one can simulate nature by creating a set of multiple field configurations, called an *ensemble*, and calculate the values of physical observables by computing ensemble averages over these states.

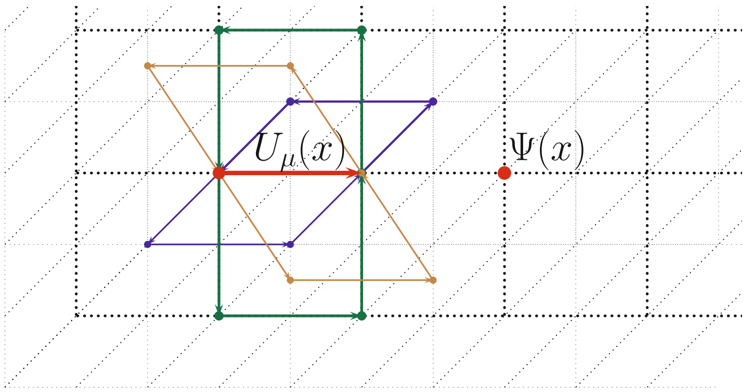


Fig. 4. The matter fields $\Psi(x)$ live on lattice sites, whereas the gauge fields $U_\mu(x)$ live on the links connecting the sites. Also depicted are the staples connecting to a single link variable that are needed in the computation of the gauge field forces.

The set of states is normally produced with the help of a Markov chain and in the most widely studied QFT, the lattice QCD, the chain is produced by combining a *molecular dynamics* algorithm together with a *Metropolis* acceptance test. Therefore, the typical computational tasks in lattice QFTs are:

1. Refresh generalized momentum variables from a heat bath (Gaussian distribution) once per *trajectory*.
2. Compute generalized forces for fields for each step

3. Integrate classical equations of motion for the fields at each step [1]
4. Perform a Metropolis acceptance test at the end of the trajectory in order to achieve the correct limiting distribution.

In order to reach satisfying statistics, normally thousands of these trajectories need to be generated and each trajectory is typically composed of 10 to 100 steps. The force calculation normally involves a matrix inversion, where the matrix indices run over the entire lattice and it is therefore the heaviest part of the computation. The matrix arises in simulations with dynamical fermions (normal propagating matter particles) and the simplest form for the fermion matrix is[2]

$$A_{x,y} = [Q^{\dagger}Q]_{x,y} \qquad \text{where}$$

$$Q_{x,y} = \delta_{x,y} - \kappa \sum_{\mu=\pm1}^{\pm4} \delta_{y+\hat{\mu},x}(1+\gamma_{\mu})U_{\mu}(x). \tag{4}$$

Here, κ is a constant related to the mass(es) of the quark(s), $\delta_{x,y}$ is the *Kronecker delta function* (unit matrix elements), the sum goes over the spacetime dimensions μ, γ_{μ} are 4-by-4 constant matrices and $U_{\mu}(x)$ are the link variable matrices that carry the force (gluons for example) from one lattice site to the neighbouring one. In normal QCD they are 3-by-3 complex matrices.

The matrix A in the equation $Ar = z$, where one solves for the vector r with a given z, is an almost diagonal sparse matrix with a *predefined sparsity pattern*. This fact makes lattice QCD ideal for parallelization, as the amount work done by each site is constant. The actual algorithm used in the matrix inversion is normally some variant of the conjugate gradient algorithm, and therefore one needs fast code to handle the multiplication of a fermion vector by the fermion matrix.

This procedure is the generation of the lattice configurations which form the ensemble. Once the set of configurations $\{U_i\}, i \in [1, N]$ has been generated with the statistical weight $e^{-S[U_i]}$, where $S[U_i]$ is the *Euclidean action* (action in imaginary time formulation), the expectation value of an operator $F[U]$ can be computed simply as

$$\langle F[U] \rangle \approx \frac{1}{N} \sum_{i=1}^{N} F[U_i], \tag{5}$$

4.2 Existing GPU Solutions to Lattice QFTs

As lattice QFTs are normally easily parallelizable, they fit well into the GPU programming paradigm, which can be characterized as parallel throughput computation. The conjugate gradient methods perform many fermion matrix vector multiplications whose arithmetic intensity (ratio of floating point operations done per byte of memory fetched) is quite low, making memory bandwidth the

[1] The integration is *not* done with respect to normal time variable, but through the Markov chain index-"time".

[2] There are multiple different algorithms for simulating fermions, here we present the simplest one for illustrative purposes.

normal bottleneck within a single processor. Parallelization between processors is done by standard MPI domain decomposition techniques. The conventional wisdom that this helps due to higher local volume to communication surface area ratio is actually flawed, as typically the GPU can handle a larger volume in the same amount of time, hence requiring the MPI-implementation to also take care of a larger surface area in the same time as with a CPU. In our experience, GPU adoption is still in some sense in its infancy, as the network implementation seems to quickly become the bottleneck in the computation and the MPI implementations of running systems seem to have been tailored to meet the needs of the CPUs of the system. Another aspect of this is that normally the GPUs are coupled with highly powered CPUs in order to cater for the situation where the users use the GPUs in just a small part of the program and need a lot of sequential performance in order to try to keep the serial part of the program up with the parallel part. The GPU also needs a lot of concurrent threads (in the order of thousands) to be filled completely with work and therefore good performance is only achievable with relatively large local lattice sizes.

Typical implementations assign one GPU thread per site, which makes parallelization easy and gives the compiler quite a lot of room to find instruction level parallelism, but in our experience this can result in a relatively high register pressure: the quantum fields living on the sites have many indices (normally color and Dirac indices) and are therefore vectors or matrices with up to 12 complex numbers per field per site in the case of quark fields in normal QCD. Higher parallelization can be achieved by taking advantage of the vector-like parallelism inside a single lattice site, but this may be challenging to implement in those loops where the threads within a site have to collaborate to produce a result, especially because GPUs impose restrictions on the memory layout of the fields (consecutive threads have to read consecutive memory locations in order to reach optimal performance [2]). In a recent paper [80], the authors solve the *gauge fixing* problem by using overrelaxation techniques and they report an increase in performance by using multiple threads per site, although in this case the register pressure problem is even more pronounced and the effects of register spilling to the L1 cache were not studied.

The lattice QCD community has a history of taking advantage of computing solutions outside the mainstream: the QCDSP [81] computer was a custom machine that used digital signal processors to solve QCD with an order of one teraflop of performance. QCDOC [82] used a custom IMB powerPC-based ASIC and a multidimensional torus network, which later on evolved into the first version of the Blue Gene supercomputers [83]. The APE collaboration has a long history of custom solutions for lattice QCD and is building custom network solutions for lattice QCD [84]. For example, QCDPAX [85] was a very early parallel architecture used to study Lattice QCD without dynamical fermions.

Currently, there are various groups using GPUs to do lattice QFT simulations. The first results using GPUs were produced as early as 2006 in a study that determined the transition temperature of QCD [86]. Standardization efforts for high precision Lattice QCD libraries are underway and the QUDA library [87] scales to hundreds of GPUs by using a local Schwarz preconditioning technique, effectively eliminating all the GPU-based MPI communications for a significant portion of the calculation. They employ various optimization techniques, such as *mixed-precision* solvers, where parts of the inversion process of the fermion matrix is done at lower precision of floating point arithmetic and using reduced representations of the SU3 matrices. Scaling to multiple GPUs can also be improved algorithmically: already a simple (almost standard) *clover improvement* [88] term in the fermion action leads to better locality and of course improves the action of the model as well, taking the lattice formulation closer to the continuum limit. Domain decomposition and taking advantage of restricted additive Schwarz (RAS) preconditioning using GPUs was already studied in 2010 in [89], where the authors get the best performance on a 32^4 lattice with vanishing overlap between the preconditioning domains and three complete RAS iterations each containing just five iterations to solve the local system of 4×32^3 sites. It should be noted though that the hardware they used is already old, so optimal parameters with up-to-date components could slightly differ.

Very soon after starting to work with GPUs on lattice QFTs, one notices the effects of Amdahl's law which just points out the fact that there is an upper bound for the whole program performance improvement related to optimizing just a portion of the program. It is quite possible that the fermion matrix inversion takes up 90% of the total computing time, but making this portion of the code run 10 times faster reveals something odd: now we are spending half of our time computing forces and doing auxiliary computations and if we optimize this portion of the code as well, we improve our performance by a factor of almost two again – therefore optimizing only the matrix inversion gives us a mere fivefold performance improvement instead of the promised order of magnitude improvement. Authors of [90] implemented practically the entire HMC trajectory on the GPU to fight Amdahl's law and recent work [91] on the QDP++ library implements *Just-in-Time* compilation to create GPU kernels on the fly to accommodate any non-performance critical operation over the entire lattice.

Work outside of standard Lattice QCD using GPUs includes the implementation of the Neuberger-Dirac overlap operator [92], which provides *chiral symmetry* at the expense of a non-local action. Another group uses the Arnoldi algorithm on a multi-GPU cluster to solve the overlap operator [93] and shows scaling up to 32 GPUs. Quenched SU2 [94] and later quenched SU2, SU3 and generic SU(N_c) simulations using GPUs are described in [95] and even compact U(1) Polyakov loops using GPUs are studied in [96]. Scalar field theory – the so-called $\lambda\phi^4$ model – using AMD GPUs is studied in [97]. The TWQCD collaboration has also implemented almost the entire HMC trajectory computation with dynamical Optimal Domain Wall Fermions, which improve the chiral symmetry of the action [98].

While most of the groups use exclusively NVIDIA's CUDA-implementation [2], which offers good reliability, flexibility and stability, there are also some groups using the OpenCL standard [99]. A recent study [100] showed better performance on AMD GPUs than on NVIDIA ones using OpenCL, although it should be noted that the NVIDIA GPUs were consumer variants with reduced double precision throughput and that optimization was done for AMD GPUs. The authors of [90] have implemented both CUDA and OpenCL versions of their staggered fermions code and they report a slightly higher performance for CUDA and for NVIDIA cards.

4.3 QFT Summary

All in all, lattice QFT using GPUs is turning from being a promising technology to a very viable alternative to traditional CPU-based computing. When reaching for the very best strong scaling performance – meaning best performance for small lattices – single threaded performance does matter if we assume that the rest of the system scales to remove other bottlenecks (communication, memory bandwith.) In these cases, it seems that currently the best performance is achievable through high-end supercomputers, such as the IBM Blue Gene/Q [101], where the microprocessor architecture is actually starting to resemble more a GPU than a traditional CPU: the PowerPC A2 chip has 16 in-order cores, each supporting 4 relatively light weight threads and a crossbar on-chip network. A 17th core runs the OS functions and an 18th core is a spare to improve yields or take place of a damaged core. This design gives the PowerPC A2 chip similar performance to power ratio as an NVIDIA Tesla 2090 GPU, making Blue Gene/Q computers very efficient. One of the main advantages of using GPUs (or GPU-like architectures) over traditional serial processors is the increased performance per watt and the possibility to perform simulations on commodity hardware.

5 Wave Function Methods

The stochastic techniques based on Markov chains and the Metropolis algorithm showed great success in the field theory examples above. There are also many-body wave function methods that use the wave function as the central variable and use stochastic techniques for the actual numerical work. These quantum Monte Carlo (QMC) techniques have shown to be very powerful tools for studying electronic structures beyond the mean-field level of for example the density functional theory. A general overview of QMC can be found from [102]. The simplest form of the QMC algorithms is the variational QMC, where a trial wave function with free parameters is constructed and the parameters are optimized, for example, to minimize the total energy [103]. This simple strategy works rather well for various different systems, even for strongly interacting particles in an external magnetic field [104].

There have been some works porting QMC methods to GPUs. In the early work by Amos G. Anderson *et al.* [105], the overall speedup compared to the CPU was rather modest, from three to six, even if the individual kernels were up to 30 times faster. More recently, Kenneth P. Esler *et al.* [106] have ported the QMCPack simulation code to the Nvidia CUDA GPU platform. Their full application speedups are typically around 10 to 15 compared to a quad-core

Xeon CPU. This speedup is very promising and demonstrates the great potential GPU computing has for the QMC methods that are perhaps the computational technologies that are the mainstream in future electronic structure calculations.

There are also many-body wave function methods that are very close to the quantum chemical methods. One example of these is the full configuration interaction method in chemistry that is termed exact diagonalization (ED) in physics. The activities in porting the quantum chemistry approaches to GPU are reviewed in [107], and we try to remain on the physics side of this unclear borderline. We omit, for example, works on the coupled cluster method on the GPU [108]. Furthermore, quantum mechanical transport problems are also not discussed here [109].

Lattice models [110,111] are important for providing a general understanding of many central physical concepts like magnetism. Furthermore, realistic materials can be cast to a lattice model [112]. Few-site models can be calculated exactly using the ED method. The ED method turns out to be very efficient on the GPU [113]. In the simplest form of ED, the first step is to construct the many-body basis and the Hamiltonian matrix in it. Then follows the most time-consuming part, namely the actual diagonalization of the Hamiltonian matrix. In many cases, one is mainly interested in the lowest eigenstate and possibly a few of the lowest excited states. For these, the Lanczos algorithm turns out to be very suitable [113]. The basic idea of the Lanczos scheme is to map the huge but sparse Hamiltonian matrix to a smaller and tridiagonal form in the so-called Krylov space that is defined by the spanning vectors obtained from a starting vector $|x_0\rangle$ by acting with the Hamiltonian as $H^m|x_0\rangle$. Now, as the GPU is very powerful for the matrix-vector product, it is not surprising that high speedups compared to CPUs can be found[113].

6 Outlook

The GPU has made a definite entry into the world of computational physics. Preliminary studies using emerging technologies will always be done, but the true litmus test of a new technology is whether studies emerge where the new technology is actually used to advance science. The increasing frequency of studies that mention GPUs is a clear indicator of this.

From the point of view of high performance computing in computational physics, the biggest challenge facing GPUs at the moment is scaling: in the strong scaling case, as many levels of parallelism as possible inherent in the problem should be exploited in order to reach the best performance with small local subsystems. The basic variables of the model are almost always vectors of some sort, making them an ideal candidate for SIMD type parallelism. This is often achieved with CPUs with a simple compiler flag, which instructs the compiler to look for opportunities to combine independent instructions into vector operations.

Furthermore, large and therefore interesting problems from a HPC point of view are typically composed of a large number of similar variables, be it particles, field values, cells or just entries in an array of numbers, which hints at another, higher level of parallelism of the problem that traditionally has been exploited using MPI, but is a prime candidate for a data parallel algorithm. Also, algorithmic changes may be necessary to reach the best possible performance: it may very well be that the best algorithm for CPUs is no longer the best one for GPUs. A classic example could be the question whether to use lookup tables of certain variables or recompute them on-the-fly. Typically, on the GPU the flops are cheap making the recomputation an attractive choice whereas the large caches of the CPU may make the lookup table a better option.

On the other hand, MPI communication latencies should be minimized and bandwidth increased to accommodate the faster local solve to help with both weak and strong scaling. As far as we know, there are very few, if any, groups taking advantage of GPUDirect v.2 for NVIDIA GPUs [114], which allows direct GPU-to-GPU communications (the upcoming GPUDirect Support for RDMA will allow direct communications across network nodes) reducing overhead and CPU synchronization needs. Even GPUDirect v.1 helps, as then one can share the *pinned memory* buffers between Infiniband and GPU cards, removing the need to do extra local copies of data. The MPI implementations should also be scaled to fit the needs of the GPUs connected to the node: currently the network bandwidth between nodes seems to be typically about two orders of magnitude lower than the memory bandwidth from the GPU to the GPU memory, which poses a challenge to strong scaling, limiting GPU applicability to situations with relatively large local problem sizes.

Another, perhaps an even greater challenge, facing GPUs and similar systems is the ecosystem: Currently a large portion of the developers and system administrators like to think of GPUs and similar solutions as *accelerators* – an accelerator is a component, which is attached to the main processor and used to speed up certain portions of the code, but as these "accelerators" become more and more agile with wider support for standard algorithms, the term becomes more and more irrelevant as a major part of the entire computation can be done on the "accelerator" and the original "brains" of the machine, the CPU, is mainly left there to take care of administrative functions, such as disk IO, common OS services and control flow of the program.

As single threaded performance has reached a local limit, all types of processors are seeking more performance out of parallelism: more cores are added and vector units are broadened. This trend, fueled by the fact that transistor feature sizes keep on shrinking, hints at some type of convergence in the near future, but exactly what it will look like is anyone's best guess. At least in computational physics, it has been shown already that the scientists are willing to take extra effort in porting their code to take advantage of massively parallel architectures, which should allow them to do the same work with less energy and do more science with the resources allocated to them.

The initial programming effort does raise a concern for productivity: How much time and effort is one willing to spend to gain a certain amount of added performance? Obviously, the answer depends on the problem itself, but perhaps even more on the assumed direction of the industry – a wrong choice may result in wasted effort if the chosen solution simply does not exist in five years time. Fortunately, what seems to be clear at the moment, is the overall direction of the industry towards higher parallelism, which means that a large portion of the work needed to parallelize a code for a certain parallel architecture will most probably be applicable to another parallel architecture as well, reducing the risk of parallelization beyond the typical MPI level.

The answer to what kind of parallel architectures will prevail the current turmoil in the industry may depend strongly on consumer behavior, since a large part of the development costs of these machines are actually subsidized by the development of the consumer variants of the products. Designing a processor only for the HPC market is too expensive and a successful product will need a sister or at least a cousin in the consumer market. This brings us back to DOOM and other performance-hungry games: it may very well be that the technology developed for the gamers of today, will be the programming platform for the scientists of tomorrow.

Acknowledgements. We would like to thank Kari Rummukainen, Adam Foster, Risto Nieminen, Martti Puska, and Ville Havu for all their support. Topi Siro acknowledges the financial support from the Finnish Doctoral Programme in Computational Sciences FICS. This research has been partly supported by the Academy of Finland through its Centres of Excellence Program (Project No. 251748) and by the Academy of Finland Project No. 1134018.

References

1. Macedonia, M.: The GPU enters computing's mainstream. Computer 36(10), 106–108 (2003)
2. NVIDIA Corporation: NVIDIA CUDA C programming guide, Version 4.2 (2012)
3. McCammon, J.A., Gelin, B.R., Karplus, M.: Dynamics of folded proteins. Nature 267(5612), 585–590 (1977)
4. Tembre, B.L., Cammon, J.M.: Ligand-receptor interactions. Computers & Amp; Chemistry 8(4), 281–283 (1984)
5. Gao, J., Kuczera, K., Tidor, B., Karplus, M.: Hidden thermodynamics of mutant proteins: a molecular dynamics analysis. Science 244(4908), 1069–1072 (1989)
6. Samish, I., MacDermaid, C.M., Perez-Aguilar, J.M., Saven, J.G.: Theoretical and computational protein design. Annual Review of Physical Chemistry 62(1), 129–149 (2011)
7. Berkowitz, M.L., Kindt, J.T.: Molecular Detailed Simulations of Lipid Bilayers, pp. 253–286. John Wiley & Sons, Inc. (2010)
8. Lyubartsev, A.P., Rabinovich, A.L.: Recent development in computer simulations of lipid bilayers. Soft Matter 7, 25–39 (2011)
9. Springel, V., White, S.D.M., Jenkins, A., Frenk, C.S., Yoshida, N., Gao, L., Navarro, J., Thacker, R., Croton, D., Helly, J., Peacock, J.A., Cole, S., Thomas, P., Couchman, H., Evrard, A., Colberg, J., Pearce, F.: Simulations of the formation, evolution and clustering of galaxies and quasars. Nature 435(7042), 629–636 (2005)

10. Chinchilla, F., Gamblin, T., Sommervoll, M., Prins, J.: Parallel N-body simulation using GPUs. Technical report, University of North Carolina (2004)
11. Brodtkorb, A.R., Hagen, T.R., Sætra, M.L.: Graphics processing unit (GPU) programming strategies and trends in GPU computing. Journal of Parallel and Distributed Computing (2012)
12. Stone, J.E., Hardy, D.J., Ufimtsev, I.S., Schulten, K.: GPU-accelerated molecular modeling coming of age. Journal of Molecular Graphics and Modelling 29(2), 116–125 (2010)
13. Nyland, L., Harris, M., Prins, J.: Fast N-Body Simulation with CUDA. In: GPU Gems 3, ch. 31, vol. 3. Addison-Wesley Professional (2007)
14. Allen, M.P., Tildesley, D.J.: Computer Simulation of Liquids. Clarendon, Oxford (2002)
15. Kipfer, P., Segal, M., Westermann, R.: Uberflow: a GPU-based particle engine. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWS 2004, pp. 115–122. ACM, New York (2004)
16. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. Technical report, NVIDIA (2008)
17. Ha, L., Krüger, J., Silva, C.T.: Fast four-way parallel radix sorting on GPUs. Computer Graphics Forum 28(8), 2368–2378 (2009)
18. Anderson, J.A., Lorenz, C.D., Travesset, A.: General purpose molecular dynamics simulations fully implemented on graphics processing units. Journal of Computational Physics 227(10), 5342–5359 (2008)
19. Moreland, K., Angel, E.: The FFT on a GPU. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWS 2003, pp. 112–119. Eurographics Association, Aire-la-Ville (2003)
20. Govindaraju, N.K., Manocha, D.: Cache-efficient numerical algorithms using graphics hardware. Technical report, The University of North Carolina (2007)
21. Gu, L., Li, X., Siegel, J.: An empirically tuned 2d and 3d FFT library on CUDA GPU. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, pp. 305–314. ACM, New York (2010)
22. Chen, Y., Cui, X., Mei, H.: Large-scale FFT on GPU clusters. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, pp. 315–324. ACM, New York (2010)
23. Ahmed, M., Haridy, O.: A comparative benchmarking of the FFT on Fermi and Evergreen GPUs. In: 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 127–128 (2011)
24. Skeel, R.D., Tezcan, I., Hardy, D.J.: Multiple grid methods for classical molecular dynamics. Journal of Computational Chemistry 23(6), 673–684 (2002)
25. Hardy, D.J., Stone, J.E., Schulten, K.: Multilevel summation of electrostatic potentials using graphics processing units. Parallel Computing 35(3), 164–177 (2009)
26. Goodnight, N., Woolley, C., Lewin, G., Luebke, D., Humphreys, G.: A multigrid solver for boundary value problems using programmable graphics hardware. In: ACM SIGGRAPH 2005 Courses, SIGGRAPH 2005, ACM, New York (2005)
27. McAdams, A., Sifakis, E., Teran, J.: A parallel multigrid poisson solver for fluids simulation on large grids. In: Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA 2010, pp. 65–74. Eurographics Association, Aire-la-Ville (2010)
28. Meagher, D.: Octree Encoding: a New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer. Rensselaer Polytechnic Institute. Image Processing Laboratory (1980)

29. Lefebvre, S., Hornus, S., Neyret, F.: Octree Textures on the GPU. In: GPU Gems 2, ch. 37, vol. 2. Addison-Wesley Professional (2005)
30. Belleman, R.G., Bédorf, J., Zwart, S.F.P.: High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. New Astronomy 13(2), 103–112 (2008)
31. Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K., Taiji, M.: 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 62:1–62:12. ACM, New York (2009)
32. Rokhlin, V.: Rapid solution of integral equations of classical potential theory. Journal of Computational Physics 60(2), 187–207 (1985)
33. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. Journal of Computational Physics 73(2), 325–348 (1987)
34. Gumerov, N.A., Duraiswami, R.: Fast multipole methods on graphics processors. Journal of Computational Physics 227(18), 8290–8313 (2008)
35. Darve, E., Cecka, C., Takahashi, T.: The fast multipole method on parallel clusters, multicore processors, and graphics processing units. Comptes Rendus Mécanique 339(2-3), 185–193 (2011)
36. Takahashi, T., Cecka, C., Fong, W., Darve, E.: Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units. International Journal for Numerical Methods in Engineering 89(1), 105–133 (2012)
37. Yokota, R., Barba, L., Narumi, T., Yasuoka, K.: Petascale turbulence simulation using a highly parallel fast multipole method on GPUs. Computer Physics Communications (2012)
38. Götz, A.W., Williamson, M.J., Xu, D., Poole, D., Le Grand, S., Walker, R.C.: Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. generalized Born. Journal of Chemical Theory and Computation 8(5), 1542–1555 (2012)
39. Kohn, W., Sham, L.J.: Self-consistent equations including exchange and correlation effects. Phys. Rev. 140, A1133–A1138 (1965)
40. Parr, R., Yang, W.: Density-Functional Theory of Atoms and Molecules. International Series of Monographs on Chemistry. Oxford University Press, USA (1994)
41. Payne, M.C., Teter, M.P., Allan, D.C., Arias, T.A., Joannopoulos, J.D.: Iterative minimization techniques for *ab initio* total-energy calculations: molecular dynamics and conjugate gradients. Rev. Mod. Phys. 64, 1045–1097 (1992)
42. Yasuda, K.: Accelerating density functional calculations with graphics processing unit. Journal of Chemical Theory and Computation 4(8), 1230–1236 (2008)
43. Yasuda, K.: Two-electron integral evaluation on the graphics processor unit. Journal of Computational Chemistry 29(3), 334–342 (2008)
44. Ufimtsev, I., Martinez, T.: Graphical processing units for quantum chemistry. Computing in Science Engineering 10(6), 26–34 (2008)
45. Ufimtsev, I.S., Martinez, T.J.: Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. Journal of Chemical Theory and Computation 4(2), 222–231 (2008)
46. Luehr, N., Ufimtsev, I.S., Martinez, T.J.: Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs). Journal of Chemical Theory and Computation 7(4), 949–954 (2011)
47. Ufimtsev, I.S., Martinez, T.J.: Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation. Journal of Chemical Theory and Computation 5(4), 1004–1015 (2009)

48. Ufimtsev, I.S., Martinez, T.J.: Quantum chemistry on graphical processing units. 3. Analytical energy gradients, geometry optimization, and first principles molecular dynamics. Journal of Chemical Theory and Computation 5(10), 2619–2628 (2009)
49. Asadchev, A., Allada, V., Felder, J., Bode, B.M., Gordon, M.S., Windus, T.L.: Uncontracted Rys quadrature implementation of up to G functions on graphical processing units. Journal of Chemical Theory and Computation 6(3), 696–704 (2010)
50. Genovese, L., Ospici, M., Deutsch, T., Méhaut, J.F., Neelov, A., Goedecker, S.: Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. The Journal of Chemical Physics 131(3), 034103 (2009)
51. Genovese, L., Neelov, A., Goedecker, S., Deutsch, T., Ghasemi, S.A., Willand, A., Caliste, D., Zilberberg, O., Rayson, M., Bergman, A., Schneider, R.: Daubechies wavelets as a basis set for density functional pseudopotential calculations. The Journal of Chemical Physics 129(1), 014109 (2008)
52. Kresse, G., Furthmüller, J.: Efficient iterative schemes for *ab initio* total-energy calculations using a plane-wave basis set. Phys. Rev. B 54, 11169–11186 (1996)
53. Maintz, S., Eck, B., Dronskowski, R.: Speeding up plane-wave electronic-structure calculations using graphics-processing units. Computer Physics Communications 182(7), 1421–1427 (2011)
54. Hacene, M., Anciaux-Sedrakian, A., Rozanska, X., Klahr, D., Guignon, T., Fleurat-Lessard, P.: Accelerating VASP electronic structure calculations using graphic processing units. Journal of Computational Chemistry (2012) n/a–n/a
55. Hutchinson, M., Widom, M.: VASP on a GPU: Application to exact-exchange calculations of the stability of elemental boron. Computer Physics Communications 183(7), 1422–1426 (2012)
56. Giannozzi, P., Baroni, S., Bonini, N., Calandra, M., Car, R., Cavazzoni, C., Ceresoli, D., Chiarotti, G.L., Cococcioni, M., Dabo, I., Corso, A.D., de Gironcoli, S., Fabris, S., Fratesi, G., Gebauer, R., Gerstmann, U., Gougoussis, C., Kokalj, A., Lazzeri, M., Martin-Samos, L., Marzari, N., Mauri, F., Mazzarello, R., Paolini, S., Pasquarello, A., Paulatto, L., Sbraccia, C., Scandolo, S., Sclauzero, G., Seitsonen, A.P., Smogunov, A., Umari, P., Wentzcovitch, R.M.: Quantum espresso: a modular and open-source software project for quantum simulations of materials. Journal of Physics: Condensed Matter 21(39), 395502 (2009)
57. Girotto, I., Varini, N., Spiga, F., Cavazzoni, C., Ceresoli, D., Martin-Samos, L., Gorni, T.: Enabling of Quantum-ESPRESSO to petascale scientific challenges. In: PRACE Whitepapers. PRACE (2012)
58. Spiga, F., Girotto, I.: phiGEMM: A CPU-GPU library for porting Quantum ESPRESSO on hybrid systems. In: 2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 368–375 (February 2012)
59. Wang, L., Wu, Y., Jia, W., Gao, W., Chi, X., Wang, L.W.: Large scale plane wave pseudopotential density functional theory calculations on GPU clusters. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 71:1–71:10. ACM, New York (2011)
60. Jia, W., Cao, Z., Wang, L., Fu, J., Chi, X., Gao, W., Wang, L.W.: The analysis of a plane wave pseudopotential density functional theory code on a GPU machine. Computer Physics Communications 184(1), 9–18 (2013)

61. Enkovaara, J., Rostgaard, C., Mortensen, J.J., Chen, J., Dułak, M., Ferrighi, L., Gavnholt, J., Glinsvad, C., Haikola, V., Hansen, H.A., Kristoffersen, H.H., Kuisma, M., Larsen, A.H., Lehtovaara, L., Ljungberg, M., Lopez-Acevedo, O., Moses, P.G., Ojanen, J., Olsen, T., Petzold, V., Romero, N.A., Stausholm-Møller, J., Strange, M., Tritsaris, G.A., Vanin, M., Walter, M., Hammer, B., Häkkinen, H., Madsen, G.K.H., Nieminen, R.M., Nørskov, J.K., Puska, M., Rantala, T.T., Schiøtz, J., Thygesen, K.S., Jacobsen, K.W.: Electronic structure calculations with GPAW: a real-space implementation of the projector augmented-wave method. Journal of Physics: Condensed Matter 22(25), 253202 (2010)
62. Hakala, S., Havu, V., Enkovaara, J., Nieminen, R.: Parallel Electronic Structure Calculations Using Multiple Graphics Processing Units (GPUs). In: Manninen, P., Öster, P. (eds.) PARA 2012. LNCS, vol. 7782, pp. 63–76. Springer, Heidelberg (2013)
63. Castro, A., Appel, H., Oliveira, M., Rozzi, C.A., Andrade, X., Lorenzen, F., Marques, M.A.L., Gross, E.K.U., Rubio, A.: Octopus: a tool for the application of time-dependent density functional theory. Physica Status Solidi (B) 243(11), 2465–2488 (2006)
64. Andrade, X., Alberdi-Rodriguez, J., Strubbe, D.A., Oliveira, M.J.T., Nogueira, F., Castro, A., Muguerza, J., Arruabarrena, A., Louie, S.G., Aspuru-Guzik, A., Rubio, A., Marques, M.A.L.: Time-dependent density-functional theory in massively parallel computer architectures: the Octopus project. Journal of Physics: Condensed Matter 24(23), 233202 (2012)
65. Isborn, C.M., Luehr, N., Ufimtsev, I.S., Martinez, T.J.: Excited-state electronic structure with configuration interaction singles and and Tamm-Dancoff time-dependent density functional theory on graphical processing units. Journal of Chemical Theory and Computation 7(6), 1814–1823 (2011)
66. Peskin, M.E., Schroeder, D.V.: An Introduction to Quantum Field Theory. Westview Press (1995)
67. Crewther, R.J.: Introduction to quantum field theory. ArXiv High Energy Physics - Theory e-prints (1995)
68. Fodor, Z., Hoelbling, C.: Light hadron masses from lattice QCD. Reviews of Modern Physics 84, 449–495 (2012)
69. Göckeler, M., Hägler, P., Horsley, R., Pleiter, D., Rakow, P.E.L., Schäfer, A., Schierholz, G., Zanotti, J.M.: Generalized parton distributions and structure functions from full lattice QCD. Nuclear Physics B Proceedings Supplements 140, 399–404 (2005)
70. Alexandrou, C., Brinet, M., Carbonell, J., Constantinou, M., Guichon, P., et al.: Nucleon form factors and moments of parton distributions in twisted mass lattice QCD. In: Proceedings of The XXIst International Europhysics Conference on High Energy Physics, EPS-HEP 2011, Grenoble, Rhones Alpes France, July 21-27, vol. 308 (2011)
71. McNeile, C., Davies, C.T.H., Follana, E., Hornbostel, K., Lepage, G.P.: High-precision f_{B_s} and heavy quark effective theory from relativistic lattice QCD. Physical Review D 85, 031503 (2012)
72. Rummukainen, K.: QCD-like technicolor on the lattice. In: Llanes-Estrada, F.J., Peláez, J.R. (eds.). American Institute of Physics Conference Series, vol. 1343, pp. 51–56 (2011)
73. Petreczky, P.: Progress in finite temperature lattice QCD. Journal of Physics G: Nuclear and Particle Physics 35(4), 044033 (2008)
74. Petreczky, P.: Recent progress in lattice QCD at finite temperature. ArXiv e-prints (2009)

75. Fodor, Z., Katz, S.D.: The phase diagram of quantum chromodynamics. ArXiv e-prints (August 2009)
76. Montvay, I., Münster, G.: Quantum Fields on a Lattice. Cambridge Monographs on Mathematical Physics. Cambridge University Press, The Edinburgh Building (1994)
77. Rothe, H.J.: Lattice Gauge Theories: An Introduction, 3rd edn. World Scientific Publishing Company, Hackendsack (2005)
78. Gupta, R.: Introduction to lattice QCD. ArXiv High Energy Physics - Lattice e-prints (1998)
79. Egri, G., Fodor, Z., Hoelbling, C., Katz, S., Nogradi, D., Szabo, K.: Lattice QCD as a video game. Computer Physics Communications 177, 631–639 (2007)
80. Schröck, M., Vogt, H.: Gauge fixing using overrelaxation and simulated annealing on GPUs. ArXiv e-prints (2012)
81. Mawhinney, R.D.: The 1 teraflops QCDSP computer. Parallel Computing 25(10-11), 1281–1296 (1999)
82. Chen, D., Christ, N.H., Cristian, C., Dong, Z., Gara, A., Garg, K., Joo, B., Kim, C., Levkova, L., Liao, X., Mawhinney, R.D., Ohta, S., Wettig, T.: QCDOC: A 10-teraflops scale computer for lattice QCD. In: Nuclear Physics B Proceedings Supplements, vol. 94, pp. 825–832 (March 2001)
83. Bhanot, G., Chen, D., Gara, A., Vranas, P.M.: The BlueGene / L supercomputer. Nuclear Physics B - Proceedings Supplements 119, 114–121 (2003)
84. Ammendola, R., Biagioni, A., Frezza, O., Lo Cicero, F., Lonardo, A., Paolucci, P.S., Petronzio, R., Rossetti, D., Salamon, A., Salina, G., Simula, F., Tantalo, N., Tosoratto, L., Vicini, P.: apeNET+: a 3D toroidal network enabling petaFLOPS scale Lattice QCD simulations on commodity clusters. In: Proceedings of The XXVIII International Symposium on Lattice Field Theory, Villasimius, Sardinia Italy, June 14-19 (2010)
85. Shirakawa, T., Hoshino, T., Oyanagi, Y., Iwasaki, Y., Yoshie, T.: QCDPAX-an MIMD array of vector processors for the numerical simulation of quantum chromodynamics. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing 1989, pp. 495–504. ACM, New York (1989)
86. Aoki, Y., Fodor, Z., Katz, S.D., Szabó, K.K.: The QCD transition temperature: Results with physical masses in the continuum limit. Physics Letters B 643, 46–54 (2006)
87. Babich, R., Clark, M.A., Joó, B., Shi, G., Brower, R.C., Gottlieb, S.: Scaling lattice QCD beyond 100 GPUs. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 70:1–70:11. ACM, New York (2011)
88. Hasenbusch, M., Jansen, K.: Speeding up the HMC: QCD with clover-improved wilson fermions. Nuclear Physics B Proceedings Supplements 119, 982–984 (2003)
89. Osaki, Y., Ishikawa, K.I.: Domain decomposition method on GPU cluster. In: Proceedings of The XXVIII International Symposium on Lattice Field Theory, Villasimius, Sardinia Italy, June 14-19 (2010)
90. Bonati, C., Cossu, G., D'Elia, M., Incardona, P.: QCD simulations with staggered fermions on GPUs. Computer Physics Communications 183, 853–863 (2012)
91. Winter, F.: Accelerating QDP++ using GPUs. In: Proceedings of the XXIX International Symposium on Lattice Field Theory (Lattice 2011), Squaw Valley, Lake Tahoe, California, July 10-16 (2011)
92. Walk, B., Wittig, H., Dranischnikow, E., Schomer, E.: Implementation of the Neuberger overlap operator in GPUs. In: Proceedings of The XXVIII International Symposium on Lattice Field Theory, Villasimius, Sardinia Italy, June 14-19 (2010)

93. Alexandru, A., Lujan, M., Pelissier, C., Gamari, B., Lee, F.X.: Efficient implementation of the overlap operator on multi-GPUs. ArXiv e-prints (2011)
94. Cardoso, N., Bicudo, P.: SU (2) lattice gauge theory simulations on Fermi GPUs. Journal of Computational Physics 230, 3998–4010 (2011)
95. Cardoso, N., Bicudo, P.: Generating SU(Nc) pure gauge lattice QCD configurations on GPUs with CUDA. ArXiv e-prints (2011)
96. Amado, A., Cardoso, N., Cardoso, M., Bicudo, P.: Study of compact U(1) flux tubes in 3+1 dimensions in lattice gauge theory using GPU's. ArXiv e-prints (2012)
97. Bordag, M., Demchik, V., Gulov, A., Skalozub, V.: The type of the phase transition and coupling values in $\lambda\phi^4$ model. International Journal of Modern Physics A 27, 50116 (2012)
98. Chiu, T.W., Hsieh, T.H., Mao, Y.Y.: Pseudoscalar Meson in two flavors QCD with the optimal domain-wall fermion. Physics Letters B B717, 420 (2012)
99. Munshi, A.: The OpenCL specification, Version 1.2 (2011)
100. Bach, M., Lindenstruth, V., Philipsen, O., Pinke, C.: Lattice QCD based on OpenCL. ArXiv e-prints (2012)
101. IBM Systems and Technology: IBM System Blue Gene/Q – Data Sheet (2011)
102. Foulkes, W.M.C., Mitas, L., Needs, R.J., Rajagopal, G.: Quantum Monte Carlo simulations of solids. Reviews of Modern Physics 73, 33–83 (2001)
103. Harju, A., Barbiellini, B., Siljamaki, S., Nieminen, R., Ortiz, G.: Stochastic gradient approximation: An efficient method to optimize many-body wave functions. Physical Review Letters 79(7), 1173–1177 (1997)
104. Harju, A.: Variational Monte Carlo for interacting electrons in quantum dots. Journal of Low Temperature Physics 140(3-4), 181–210 (2005)
105. Anderson, A.G., Goddard III, W.A., Schröder, P.: Quantum Monte Carlo on graphical processing units. Computer Physics Communications 177(3), 298–306 (2007)
106. Esler, K., Kim, J., Ceperley, D., Shulenburger, L.: Accelerating quantum Monte Carlo simulations of real materials on GPU clusters. Computing in Science and Engineering 14(1), 40–51 (2012)
107. Wölfle, A.W.G., Walker, R.C.: Quantum chemistry on graphics processing units. In: Wheeler, R.A. (ed.). Annual Reports in Computational Chemistry, ch. 2, vol. 6, pp. 21–35. Elsevier (2010)
108. DePrince, A., Hammond, J.: Quantum chemical many-body theory on heterogeneous nodes. In: 2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC), pp. 131–140 (2011)
109. Ihnatsenka, S.: Computation of electron quantum transport in graphene nanoribbons using GPU. Computer Physics Communications 183(3), 543–546 (2012)
110. Hubbard, J.: Electron correlations in narrow energy bands. Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences 276(1364), 238–257 (1963)
111. Gutzwiller, M.C.: Effect of correlation on the ferromagnetism of transition metals. Physical Review Letters 10, 159–162 (1963)
112. Meredith, J.S., Alvarez, G., Maier, T.A., Schulthess, T.C., Vetter, J.S.: Accuracy and performance of graphics processors: A quantum Monte Carlo application case study. Parallel Computing 35(3), 151–163 (2009)
113. Siro, T., Harju, A.: Exact diagonalization of the Hubbard model on graphics processing units. Computer Physics Communications 183(9), 1884–1889 (2012)
114. NVIDIA Corporation: NVIDIA GPUDirect$^{\mathrm{TM}}$ Technology (2012)

Preparing Scientific Application Software for Exascale Computing

J.A. Åström[1], A. Carter[2], J. Hetherington[3], K. Ioakimidis[4], E. Lindahl[5], G. Mozdzynski[6], R.W. Nash[3], P. Schlatter[7], A. Signell[8], and J. Westerholm[8]

[1] CSC – IT Center for Science, P.O. Box 405, FIN-02101, Espoo, Finland
[2] EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK
[3] Dept of Chemistry, Faculty of Maths & Physical Sciences, University College London, Gower Street, London, WC1E 6BT, UK
[4] Institute of Fluid Mechanics and Hydraulic Machinery, University of Stuttgart, Pfaffenwaldring 10, D - 70550 Stuttgart, Germany
[5] Science for Life Laboratory, KTH & Stockholm University, Box 1031, 17121 Solna, Sweden
[6] European Centre for Medium-Range Weather Forecasts, Shinfield Park, Reading RG2 9AX, UK
[7] Department of Mechanics, KTH, SE-100 44 Stockholm, Sweden
[8] Åbo Akademi, Institutionen för Informationsteknologi, Datateknik/ högeffektiva datorberäkningar, Joukahainengatan 3-5 A 20520, Turku, Finland

Abstract. Many of the most widely used scientifc application software of today were developed largely during a time when the typical amount of compute cores was calculated in tens or hundreds. Within a not too distant future the number of cores will be calculated in at least hundreds of thousands or even millions. A European collaboration group CRESTA has recently been working on a set of renowned scientific software to investigate and develop these codes towards the realm of exascale computing. The codes are ELMFIRE, GROMACS, IFS, HemeLB, NEK5000, and OpenFOAM. This paper contains a summary of the strategies for their development towards exascale and results achieved during the first year of the collaboration project.

Keywords: Exascale, Science Application Software, Optimization, Elmfire, Gromacs, IFS, HemeLB, NEK5000, OpenFOAM.

1 Introduction

Many of the most widely used scientific application software have been subject to constant development during several decades. This easily results in codes that have primary structures that are optimized for the typical computer architecture of the early phase of development. The most dramatic change in high-performance computers (HPCs) during the last decade has been that the number of compute cores have risen dramatically. This trend also seems to continue in the near future. Another modern trend is the introduction of accelerators, like

GP-GPUs. This trend further complicates the use of legacy software in HPCs. Very few of the most important application software of today are ready to be used on the massively parallel architecture of the exascale HPC facilities which are planned within the next 5-10 years targeted at exascale performance.

Within the EC-funded project CRESTA (Collaborative Research into Exascale Sytemware, Tools and Applications) [1], a set of six applications codes are being investigated and optimized with the objective of preparing them for exascale. The codes are ELMFIRE, GROMACS, IFS, HemeLB, NEK5000 and OpenFOAM.

ELMFIRE [2] is a gyro-kinetic particle-in-cell code that simulates movement and interaction between high-speed particles on a three dimensional grid in torus-shaped atomic fusion reactors. The particles are held together by an external magnetic field. The objective is to simulate significant portions of large-scale reactors like the Joint European Torus JET [3] or the International Thermonuclear Experimental Reactor ITER [4].

GROMACS [5] is a molecular dynamics code that is extensively used for simulation of biomolecular systems. It is primarily designed for biochemical molecules like proteins, lipids and nucleic acids with a large amount of complicated bonded interactions, but since GROMACS is optimized for fast computing it is also fairly extensively used for simulating e.g. non-organic polymers.

IFS is the production weather forecasting application used at the European Centre for Medium Range Weather Forecasts (ECMWF). The objective is to develop more reliable 10-day weather forecasts that can be run in an hour or less using denser grids of measurement values.

HEMELB [9] is ultimately intended to form part of a clinically deployed exascale virtual physiological human. HemeLB simulates blood flows in empirical blood vessel geometries. The objective is to develop a clinically useful exascale tool.

NEK5000 [8] is an open-source code for the simulation of incompressible flow in complex geometries. Simulation of turbulent flow is of one of the major objectives of NEK5000.

OPENFOAM [6] is an open source software for computational fluid dynamics. The program is a "toolbox" which provides a selection of different solvers as well as routines for various kinds of analysis, pre- and post-processing. Within the present project the focus will be on a specialized code for turbine machinery. The future objective is to be able to simulate a whole hydraulic machine on exascale architectures [6,7].

In the following sections we present more detailed descriptions of the exascale development strategies of the these codes.

2 Exascale Strategies and Development

2.1 ELMFIRE

ELMFIRE is a particle-in-cell code that simulates the movement and interaction between extended charged gyrokinetic particles moving at high speed in a torus-shaped geometry. The particles are confined by a strong external magnetic field. ELMFIRE approximates the Coulomb interaction between particles by solving a global electrostatic field on a grid, using the particle charges as sources. ELMFIRE then advances particles in time by free streaming along the magnetic field line and particle drift perpendicular to the magnetic field. Typically, time steps correspond to 30-50ns real time. Today the time step based simulation in ELMFIRE can be roughly divided into seven parts: (I) Perform momentum and energy conserving binary collisions between particles close to each other, (II) Using a 4th order Runge-Kutta, calculate particle movements in continuous space during the time step based on the electric field, (III) Collect grid cell charge data from the particles for the electrostatic field. (IV) Combine and split the grid charge data so each processor has a smaller part of it, (V) Construct a large modified gyro-kinetic Poisson equation based on the data and solve it in parallel, (VI) Calculate additional movement caused by polarization drift of particles based on the acquired electric field, (VII) Write diagnostics output.

The most CPU heavy part of the code presently is calculating particle movements but as each processor is assigned a fixed number of particles this scales linearly with the number of processors and is therefore not an issue when scaling to larger systems. The most interesting part is the collection and distribution of grid cell charge data. In the current version each processor can have its assigned particles moving in any part of the torus, leading to all processor contributing charge data to all grid cells in the system. Charge neutrality, a central requirement in plasma physics, is achieved by forcing the ion polarization drift and electron parallel acceleration by the electric field to create such shifts in the particle positions that each cell charge becomes zero. To accomplish local charge neutrality, Elmfire uses a modified Poisson equation where the righthand side source terms are calculated based on the positions of the particles after they have been moved based on the current electric field. The ion polarization drift and electron parallel acceleration can be expressed as a movement of a small charge from a cell to another. However, as the drift and acceleration depend on the new electric field, part of the charge movement has to be expressed as a function of the new electric field and therefore included in the lefthand side Poisson matrix. The effect of the ion polarization drift and the electron parallel acceleration is limited to the nearby grid cells. Solving the modified Poisson equation gives the new electric field, which is used to calculate the actual particle movement caused by the polarization drift and parallel acceleration. The final result, at the end of the time step, is neutrality in each grid cell. As a consequence each processor stores the full electrostatic grid data and a huge sparse matrix (# grid cells x # grid cells) for collecting charge data for the grid cells. The matrix has been optimized by reducing the second dimension to a constant, which is the number of cells around a given cell to which charges due to gyrokinetic motion and polarization drift can be moved from the given cell. This reduces memory usage significantly but not enough for large-scale simulations. It also introduces an extra index conversion when gathering the data. Once the grid cell charge data has been combined and split among the processors, each processor can construct its own part of the Poission equation individually. The Poisson equation is then solved in parallel using PETSc [10]. The solution (the electric potential) is then distributed to all processors to be used in the next time step. Focus of the initial work on ELMFIRE will be on basic scalability, mostly related to memory usage. The version provided for the project does not implement any spatial domain decomposition that leads to massive memory usage and data duplication. This currently completely prevents simulations on large grids. The items mentioned below are what currently have been identified as possible solutions to problems problems preventing ELMFIRE from scaling to simulations larger than 100 000 processors. It is however expected that we find additional, and more important, problems once the initial domain decomposition has been done.

(I) Implement a 3D domain decomposition : The version provided for the project does not implement any spatial decomposition of the particles. Particles are distributed evenly among processors but the electrostatic grid data is duplicated in all processors. This prevents scaling to larger electrostatic grids than approximately 120x150x8 regardless of the number of cores available. For large scale simulations of e.g. JET or ITER it would be beneficial to be able to simulate electrostatic grids up to 3000x4000x16 i.e. almost 1500 times larger than today. An estimate for an ITER simulation is that 640 000 cores would be needed for 590 billion particles. With the current version this would require approximately 28TB memory per core. We plan to implement an electrostatic grid cell based domain decomposition of the code so that each processor can have particles only inside its own grid cells. This should restrict the grid cell data needed in each processor to its own grid cells and a few surrounding grid cells (in order to propagate the particles in time). It should also remove the need to communicate large amount of data for the charge data with the downside of having to send particle data between processors in each time step.

(II) Improve load balancing : In the current version load balancing is not a large problem but it is expected that the 3D domain decomposition will introduce load-balancing issues, as the particles are not evenly distributed between all grid cells in the simulation. These need to be investigated and addressed after the initial domain decomposition has been performed. One approach would be to dynamically reallocate the electrostatic grid based on the workload, that is, the size of the grid and the number of particles.

(III) Improve memory usage for binary collisions: ELMFIRE calculates collisions between randomly chosen particles close to each other in each time step. In order to assess how close particles are to each other, a separate collision grid is set up. Currently this uses 10 times the memory it really needs. By introducing data structures that avoids duplications this could be improved.

(IV) Parallelize file writing : File writing in ELMFIRE is presently done by all processes sending data to process 0, which then writes the data to disk. For small simulations this is typically not an issue ($< 5\%$ of the each time steps goes to writing diagnostics) but it will likely block large scale simulations and input

files for visualizations. The file writing needs to be parallelized for ELMFIRE to scale to ITER sized problems.

2.2 GROMACS

The work in GROMACS is focused on achieving significant improvements for real applications. Seen from the users side, there are three overall important objectives in order to advance the state-of-the-art for different applications: *(I)* to reduce the computation time per iteration in order to achieve longer simulations, *(II)* to improve the capacity to handle larger application systems so that e.g. mesoscopic phenomena can be simulated, and *(III)* to improve the accuracy and statistics for small application systems through massive sampling. All three aspects are critically important, but they require slightly different approaches. The wallclock time for a single time-step iteration is already today in the range of a few milliseconds for some systems, and while there are some possibilities to improve this further, it is not likely that it can be improved by more than perhaps down to the order of 0.1 milliseconds. In contrast, handling much larger systems, i.e. more atoms, is easier (although not trivial) from a parallelization algorithm point-of-view, but it will involve challenges related to handling of data when a single master node no longer can control all input and output, both when starting execution and for checkpointing or output. Finally, for small systems the main approach will be ensemble techniques to handle thousands of simulations that each will use thousands of cores. Detailed improvements wiil be as follow:

(I) Benchmarking new GROMACS releases, and GPU coding : GROMACS version 4.6, which has been developed during the first part of the project, is currently in the beta stage, and will bring some important new advances in domain decomposition and scaling over previous versions. We have developed a new set of computational kernels that have departed from the classical implementation with neighbor lists, which will make it much easier to parallelize both with SIMD and multithreading, and achieve a higher fraction of the hardware peak floating-point performance. These kernels are also being implemented on GPUs, and Gromacs 4.6 will use heterogeneous acceleration with some kernels running on the GPU while other execute simultaneously on the CPU, where the domain decomposition is also done. It will be an important step to benchmark all these new kernels on different hardware, in particular large clusters with GPU co-processors (such as Cray XK6), and in this frame we will also implement support for the next-generation Nvidia Kepler architecture scheduled for release in the spring of 2012. These cards in particular will be used on several new Cray installations.

(II) Multi-grid solvers for efficient PME electrostatics : The vast majority of biomolecular simulations rely on particle-mesh Ewald (PME) lattice summation to handle long-range electrostatic interactions. Since this in turn relies on 3D FFTs, the associated all-to-all communication pattern is a major bottleneck for scaling. We are developing improved FFT algorithms and communication patterns, but to improve support for heterogeneous architectures such as CPU-GPU parallelism on each node, we need to develop algorithms that avoid

communicating grids over all processors. This can currently be achieved either through multipole-based [11], or multigrid-based [12] methods, and we intend to investigate both. This part is targeting "medium size " parallelization for normal-size atomic systems (10k-100k cores), and the O(N) algorithms will provide virtually perfect weak scaling, even for systems including long-range electrostatics (currently this is only true for simple cut-off interactions).

(III) Efficient large-scale I/O : With the completion of long-range electrostatics algorithms that exhibit O(N) scaling, it should be possible to reach multi-petascale for normal simulations of very large systems such as virus particles, complexes of several molecules, or standard material science studies. Typical simulations of this kind may involve a few hundred million particles. To support this, we need to rewrite the input/output layer of Gromacs so that a large set of the I/O tasks participate in reading the data from files to avoid running out of memory on the master node, and to avoid global communication during start-up. This will ideally use a minimalistic PGAS-like library that is fully portable (or even included in the code), so that all I/O code does not have to do explicit communication. We will also implement code for check-pointing and for trajectory output that supports asynchronous output by sending the data to a subset of the I/O nodes which then transpose the data, and write it to trajectories while the simulation continues. This should be decomposed over time-frames rather than space.

(IV) Task-based parallelism : One of the most significant long-term changes will be a complete code re-write to support introduction of task-based parallelism for improved efficiency inside many-core nodes, to enable better simultaneous utilization of CPUs and GPUs, and to overlap computation and communication between nodes. Of these, the last item will be particularly critical for increased scaling, since system size growth means that gradually more time is spent on communication than computation. At this point we will also investigate the usage of lower-level communication libraries to improve scaling further. Presently, our preliminary tests indicate that automated tools such as OpenMP do not provide sufficiently fine-grained control over the execution, and we might therefore have to use threads directly unless better alternatives are found.

(V) Ensemble computing and parallel adaptive molecular dynamics : Our main disruptive long-term path to true exascale performance will be to combine direct domain-decomposition scaling in individual simulations with ensemble approaches to support simultaneous execution of thousands of coupled simulations. This will be accomplished by using Markov State Models and kinetic clustering for parallel adaptive simulation [13]. In contrast to the distributed computing approach used e.g. in Folding@Home [14], exascale resources will enable extremely tight coupling between simulations each using 1k-100k cores. This will make it possible to employ kinetic clustering for slow dynamics (e.g. multi-millisecond structural transitions in proteins) where even single state transitions will require petascale-level simulations, and complete mapping of the processes is simply not possible with todays resources. This will initially be implemented as a separate layer of code, where our idea is to formulate dynamic data flow networks that

execute a set of simulations, perform analysis, and based on the result of the analysis a second generation of simulations is executed. The advantage of this approach is that the resulting code will be very easy to adapt to other simulation programs (in principle anything that relies on sampling). In particular, this setup will enable us to achieve exascale performance for typical application systems. A target setup is a normal membrane protein system with around 250,000 atoms. With the new electrostatics solvers and task parallelism, we expect to achieve efficient scaling over 1k-10k cores (including heterogeneous CPU-GPU parallelism), and an ensemble could then typically include 1,000 such simulations, which means efficient use of well over a million cores. Larger systems will enable us to push this to even larger supercomputers, and approach a billion cores on future exascale resources.

2.3 IFS

The Integrated Forecasting System (IFS) is the production numerical weather forecast application at ECMWF. IFS comprise several component suites, namely, a 10-day deterministic forecast, a four dimension variational analysis (4D-Var), an ensemble prediction system (EPS) and an ensemble data assimilation system (ENDA). The use of ensemble methods are well matched to todays HPC systems, as each ensemble application (model or data assimilation) is independent and can be sized in resolution and by the number of ensemble members to fill any supercomputer. However, these ensemble applications are only part of the IFS production suite and the high resolution deterministic model (referred to as 'IFS model' from now on) and 4D-Var analysis applications are equally important in providing forecasts to ECMWF member states of up to 10 to 15 days ahead. For the CRESTA project it has been decided to focus on the IFS model to understand its present limitations and to explore approaches to get it to scale well on future exascale systems. While the focus is on the IFS model, it is expected that developments to the model should also improve the performance of the other IFS suites (EPS, 4D-Var and ENDA) mentioned above. The resolution of the operational IFS model today is T1279L91 (1279 spectral waves and 91 levels in the atmosphere). For the IFS model, it is paramount that it completes a 10-day forecast in less than one hour so that forecast products can be delivered on time to ECMWF member states. The IFS model is expected to be increased in resolution over time as shown in Table 1.

Table 1. IFS model: current and future model resolutions

IFS model resolution	Envisaged Operational Implementation	Grid point spacing (km)	Time-step (seconds)
T1279L91	2011	16	600
T2047L137	2014-2015	10	450
T3999L200	2023-2024	5	240
T7999L300	2031-2032	2.5	120

As can be seen in this table, the time-step reduces as the model resolution increases. In general halving the grid spacing increases the computational cost by 12, a doubling of cost for each of the horizontal coordinate directions plus the time-step and only 50 percent more for the vertical. However, in reality the cost can be greater than this, when some non-linear items are included such as the Legendre transforms and Fourier transforms. It is clear from this that the IFS model from a computational viewpoint can utilize future supercomputers at Exascale and beyond. What is less clear is whether the IFS model can continue to run efficiently on such systems and continue to meet the operational target of one hour when running on 100,000 or more cores which it would have to do. In a nutshell, IFS is a spectral, semi-implicit, semi-Lagrangian code, where data exists in 3 spaces, namely, grid-point, Fourier and spectral space. In a single time-step data is transposed between these spaces so that the respective grid-point, Fourier and spectral computations are independent over two of the three co-ordinate directions in each space. Fourier transforms are performed between grid-point and Fourier spaces, and Legendre transforms are performed between Fourier and spectral spaces. A full description of the above IFS parallelization scheme is contained in [15]. The performance of the IFS model has been well documented over the past 20 years, with many developments to improve performance, with more recent examples described in presentations on the ECMWF web-site (http://www.ecmwf.int). In recent years focus has turned to the cost of the Legendre transform, where the computational cost is $O(N^3)$ for the global model, where N denotes the cut-off wave number in the triangular truncation of the spherical harmonics expansion. This has been addressed by a Fast Legendre Transform (FLT) development, where the computational cost is reduced to $C_L N^2 LOG(N)$ where C_L is a constant and $C_L \ll N$. The FLT algorithm is described in [16,17,18]. While the cost of the Legendre transforms has been addressed, the associated TRMTOM and TRMTOL transposition routines between Fourier and spectral space are relatively expensive at T3999 ($> 10\%$ of wall time). Today, these transpositions are implemented using efficient MPI_allgatherv collective calls in separate communicator groups, which can be considered the state of the art for MPI communications. Within the CRESTA project we plan to address this performance issue by using Fortran90 coarrays to overlap these communications with the computation of the Legendre transforms, this being done per wave number within an OpenMP parallel region. If this approach is successful, it could pave the way for other areas in the IFS where similar communication can be overlapped with computation. The semi-implicit semi-Lagrangian (SL) scheme in IFS allows the use of a relatively long time-step as compared with a Eulerian solution. This scheme involves the use of a halo of data from neighbouring MPI tasks which is needed to compute the departure-point and mid-point of the wind trajectory for each grid-point ('arrival' point) in a tasks partition. While the communications in the SL scheme are relatively local the downside is that the location of the departure point is not known until run-time and therefore the IFS must assume a worst case geographic distance for the halo extent computed from a maximum assumed wind speed of 400 m/s and the time-step. Today, each task must perform MPI communications for this halo of data before the iterative scheme can execute to determine the departure-point and mid-point of the wind trajectory. This approach is clearly non-scaling as the same halo of data must be communicated, even if a task only has one grid-point (a rather extreme example). To address this non-scaling issue, the SL scheme will be optimized to use Fortran90 coarrays to only get grid-columns from neighbouring tasks as and when they are required in the iterative scheme to compute the departure-point and mid-point of the trajectory. In IFS the cost for computing Fourier transforms is $C_F N_J LOG(N_J)$, for each varying length latitude $J = 1..N$ (N as above), where C_F is a constant and N_J is the number of grid points on latitude J. For optimal performance of the fourier transforms, full latitudes are statically load-balanced to tasks, where each task is responsible to computing FFTs for a subset of latitudes and a subset of atmospheric levels. The heuristic currently used will be reviewed as part of the CRESTA project and to explore an improved cost function for this load-balancing problem. The improved scheme should be applicable to all model resolutions. Based on the above background description of IFS, we propose the following schedule of developments within the CRESTA project. It should be noted that some of these developments will overlap in time.

(I) Coarray kernel : Develop kernel to investigate overlapping computation and communication using Fortran 2008 coarrays in an OpenMP parallel region.

(II) Exascale "Legendre transform" optimization : The IFS transform library will be optimized to overlap the computation of the Legendre transforms with the associated communications. These code developments will use the same strategy as prototyped in the Coarray Kernel, where the Legendre transform computation and associated coarray communications will execute in the same OpenMP parallel region. This development will be tested using IFS model resolutions up to T2047.

(III) Exascale "Semi-Lagrangian" optimization : Developments to the IFS semi-Lagrangian scheme to use Fortran 2008 coarrays to improve scalability by removing the need to perform full halo wide communications.

(IV) Optimization of Fourier latitude load-balancing heuristic : Optimization of the heuristic used to statically load-balance the distribution of variable length latitudes in grid-space. An optimal distribution of latitudes is required to load-balance the cost of performing Fourier transforms as IFS transforms data from grid to Fourier space.

(V) Development of a future solver for IFS : Research into a new multigrid solver for extreme scaling of IFS and a replacement of the spectral method. Such a solver could be initially tested using a shallow water model code and not IFS. Please note, this development is not part of ECMWF's current research plans and should be considered more speculative.

2.4 HemeLB

The Lattice-Boltzmann method for solution of partial differential equations has nearly ideal weak scaling properties. The HemeLB code has, consequently, proved to have excellent scalability. This has been explicitly proved up to roughly 20,000 cores and it is reasonable to expect good scaling far beyond this number. For HemeLB to function properly on multi-petascale and exascale and to be useful for clinical applications a set of libraries and systemware must be able to support the code. It is not sufficient that these libraries be delivered as research code capable only of use on specific platforms, each of these must be usable, manageable, deployable well-engineered, well-tested code.

(I) Visualisation and steering : Support for standard flow field visualisation for exascale simulations is a prerequisite for HemeLB to work at the exascale. As a first step, standard tools for flow visualisation, such as COVISE [17] will be linked to HemeLB in an ad-hoc fashion. However, to move forward, we will need to work with CRESTA collaborators to define a configuration system (API or DSL) so that visualisation tools can work with HemeLBs data in-situ, to support co-visualisation. In order to handle remote visualisation for steering at the exascale, data-volumes will need to be reduced by in-situ extraction of medically relevant properties, such as vessel wall stress, so that these smaller datasets can be shared. As HemeLB will form part of an ecosystem of computational physiology models within the Virtual Physiological Human, these systems will need to be made sufficiently configurable so that HemeLB results can be visualised alongside those of collaborating codes as part of a multiscale simulation.

(II) Pre-processing : HemeLB uses the Parmetis [16] library to achieve domain decomposition for sparse geometries. Effort will be required within CRESTA to ensure this library scales appropriately. CRESTA enhanced or developed domain decomposition tools must support configurable interfaces for application specific domain decomposition. Later efforts will support continuous dynamic domain decomposition, in response to both simulation and system variability, including support for fault-tolerance.

(III) Environments and operating systems : The vision of HemeLB as part of a clinically deployed exascale virtual physiological human will require usable environments for exascale deployment and job management. Job management infrastructure must support remote on-demand access from clinical settings, and appropriate algorithms for resource sharing must be developed for exascale hardware for this context. Operating system support for applications must be robust and easy-to-use, supporting multiple interacting applications using heterogeneous languages and paradigms for multiscale simulation.

Environmental support for auto-tuning of application configuration will be necessary, and this will require effort to support interaction with HemeLBs compile-time auto configuration facilities through CMake.

(III) Introspection : HemeLB, as with many other applications, needs to be aware of its own progress as time passes. This application introspection, if it is not to be a blocker to exascale performance, will require attention from HemeLB developers and CRESTA tool effort. This will require not only performance measurement, but also support for report generation, visualisation of the correctness of the lattice-Boltzmann simulation. Within the multiscale VPH context, HemeLB introspection will need to interact with that of other applications. A

clear API allowing application developers to discover on-going changes in the host environment, responding to faults and slow-downs, will be required for performance at Exascale.

2.5 NEK5000

Nek5000 [8] is an open-source code for the simulation of incompressible flow in complex geometries. The discretization is based on the spectral-element method (SEM) that combines the higher-order accuracy from spectral methods with the geometric flexibility of finite element methods. Nek5000 is written in mixed Fortran77/C and designed to employ fully large-scale parallelism. The code has a long history of HPC development. Recently the large-scale simulations were successful performed on the Cray XE6 system at PDC, KTH with 32,768 cores [19] and on the IBM BG/P Eugene with 262,144 cores [20]. An overview of the capabilities and recent developments within the Nek5000 community is given in the presentation by Paul Fischer, Main developer [20]. Within the CRESTA project, main focus will be on the development of the following software environment and tools:

(I) Adaptive refinement : Current version of Nek5000 code uses conformal grid with uniform order of the spatial interpolations throughout the domain. The principal way for grid refinement is by global p-refinement, i.e. by increasing the approximation order globally. There are two methods of introducing adaptive mesh refinement (AMR): adaptive h-refinement, i.e. the splitting of cells into smaller ones, and adaptive p-refinement, i.e. increasing polynomial order in given element. Giving possibility to resolve particular region of the flow, AMR makes a challenge as it can have negative .effect on scalability. However, local refinement, either adaptive or by user intervention, is a desirable feature for nek5000 which will be crucial for the future scalability of the code, in particular for the simulation of large-scale problems involving turbulence. In the CRESTA project we will work on framework of adaptive refinement in h-types. The basic idea is that the refinements are only used in the regions with significant errors. Such error estimators can be formulated based on the solution of the adjoint equations (dual problem) that can be thought as a measure of the sensitivity of certain observables to the local mesh quality. Such estimators have been developed at KTH. Though consideration of multiple local observables such as drag, shedding frequency etc. it is proposed to decide when to divide the element or switch from lower-order to higher-order (or vice versa).

(II) Alternative discretisation : So far, nek5000 is designed to have a spectral-element discretization in all directions, either 2D or 3D. For certain cases, in particular flows in which spatial homogeneity can be assumed in at least one direction, the SEM discretization could be replaced by a more optimal Fourier-Galerkin discretization. A substantial gain in performance can be expected for such flow cases. The algorithmic changes implied by this new discretization, and in particular the impact on scalability will be studied within CRESTA.

(III) Hybrid parallelization : In the present state, Nek5000 does not employ any hybrid approach to parallelization. All communication is handled by MPI, which

has proven to be very efficient, mainly due to the element structure of the mesh. However, in the light of alternative discretization that might include an additional level into the mesh topology, a hybrid approach should be reconsidered.

(IV) Boundary conditions for exascale computing : The definition of boundary conditions requires special attention, especially in cases where large parts of the domain are in the turbulent state. In particular for exascale computations, which are aimed at realistic geometries in large domains, a faithful prescription of boundary conditions is crucial. The challenge is two-fold: First, reflections in the form of pressure waves need to be avoided at boundaries, and secondly, proper convective properties need to be maintained as to reduce the upstream influence of the condition, even in the presence of highly unsteady flow towards the boundary. Similar issues need to be dealt with at inflow boundaries when transient turbulent velocity profiles are required: Simply adding random fluctuations to the DNS profiles lacks the temporal and spatial correlation of real turbulence. The fluctuations must be pre-computed and stored in a database or computed on the fly from an auxiliary computation. In the framework of exascale simulations, the handling of such unsteady conditions needs to be assessed and refined.

(V) Pre- and post-processing : In the CRESTA project it is decided to focus on h-type refinements, so we have to consider simple mechanism of the mesh generations build into the code. However, only quadrilateral (2D) and hexahedral (3D) elements are used in the types of mesh used in the Nek5000 making the problem easier. For the real-life and industrial applications, it is necessary to employ scalable pre-processing tools for complex geometries. Within the CRESTA collaboration an interface to optimized solutions of meshes with domain decomposition and load balancing should be created.

(VI) Load balancing : Nek5000 can obtain full scaling with uniform order for petascale computations. When adaptive mesh refinement is introduced the load balancing should be carefully considered due to the fact that computation on different cores begin to differ as a result of varying order of accuracy.

2.6 OpenFOAM

OpenFOAM is an open source application for computational fluid dynamics (CFD). The program is a 'toolbox' which provides a selection of different solvers as well as routines for various kinds of analysis, pre- and post-processing. OpenFOAM is licenced under the GPL. As such, modifications have been made to the code by different parties at different times and several versions are in common use. In this project, we consider the official release from the OpenFOAM foundation (a not-for profit organisation, wholly owned by OpenCFD Ltd.), and the release from the OpenFOAM Extend project. It is hoped that any changes to the code contributed by the CRESTA project could be made available for inclusion in both distributions, but if there are good reasons to make optimisations or improvements to one particular version, we will do so. For example, there is code specific to the Extend project for dealing with moving geometries. If it turns out that this code introduces a performance bottleneck, then this would be a valid candidate for optimisation during the project. Since the code can be used in many different ways, it is challenging to identify ways to enable the application for exascale systems in general. It is likely that there are some problems that are much more amenable to large-scale systems, but it is not obvious a priori that there is much to be gained in making simulations of simple systems (such as Lid-driven Cavity Flow) scale to many more processors than at present. In conjunction with contacts at OpenCFD Ltd., we have identified a use-case that is considered a realistic candidate for simulation at exascale. This specific example, which consists of modelling the flow of air around a motorbike, is representative of a wider class of problems that could benefit from simulation on exascale systems. There is no published roadmap for the development of OpenFOAM, so this activity will have to be fairly reactive to any developments in the releases of the code. Having said that, it is expected that the following approach will be taken to prepare OpenFOAM for exascale systems:

(I) Benchmarking of the latest version of the code : Version 2.1.0 of OpenFOAM has been released since the CRESTA project started. There have been some fairly major changes to the code since version 1, including the incorporation of parallel mesh generation. Benchmarking and profiling of OpenFOAM have been undertaken on previous versions, but before we know where to concentrate our efforts in optimization for future systems, we need to understand the impacts of recent changes on the codes performance. In addition to providing an update of previous results on the performance of OpenFOAM based on current systems and the newest version of the code, we will adjust parameters of our profiling runs in order to attempt to measure how the performance would vary as the ratios of computation, communication and memory access vary. In addition, we will specifically investigate the I/O performance of the code and seek to identify how these I/O patterns are likely to change when scaling up to exascale.

(II) Code analysis of the latest version of the code : In tandem to measuring the performance of the code, an analysis of the codes structure will be undertaken in order to, for example: Determine internal interfaces in the code where alternative solvers, libraries, etc. could be swapped in if it was determined that these could provide better performance; Determine the parallelisation patterns currently used in the code and evaluate these with respect to exascale issues such as fault-tolerance. A simple example of this might be that a synchronous domain-decomposition might not be intolerant to a process failing, whereas a tracked task-farm approach might be able to recover from a process failing. (Note that this is example is illustrative. At present, there is no evidence that either of these patterns is directly relevant to OpenFOAM.)

(III) Performance analysis of kernels, libraries : In the course of the activities above, we will have been able to quantitatively measure the characteristics of the sub-problems solved by libraries and routines used for linear algebra and meshing. We will then engage with the developers of these libraries and seek comparisons with the other applications to determine possible optimisations.

(IV) Iterative performance improvement : Concentrating on those parts of the code which have been determined to be potential future bottlenecks, we will use standard optimisation techniques to seek to improve the scaling of the code (including, for example, overlapping communication and computation, possibly through the use of more asynchronous communications, investigating the effects of compiler optimization, changing memory access patterns, introducing further (hybrid) levels of parallelisation).

(V) Investigation of alternative parallelisation approaches : This is a riskier approach to improving parallel performance scaling, but potentially has large rewards, especially if it emerges that future architectures look like they will be qualitatively different from those of today. With a large code like OpenFOAM, it is very difficult to make non-incremental changes to the code, but having gained a good understanding of the codes structure and performance over the first two years of the project, it is likely that proof-of-concept code could be written to demonstrate alternative parallelisation patterns that could eventually be adopted by the codes developers. These will probably involve exposing more potential parallelism in the problem so that the code can make use of the millions of cores expected to feature in the machines of the future. Such patterns could include hybrid message-passing / shared memory approaches, adding task parallelism, or re-computing certain data to reduce communications.

(VI) Hydraulic machinery : The application of OpenFOAM at the Institute of Fluid Mechanics and Hydraulic Machinery, University of Stuttgart, is the simulation of the flow in an entire hydraulic turbine using a Large Eddy Simulation (LES). This means that a great part of the turbulence in the flow will be resolved in the computation up to very fine turbulent scales. Since the Reynolds number of this flow is very high this simulation needs very fine computational grids, very fine time steps and long simulation times. Consequently a very high computational effort is required. According to a publication of Chapman [22] and Fröhlich [23] the number of vertices in the computational domain can be estimated to approximately 1000 million for all parts of a hydraulic machine. In order to do LES for a whole hydraulic machine (including rotor/stator interaction) the General Grid Interface (GGI) implemented in OpenFOAM is needed. For this reason the version OpenFOAM-1.6-extend [7] is required. In our knowledge no work has been done on exascale systems with the OpenFOAM-1.6-extend version. GGI was a bottleneck in the OpenFOAM-extend version but due to a new implementation performs well when running on 512 cores. Further performance and scale up tests will be carried out to find out if GGI is a possible bottleneck on exascale systems. In case GGI could be bottleneck on exascale systems, an upgrade must be carried out. Furthermore, the standard simulation technique in OpenFOAM for incompressible flows is an implicit time discretization with a SIMPLE or PISO type pressure-velocity coupling. These algorithms could be computationally time expensive because of the need to repeatedly solve global systems of linear equations in an iterative loop. The solution of these global linear equation systems could be a bottleneck for a LES on very fine grids. Performance and scale up tests will be carried out in order to identify if the

algorithms mentioned before are able to get good results, as well as a good performance with OpenFOAM-extend. If it is not the case, the algorithms will be changed towards an explicit formulation. A version of the Fractional Step Method would be proposed to solve the equations. It is well known that the Fractional Step Method (FSM) is used for Direct Numerical Simulation (DNS) and LES to enhance the stability of the solution. It is expected, that this method will reach a higher performance for very large computational grids. To realize the tests mentioned above two test cases have been prepared at IHS. To check if the physics is correct quite quickly, we have prepared the ERCOFTAC square cylinder with about 15 million grid vertices. The ERCOFTAC square cylinder is a unique test case that is experimentally measured [24]. Furthermore, the final scope is to compute a whole hydraulic machine and therefore we have as final test case a whole hydraulic machine.

3 Summary

In summary, the experience so far from the CRESTA project is that exascale strategies are rather code specific. The rather expected exceptions from this are parallel I/O and hybrid parallelization. These two seems to be more or less necessary for all codes with exascale ambitions, but neither of them receive much emphasis in any of the above application code strategies.

Fault tolerance is another issue that should concern all code to some extent. This problem has been discussed rather extensively within the HPC community for some time, but the problem is, more or less, theoretical until hardware faults begin to occur much more frequently in practice.

Finally, an issue that should concern many scientific applications is the type of scaling strategy for exascale. These can be crudely divided in three cathegories: strong, weak and ensemble scaling. For the strong scaling case it is in general probably impossible to reach exascale, for weak scaling it seems realistic to some extent, and for the ensemble it is in general possible for any code. This kind of scaling strategy thinking is likely to become one of the key components of the practical side of exascale computing in the future: How to first maximize strong scaling, then weak scaling, and thereafter to find the optimal way to govern a large set of simultaneous large parallel computations in order to maximize the scientific output. Finally, there must also be a stratgey on how to handle the massive data output from such an exersice. Of these, the first two parts have been rather well taken care of already as they are key components of code optimization on smaller computers. The latter two become important on multi-petaflop or exaflop scale.

References

1. http://www.cresta-project.eu
2. Heikkinen, J.A., Janhunen, S.J., Kiviniemi, T.P., Ogando, F.: Full f gyrokinetic method for particle simulation of tokamak transport. Journal of Computational Physics 227, 5582–5609 (2008)

3. http://www.efda.org/jet/
4. http://www.iter.org
5. Berendsen, et al.: Comp. Phys. Comm. 91, 43–56 (1995)
6. OpenFOAM web site, See, http://www.openfoam.org/download/
7. Extend project web site. See, http://www.extend-project.de/the-extend-project
8. nek5000.mcs.anl.gov
9. Mazzeo, M.D., Coveney, P.V.: Computer Physics Communications, 178(12), 894–914 (2008)
10. http://www.mcs.anl.gov/petsc/index.html
11. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. J. Comput. Phys. 73, 325 (1987)
12. Izaguirre, J.A., Hampton, S.S., Matthey, T.: Parallel multigrid summation for the N-body problem. J. Parallel Dist. Comp. 65, 949–962 (2005)
13. Pande, V.S., Beauchamp, K., Bowman, G.R.: Everything you wanted to know about Markov State Models but were afraid to ask. Methods 52, 99–105 (2010)
14. http://folding.stanford.edu/English/HomePage
15. Barros, S.R.M., Dent, D., Isaksen, L., Robinson, G., Mozdzynski, G., Wollenweber, F.: The IFS Model: A parallel production weather code. Parallel Computing 21, 1621–1638 (1995)
16. Rokhlin, V., Tygert, M.: Fast algorithms for spherical harmonic expansions. SIAM Journal on Scientific Computing 27(6), 1903–1928 (2006)
17. Tygert, M.: Fast algorithms for spherical harmonic expansions, II. Journal of Computational Physics 227(8), 4260–4279 (2008)
18. Tygert, M.: Fast algorithms for spherical harmonic expansions, III. Journal of Computational Physics 229(18), 6181–6192 (2010)
19. Malm, J., Schlatter, P., Henningson, D.S.: Coherent structures and dominant frequencies in a turbulent three-dimensional diffuser. J. Fluid Mech (2012)
20. Extreme Scaling Workshop 2010 Report
21. http://www.mcs.anl.gov/~fischer/nek5000/fischer_nek5000_dec2010.pdf
22. Chapman, D.: Computational Aerodynamics Development and Outlook. AIAA Journal 17(12), 1293–1313 (1979)
23. Fröhlich, J.: Large Eddy Simulation turbulenter Strïmungen, 1st edn. Teubner Verlag, Auflage (2006)
24. ERCOFTAC web site. See, http://www.ercoftac.org/

PRACE DECI (Distributed European Computing Initiative) Minisymposium

Chris Johnson[1], Adam Carter[1], Iain Bethune[1], Kevin Statford[1], Mikko Alava[2], Vitor Cardoso[3], Muhammad Asif[4], Bernhard S.A. Schuberth[5], and Tobias Weinzierl[6]

[1] EPCC, Department of Physics and Astronomy, The University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JZ, Scotland
{chrisj,adam,ibethune,kevin}@epcc.ed.ac.uk

[2] Aalto University, School of Science, Department of Applied Physics, Finland
mikko.alava@aalto.fi

[3] CENTRA, Departamento de Física, Instituto Superior Técnico, Universidade Técnica de Lisboa - UTL, Av. Rovisco Pais 1, 1049 Lisboa, Portugal
Department of Physics and Astronomy, The University of Mississippi, University, MS 38677, USA
vitor.cardoso@ist.utl.pt

[4] Catalan Institute of Climate Sciences, Barcelona, Spain
muhammad.asif@ic3.cat

[5] Dept. of Earth and Environmental Sciences, Geophysics Section, Ludwig-Maximilians-Universität München, Theresienstr. 41, 80333 Munich, Germany
mail@bernhard-schuberth.de

[6] Technische Universität München, Garching, Germany
tobias.weinzierl@mytum.de

1 Introduction

This article gives an overview of the DECI (Distributed European Computing Initiative) Minisymposium held within the PARA 2012 conference taking the form of a short set of articles for each of the talks presented. The work presented here was carried out under either the DEISA (receiving funding through the EU FP7 project RI-22291) or PRACE-2IP (receiving funding from the EU FP7 Programme (FP7/2007-2013) under grant agreement no RI-283493) projects.

2 How Strong Are Materials? (Alava[2])

2.1 Introduction

Large-scale simulations of fracture models were done to resolve the question of how strong are materials. We investigated systematically [1] the behaviour of two dimensional lattice models - the so-called random fuse networks - by varying the system size and the disorder present in the "material". This class of models [2] simplifies continuum fracture by putting it into a lattice, and doing a scalar approximation of elasticity; it is known that qualitatively this is not essential and the resulting speedup is important. For the same reason the simulations are tried in two dimensions instead of three.

These results, combined with theoretical advances made to explain them, establish the fundamental basis of the strength of brittle materials in the presence of heterogeneities - in other words for anything which is hard and something other than a pure single crystal. The main efforts in the DEISA part consisted of porting the simulation software to the KTH Ekman-cluster and running it. The numerical effort is to solve a series of mechanical equilibria - which amounts to a damage mechanics study - the evolution of the scalar fracture model by rank-I updates. These work only in 2D, so extensions to 3D systems are much tougher. The original version was written at ORNL by Phani Nukala and the current version is an adaptation of a serial version from Cornell to the DEISA HPC environment.

The main issue is to produce massive amounts of data from "fracture experiments" on systems that evolve to the fracture point with accumulating damage. This then produces empirical probability distributions of strength (the peak or maximum stress along the stress-strain curve). These are illustrated in Fig. 1. The crucial question now becomes, what is the physical mechanism underlying the observed distributions? We found out that the "disordered materials" under study evolve with damage accumulation such that finally two crack populations exist: the original one, and a set of larger ones from crack coalescence. Both of these have an exponential shape.

The mathematics of the problem has been postulated to be "simply" the statistics of extremes. This assumes that the systems (as in laboratory samples) can be split into independent sub volumes, and that these "representative elements" have the same microscopic detail or physics irrespective of the system size. Such limiting distributions arise from a rescaling procedure, which in physics is well known as "renormalization". Thus we expect the strength distributions to scale with system size so as to converge to one of the limiting distributions of extreme statistics. Engineers have traditionally used the Weibull one of these to describe fracture. However, we discovered by the very high quality data obtained, up to 10^5 samples for a given case (disorder, system size) that this is not true. Duxbury, Beale, and Leath argued in the 1980s that the limiting one should actually be the Gumbel one. Our simulations show this to be correct, but even more importantly they allow us to capture the tail-behaviour of the extreme statistics of the problem. In other words, as is the case for the Gaussian or normal distribution for finite samples (finite "N") the tails of the probability distribution do not follow the asymptotic form, since the convergence does not apply there yet. We were able to establish this thanks to the numerical results, and moreover to obtain motivated by this discovery several important estimates of the deviations, in particular in the important low-strength tail of the distribution.

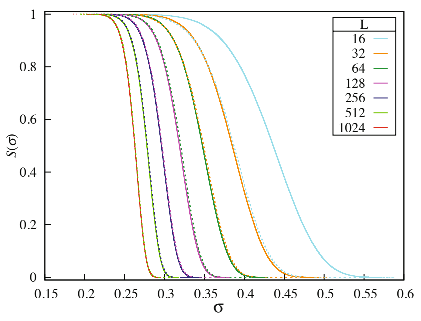


Fig. 1. Testing the weakest link hypothesis. Comparing the survival probability for a L^2 network (solid lines) with that predicted by the weakest link hypothesis or renormalization, $S^4_{L/2}$, (dotted lines) for one strength of material disorder. Note the excellent agreement even for moderate system sizes.

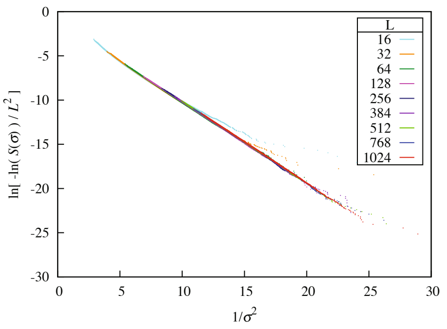


Fig. 2. Testing the Duxbury-Beale-Leath distribution of failure stresses. A collapse of the strength distribution for different system sizes at the same disorder as in Fig. 1, such that the DBL form would collapse onto a straight line.

Acknowledgements. Claudio Manzato (Universitá di Modena, Italy and Aalto), science done in collaboration with Stefano Zapperi (CNR, Milan, Italy), James Sethna (Cornell), Phani Nukala (Oak Ridge), Ashivni Shekhawat (Cornell).

References

1. Manzato, C., et al.: Fracture strength of disordered media: Universality, interactions and tail asymptotics. Physical Review Letters 108, 065504 (2012)
2. Alava, M.J., Nukala, P.K.N.N., Zapperi, S.: Statistical Models of Fracture. Advances in Physics 55, 349–476 (2006)

3 Dynamics of Black Holes (Cardoso[3])

3.1 Introduction

The two-body problem in General Relativity has been an outstanding problem since Einstein's original formulation of the theory, which was solved satisfactorily only recently; in 2005 several groups were able to numerically solve Einstein's equations and generically evolve black hole (BH) binaries for the first time [1]. This breakthrough paved the way for an exciting journey into fundamental physics, astrophysics, high energy physics and particle physics [2]. The PRACE/DECI project of the Lisbon group aimed at substantially advancing the state-of-the-art in several aspects.

3.2 Astrophysics

Stellar-mass or supermassive BHs are an important component of most galaxies. They are thought to interact (at significant rates) in two-body processes, making them the most attractive source of gravitational waves to be observed with interferometers such as LIGO, VIRGO, TAMA, etc. There has been significant effort and progress in understanding the inspiral and merger of equal-mass BHs; unfortunately, the more realistic case of extreme mass ratio binaries is technically challenging due to the large number of scales in the problem. We have partially solved this problem by considering the head-on collisions of highly unequal mass BH binaries [3]. In Fig. 3 we show waveforms (Newman-Penrose scalar Ψ_4) for the full nonlinear problem, together with a perturbative, point particle (PP) calculation [4]. The overall good agreement for waveforms demonstrates that numerical techniques are capable of bridging the gap between linear analysis and the fully non-linear regime of general relativity. For more details we refer the reader to Ref. [3].

3.3 Fundamental Physics

The ability to collide BHs at arbitrary speed opens up the possibility to test the Cosmic Censorship Conjecture (CCC): are black holes always the outcome of such collisions or can one form naked singularities? In particular BHs spin slowly, with angular momentum J satisfying the constraint $Jc/(GM^2) \leq 1$. where G is Newton's constant, c is the light speed and M the BH's mass. The high energy, finite-impact collision of two BHs is a prime candidate to give rise to an object other than a BH, potentially violating the CCC. Are BHs destroyed in high-energy collisions? Our results show a fascinating outcome: potentially hazardous (for the CCC) collisions radiate their excess angular momentum in zoom-whirl orbits before merging. In other words, BHs that approach each other with too large an angular momentum, zoom out while radiating this excess and are then "allowed" to merge. This is indicated in Fig. 4. The CCC is not violated [5,6].

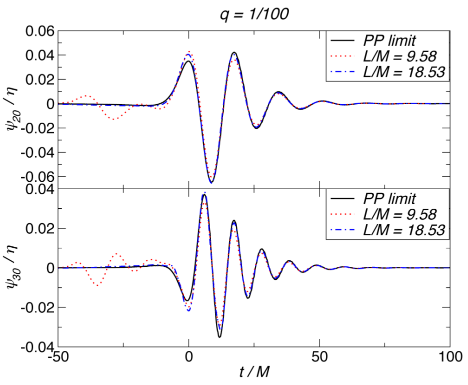


Fig. 3. (Color online) Waveforms for head-on collisions of two BHs with masses M_1, M_2 and mass ratio $q \equiv M_1/M_2 = 1/100$, with $\eta = M_1 M_2/M^2$. We have decomposed the waveform in spin-2 spherical harmonics. The modes are shown for $l = 2$ (upper panel) and $l = 3$ (lower panel), for two different initial separations. Also shown is the waveform in the PP limit (black solid lines). Taken from [3].

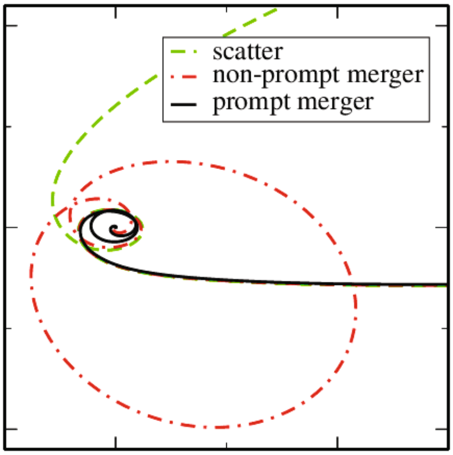


Fig. 4. Puncture trajectories of two BHs thrown at each other with $v = 75\% c$. Here we show the trajecotry of a single BH, for a scattering orbit ($b[= 3.40M] > b_{\rm scat}$), a prompt merger ($b[= 3.34M] < b^*$) and a nonprompt merger ($b^* < b[= 3.39M] < b_{\rm scat}$). Taken from Ref. [6]. The zoom-whirl orbits typically radiated excess angular momentum before merger.

3.4 High Energy Physics

The high-energy processes described above are directly relevant for many HEP scenarios, including TeV-scale gravity and the gauge/gravity scenarios. In the former, BHs can be created from point particle collisions in accelerator experiments. The ATLAS team at CERN is actively looking for BH signatures from such events [2]; one crucial input for these searches is the production cross-section, i.e, the critical impact parameter to produce a BH. This calculation requires the full nonlinear numerical evolution of Einstein's equations. Preliminary results in four-dimensional asymptotically flat spacetime yield $b_{\rm scat} \sim 2.5(M/v)$ [7,5,6]. The total gravitational radiation released in such collisions can go up to 35% of the CM energy or higher, making these the most (radiative-)efficient processes known to mankind. The extension of these results to higher dimensions is on-going [8,9,10].

3.5 Particle Physics

Finally, an unexpected mechanism of using dynamical BHs to study particle physics was recently uncovered. Rotating BHs display an interesting effect known as "superradiance," whereby an incident beam of light gets scattered with higher amplitude. This happens at the expense of the hole's kinetic energy: after the reflection, the BH spin decreases. If the scattered wave is massive, the entire setup produces a "black hole bomb": the scattered and amplified beam gets re-sent into the BH by a mass term. This leads to an exponential energy extraction cascade from the BH, that would extract energy from the black hole very quickly. Therefore the very existence of such particles is constrained by the observation of spinning black holes. Supermassive BHs can be used to measure the mass of extremely light particles to unprecedented levels and rule out the existence of new exotic particles, perhaps constraining the nature of dark matter. With this technique we have succeeded in constraining the mass of the photon to unprecedented levels: the mass must be smaller than 10^{-20} eV, or one hundred times better than the current bound [11,12]. To put this in context, this mass is one hundred billion billion times smaller than the present constraint on the neutrino mass (~2eV).

Acknowledgements. Vitor Cardoso's work was supported by the DyBHo–256667 ERC Starting Grant.

References

1. Sperhake, U., Berti, E., Cardoso, V, arXiv:1107.2819 [gr-qc]
2. Cardoso, V., Gualtieri, L., Herdeiro, C., Sperhake, U., Chesler, P.M., Lehner, L., Park, S.C., Reall, H.S., et al., arXiv:1201.5118 [hep-th]
3. Sperhake, U., Cardoso, V., Ott, C.D., Schnetter, E., Witek, H.: Phys. Rev. D 84, 084038 (2011), arXiv:1105.5391 [gr-qc]
4. Davis, M., Ruffni, R., Press, W.H., Price, R.H.: Phys. Rev. Lett. 27, 1466 (1971)
5. Sperhake, U., Cardoso, V., Pretorius, F., Berti, E., Gonzalez, J.A.: Phys. Rev. Lett. 101, 161101 (2008), arXiv:0806.1738 [gr-qc]
6. Sperhake, U., Cardoso, V., Pretorius, F., Berti, E., Hinderer, T., Yunes, N.: Phys. Rev. Lett. 103, 131102 (2009), arXiv:0907.1252 [gr-qc]

7. Shibata, M., Okawa, H., Yamamoto, T.: Phys. Rev. D 78, 101501 (2008), arXiv:0810.4735 [gr-qc]
8. Zilhao, M., Witek, H., Sperhake, U., Cardoso, V., Gualtieri, L., Herdeiro, C., Nerozzi, A.: Phys. Rev. D 81, 084052 (2010), arXiv:1001.2302 [gr-qc]
9. Witek, H., Zilhao, M., Gualtieri, L., Cardoso, V., Herdeiro, C., Nerozzi, A., Sperhake, U.: Phys. Rev. D 82, 104014 (2010), arXiv:1006.3081 [gr-qc]
10. Witek, H., Cardoso, V., Gualtieri, L., Herdeiro, C., Sperhake, U., Zilhao, M.: Phys. Rev. D 83, 044017 (2011), arXiv:1011.0742 [gr-qc]
11. Pani, P., Cardoso, V., Gualtieri, L., Berti, E., Ishibashi, A.: Phys. Rev. Lett. (in press), arXiv:1209.0465 [gr-qc]
12. Pani, P., Cardoso, V., Gualtieri, L., Berti, E., Ishibashi, A., arXiv:1209.0773 [gr-qc]

4 Framework to Run Ensemble Climate Simulations (Asif[4])

4.1 Introduction

A typical climate forecast experiment is a run of a climate model having variable range of forecast length from a few months to a few years. Such an experiment may have one or more than one start-dates and every start-date may comprise of single or many members. The full length of forecasting period for the experiment could be divided into number of chunks of fixed forecast length by exploiting the model restart options. Furthermore, in the context of computing operations, every chunk could have two big sections; a parallel section where the actual model run would be performed and a serial section for performing other necessary operations like post-processing of the model output, archiving the model output and cleaning the disk space for the smooth proceeding of the experiment.

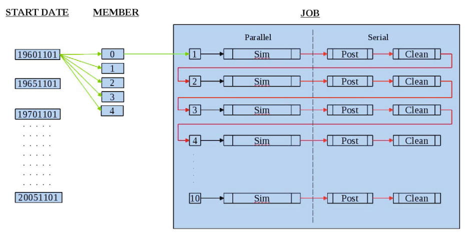


Fig. 5. Sample experiment setup

Fig. 5 shows a sample experiment where ten start-dates and five members are under consideration and each start-date and member is being run for ten years. Many EC-Earth partners run simulations (Sim) using 10 chunks of one year forecast length, with accompanying post-processing (Post) and cleaning (Clean) jobs. In this fashion, the experiment will be made of 50 independent simulations, each submitting 30 jobs (10 Sim, 10 Post and 10 Clean) with specific dependencies between them. In short, there is high need of a system to automate such types of typical experiments in order to optimize the utilization of computing resources.

4.2 Autosubmit

IC3 has developed Autosubmit, which is a tool to manage and monitor climate forecasting experiments by using supercomputers remotely. It is designed with the following goals:

1. Supercomputer-independent framework to perform experiments
2. Efficient utilization of available computing resources on supercomputers
3. User-friendly interface to start, stop and monitor experiments
4. Auto restarting the experiment or some part of experiment
5. Ability to reproduce the completed experiments

The current version of Autosubmit has an object-oriented design and uses Python as its programming language and SQLite as a database. Autosubmit acts as a wrapper over the queuing system of a supercomputer remotely via ssh. So far, queuing systems such as PBS, SGE and SLURM has been tested with it. In an experiment, as a first step Autosubmit creates the entire sequence of jobs and thereafter submits and monitors the jobs one by one after resolving dependencies among them until the end of sequence.

The development of Autosubmit is quite relevant compared with other similar tools such as SMS and ecFLOW (developed at ECMWF). The main idea is to increase the portability and improve the interactions with other systems/tools such as PRACE/ENES tools (e.g. SAGA) and METAFOR (e.g. CIM).

4.3 Wrapping Exercise

Currently supercomputing centres are increasing their computing capacity such as number of cores, etc. Meanwhile, the rules to make use of those resources are also becoming more strict. For example running the model on a supercomputer where the minimum scalability is restricted (e.g. PRACE Tier-0 machines: 8,192 cores at JUGENE, 4,096 at SuperMUC, HERMIT and MareNostrum with 2,048 minimum number of cores, or US DOE INCITE project: 60,000 cores at Oak Ridge Leadership Computing Facility (OLCF)). Hence, as it is difficult to scale the current version of EC-Earth beyond a few hundred cores there is a need to adopt some mechanism to deal with minimum scalability restrictions on supercomputers.

Therefore, in order to provide a solution to the climate community for the restricted scalability issue, a job wrapping exercise has been made using Lindgren (PDC supercomputer) where several jobs are wrapped at the same time by using python threading techniques. Say for example, 10 jobs of 346 cores each could be run as a big single job of 3460 cores.

4.4 Future Work

Future Autosubmit will be a flexible platform, released under GNU/GPL license, prepared for running multi-model and multi-jobs in Tier-0 and Tier-1 machines:

– Explore options to implement wrapper to ensemble simulation jobs (this piece of work will be done by IC3 under IS-ENES2)
– Integration of HPCs using SAGA (Simple API for Grid Applications)
– BLISS-SAGA (a light-weight implementation for Python) comply with OGF (Open Grid Forum) standards (how to interact with the middleware)
– A number of adaptors are already implemented, to support different grid and cloud computing backends SAGA provides units to compose high-level functionality across distinct distributed systems (e.g. submit jobs from same experiment to different platforms)
– Documenting experiments on simplified METAFOR standards by using relational databases (MySQL)
– Designing a web front-end for experiment creation and monitoring (Django)
– Storing user-defined job dependency tree in XML Scheme file
– Installation package and open source license.

5 CP2K in PRACE (Carter[1], Bethune[1] and Statford[1])

5.1 Introduction

This short article summarises work undertaken during various different activities within PRACE, and the work is described in more detail in [1]. The article includes a very brief introduction to CP2K[2], and a brief discussion of CP2K as an archetypal mixed-mode code. It then describes some of the work done to introduce mixed-mode parallelisation into the code, and presents some results. It concludes with a mention of ongoing work.

CP2K performs atomistic and molecular simulations of solid state, liquid, molecular and biological systems. It is a Density Functional Theory code with support for both classical and empirical potentials. The code is freely available, and is GPL licensed.

5.2 A Mixed-Mode Strategy

The code was originally parallelised with MPI only. OpenMP has been incrementally added to the code to introduce mixed-mode parallelism. The idea is to use OpenMP to parallelise those areas of the code that consume the most CPU time. Setting up parallel regions is relatively cheap, allowing the creation of micro-parallel regions. There are various different strategies for communication between processes within a mixed-mode code. CP2K adopts what is arguably the cleanest approach, and the safest to implement, whereby MPI communication takes place only *outside* parallel regions.

The decision to introduce OpenMP to CP2K was motivated by a desire to improve both performance and scaling. Performance of the code was expected to improve for the following reasons: First, using a mixed-mode approach should reduce the impact of those algorithms that scale poorly with the number of MPI tasks. For example, when using T threads, the switchover point from where it is necessary to use a less efficient 2D-decomposed FFT (as opposed to a more efficient 1D version) is increased by a factor of T. Second, better load balancing is to be expected. Existing MPI load balancing algorithms do a coarse load-balance. Finer-grained balance can then be achieved over OpenMP threads. Finally, there should be a significant reduction in the number of messages. This was particularly true on pre-Gemini networks, and the less sophisticated networks found on standard clusters. For all-to-all communications, the message count should be reduced by a factor of T^2.

As an example of some of the optimisation work on CP2K undertaken in PRACE, it was found that the calculation of the core Hamiltonian matrix could take a significant amount of time for certain calculations, particularly those with large basis sets. There was no existing OpenMP in this part of the code, so adding OpenMP to this part of the code was tried, motivated by the observations described above. The main change to the code was the introduction of a parallel region around a loop over all particles in a neighbour list: `DO WHILE (neighbor_list_iterate(nl_iterator)==0)`.

A challenge in such a parallelisation is determining the breakdown between shared and private variables. The choice as to which class the variables fall into was determined through inspection of the code. The code uses an iterator object to iterate over a fairly complex data structure, so it is difficult to tell *a priori* to what extent the iterations of the loop are independent. From examining output from instrumented test runs, and again through inspection of the code, it was determined that it was possible to break down the iterations of the loop into independent tasks, each corresponding to one or more iterations of the original loop.

The approach was to introduce a new data structure to hold the data describing a task and to iterate over the original data structure in serial, building an array of tasks. A parallel region was introduced around a new loop which loops over the independent tasks. In fact, the tasks are not completely independent, as they all update a shared array containing forces between particles.

The performance of the resulting code (for a representative input) is shown in Fig. 6. In general, performance only improves when using a small number of threads, but in this case at least, this is an improvement over the original MPI-only version. Considerable improvements were also made to other parts of the code. OpenMP was added for 11 cross-correlation functionals and improvements were made to a further 6. All twenty-six now use OpenMP. For the functionals to which OpenMP was added the efficiency was over 95% on a single NUMA region and up to 92% on the entire node. Also, three new grid-reduction strategies were implemented for a key ‘collocate’ kernel. The best result was a $\sim 50\%$ speedup over the existing version when running on 24 threads. A further inefficiency was

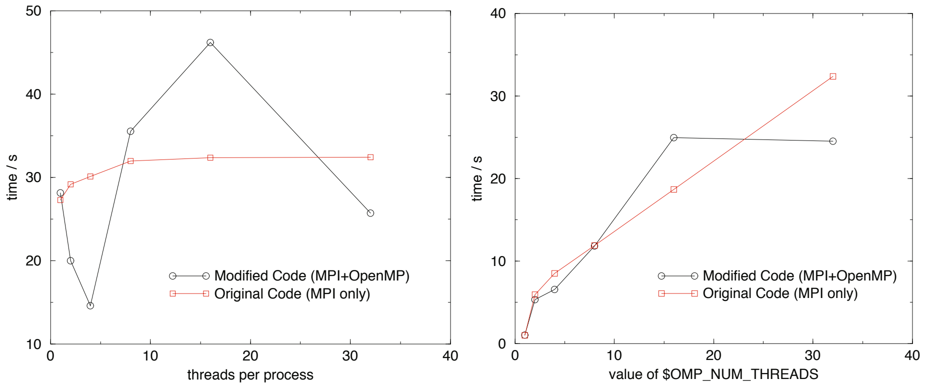


Fig. 6. Performance comparisons of original and modified code (left) 16 MPI processes, varying number of threads; (right) 512 total cores,varying number of MPI processes and threads. System: HECToR. Benchamrk: molopt. Details in [1].

identified in memory re-use, giving a $\sim$ 400% speedup for the routine as a whole. Details are available in [1].

References

1. Bethune, I., Carter, A., Stratford, K., Korosooglou, P.: CP2K Scalable Atomistic Simulation for the PRACE Community (2012)
2. VandeVondele, J., Krack, M., Mohamed, F., Parrinello, M., Chassaing, T., Hutter, J.: Quickstep: fast and accurate density functional calculations using a mixed Gaussian and plane waves approach. Comp. Phys. Comm. 167, 103 (2005)

6 Synthetic Seismograms for a Synthetic Earth – Joint Modeling of Mantle Flow, Mineral Physics and 3D Seismic Wave Propagation (Schuberth[5])

6.1 Introduction

Section 6 is a shortened and modified version of [1]. Long-standing questions in the study of Earth's deep interior are about the origin of seismic heterogeneity and the nature of flow in the mantle. Understanding the dynamic behaviour is important as the flow drives plate tectonics and controls the way the Earth looses its heat. Thus, it is a crucial factor in tectonic modelling or in simulations of the geodynamo and the thermal evolution of the Earth. A better understanding of these aspects is also of great societal importance. For example, the continuous drift of tectonic plates relative to each other results in a build up of stress at the plate boundaries. This stress can eventually exceed the yield stress of rock thus leading to (often disastrous) earthquakes.

In order to improve conceptual models of mantle flow, the major challenges today are to efficiently mine the wealth of information contained in seismic waveforms (which are our main source of information on Earth's deep interior) and to constrain the relative contributions of thermal anomalies and compositional variations to the observed seismic heterogeneity. High expectations to gain new insight currently lie within numerical simulations of wave propagation through complex three-dimensional structures. Modern computational tools for seismic wave propagation incorporate a large range of physical phenomena and are able to produce synthetic datasets that show a complexity comparable to real observations. Also, computing whole waveform synthetic seismograms at relevant frequencies became feasible on a routine basis in recent years thanks to rapidly growing computational resources. However, it has long been not clear how to introduce geodynamic considerations into seismological forward simulations in an efficient and consistent manner, and how to benefit from expensive large-scale simulations for our understanding of deep Earth structure and dynamics. This was the motivation to develop a novel method, in which we generate synthetic 3D mantle structures based on dynamic flow calculations that serve as input models in the simulation of seismic wave propagation.

Here, we present the results of this new multi-disciplinary approach that combines forward modelling techniques from geodynamics, mineral physics and seismology. The thermal state of Earth's mantle at present-day geologic time is predicted by 3D high-resolution mantle circulation models using a finite-element method. The temperature field is then mapped to seismic velocities. For this task, we take advantage of recent progress in describing the state of dynamic Earth models in terms of elastic properties through thermodynamically self-consistent models of mantle mineralogy. The predicted seismic structures are then implemented in a spectral element code for the simulation of 3D global wave propagation [2]. Using state-of-the-art techniques to solve the wave equation in 3D heterogeneous media, this approach allows us to capture the full physics of wave propagation.

Both the geodynamic as well as the seismic simulations require large-scale high-performance calculations. The computational resources provided through the DECI-5 call offered by DEISA allowed for the first time to simulate seismic wave propagation in synthetic Earths; that is, we are now able to compute synthetic seismograms completely independent of seismic observations. This means that we can test geodynamic hypotheses directly against seismic observations, which may serve as a complementary tool to tomographic inversions. More specifically, it is for the first time possible to study frequency-dependent waveform effects, such as wavefront healing and focusing/defocusing in mantle structures with realistic length-scales; that is, in a physically consistent manner.

6.2 Results

One specific question that we addressed with our joint forward modelling approach is the origin of two large regions of strongly reduced seismic velocities in the lowermost mantle (the so-called African and Pacific superplumes). Several

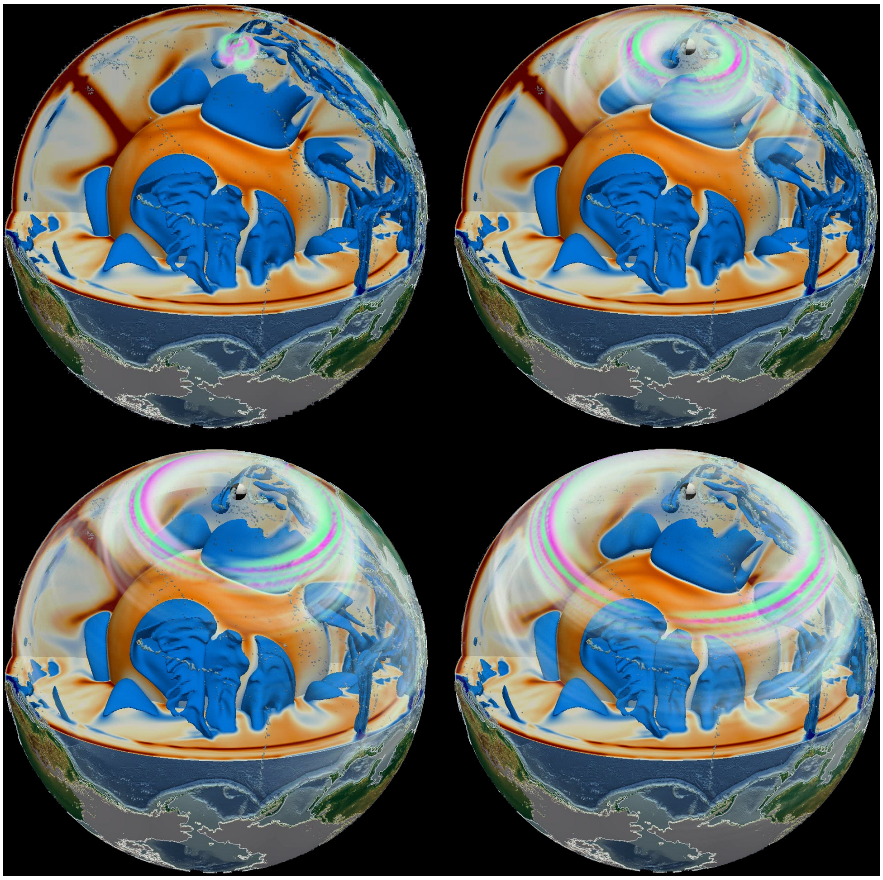


Fig. 7. Snapshots of the three-dimensional wavefield in one of our geodynamic models. 3D global wave propagation was simulated for an earthquake in the Fiji Islands region using a spectral element technique. The wavefield is depicted by green and magenta colours together with the shear wave velocity variations in the model, for which vertical cross-sections and iso-surfaces are shown on a blue to brownish colour scale ranging from -2% to 2%. Surface topography is also shown for parts of the globe for geographic reference [Schuberth et al., 2012].

seismological observations are typically taken as an indication that the superplumes are being caused by large-scale compositional variations and that they are piles of material with higher density than normal mantle rock. However, a large number of recent geodynamic, mineralogical and also seismological studies argue for a strong thermal gradient across the core-mantle boundary (CMB) that might provide an alternative explanation through the resulting large lateral temperature variations. We tested the hypothesis whether the presence of such a strong thermal gradient in isochemical whole mantle flow is compatible with geophysical observations.

We have computed the 3D seismic wavefield and synthetic seismograms for a total of 17 earthquakes distributed evenly over the globe. To obtain the necessary numerical accuracy for the period range of interest (i.e., down to a shortest period of 10s), we used a spectral element mesh with around 1.3 billion grid points and 3.7 billion degrees of freedom distributed on 486 compute cores of the supercomputing facility HLRB2 of the Leibniz Supercomputing Centre (LRZ). The wavefield of each earthquake was "recorded" by a very large number of virtual seismic stations in order to achieve a relatively homogeneous illumination of our model even with a low number of seismic sources. From the synthetic seismograms, we obtained ∼350,000 traveltimes each for compressional (P) and shear (S) waves by an automated cross-correlation measurement technique.

The results from our full wavefield simulations demonstrate that P- and S-wave traveltime variations in our geodynamic model are compatible with the observed seismic data: The standard deviation of P-wave traveltime variations

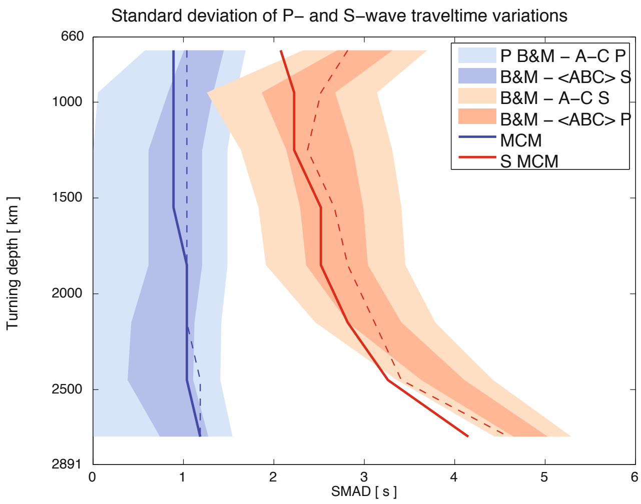


Fig. 8. Comparison of the standard deviation (SMAD = scaled median average deviation) of traveltime variations in our geodynamic model to that of the observations. Intermediate and light shaded areas show the range of values inferred from the data [2]. Blue lines: simulated P-wave traveltime variations. Red lines: same for S-waves. Solid and dashed lines show SMAD curves for two different measurement techniques [3].

stays almost constant with depth in the mantle, while that of the S-wave traveltimes increases strongly towards the CMB (cf. Fig. 8). Most important, the standard deviations of our synthetic P- and S-wave traveltimes do not only show different trends with depth, but are also matching those of the observations well in terms of their magnitude. This is a remarkable result, as it shows that isochemical whole mantle flow with strong core heating and a pyrolite composition can be reconciled with seismic observations. While this finding does not necessarily mean that there is no chemical heterogeneity present in the lower mantle, it shows that complex large-scale variations in chemical composition are not required by the dataset studied here.

References

1. Schuberth, B.S.A., et al.: Simulation of 3-D seismic wave propagation in a synthetic Earth. In: Wagner, S., Bode, A., Satzger, H., Brehm, M. (eds.) High-Performance Computing in Science and Engineering, Garching/Munich 2012, Bayerische Akademie der Wissenschaften (2012)
2. Schuberth, B.S.A., Zaroli, C., Nolet, G.: Geophys. J. Int. 188(3), 1393–1412 (2012)
3. Bolton, H., Masters, G.: J. Geophys. Res. 13(B7), 13,527–13,540 (2001)

7 A Case Study for the Deployment of CFD Simulation Components to the Grid(Weinzierl[6])

IO is predicted to become a major challenge on exascale computers [1]. The IO facilities of future computer generations will not scale with the increasing compute power, the increasing number of cores [1,2,3,4], and the increasing complexity and data richness of exascale simulations. Already today, loading data to the supercomputer and downstreaming data from the simulation code requires significant time. Already today it is thus possible to study and tackle the IO challenge and to study strategies of how to avoid IO-bound applications [3]. With DEISA, the deployment of CFD simulations, for example due to Grid interfaces, is straightforward and the information where insight is computed can, theoretically, be hidden, Our work studies how to get the data to and from a Grid-like environment.

For the solution of partial differential equations with mesh-based methods, one approach is, on the one hand, to generate the meshes and, hence, study data in-situ. On the other hand, it is natural to examine approaches of downstreaming only data really of interest to the scientists. Analogous to the *more science per flop* [5], the aim here is *more insight per megabyte.* The latter comprises traditionally in-situ visualisation and postprocessing, but it also induces use of all IO facilities efficiently and reduction of the memory footprint of the streamed data.

In the present DEISA project, we study, on the methodological side, the impact of octree-based mesh to the IO challenge. Our generalised octree approach embeds the computational domain into a unit hypersquare and divides equidistantly this hypersquare into k pieces along each coordinate axis. Let d be the dimension. Then the result are k^d new subcubes and we can continue recursively while we decide independently for each subcube whether to continue or not. The spacetree generalising the octree idea beyond bipartition yields a cascade of adaptive Cartesian grids that are very simple both to encode and to generate in-situ. While spacetree-based approaches look back to a long tradition in computer graphics, they attract a lot of attention in particular for mesh generation and management on supercomputers (e.g., [6,7,8,9,10,11,12,13,14,15,18]).

In-situ visualisation and postprocessing is beyond the scope of the present DEISA project, and it also is a misfit to the traditional batch processing. However, it turns out that the spacetree method is nevertheless advantageous for the IO output. We propose not to downstream all the simulation data, but to replace the fire-and-forget data flow with a demand-driven approch: The user—in our example a simplified fluid-structure interaction code—specifies the region of interest; typically around the structure in this case. There might be multiple regions of interest formalised by queries. A query comprises the spatial region of interest, which subset of data (only pressure values in CFD, e.g.) are relevant, and in which resolution the data shall be downstreamed [16]—it describes a regular Cartesian grid. Furthermore, we augment each query with an identifier to which software component on which computer aims to postprocess these data [17].

The queries are distributed among the spacetree. Now, the multiresolution tessellation pays off: queries can be decomposed along any domain decomposition, they can be mapped to a spacetree refinement level corresponding to the query resolution, and the mapping of spacetree-associated data to a query is trivial due to the simplicity of the grids used. Queries befilled by different compute nodes due to a domain decomposition can be merged directly on the supercomputer again, before they are sent back to the postprocessing device. This way, we deliver exclusively data that is of relevance, we deliver data in the required resolution and accuracy, and we provide low overhead answers—there is no simpler data structure than a Cartesian grid.

Such an approach can be tailored to the IO topology of a supercomputer. Queries decompose along the domain decomposition, i.e. each compute node might befill only subparts of a query answer. While we do merge these subparts on the supercomputer, we ensure that the data is merged such that each merging node has an IO node exclusively (if possible) i.e. answers to multiple queries do not compete for IO resources. Furthermore, not all data merges have necessarily to happen on the supercomputer—it might pay off to merge some subqueries, send incomplete answers to the postprocessing nodes, and to merge the results there. Such a dataflow scenario is advantageous if the individual fragments sent back can be sent due to multiple IO devices. First experiments reveal that with a tailoring to the IO topology and architecture as well as a demand-driven data flow that streams exclusively the data required, one can even make a great step towards on-the-fly visualisation of supercomputer simulations.

The octree paradigm's multiscale mesh representation furthermore enables the data management to exploit smoothness and multiscale properties of any data streamed. Following ideas of full approximation storage ([19], e.g.), any data mapped onto a spacetree can be induced to any level of the tree: if a solution on a fine adaptive Cartesian grid is given, simple induction, i.e. copying data from fine grids to coarser within the tree, yields multiple solution resolutions. It then is straightforward to switch from a nodal data representation to a hierarchical one. Data is not stored as-is, but we suggest to store on each level only the difference to the coarser levels: the hierarchical surplus [20,21]. While there are multiple advantages of such a storage schemes, the interesting property for IO streaming is that the gain in accuracy per level usually is limited and can be analysed. Typically, only few bits of the hierarchical surplus carry relevant additional information compared to coarser resolutions. It is thus straightforward, to hold only these bits, i.e. not to work with full double precision but with restricted accuracy [22].

References

1. Dongarra, J., Beckman, P.H.: The International Exascale Software Project roadmap. IJHPCA 25(1), 3–60 (2011)
2. Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R., Ward, L., Sadayappan, P.: Scalable I/O forwarding Framework for High-Performance Computing Systems. In: IEEE International Conference on Cluster Computing and Workshops, CLUSTER 2009, pp. 1–10 (2009)
3. Fabian, N., Moreland, K., Thompson, D., Bauer, A., Geveci, B., Rasquin, M., Jansen, K.: The ParaView Coprocessing Library: A scalable, general purpose in situ visualization library. In: LDAV 2011 IEEE Symposium on Large-Scale Data Analysis and Visualization (2011)
4. Iskra, K., Romein, J.W., Yoshii, K., Beckman, P.: I/O-forwarding infrastructure for petascale architectures. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008) ISBN: 978-1-59593-795-7
5. Keyes, D.E.: Euro-Par 2000: Four Horizons for Enhancing the Performance of Parallel Simulations Based on Partial Differential Equations. In: Proceedings from the 6th International Euro-Par Conference on Parallel Processing, pp. 1–17 (2000) ISBN: 3-540-67956-1
6. Weinzierl, T., Mehl, M.: Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. SIAM Journal on Scientific Computing 33, 2732–2760 (2011) ISSN: 1064-8275
7. Weinzierl, T.: A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids (Dissertation) ISBN: 9783868531461 (2009)
8. Burstedde, C., Ghattas, O., Gurnis, M., Stadler, G., Tan, E., Tu, T., Wilcox, L.C., Zhong, S.: Scalable adaptive mantle convection simulation on petascale supercomputers. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–15 (2008) ISBN: 978-1-4244-2835-9
9. Burstedde, C., Ghattas, O., Stadler, G., Tu, T., Wilcox, L.C.: Towards Adaptive Mesh PDE Simulations on Petascale Computers. In: Proceedings of Teragrid 2008 (2008)
10. Griebel, M., Zumbusch, G.W.: Hash-Storage Techniques for Adaptive Multilevel Solvers and their Domain Decomposition Parallelization. In: Proceedings of Domain Decomposition Methods 10, DD10, vol. 218, pp. 279–286 (1998)

11. Griebel, M., Zumbusch, G.: Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. In: Parallel Computing, vol. 25(7), pp. 827–843 (1991) ISSN: 0167-8191
12. Sundar, H., Biros, G., Burstedde, C., Rudi, J., Ghattas, O., Stadler, G.: Parallel Geometric-Algebraic Multigrid on Unstructured Forests of Octrees. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2012 (2012)
13. Burstedde, C., Wilcox, L.C., Ghattas, O.: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. SIAM Journal on Scientific Computing 33(3), 1103–1133 (2011)
14. Bangerth, W., Burstedde, C., Heister, T., Kronbichler, M.: Algorithms and Data Structures for Massively Parallel Generic Adaptive Finite Element Codes. ACM Transactions on Mathematical Software 38(2) (2011)
15. Sampath, R.S., Adavani, S.S., Sundar, H., Lashuk, I., Biros, G.: Dendro: parallel algorithms for multigrid and AMR methods on 2: 1 balanced octrees. In: SC, p. 18 (2008)
16. Atanasov, A., Weinzierl, T.: Query-driven Multiscale Data Postprocessing in Computational Fluid Dynamics. Procedia CS 4, 332–341 (2011)
17. Atanasov, A., Srinivasan, M., Weinzierl, T.: Query-driven Parallel Exploration of Large Datasets. In: IEEE LDAV (2012)
18. Sundar, H., Sampath, R.S., Biros, G.: Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel. SIAM J. Sci. Comput. 30(5), 2675–2708 (2008) ISSN: 1064-8275
19. Trottenberg, U., Schuller, A., Oosterlee, C.: Multigrid (2000) ISBN: 9780127010700
20. Griebel, M.: Zur Lösung von Finite-Differenzen- und Finite-Element-Gleichungen mittels der Hiearchischen-Transformations-Mehrgitter-Methode (dissertation), vol. 342/4/90 A (1990)
21. Griebel, M.: Multilevelmethoden als Iterationsverfahren über Erzeugendensystemen. In: Teubner Skripten zur Numerik (1994)
22. Bungartz, H.-J., Eckhardt, W., Weinzierl, T., Zenger, C.: A Precompiler to Reduce the Memory Footprint of Multiscale PDE Solvers in C++. Future Generation Computer Systems 26, 175–182 (2010) ISSN = 0167-739X

Part II

Advances in HPC Applications

Parallel Electronic Structure Calculations Using Multiple Graphics Processing Units (GPUs)

Samuli Hakala, Ville Havu, Jussi Enkovaara, and Risto Nieminen

COMP Center of Excellence, Department of Applied Physics, School of Science, Aalto University,
P.O. Box 11100, FI-00076 AALTO, Finland
samuli.hakala@aalto.fi

Abstract. We present an implementation of parallel GPU-accelerated GPAW, a density-functional theory (DFT) code based on grid based projector-augmented wave method. GPAW is suitable for large scale electronic structure calculations and capable of scaling to thousands of cores. We have accelerated the most computationally intensive components of the program with CUDA. We will provide performance and scaling analysis of our multi-GPU-accelerated code staring from small systems up to systems with thousands of atoms running on GPU clusters. We have achieved up to 15 times speed-ups on large systems.

Keywords: electronic structure calculations, density functional theory, graphics processing units.

1 Introduction

Various electronic structure calculations are a large consumer of supercomputing resources around the world. Density functional theory (DFT) is a popular method for *ab-initio* electronic structure calculations in material physics and quantum chemistry. There exists several program packages and codes intended for these kinds of simulations. We have implemented an accelerated version of the GPAW code using multiple graphics processing units (GPUs). GPAW [1, 2] is a DFT program package based on the projector augmented wave (PAW) method. It is suitable for large scale parallel simulations and it is used by several research groups world wide. Time-depended density-functional theory is implemented in the linear response and in the time propagation schemes. In this paper we describe the work done in implementing the most computationally intensive routines in GPAW with GPUs. We also present and analyze the performance of our implementation.

A modern GPU is an efficient stream processor suitable for parallel computations. In the last few years the usage of GPUs in scientific calculations and in high performance computing has increased considerably. In a recent Top500 list [3] (11/2012) 62 out of the top 500 supercomputers in the world used accelerators or co-processors with 50 of these employing NVIDIA GPUs. Several computer codes in physics and chemistry have already been modified or written from scratch to take advantage of GPUs. More information on the use of GPUs in computational physics can be found for example in a review paper by Harju *et al.* [4].

In DFT simulations numerical approximations are needed. They relate to the treatment of the core electrons and to the discretization of the equations. The most common discretization methods are localized orbitals, plane waves, real space grids and finite elements. Normally an iterative minimization procedure is used to find the solution to the problem starting from an initial guess [5]. Depending on the discretization method, the known numerical bottlenecks are vector operations, matrix products, stencil operations and fast Fourier transforms. These computationally intensive parts are prime targets for GPU acceleration. However in order to attain high performance is it usually also necessary to implement GPU versions of a lot of the non-intensive routines.

Previously GPUs have been used in DFT calculations based on Gaussian type orbitals [6–10], wavelet basis sets [11], plane waves [12–16] and to some extent real space grids [17]. But to our knowledge this is the first DFT implementation running on large GPU clusters using real-space grids and the PAW method.

2 Overview of GPAW

We will give only a short overview on GPAW and the PAW [18] method. Atomic units are used in all equations. We use the DFT formulation introduced by Kohn and Sham [19], where the problem of interacting electrons is mapped to one with non-interacting electrons moving in an effective potential so that the total electron density is the same as in the original many-body problem [20]. The single-particle Kohn-Sham (KS) wave functions $\psi_n(\boldsymbol{r})$ are solutions to the equation

$$H\psi_n(\boldsymbol{r}) = \epsilon_n \psi_n(\boldsymbol{r}), \tag{1}$$

where the Hamiltonian is $H = -\frac{1}{2}\nabla^2 + v_H(\boldsymbol{r}) + v_{ext}(\boldsymbol{r}) + v_{xc}(\boldsymbol{r})$. The last three terms in the Hamiltonian define the effective potential, consisting of the Hartree potential v_H defined by the Poisson equation $\nabla^2 v_H(\boldsymbol{r}) = -4\pi\rho(\boldsymbol{r})$, external ionic potential v_{ext} and the exchange-correlation potential v_{xc}. The exchange-correlation potential contains all the complicated electron interactions that the KS formulation hides and in practical calculations it is approximated. The electronic charge density $\rho(\boldsymbol{r})$ is determined by the wave functions $\rho(\boldsymbol{r}) = \sum_i f_i |\psi_i(\boldsymbol{r})|^2$, where the f_i:s are the orbital occupation numbers.

PAW method is based on a linear transformation $\hat{\mathcal{T}}$ between smooth valence pseudo wave functions $\tilde{\psi}_n$ and all electron Kohn-Sham wave functions ψ_n. Core states of the atoms $\phi_i^a(\boldsymbol{r})$ are frozen. The transformation operator $\hat{\mathcal{T}}$ can be constructed from the atom centered all electron wave functions $\phi_i^a(\boldsymbol{r})$, the corresponding smooth partial waves $\tilde{\phi}_i^a(\boldsymbol{r})$ and the projector functions $\tilde{p}_i^a(\boldsymbol{r})$. The transformation is exact with infinite number of partial waves and projector functions. In practical calculations one or two functions per angular momentum channel are usually enough.

In PAW formalism KS wave functions can be expressed as

$$\psi_n(\boldsymbol{r}) = \tilde{\psi}_n(\boldsymbol{r}) + \sum_a \Big(\psi_n^a(\boldsymbol{r} - \boldsymbol{R}_a) - \tilde{\psi}_n^a(\boldsymbol{r} - \boldsymbol{R}_a)\Big) \tag{2}$$

where ψ_n^a and $\tilde{\psi}_n^a$ are the all electron and the smooth continuation of the wave function ψ_n inside the augmentation region of the atom a at position $\boldsymbol{R}_a$. The functions ψ_n^a and $\tilde{\psi}_n^a$ may be expressed in terms of projector functions and atom centered wave functions

$$\psi_n^a = \sum_j P_{nj}^a \phi_j^a(\boldsymbol{r}) \tag{3}$$

$$\tilde{\psi}_n^a = \sum_j P_{nj}^a \tilde{\phi}_j^a(\boldsymbol{r}) \tag{4}$$

where the expansion coefficient in terms of projector functions are $P_{nj}^a = \langle \tilde{p}_j | \tilde{\psi}_n \rangle$.

Similarly we can construct the all-electron (AE) density in terms of smooth part and atom-centered corrections.

$$n(\boldsymbol{r}) = \tilde{n}(\boldsymbol{r}) + \sum_a (n^a(\boldsymbol{r}) - \tilde{n}^a(\boldsymbol{r})) \tag{5}$$

The pseudo electron density is defined as

$$\tilde{n}(\boldsymbol{r}) = \sum_n f_n |\tilde{\psi}_n(\boldsymbol{r})|^2 + \sum_a \tilde{n}_c^a(\boldsymbol{r}) \tag{6}$$

where f_n are the occupation numbers and $\tilde{n}_c^a$ is a smooth pseudo core density. With the atomic density matrix

$$D_{i_1,i_2}^a = \sum_n \langle \tilde{\psi}_n | \tilde{p}_{i_1}^a \rangle f_n \langle \tilde{p}_{i_2}^a | \tilde{\psi}_n \rangle \tag{7}$$

the all electron density inside the augmentation sphere is expressed as

$$n^a(\boldsymbol{r}) = \sum_{i_1,i_2} D_{i_1,i_2}^a \phi_{i_1}^a(\boldsymbol{r}) \phi_{i_2}^a(\boldsymbol{r}) + n_c^a(\boldsymbol{r}) \tag{8}$$

and its smooth counterpart as

$$\tilde{n}^a(\boldsymbol{r}) = \sum_{i_1,i_2} D_{i_1,i_2}^a \tilde{\phi}_{i_1}^a(\boldsymbol{r}) \tilde{\phi}_{i_2}^a(\boldsymbol{r}) + \tilde{n}_c^a(\boldsymbol{r}) \tag{9}$$

An iterative procedure called Self-Consistent Field (SCF) calculation is used to find the solution to the eigenproblem (1) starting from an initial guess for the charge density. In GPAW the most time consuming parts of a single SCF-iteration are: solving the Poisson equation, iterative refinement of eigenvectors, subspace diagonalization and orthonormalization of wave functions.

The Hartree potential is obtained from the electron density by solving the Poisson equation using a multigrid algorithm [21]. Iterative updating of the eigenvectors is done with the residual minimization method – direct inversion in iterative subspace (RMM-DIIS) [22, 23]. Basically, at each step the wave functions are updated with the residuals.

$$R_n = (\hat{H} - \epsilon_n \hat{S})\tilde{\psi}_n \tag{10}$$

The convergence of this iteration is accelerated with the use of preconditioned residuals by solving approximately a Poisson equation $\frac{1}{2}\nabla^2 \tilde{R}_n = R_n$ with a multi-grid method [24]. A subspace diagonalization and the orthonormalization of eigenvectors is performed at each step.

GPAW uses uniform real space grids to discretize the KS equations. Wave functions (ψ_{nG}), potentials ($v_{H,G}$) and densities (ρ_G) are represented by their values at grid (G) points. Derivatives and Laplacians ($L_{GG'}$) are calculated using finite difference stencils. A coarse grid is used for wave functions and a fine grid for potentials and densities. A radial grid is used for the projector functions defined inside the augmentation sphere. The discretized Hamiltonian in PAW formalism is defined as

$$H_{GG'} = -\frac{1}{2}L_{GG'} + v_{eff}\delta_{GG'} + \sum_{i_1,i_2} p^a_{i_1 G} H^a_{i_1 i_2} p^a_{i_2 G'}, \tag{11}$$

where p^a_{iG} are the discretized projector functions and $H^a_{i_1 i_2}$ the PAW non-local atomic Hamiltonian contributions.

Subspace diagonalization requires applying the Hamiltonian operator to the wave functions and diagonalizing the resulting smaller Hamiltonian matrix.

$$H_{nn'} = \sum_G \psi_{nG} \sum_{G'} H_{GG'}\psi_{n'G'} = \sum_G \psi_{nG}(H\psi)_{n'G} \tag{12}$$

The wave functions are then multiplied by the matrix of eigenvectors

$$\psi'_{nG} = \sum_{n'} \Lambda^H_{nn'}\psi_{n'G}. \tag{13}$$

In orthonormalization an overlap matrix is constructed by applying overlap operator to the wave functions

$$S_{nn'} = \sum_G \psi_{nG} \sum_{G'} S_{GG'}\psi_{n'G'}. \tag{14}$$

This is then Cholesky decomposed and multiplied with the wave functions to obtain orthonormal wave functions

$$\psi'_{nG} = \sum_{n'} L^{-T}_{nn'}\psi_{n'G}. \tag{15}$$

Both the subspace diagonalization and orthonormalization also involve integrals of projector functions multiplied by the wave functions and addition of projector function multiplied by a matrix to the wave functions.

3 GPU Implementation

A NVIDIA Fermi [25] GPU architecture consists of several streaming multiprocessors, each with its own set of CUDA cores. Each core has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU) supporting both single and double precision floating point arithmetic. All multiprocessors have small amount of local memory and have an access to the global memory. Local memory is split between L1 cache and shared memory, which can be used as a user-managed cache. A global L2 cache is shared by all multiprocessors. High global memory latency is masked by executing thousands of threads concurrently. Local memory and registers are partitioned among currently executing threads.

GPAW is implemented using Python programming language with extensions written in C for the performance critical parts. Our GPU accelerated version uses the PyCUDA [26] toolkit to enable the use of GPU in Python code and several custom CUDA kernels [27] to speed up the GPAW C-extensions. We have used GPUs to speed up most of the performance critical parts of SCF iteration. All of our calculations use double precision arithmetic.

The Poisson equation is solved on a fine grid using a multigrid solver. The basic operations are: finite difference stencils for the Laplace operator and restriction and interpolation operations between coarser and finer grids. We have implemented CUDA kernels for all these operations and the entire Poisson solver is done with GPUs.

The 3D finite difference kernel processes the grid slice-by-slice [28]. We define global memory read redundancy as the ratio between output points written to and input points read from global memory. Global memory read redundancy is reduced by performing the calculations from shared memory. Each YZ-slice of the grid is divided into 2D thread blocks. Each thread reads one grid point from global memory to shared memory. Also data required for the stencil halos is added to shared memory. Each thread then calculates the finite difference operator for one grid point. For the YZ-slice data is read from the shared memory. Data required for the X-axis calculations is stored in registers for each thread. The working slice is then moved along the X-axis of grid to completely process the grid. Our implementation automatically generates custom CUDA kernels for each order-k stencils from a single C source code base. This is done to minimize the shared memory and register consumption and to completely unroll all the inner loops required for the finite difference operation. For small grids we also divide the grid into concurrent slices along the X-axis of grid to increase the number of threads performing calculations at the same time even though it hurts the overall read redundancy.

Figure 1 shows a performance comparison between CPU and GPU version of 3rd order finite difference operation for different grid sizes. The GPU used in testing was NVIDIA Tesla M2070 which has double precision performance of 515 GFLOPS and maximum memory bandwidth of 150 GB/s. The CPU used was Intel Xeon X5650 with six cores each with double precision performance of 10.64 (12.24 in turbo mode) GFLOPS and memory bandwidth of 32 GB/s. A single

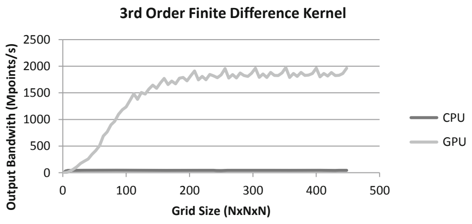


Fig. 1. The performance comparison of a 3rd order finite difference operator. GPU: NVIDIA Tesla M2070 CPU: Intel Xeon X5650 using a single core.

CPU core was used in the tests. For large grid sizes the GPU version of the code is over 40 times faster. The peak output performance for the kernel is around 1975 Mpoints/s. Ignoring boundary effects and using 16x8 thread blocks, the 3rd order finite difference operator (with 19 input elements for every output value) has global memory read redundancy of 2.1250. For each output value one global memory write and 2.1250 reads are performed, resulting in around 49 GB/s peak bandwidth usage for the whole kernel. For each output value 37 floating point operations are performed, which means that kernel has peak floating point performance at around 73 GFLOPS. Clearly for large grids the finite difference kernel is memory bandwidth bound.

For the restriction and interpolation operations we use similar strategy as for finite difference operations. Calculations are performed using combination of shared memory and registers. Figure 2 shows a performance comparison between CPU and GPU versions of the restriction and the interpolation operations for different grid sizes. All operations support real and complex grids and finite and periodic boundary conditions. The restriction, interpolation and the finite difference kernels all have the same problem: the output bandwidth for small grid sizes is much worse than for large grids. This can cause performance issues with multigrid methods. Also, in practical GPAW calculations one generally wants to use coarser grid sizes to speed up the calculations and to decrease memory consumption.

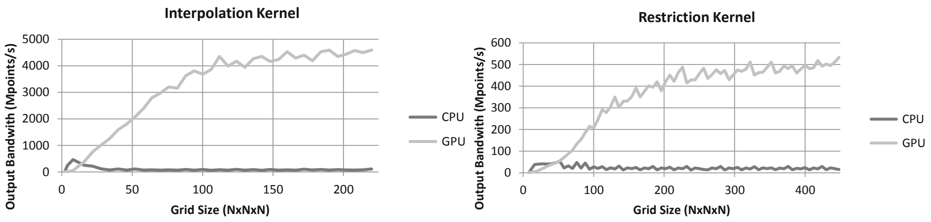


Fig. 2. The performance comparison of interpolation and restriction operators. GPU: NVIDIA Tesla M2070 CPU: Intel Xeon X5650 using a single core.

To avoid any slow and unnecessary transfers between the GPU and the host computer, during the SCF-iteration all the wave functions are stored in the GPU memory and the operations involving them are performed using the GPU. The iterative updating of the eigenvectors (RMM-DIIS algorithm) is performed entirely on GPUs. Most of the basic linear algebra operations are done with NVIDIA's CUBLAS library. Since large part of the operations is performed on all of the wave functions, we have implemented blocking versions of most of our kernels which allow us to update a block of eigenvectors simultaneously on a GPU. These include custom versions of several of the level 1 BLAS routines. For example, we have a implemented a custom blocking dot product using a GPU optimized parallel reduction sum kernel. The preconditioner uses same basic operations as the Poisson solver, but we use blocking versions of the restriction, the interpolation and the finite difference kernels. Figure 3 shows the effect of different block sizes on the operations used by the preconditioner. Especially on small grid sizes the use of blocking increases the performance considerably.

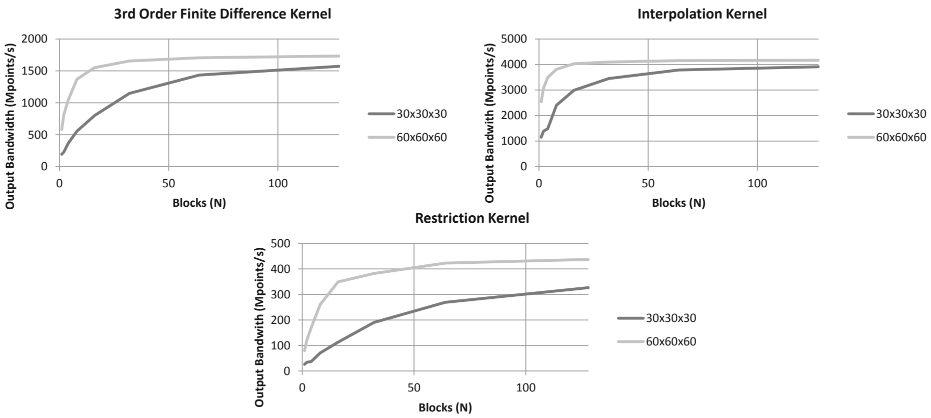


Fig. 3. The effect of different block sizes on the 3rd order finite difference, the interpolation and the restriction kernels on two small grid sizes

The most time consuming parts of subspace diagonalization and orthonormalization are matrix-matrix multiplications which are performed with CUBLAS on GPU. Also, the Hamiltonian operator and the overlap operator are applied to the wave functions on GPU. For the integrals of projector functions multiplied by the wave functions and addition of projector function multiplied by a matrix to the wave functions we have developed several custom CUDA kernels which perform blocking parallel integrations and additions between the projector functions defined on a dense radial grids and wave functions defined on coarse grids. We use GPU also for some smaller operations, like the calculation of the electron density from the wave functions.

The high-level parallelization of the code is done with MPI (Message Passing Interface). Our GPU version supports multiple GPUs using domain decomposition of the real space grid or by parallelizing over **k**-points. For **k**-points the parallelization is almost trivial since the wave functions in different k-points don't interact numerically with each other. However, usually in periodic systems with several hundred atoms or more only one **k**-point is needed.

Domain decomposition for the finite difference stencils and restriction and integration operations involves communication with the nearest neighbor domains. In the GPU version this requires data movement from the device memory to the main memory, transferring the data to the destination node using MPI and moving the data from main memory to the device memory in the destination node. We have implemented several different approaches to speed up this process which involve overlapping receives, sends, memory transfers and computations in the middle part of the grids and blocking of several wave functions and boundary regions into few large transfers. These are used depending on the grid and the transfer sizes.

4 Performance Evaluation

We evaluated the performance and scalability of our code by comparing the ground state DFT calculation times between the GPU and the CPU versions of the code. For benchmarking purposed only a fixed number of SCF iterations (usually 10) were calculated for each system and an average time for a single SCF iteration was used to compare the performance. For small systems testing was performed with Vuori cluster at CSC, which has 7 GPU nodes connected to an Infiniband network. Each node has two Intel Xeon X5650 processors and two GPU cards, either NVIDIA Tesla M2050 or M2070. All calculations were performed using double precision floating point accuracy and the same number of CPU cores as GPU cards were used in testing.

For the serial performance we used two simple test systems: the fullerene molecule C_{60} and 95 atom bulk silicon Si_{95}. The results are shown in Tables 1 and 2. The overall speed-ups for these systems were 7.7 for the silicon one and 8.3 for the fullerene. However the speed-ups for the individual GPU accelerated parts were much higher ranging from 7.7 to 20. The reason for this is that about one third of the time in the GPU accelerated version is taken by routines which don't have GPU implementations yet. A large chunk of that time is taken by the calculation of the exchange-correlation potential.

We tested the parallel scalability the our multi-GPU code with a carbon nanotube system. The scalability was tested in a weak sense meaning that we attempted to keep the problem size per MPI task constant when the number of tasks was increased. The length of the carbon nanotube was increased concurrently with the MPI tasks. The size of the system varied form one MPI task, 80 atoms and 320 valence electrons to 12 tasks, 320 atoms and 1280 valence electrons. The performance of the CPU and GPU versions of the code and the achieved speed-ups are demonstrated in Figure 4. The same number of CPU

Table 1. Bulk silicon with 95 atoms with periodic boundary conditions, 360 bands and 1 **k**-point. Times are in seconds per one SCF iteration. Grid size: 56x56x80 CPU: Intel Xeon X5650 using a single core GPU: NVIDIA Tesla M2070.

Si_{95}	**CPU (s)**	**GPU (s)**	**Speed-up**
Poisson solver	1.8	0.13	14
Orthonormalization	23	3.0	7.7
Precondition	9.4	0.77	12
RMM-DIIS other	32	3.2	10
Subspace diag.	23	2.1	11
Other	2.7	2.7	1.0
Total (SCF-iter)	**93**	**13**	**7.7**

Table 2. Fullerene molecule C_{60} with 240 electronic states. Times are in seconds per one SCF iteration. Grid size: 84x84x84 CPU: Intel Xeon X5650 using a single core GPU: NVIDIA Tesla M2070.

C_{60}	**CPU (s)**	**GPU (s)**	**Speed-up**
Poisson Solver	13	0.64	20
Orthonormalization	11	1.2	9.2
Precondition	16	0.99	16
RMM-DIIS other	8.1	0.6	13
Subspace Diag.	22	2.1	10
Other	3.5	3.2	1.1
Total (SCF-iter)	76	9.1	8.3

cores and GPUs were used in all the tests. The weak scaling efficiency of the multi-GPU code is very good. The largest speed-up was observed with 12 GPUs.

A larger test for weak scaling was performed with CURIE supercomputer based in France, which has a large hybrid GPU partition with 144 nodes connected to an Infiniband network. Each node has two Intel Xeon E5640 processors and two NVIDIA Tesla M2090 GPU cards. Bulk silicon with periodic boundary conditions was selected as a test system. Again the number of atoms in the test system was increased concurrently with the MPI tasks. The size of the system varied form one MPI task, 95 atoms and 380 valence electrons (grid size: 80x56x56) to 192 tasks, 320 atoms and 6908 valence electrons (grid size: 164x164x164). The largest system requires about 1TB of memory for calculations. The performance, speed-ups and scaling behavior of the CPU and GPU versions of the code is demonstrated in Figure 5. Again, the scalability and the performance of the multi-GPU code seems to be very good and consistent even on massive systems using 192 GPUs. The achieved speed-ups varied from 10 to 15.8.

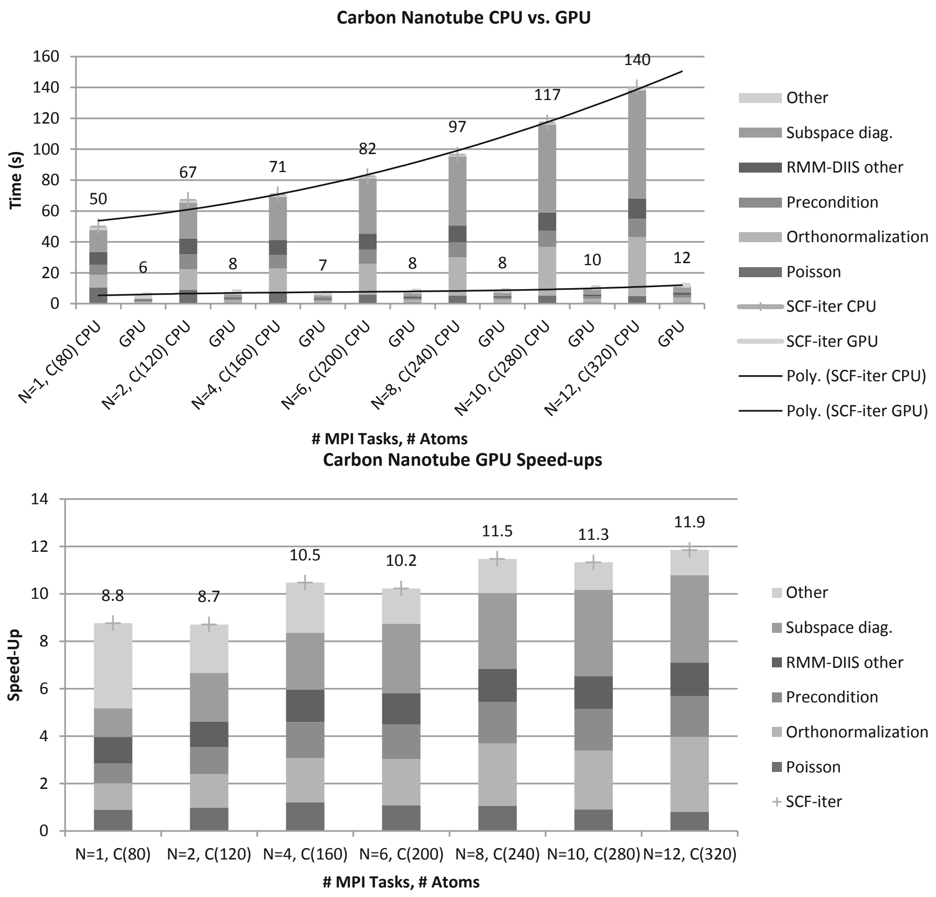


Fig. 4. Upper figure: The weak scaling performance of the CPU and GPU versions of the code using carbon nanotubes. A third degree polynomial is fitted to the figure, since the largest chunk of the total time is taken by matrix-matrix multiplications. Lower figure: The achieved speed-ups with GPUs. Equal number of GPUs and CPU cores were used in all the tests. CPU: Intel Xeon X5650 GPU: NVIDIA Tesla M2070.

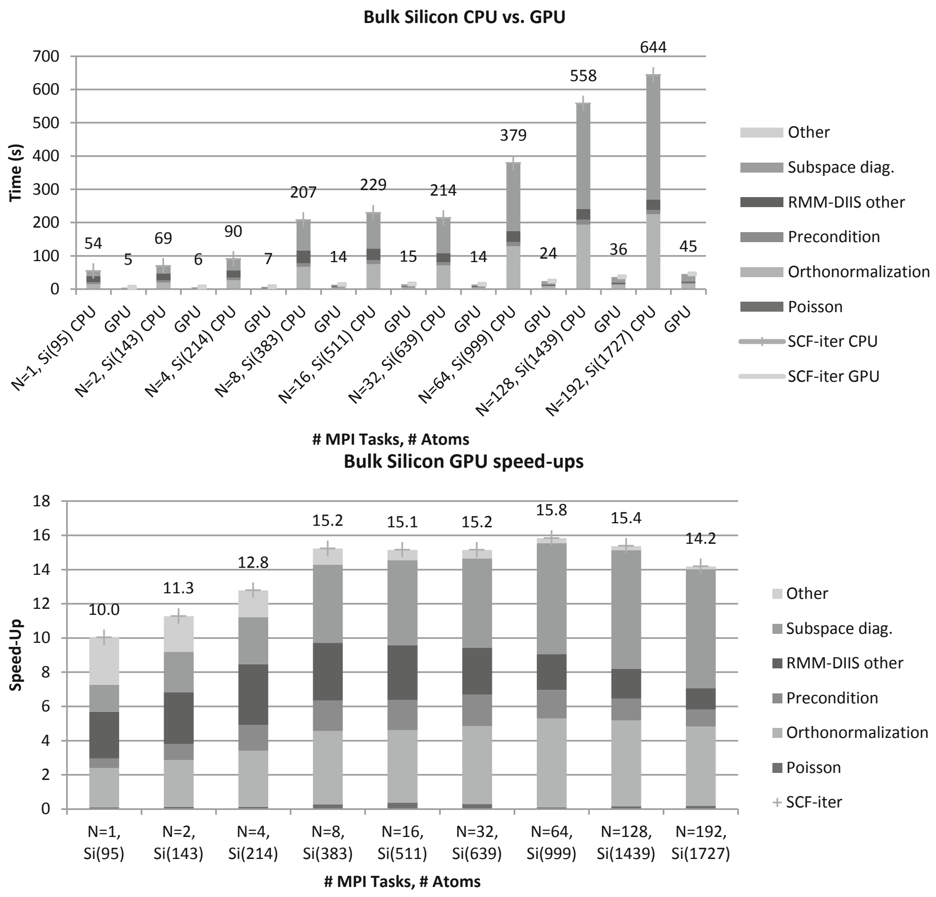


Fig. 5. Upper figure: The weak scaling performance of the CPU and GPU versions of the code using bulk silicon systems. Lower figure: The achieved speed-ups with GPUs. Equal number of GPUs and CPU cores were used in all the tests. CPU: Intel Xeon E5640 GPU: NVIDIA Tesla M2090.

5 Conclusions

We have provided an electronic structure calculation code capable of running on large GPU clusters. We have accelerated with GPUs most of the numerically intensive parts in DFT GPAW calculations: solving the Poisson equation, subspace diagonalization, the RMM-DIIS algorithm and orthonormalization of the wave functions. High performance was achieved by carefully optimizing the CUDA kernels and minimizing the data transfers between the GPU and the host computer. For the serial version of the code we observed speed-ups between 7.7 and 8.3. The performance could be improved by implementing more routines using GPUs.

Multiple GPUs and nodes can be utilized with MPI using domain decomposition or by parallelizing over **k**-points. Our parallel implementation overlaps computations and data transfers between different GPUs. With the parallel version of the code we were able to get significant speed-ups (up to 15 times) in ground state DFT calculations using multiple GPUs when compared to equal number of CPU cores. Also, an excellent weak scaling efficiency for the multi-GPU code was achieved in the tested systems running up to 192 GPU cards.

Acknowledgments. We acknowledge that some of the results in this paper have been achieved using the PRACE Research Infrastructure resource CURIE based in France at GENCI/CEA. We also acknowledge CSC – IT Center for Science Ltd. for the allocation of computational resources.

References

[1] Mortensen, J.J., Hansen, L.B., Jacobsen, K.W.: Real-space grid implementation of the projector augmented wave method. Phys. Rev. B 71, 35109 (2005)

[2] Enkovaara, J., Rostgaard, C., Mortensen, J.J., Chen, J., Dulak, M., Ferrighi, L., Gavnholt, J., Glinsvad, C., Haikola, V., Hansen, H.A., Kristoffersen, H.H., Kuisma, M., Larsen, A.H., Lehtovaara, L., Ljungberg, M., Lopez-Acevedo, O., Moses, P.G., Ojanen, J., Olsen, T., Petzold, V., Romero, N.A., Stausholm-Møller, J., Strange, M., Tritsaris, G.A., Vanin, M., Walter, M., Hammer, B., Häkkinen, H., Madsen, G.K.H., Nieminen, R.M., Nørskov, J.K., Puska, M., Rantala, T.T., Schiøtz, J., Thygesen, K.S., Jacobsen, K.W.: Electronic structure calculations with GPAW: a real-space implementation of the projector augmented-wave method. Journal of Physics: Condensed Matter 22(25), 253202 (2010)

[3] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: Top500 supercomputer sites (November 2012), `http://www.top500.org/lists/2012/11/` (accessed December 5, 2012)

[4] Harju, A., Siro, T., Canova, F.F., Hakala, S., Rantalaiho, T.: Computational Physics on Graphics Processing Units. In: Manninen, P., Öster, P. (eds.) PARA 2012. LNCS, vol. 7782, pp. 3–26. Springer, Heidelberg (2013)

[5] Payne, M.C., Teter, M.P., Allan, D.C., Arias, T.A., Joannopoulos, J.D.: Iterative minimization techniques for *ab initio* total-energy calculations: molecular dynamics and conjugate gradients. Rev. Mod. Phys. 64, 1045–1097 (1992)

[6] Yasuda, K.: Accelerating density functional calculations with graphics processing unit. Journal of Chemical Theory and Computation 4(8), 1230–1236 (2008)

[7] Yasuda, K.: Two-electron integral evaluation on the graphics processor unit. Journal of Computational Chemistry 29(3), 334–342 (2008)
[8] Ufimtsev, I.S., Martinez, T.J.: Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. Journal of Chemical Theory and Computation 4(2), 222–231 (2008)
[9] Ufimtsev, I.S., Martinez, T.J.: Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation. Journal of Chemical Theory and Computation 5(4), 1004–1015 (2009)
[10] Asadchev, A., Allada, V., Felder, J., Bode, B.M., Gordon, M.S., Windus, T.L.: Uncontracted Rys quadrature implementation of up to G functions on graphical processing units. Journal of Chemical Theory and Computation 6(3), 696–704 (2010)
[11] Genovese, L., Ospici, M., Deutsch, T., Méhaut, J.F., Neelov, A., Goedecker, S.: Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. The Journal of Chemical Physics 131(7), 34103 (2009)
[12] Maintz, S., Eck, B., Dronskowski, R.: Speeding up plane-wave electronic-structure calculations using graphics-processing units. Computer Physics Communications 182(7), 1421–1427 (2011)
[13] Hacene, M., Anciaux-Sedrakian, A., Rozanska, X., Klahr, D., Guignon, T., Fleurat-Lessard, P.: Accelerating VASP electronic structure calculations using graphic processing units. Journal of Computational Chemistry (2012) n/a–n/a
[14] Spiga, F., Girotto, I.: phiGEMM: A CPU-GPU library for porting Quantum ESPRESSO on hybrid systems. In: 2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 368–375 (February 2012)
[15] Wang, L., Wu, Y., Jia, W., Gao, W., Chi, X., Wang, L.W.: Large scale plane wave pseudopotential density functional theory calculations on GPU clusters. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 71:1–71:10. ACM, New York (2011)
[16] Jia, W., Cao, Z., Wang, L., Fu, J., Chi, X., Gao, W., Wang, L.W.: The analysis of a plane wave pseudopotential density functional theory code on a GPU machine. Computer Physics Communications 184(1), 9–18 (2013)
[17] Andrade, X., Alberdi-Rodriguez, J., Strubbe, D.A., Oliveira, M.J.T., Nogueira, F., Castro, A., Muguerza, J., Arruabarrena, A., Louie, S.G., Aspuru-Guzik, A., Rubio, A., Marques, M.A.L.: Time-dependent density-functional theory in massively parallel computer architectures: the octopus project. Journal of Physics: Condensed Matter 24, 233202 (2012)
[18] Blöchl, P.E.: Projector augmented-wave method. Phys. Rev. B 50, 17953–17979 (1994)
[19] Kohn, W., Sham, L.J.: Self-consistent equations including exchange and correlation effects. Phys. Rev. 140, A1133–A1138 (1965)
[20] Parr, R., Yang, W.: Density-Functional Theory of Atoms and Molecules. International Series of Monographs on Chemistry. Oxford University Press, USA (1994)
[21] Brandt, A.: Multi-level adaptive solutions to boundary-value problems. Math. Comp. 31, 333–390 (1977)
[22] Wood, D., Zunger, A.: A new method for diagonalising large matrices. Journal of Physics A: Mathematical and General 18(9), 1343 (1999)
[23] Kresse, G., Furthmüller, J.: Efficient iterative schemes for *ab initio* total-energy calculations using a plane-wave basis set. Phys. Rev. B 54, 11169–11186 (1996)

[24] Briggs, E.L., Sullivan, D.J., Bernholc, J.: Real-space multigrid-based approach to large-scale electronic structure calculations. Physical Review B 54, 14362–14375 (1996)

[25] NVIDIA Corp: Whitepaper: NVIDIA's next generation CUDA compute architecture: Fermi, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (accessed October 20, 2012)

[26] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. Parallel Computing 38(3), 157–174 (2012)

[27] NVIDIA Corp: CUDA parallel computing platform, http://www.nvidia.com/object/cuda_home_new.html (accessed October 14, 2012)

[28] Micikevicius, P.: 3D finite difference computation on GPUs using CUDA. In: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, pp. 79–84. ACM, New York (2009)

An Environment for Service Composition, Execution and Resource Allocation

Jan Kwiatkowski, Krzysztof Juszczyszyn, and Grzegorz Kolaczek

Institute of Informatics
Wroclaw University of Technology
Wybrzeze Wyspianskiego 27, 50-370 Wroclaw, Poland
{jan.kwiatkowski,krzysztof.juszczyszyn,grzegorz.kolaczek}@pwr.wroc.pl

Abstract. In recent years the evolution of software architectures led to the rising prominence of the Service Oriented Architecture (SOA) concept. The services can be deployed in distributed environments and executed on different hardware and software platforms. In the paper a configurable and flexible environment, allowing composition, deployment and execution of composite services, which can be applied in the wide range of SOA-based systems is presented. It supports service semantic description, composition and the distribution of service requests guaranteeing services quality, especially efficient allocating communication and computational resources to services. We present an unified approach, which assumes the semantic description of Web service functionalities with an XML-based language - Smart Service Description Language which provides similar features to that of OWL-S or WSDL, however, it was designed to support services execution and monitoring. These unique features allow to design a service execution engine, compatible with the underlying execution environment and providing support for service QoS guarantees.

Keywords: Service Oriented Architecture, service composition, service request distribution, service security.

1 Introduction

In recent years the evolution of software architectures led to the rising prominence of the Service Oriented Architecture (SOA) concept. This architecture paradigm facilitates building flexible service systems. The services can be deployed in distributed environments, executed on different hardware and software platforms, reused and composed into composite services.

Service composition and execution were addressed in numerous works [6]. Many approaches [7] require a well-defined business process to compose a composite service. Semantic analysis of user requirements, service discovery (meeting the functional requirements) and the selection of specific services against non-functional requirements (i.e. execution time, cost, security) are common issues in service composition. However, there many disadvantages in the solutions developed so far, which prevent their successful introduction to the market. In many cases only one aspect of service composition is considered. For example the work [10] focuses on services selection based only on one functional requirement at a time. Other works show that non-functional requirements are considered to be of a key importance, however approaches still ignore the aspect of building a proper structure of a composite service which is a key to optimization of i.e. execution time.

Various methods for services selection or composite service QoS-based optimization were presented. However, those solutions are not widely used by other researchers. Some of them propose complete end-to-end composition tools introducing a concept of two-staged composition: logical composition stage to prune the set of candidate services and then composing an abstract work-flow. METEOR-S [1] presents a likewise concept of binding web services to an abstract process and selecting services fulfilling the QoS requirements. Notions of building complete composition frameworks are also clear in SWORD [13], which was one of the initial attempts to use planning to compose web services. However, it should be noted that the proposed approaches are closed and do not support incorporations of other methods and algorithms. On the other hand, an extensible framework-based approach is what is currently needed in SOA field in order to create composition approaches that are fitted to different domains and problems characteristic for them. An approach to service composition described below was developed in order to be compatible with this assumption.

The paper is organized as follows. Section 2 briefly describes the architecture of developed and implemented environment. Description of Service Composer and Work-flow Engine which are responsible for service composition and execution control are presented in section 3. In the section 4 components responsible for service execution at the lowermost level are presented. Section 5 describes the Validation Unit, it covers the presentation of the general idea as well as some its implementation details. Finally, section 6 outlines the work and discusses the further works.

2 The General Architecture of Proposed Environment

The architecture of proposed environment is presented in the figure 1. It is composed of a number of independent modules providing separate functionalities and interacting with each other using specified interfaces. Its components are responsible for service composition and execution control provided by Service Composer and Work-flow Engine, service execution and monitoring performed by: Broker, Facade, Controller, Virtualizer, and Validation unit responsible for secure execution of services. The main components of design and implemented environment are as follows:

– Broker - handles user requests and distribute them to proper instances of services. The decision is taken using information about current loading of available communication and computation resources. Functional and non-functional requirements are taken into consideration. It also performs internal requests to coordinate the operation of the system components as well as obtain some necessary information.

- Facade - supports communication with components behind it, and collects necessary information for the Broker. It also provides special services for the Broker to test current state of processing environment (e.g. characteristic of communication links).
- Controller - manages all components behind the Facade. It is responsible for control processing according to capabilities of the environment and current state of it. It can also route the requests to the services independently, taking into account computational resource utilization and performs decision to start/stop another instance of service.
- Virtualizer - offers the access to hypervisor commands. Uses *libvirt* to execute commands which gives the environment independence from particular hypervisor.
- Service Composer - is responsible for composite service composition.
- Work-flow Engine - governs the process of services execution and the dynamic interpretation of requirements leading to service execution.
- Validation unit - provides the value of security level which is an important non-functional requirement used by Service Composer. More specifically returns a security level which is equal to the current anomaly level of the service.

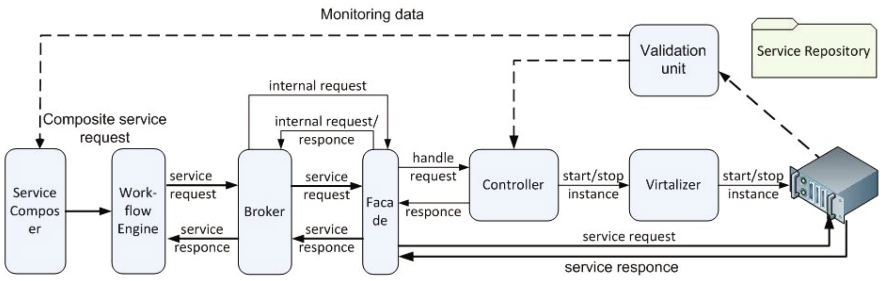


Fig. 1. The architecture of developed and implemented environment

Above specified modules interact using two interfaces. Internal communication is XML-RPC based, for components behind the Facade, and uses SOAP messages for communication between Broker and Facade. This allows flexibly manage distributed computational resources as well. Interaction with external components is based on Broker services with SOAP messages.

3 Service Composer and Work-Flow Engine

This section presents a general composition scenario and indicates that each of its stages could be performed using different methods; one could use different semantic selection methods when searching for services; finally, optimization techniques could be used to produce the composite service fulfilling the non-functional requirements [14]. All these led to designing a composition framework with an flexible architecture that would allow composition service designers to incorporate various approaches, test them and deploy in a form of service enabled composition tools. This approach is fully consistent with the assumptions of the SOA paradigm and stems from our previous experiences with service composition frameworks [15]. It consists of Service Composer, equipped with a front-end Web interface allowing a business client to define his domain by connecting to external service and knowledge repositories and Work-flow Engine. Both are service-based and can be called from an external application through SOAP protocol. The composite services are described in SSDL (Smart Service Description Language) which is proposed as a solution allowing simple description of composite service execution schemes, supporting functional and non-functional description of services. Its functionality includes the Web Service Description Language (WSDL), but offers important extensions. A definition of SSDL node types contains all basic data types which allow for the functional and non-functional description of a service. Each SSDL node, is used to describe a basic functionality requirement for a service has several important sections used during service selection, composition and the final execution plan optimization:

- physical description - used by every type of input and output data for a specific Web Service,
- functional description - used to semantically describe the capabilities of a service, expressed in terms of domain ontology concepts,
- non-functional description - used to describe non-functional parameters of a service such as: time, cost, availability and others; non-functional parameters that can be requested for composition purpose are not limited in any way - external validation can be performed using, e.g. user defined ontology and rules.

To compose a composite service means to find a set of atomic services and bind them together so that they, as a new service, fulfil all functional and non-functional requirements. Typically automated composition process requires a semantic query and consists of three stages: building of a composite service structure, building of a composite service scenario, and finding an optimal execution plan of a composite service. Each of the composition stages could be performed using different methods. In order to provide a flexible approach to composition tasks, the Service Composer is a composite service itself and is built up from the services responsible for selection, data flow control and QoS optimization of the atomic services which take part in a composite service execution plan. The services of the Service Composer (semantic filters used for service selection, or QoS assessment services) may use different strategies and algorithms. The result of the composition process is a composite service plan which is passed to the Work-flow Engine.

Execution engines which support the process-driven composite service execution were described in [12], while the engine support for execution of BPEL-defined processes was proposed in [4] [5]. These solutions, however, assumed a fixed architecture of the service framework, which was based on certain devices and did not allow the reconfiguration of execution engine.

After the composition of the composite service is completed, the service is executed by the SSDL execution engine (Work-flow Engine). It assumes engine-as-a-service approach, and offers an execution engine as a configurable composite service. It has a built-in SSDL language interpreter and supports dynamic interpretation of service description files, service configuration, and execution control. The Work-flow Engine is implemented as a lightweight virtual machine which may be duplicated and migrated upon decision taken on the SOA infrastructure level. The core feature distinguishing Work-flow Engine from other execution engines is its focus on composition mechanism and, together with the expressive nature of SSDL language it interprets, ability to configure its own behaviour.

The Work-flow Engine can be configured to interpret and execute different kinds of node classes defined in SSDL. SSDL is executed by the Work-flow Engine in two phases. First, in the initialisation phase performs validation of the SSDL service execution request and can run external services transforming the whole input SSDL. Then, the engine executes each of the nodes of the SSDL by sending requests to the Broker (processing phase). The Work-flow Engine architecture allows also for further extension of its capabilities. For example both in the initialisation and the processing phase chosen actions, can be performed by appropriate external Web services indicated by the Work-flow Engine configuration.

The main role of the Work-flow Engine is to govern the process of services execution. Actions focused on the SSDL and performed in the preprocessing phase could be internal methods or, in configuration-driven approach, composite web services. Based on this fundamental engine model, other phases could be added to further personalize the behaviour and expand engine functionalities. This should ultimately lead to further extensibility of the engine, incorporating various composite service definitions as modules and various behaviours depending on chosen events.

The key components of the Workflow Engine as a composite service, along their functionalities are presented in the figure 2. Work-flow Engine is implemented in Java and supports multi-threading to process multiple service execution plans at the same time. It is capable of executing composite services defined in SSDL but it can also automatically generate web interfaces to composite services stored inside the engine, broadcasting as those services. In this mode the Execution Engine can emulate any composite service defined in SSDL. As a result of multi-threading, single execution engine could act as multiple services or in extreme case, multiple engines could act as atomic services - hiding composite services behind a layer of abstraction. For optimization purposes not only the Work-flow Engine could be maintained in various localizations but also it can delegate parts of its composite service to other instances of execution engine.

The scenario for using presented framework to compose and execute services assumes the following three phases:

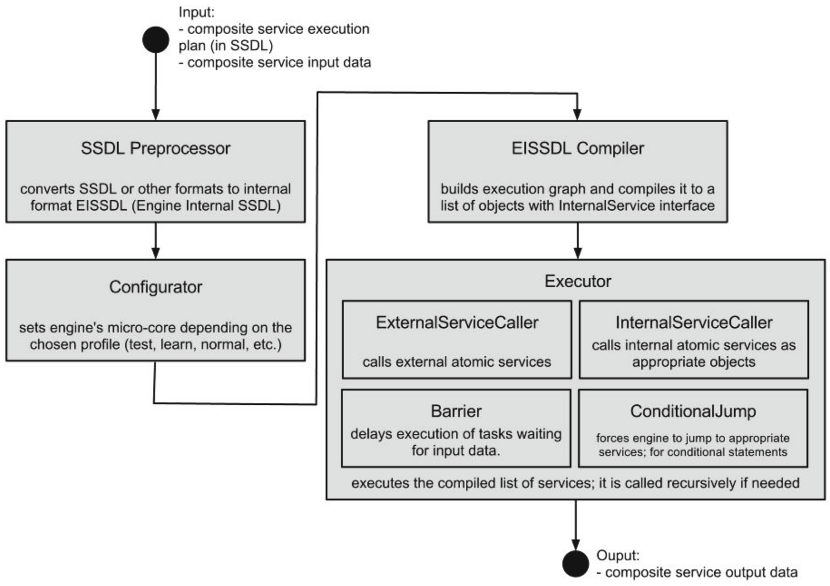

Input:
- composite service execution plan (in SSDL)
- composite service input data
SSDL Preprocessor
converts SSDL or other formats to internal format EISSDL (Engine Internal SSDL)
Configurator
sets engine's micro-core depending on the chosen profile (test, learn, normal, etc.)
EISSDL Compiler
builds execution graph and compiles it to a list of objects with InternalService interface
Executor
ExternalServiceCaller
calls external atomic services
InternalServiceCaller
calls internal atomic services as appropriate objects
Barrier
delays execution of tasks waiting for input data.
ConditionalJump
forces engine to jump to appropriate services; for conditional statements
executes the compiled list of services; it is called recursively if needed
Ouput:
- composite service output data

Fig. 2. The Execution Engine scheme and basic functionalities

- Requirement definition - using Service Composer module interface operator defines requirements for a composite service, which have a form of directed graph, where nodes and links represent service requirements and the data flow, respectively,
- Service composition - the requirements graph is processed by the Service Composer, which executes composition algorithm and produces a composite service graph which may be edited or checked by the operator. A composite service graph is stored in the SSDL format and may be passed directly to the Workflow Engine when execution is needed,
- Service Execution Work-flow Engine send the service execution request to Broker in order to execute the service.

The basic workflow starts with the business process query, complemented by its associated non-functional requirements (cost, security, time constraints, etc.). It is possible to define the business process in the GUI of Service Composer or to translate it from Aris BPM format via dedicated translation service which is also provided as a part of presented environment. Future translation services (for example from BPMN format etc.) are under development. The result of process translation or its definition using Service Composer GUI has the same result it is a composite service requirement graph which is stored in SSDL format.

In order to compose a composite service responsible for business process execution, the requirements graph is read by the Service Composer and the composition process starts. The requirements graph is processed by the Service Composer, which executes composition algorithm and produces a composite service graph which may be edited or checked by the operator. The GUI of the Service Composer is implemented in JavaScript and may be viewed from any Web browser. It supports access to the service repository, domain anthologies and provides a graphical view of any composite service or business process represented in SSDL. Figure 3 presents an example of composite service (responsible for processing video monitoring data streams). The Service Composer GUI allows editing and detailed inspection of the graph of any composite service. Any changes being made are directly written in SSDL file, which supports the operator in controlling and altering the composition results. The composite service graph contains all the atomic services and represents the dataflows between them. Again, a composite service graph is stored in the SSDL format and may be passed directly to the Work-flow Engine when execution is needed.

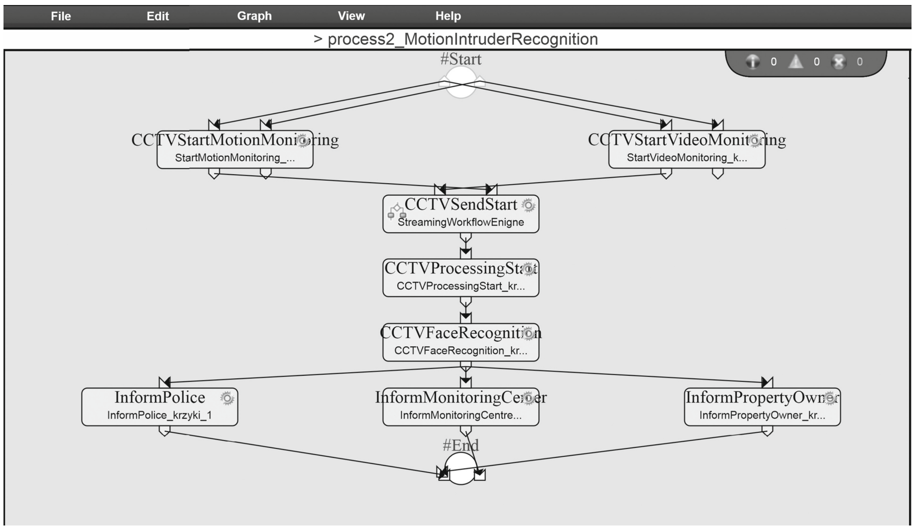

File
Edit
Graph
View
Help
> process2_MotionIntruderRecognition
#Start
0
0
0
CCTVStartMotionMonitoring
StartMotionMonitoring_...
CCTVStartVideoMonitoring
StartVideoMonitoring_k...
CCTVSendStart
StreamingWorkflowEnigne
CCTVProcessingStart
CCTVProcessingStart_kr...
CCTVFaceRecognition
CCTVFaceRecognition_kr...
InformPolice
InformPolice_krzyki_1
InformMonitoringCenter
InformMonitoringCentre...
InformPropertyOwner
InformPropertyOwner_kr...
#End

Fig. 3. The graph of an exemplary composite service (dataflow view)

In order to execute a composite service, its definition in SSDL is passed to the Work-flow Engine which communicates with the Broker and registers the service. From this point on, the service may be executed by the Work-flow Engine, which interprets the SSDL definitions of composite services and maintains the non-functional parameters. The framework also allows the use of external execution engines (in this scenario, the Work-flow Engine serves as an SSDL-driven interface for them). In the course of execution, the Work-flow Engine calls the atomic services via the Broker, which distributes calls to the chosen instances of services.

4 Request Distribution Manager Components

Broker, Facade, Virtualizer and Controller together create so called Request Distribution Manager (RDM) that is responsible for atomic service execution at the lowermost level. The architecture of RDM resolves the problem with traditional software lack of flexibility. Software composition and distribution in the traditional form, where applications are not open enough to follow rapidly changing needs of business, had to be replaced with something more flexible. The idea to compose the processes from services publicly or privately available, mix and match them as needed, easily connect to business partners, seems like the best way to solve it.

The RDM receives from Work-flow Engine request for service execution and takes decision where requested service will be executed in the distributed environment taken into consideration current loading of available communication as well as computation resources. Request for service execution is send by Work-flow engine or any other client which, used defined for RDM (Broker) interface in form of SOAP message. It means that RDM delivers to clients access to services available in used distributed environment. Clients see available services at Broker localization and don't know where service is located and that services may be multiplied. From the client's point of view, services are described at Service Repository using WSDL standard, and are accessible using standard SOAP calls and its physical location is hidden from the client.

The module of RDM that as the first received service request is Broker, which acts as service delivery component. It distributes requests for services to known service processing resources. The Broker collects data about instances of atomic services based on measured or calculated values of non-functional parameters. To make allocation decision the Broker estimates completion time of service execution and data (request/response) transfer time with use of adaptive models of execution systems and communication links, built as a fuzzy-neural controllers. However Broker distinguishes individual processing resources, simultaneously coordinates activities with Controller, and passes requests on. Each and every request is redirected to proper service instance based on the values of non-functional parameters of the requested service. Proper instance of service is either found from the working and available ones or the new one is started to serve the request. Such an approach gives the possibility to serve clients requests and manage resource virtualization and utilization automatically with minimal manual interaction.

The Facade supports communication between Broker and components inside the execution system. The Facade collects and delivers some essential information necessary to control request distribution. It accepts an interprets defined SOAP messages of internal services used to support request distribution and service virtualization. For standard service requests the Facade processes header section of SOAP messages. It completes especially defined section with essential data of service execution, currently real completion time of service execution. Virtualizer for virtualization management used an open source toolkit *libvirt*. It offers the virtualization API supporting most of existing hypervisors.

Controller tasks is to prevent the system from taking virtual sprawl route, which is pointed as one of the biggest challenges in virtualization. The data coming from monitoring could be used to determine the service instances which are not used and can be stopped. It also determine the need to increase or decrease the amount of memory assigned to particular virtual machine. Furthermore Controller shall understand service importance automatically starting the services which require constant accessibility. Handling the request by the Controller can lead to one of four situations. There is a running service instance which can perform it and it will be returned as the target to which the SOAP request shall be forwarded. All running instances of requested service do not satisfied non-functional requirements from the request.Then the new service instance is started. There is no running service, but there is an image which satisfies the conditions. In such a case the image will be instantiated and it will be used as the one to perform the request. Last possibility is the lack of proper service and image in which case the error will be thrown and finally returned as a SOAP Fault message to the client.

Below an example how the RDM can be used when services are provided by an application that is built using client-server architecture is presented. The application that provides services is multi-user application, moreover services can be provided by the number of application instances available at different locations at the same time. The application is equipped with own monitoring system that presents to the service administrator information about current loading for each running application instance. Additionally application monitoring system is able to predict how more users can used the application instance also. The screenshot of application monitoring system is presented in figure 4. The upper view presents the loading of selected by service administrator server and lower prediction how many more service requests can be provided by instance of application running on the specified server. When application is not able to provide service for some user because of application overloading, the request is store in the queue or the message that system is not able to provide the service for user is generated. Then to solve this problem service administrator starts manually the new instance of application and redirect all request from the queue.

Using RDM solves the problem the necessity of service administrator actions when the services are overloaded. Design and implemented Broker can works in two modes, the first when all actions related to the service requests are maintenance directly by the Broker and the second one when the Broker works in the BaaS mode (Broker as a Service). When Broker works in the BaaS mode, the IP address of service is return as response for service request. Then, the client is able to use the service directly.

Figure 5 presents the scenario for service request processing when available service instances are close to overloading. The client send the request to Broker that woks in the BaaS mode. Using the monitoring data related to current loading of communications lines and available in the distributed environment servers takes decision about the localization of Computer Centre where the new instance of service should be started. Then the request is send to Facade of chosen by the

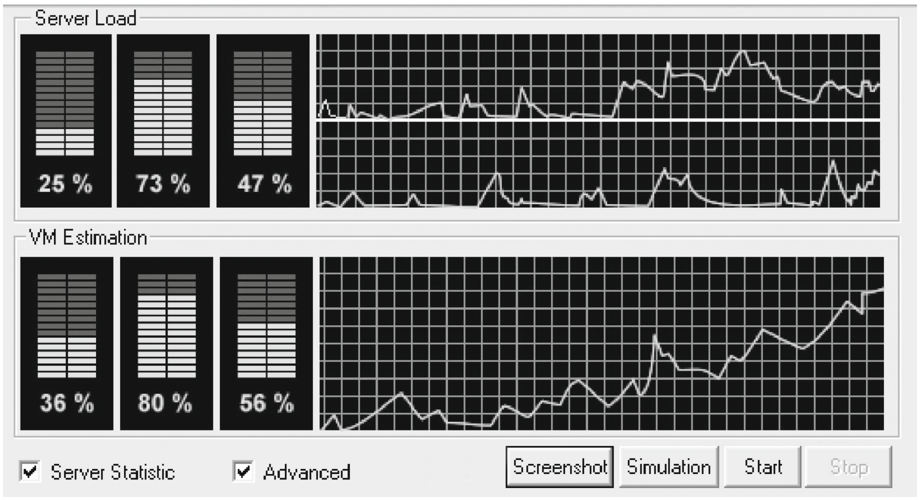


Fig. 4. Visualization of monitoring data

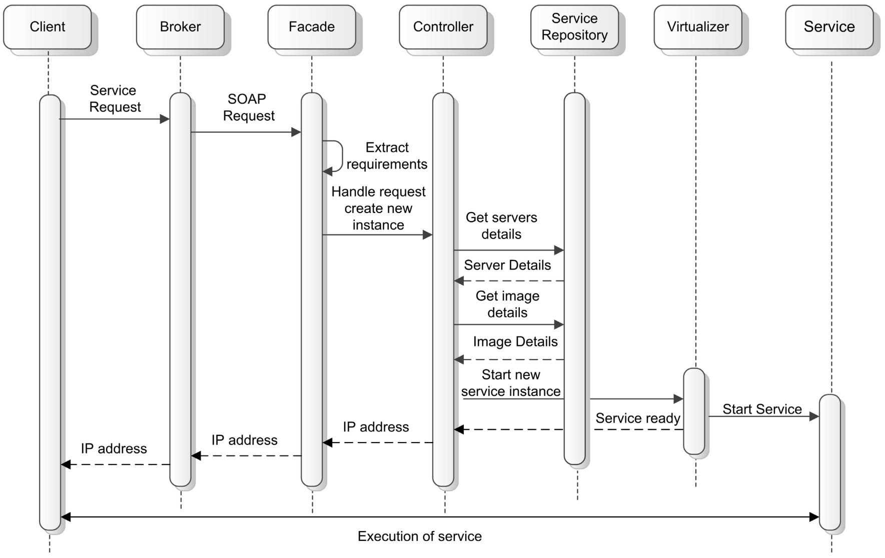


Fig. 5. Scenario of new service instance creation

Broker Computer Centre. It is then the responsibility of the Facade to get the details about the localization of the new created instance of service (IP address) and forwarding the message with this information to Broker. Then the answer is sent back by Broker to the client and the client can use new created instance of requested service.

5 Service Execution Monitoring and Validation

Each IT system must be verifiable secure, especially wherever sensitive data is handled. As a result, there are natural tendency to validate systems security level. The main idea of SOA based system validation presented in this paper is that while the atomic service security level has been evaluated also the composite service can be validated using some formal operators taken from Subjective Logic and execution plan defined in SSDL [8].

The Broker collects and provides to Validation unit module data about instances of atomic services. These data are values of non-functional parameters of registered services instances. Some of them, e.g. completion time of service execution and data volume (request/response), are used by Validation unit to validate security level of executed services while the others are used to estimate values of non-functional parameters and to pass them to Service Composer. Service Composer uses actualized values of service non-functional parameters to prepare the execution plan for next users' request corresponding to the current status of execution environment and users' SLA [3]. Another task of Validation unit is validation of security level of executed services. Values of non-functional parameters obtained from Broker are used to obtain useful information about system deviations from normal state. Validation unit uses anomaly detection which must pre-process them to reduce the probability of misinterpretation and false-positive alarms. The general idea of security level validation is that Validation unit creates time series related to features describing the executed service behaviour [9]. Time series is a sequence of data points, measured typically at successive times, spaced at (often uniform) time intervals and analysed to detect anomalies. Anomaly in time series data are points that significantly deviate from the normal pattern of the data sequence and are related to security breaches. The evaluated level of observed anomaly in executed services is also passed to Service Composer. Service Composer module uses this information to prepare execution plan of composite services in regard to security level requirements.

5.1 Implementation Issues of the Validation Unit

The Validation unit has been designed and implemented in accordance with the service oriented paradigm. This means that the Validation unit is fully operable and independent service which takes as an input a sequence of numerical values comprising time series and which returns a value supposed to be the security level of the monitored object. There are also two modes of operation of the Validation unit. The first one is dedicated to validation of the single instance of a service. The second one allows validating all available services simultaneously. However this second mode of operation is nothing more than the concurrently executed validation for each service available in Request Distribution Manager.

As Broker provides the list of currently available services the user of the Validation unit may deliberately select one service that will be validated (in second mode of operation no selection step is needed all instances of all available services will be validated by default). When a service has been selected the

Validation unit starts to send SOAP requests to the Broker. The Validation unit requests the Broker for the relevant execution information corresponding to the selected service. As it has been described in earlier sections, the Broker returns values of several non-functional parameters of registered services instances, e.g. completion time of service execution and data volume (request/response). These requests are generated periodically with the predefined time interval. As a result, the Validation unit creates corresponding time series describing the changes in parameter values of the observed object.

The implemented method of time series analysis checks for anomalies in behavioural features of executed services, e.g., abnormally high input data volume or long execution time can denote some type of DoS (Denial of Service) attacks against a service. The time series analysis is performed by the Validation unit in three steps. In the feature selection step, a relevant behavioural features is selected e.g., the Validation unit can decide to compute the number of transmitted bytes by a service during successive time windows. In the parameter estimation step, historical (training) data on the selected feature values are compared with the current feature value to learn how indicative the feature is of possible anomalies. A model of a service behaviour is constructed by iterating these two steps. The last step is detection of anomalies, as indicated by large discrepancy between the statistics of the selected feature values and baseline statistics derived from training data. This is done, among others, by noticing the periodicity in successive feature values and selecting a characteristic period to capture significant correlations between them. The anomaly detection algorithm takes a multidimensional view upon the time series of feature values (Figure 6); this is known to better capture trends and seasonal changes on various time-scales [2]. Let the time series of our feature values be

$$X = (x_1, ..., x_l, ...) \tag{1}$$

where x_i is the number of received bytes in a subsequent time window. Two types of time sub-series derived from X are also analysed by the Validation unit, namely X_P and X_T, with

$$X_{P,l} = \frac{1}{P}\sum_{k=0}^{P-1} X_{l-k}, X_{T,l} = \frac{1}{T}\sum_{k=0}^{T-1} X_{l-kP} \tag{2}$$

where P and T are averaging intervals (P corresponds to the characteristic period of X). That is, X_P and X_T represent the series of current averages of the feature values over, respectively, the last characteristic period and a number of recent characteristic periods given a fixed time shift with respect to the period start. For the time series X_P and X_T, exponential moving averages are computed as

$$\overline{X}_{P,l} = \overline{X}_{P,l-1} + w * (x_l - \overline{X}_{P,l-1}), \overline{X}_{T,l} = \overline{X}_{T,l-1} + w * (x_l - \overline{X}_{T,l-1}) \tag{3}$$

whereas standard deviations over appropriate averaging intervals are computed as

$$\sigma_{P,l} = \sqrt{\frac{1}{P}\sum_{j=l+P-1}^{l} (x_j - \overline{X}_{P,l})^2}, \sigma_{T,l} = \sqrt{\frac{1}{T}\sum_{k=0}^{T-1}(x_{l-kP} - \overline{X}_{T,l})^2} \tag{4}$$

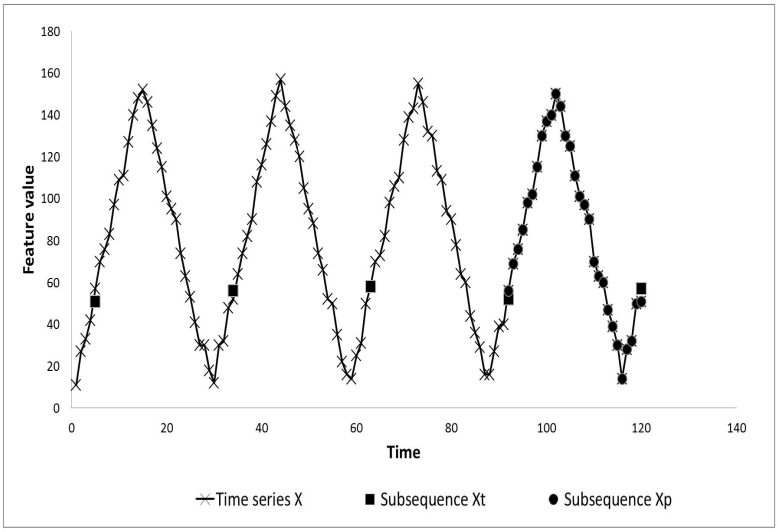


Fig. 6. Time series analysis for anomaly detection

where w is an empirically adjusted smoothing coefficient. Finally, local deviations are expressed as

$$\delta_{P,l} = \mid x_l - \overline{X}_{P,l} \mid, \delta_{T,l} = \mid x_l - \overline{X}_{T,l} \mid \tag{5}$$

Using the above quantities, one can estimate an anomaly level (if any) accompanying the observation of x_i as

$$a_i = \sqrt{(\delta_{P,l}/\sigma_{P,i})^2 + (\delta_{T,l}/\sigma_{T,i})^2}/3\sqrt{2} \tag{6}$$

unless it exceeds 1, in which case $a_i = 1$. When the current feature value is close to average, the anomaly level is close to 0, whereas the local differences exceeds three times the corresponding standard deviations then the Validation unit will return maximum anomaly level. The anomalies detected in service execution are used to assign a security level to a service. In the implemented approach, the security level is simply equal to the current anomaly level of a service.

The information about anomaly level is returned by the Validation unit to the Service Composer (in a form of corresponding SOAP message). Service Composer using up to date information about service behaviour and so about its security level can perform two following tasks. The first task is a verification of a correspondence between non-functional user requirements concerning composite service security level and the current security level. The second task is performed when there are found discrepancies between expected and real security levels. Then, the Service Composer performs a new composition which uses the current security levels returned by the Validation unit. To achieve its goals and to

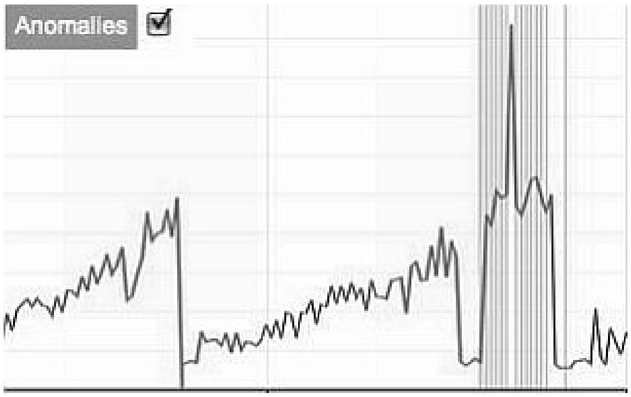


Fig. 7. Visualization of the service execution history

provide the value of up to date security level, the Validation unit starts to send SOAP requests to the Broker. In response to the Validation unit request, Broker returns values of parameters describing execution of service instances within a given period of time: IP address of the service caller, executed service ID (name), execution starting time, execution ending time, input data size (in bytes), output data size (in bytes). This type of requests are generated periodically with the predefined time interval. Using collected information, the Validation unit creates time series describing the changes in parameter values of the executed services. After that the Validation unit computes a set of statistics which are used to detect abrupt changes and to evaluate security level of the monitored services. The information about anomaly level is returned by the Validation unit to the Service Composer (in a form of corresponding SOAP message).

The selection of the service for which the security level is calculated as well as the values required by anomaly detection algorithm can be set using user friendly web interface. The same interface provides information about the detected anomalies and visualizes the services execution history (Figure 7).

6 Conclusions and Future Work

Presented approach offers a set of unique tools and techniques which provide integrated approach to service composition, execution and QoS monitoring. Originally developed service description language allows to include QoS requirements and execution-related parameters in service description, which is directly used by the Work-flow Engine. All the parameters are measured during service execution and may be used when needed by the Service Composer, which feature forms feedback between composition and service execution. Our framework is extensible in terms of adding new algorithms and techniques for service composition, prediction of resource consumption, etc. The results of first experiments with developed framework are very promising.

Acknowledgements. The research presented in this paper has been partially supported by the European Union within the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

References

1. Aggarwal, R., Verma, K., Miller, J., Milnor, W.: Constraint Driven Web Service Composition in METEOR-S. In: Proceedings of the 2004 IEEE International Conference on Services Computing, pp. 23–30 (2004)
2. Burgess, M.: Two dimensional time-series for anomaly detection and regulation in adaptive systems. In: IFIP/IEEE 13th International Workshop on Distributed Systems: Operations and Management, pp. 169–185 (2002)
3. Grzech, A., Witek, P.: Modeling and optimization of complex services in service-based systems. Cybernetics and Systems 40(8), 706–723 (2009)
4. Hackmann, G., Gill, C., Roman, G.: Extending BPEL for interoperable pervasive computing. In: Proceedings of the 2007 IEEE International Conference on Pervasive Services, pp. 204–213 (2007)
5. Hackmann, G., Haitjema, M., Gill, C.D., Roman, G.-C.: Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 503–508. Springer, Heidelberg (2006)
6. Jinghai, R., Xiaomeng, S.: A Survey of Automated Web Service Composition Methods. In: First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC, San Diego, CA, USA, pp. 43–54 (2004)
7. Ko, J.M., Kim, C.O., Kwon, I.-H.: Quality-of-service oriented web service composition algorithm and planning architecture. The Journal of Systems and Software 81, 2079–2090 (2008)
8. Juszczyszyn, K.: Subjective logic-based framework for the evaluation of Web services' security. In: Ruan, D. (ed.) Computational Intelligence: Foundations and Applications, pp. 838–843. World Scientific, New Jersey (2010)
9. Kolaczek, G., Juszczyszyn, K.: Smart Security Assessment of Composed Web Services. Cybernetics and Systems 41(1), 46–61 (2010)
10. Klusch, M., Fries, B., Sycara, K.: OWLS-MX: A hybrid Semantic Web service matchmaker for OWL-S services. Web Semantics: Science, Services and Agents on the World Wide Web 7, 121–133 (2009)
11. Kwiatkowski, J., Fras, M., Pawlik, M., Konieczny, D.: Request Distribution in Hybrid Processing Environments. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 246–255. Springer, Heidelberg (2010)
12. Nanda, M., Chandra, S., Sarkar, V.: Decentralizing execution of composite web services. In: Proceedings of OOPSLA 2004, Vancouver, Canada (2004)
13. Ponnekanti, S.R., Fox, A.: SWORD: A developer toolkit for Web service composition. In: Proceedings of the 11th World Wide Web Conference, Honolulu, HI, USA (2002)
14. Stelmach, P., Grzech, A., Juszczyszyn, K.: A Model for Automated Service Composition System in SOA Environment. In: Camarinha-Matos, L.M. (ed.) DoCEIS 2011. IFIP AICT, vol. 349, pp. 75–82. Springer, Heidelberg (2011)
15. Stelmach, P., Grzech, A., Juszczyszyn, K.: A Model for Automated Service Composition System in SOA Environment. In: Camarinha-Matos, L.M. (ed.) DoCEIS 2011. IFIP AICT, vol. 349, pp. 75–82. Springer, Heidelberg (2011)

Multicore and Accelerator Development for a Leadership-Class Stellar Astrophysics Code

O.E. Bronson Messer[1,2], J. Austin Harris[2], and Suzanne Parete-Koon[1], and Merek A. Chertkow[2]

[1] National Center for Computational Sciences, Oak Ridge National Laboratory Oak Ridge, TN, 37831

[2] Department of Physics & Astronomy, University of Tennessee, Knoxville, TN 37996

Abstract. We describe recent development work on the core-collapse supernova code CHIMERA. CHIMERA has consumed more than 100 million cpu-hours on Oak Ridge Leadership Computing Facility (OLCF) platforms in the past 3 years, ranking it among the most important applications at the OLCF ([1]). Most of the work described has been focused on exploiting the multicore nature of the current platform (Jaguar) via, e.g., multithreading using OpenMP. In addition, we have begun a major effort to marshal the computational power of GPUs with CHIMERA. The impending upgrade of Jaguar to Titan – a 20+ PF machine with an NVIDIA GPU on many nodes – makes this work essential.

Keywords: OpenMP, GPU, OpenACC, supernovae, stellar astrophysics.

1 Introduction

1.1 Overview of the Core-Collapse Supernova Problem

Core-collapse supernovae (CCSN) are among the most energetic events in the Universe, releasing 10^{53} erg (10^{47} Joules) of energy on timescales of a few tens of seconds. They produce and disseminate many of the elements heavier than helium, making life as we know it possible. They mark the birth of neutron stars and black holes and in recent years, it has become apparent that core-collapse supernovae from massive progenitors are associated with long gamma-ray bursts. ([2]; [3]; [4])

As the name suggests, core-collapse supernovae are initiated by the collapse of the iron cores of massive stars at the ends of their lives. The collapse proceeds to ultrahigh densities, in excess of the densities of nucleons in the nucleus of an atom (super-nuclear densities). The inner core becomes incompressible under these extremes, bounces, and, acting like a piston, launches a shock wave into the outer stellar core. This shock wave will ultimately propagate through the stellar layers beyond the core and completely disrupt the star in an explosion. However, in all realistic simulations to date, the shock stalls in the outer core, losing energy as it plows through the still infalling material. Exactly how the shock is revived is unknown. This is the central question in core-collapse supernova theory.

Evidence has accumulated indicating that multidimensional effects play an important and perhaps essential role in the mechanism. On the observational side, spectropolarimetry, the large average pulsar velocities, and the morphology of highly resolved images of SN 1987A all suggest that anisotropy develops very early on in the explosion [e.g., see ([5]) and ([6]) for reviews and references]. On the theoretical side, analyses of immediate post-bounce core profiles given by computer simulations show that a variety of fluid instabilities are present and may play a role in the explosion mechanism.

Supernova simulations must be carried out in two, and preferably three, spatial dimensions for these reasons and others. In addition, 3 x 10^{53} ergs of energy is released by the core in neutrinos of all flavors, and their interaction with the stellar core and mantle will either power the explosion itself or play a major role in the explosion dynamics. An inaccurate treatment of neutrino transport can qualitatively change the results of a simulation. Since neutrinos can originate deep within the core, where neutrino mean free paths are small compared with other relevant length scales, and propagate out to regions where the reverse is true, the transport scheme must be accurate in both regimes plus the all-important intermediate regime where the critical neutrino energy deposition occurs.The nuclear abundances must be evolved in regions where nuclear statistical equilibrium (NSE) cannot be maintained. This will enable the potentially observable products of nucleosynthesis to be followed and the energy released by nuclear burning to be fed back into the computation of the explosion dynamics. Finally, general relativistic effects must be incorporated, as they influence the size of the neutrino heated region, the rate of matter advection through this region, and the neutrino luminosities and RMS energies ([7]). To meet all these requirements, we and our collaborators have developed the CHIMERA code over the past several years ([8]).

1.2 The CHIMERA Code

CHIMERA can well be described as a “chimera” of three, separate, rather mature codes. The codes are tightly coupled in a single executable through a set of interface routines. The primary code modules are designed to evolve the stellar gas hydrodynamics (VH1), the “ray-by-ray-plus” neutrino transport (MGFLD-TRANS), and the thermonuclear kinetics (XNET). These three “heads” are augmented by a sophisticated equation of state for nuclear matter (e.g. LS-EOS ([9])) and a self-gravity solver capable of an approximation to general-relativistic gravity. All of the constituent parts of CHIMERA are written in FORTRAN: MGFLD-TRANS and the LS-EOS are primarily FORTRAN-77, while VH1 and XNET, and all of the associated driver and data-management routines, are written in FORTRAN 90.

The hydrodynamics is directionally split, and the ray-by-ray transport and the thermonuclear kinetics solve occur after the radial sweep occurs, when all the necessary data for those modules is local to a processor (see Figure 1). The individual modules are algorithmically coupled in an operator split approach. This approach is well-motivated, as the characteristic time scales for each module are widely disparate. Specifically, during the radial sweep of the hydrodynamics, the neutrino transport and the thermonuclear burning are computed along each radial ray, using only data that is local to that ray and, therefore, local to the current process. This combination of directionally-split hydrodynamics and operator-split local physics provides the backdrop for the communication and computation patterns found in CHIMERA. Hydrodynamic sweeps are made on "pencils" along one direction of a logically Cartesian mesh. Then, a data transpose (via MPI_ALLTOALLs across sub communicators on the mesh) is performed to switch the sense of the sweeps to one of the orthogonal directions, followed by the next sweep. This procedure can be interleaved in various ways within the operator-split scheme, but a canonical hydro timestep would have sweeps like X-Y-Z-Z-Y-X, i.e. a sweep in the "X-direction" (or, e.g. radius), followed by a Y sweep, followed by a Z sweep, followed by a reverse of that sequence. This decomposition is necessary for the ray-by-ray neutrino transport, as it allows a single "ray" to be resident on a processor at some point in a timestep. This makes the neutrino transport solve a wholly local computation, requiring no communication. Additionally, as the nuclear kinetic equations do not couple neighboring spatial cells, no off-node communication is required for the XNET module. Typical spatial resolutions for the hydrodynamics are 512 radial zones and 64 and 128 zones in the θ and ϕ directions.

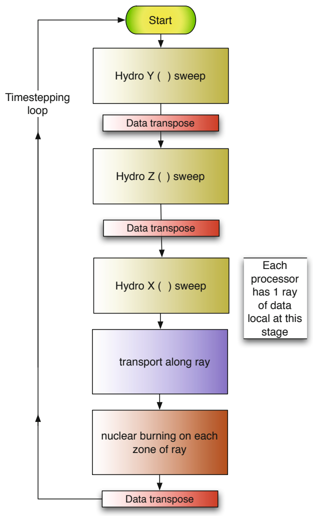


Fig. 1. Schematic CHIMERA flowchart

The hydrodynamics module in CHIMERA is a modified version of the PPM code VH-1, which has been widely used in astrophysical fluid dynamics simulations and as an important benchmark code for a variety of platforms. VH-1 is a Lagrangian remap implementation of the Piecewise Parabolic Method (PPM)

([10]). Being third order in space (for equal zoning) and second order in time, the code is well suited for resolving shocks, composition discontinuities, etc. with modest grid requirements. To avoid the odd-even decoupling and carbuncle phenomenon for shocks aligned parallel to a coordinate axis we have employed the local oscillation filter method of Sutherland et al. (2003) which subjects only a minimal amount of the computational domain to additional diffusion. We have also found it necessary to incorporate the geometry corrections of ([11]) in the hydrodynamics module to avoid spurious oscillations along the coordinate axes. Redshift and time dilation corrections are included in both the hydrodynamics and neutrino transport (to be described later). A moving radial grid option, where the radial grid follows the average radial motion of the fluid, makes it possible for the core infall phase to be followed with good resolution.

Ideally, neutrino transport would be implemented with full multidimensional Boltzmann transport. As a compromise between accuracy and computational intensity, we employ a "ray-by-ray-plus" approximation (cf. ([12])) for neutrino transport, whereby the lateral effects of neutrinos such as lateral pressure gradients (in optically thick conditions), neutrino advection, and velocity corrections are taken into account, but transport is performed only in the radial direction.

The neutrino opacities employed for the simulations are the "standard" ones described in ([13]), with the isoenergetic scattering of nucleons replaced by the more complete formalism of ([14]), which includes nucleon blocking, recoil, and relativistic effects, and with the addition of nucleon–nucleon bremsstrahlung ([15]) with the kernel reduced by a factor of five in accordance with the results of ([16]). Typical energy-space resolutions for the neutrino transport are 20 geometrically spaced groups spanning the range from 5 MeV to 300 MeV.

The equation of state (EOS) of ([9]) is currently employed for matter at high densities. For regions not in NSE, an EOS with a nuclear component consisting of 14 alpha-particle nuclei (^{4}He to ^{60}Zn), protons, neutrons, and an ironlike nucleus is used. An electron-positron EOS with arbitrary degeneracy and degree of relativity spans the entire density-temperature regime of interest. The nuclear composition in the non-NSE regions is evolved by the thermonuclear reaction network of ([17]). This is a fully implicit general purpose reaction network; however, currently we have implemented only a so-called α-network, i.e. only reactions linking the 14 alpha nuclei from ^{4}He to ^{60}Zn are used. Because the α-network neglects reaction flows involving neutron-rich nuclei, it provides only estimates of the energy generation rates for nuclear burning stages encountered in the supernova ($\pm 50\%$ for oxygen burning and $\pm 10\times$ for silicon burning) ([18]).

2 The Need for Hybridization

Many modern codes in use today rely wholly on domain decomposition via MPI for parallelization. New hybrid multicore architectures will demand that this level of parallelism be augmented with SMP-like and vector- like parallelism. Rather, those operations that are performed serially in the MPI-only code will need to be parallelized via a threading mechanism or, perhaps, local MPI communicators on the node, and those operations within the implied loop nests will profit from vector-like parallelization via the GPU's. This discovery or, perhaps in some case, the rediscovery of hierarchical levels of parallelism in current codes will form the heart of a successful programming model on modern platforms, like the newly-installed Titan at Oak Ridge National Laboratory.

2.1 Titan: The First Petascale Hybrid Platform

Titan, a hybrid Cray XK6 system, is the third generation of major capability computing systems at the Department of Energy (DOE) Office of Sciences Oak Ridge Leadership Computing Facility (OLCF) located at the Oak Ridge National Laboratory (ORNL). It is an upgrade of the existing Jaguar system first installed at the OLCF in 2008. The initial upgrade from Cray XT5 to Cray XK6 compute nodes was accepted in February 2012 and consists of 18,688 compute nodes for a total of 299,008 AMD Opteron 6274 "Interlagos" processor cores and 960 NVIDIA X2090 "Fermi" GPUs. The peak performance of the Opteron cores is 2.63 PFLOPS and the peak performance of the GPUs is 638 TFLOPS. In late 2012, the 960 NVIDIA X2090 processors will be removed and replaced with at least 14,592 of NVIDIAs next generation "Kepler" processors with a total system peak performance in excess of 20 PFLOPS.

Compute Nodes. Each of the hybrid compute nodes in Titan consists of one AMD Series 6200 16-core Opteron processor and one NVIDIA Tesla GPU. The GPU and CPUs are connected by a PCI Express Gen 2.0 bus with an 8 GB/second data transfer rate.

The x86 portion of the Titan nodes contain Opterons built of two Interlagos dies per socket. Each of these incorporates four processor groups called Bulldozer modules. Each Bulldozer module contains two independent integer unit cores which share a 256-bit floating point unit, a 2 MB L2 cache, and instruction fetch. A single core can make use of the entire floating point unit with 256-bit AVX instructions.

The four Bulldozer modules share a memory controller and 8MB L3 data cache. The processor die incorporating the four Bulldozer modules is configured with two DDR3 synchronous dynamic random access memory channels and multiple HT3 links. It is important to note here that each Titan node therefore contains 2 NUMA domains, defined by the Interlagos dies on each socket. Memory operations across dies traverse the multiple HT3 links between the dies in a socket ([19]).

The Tesla Kepler GK 110 GPU is composed of groups of streaming multiprocessors (SMX). Each SMX contains 192 single precision streaming processors called CUDA cores. Each CUDA core has pipelined floating point and integer arithmetic logic units. Kepler builds on the previous generation of NVIDA Tesla Fermi GPUs with the same IEEE 754-2008-compliant single and double precision arithmetic, including the fused multiply add operation. A Kepler GK110 GPU has between 13 and 15 SMX units and six 64-bit memory controllers.

In addition to the six fold increase in the number of CUDA cores per SMX compared to the Fermi's 32 CUDA cores per SMs, Kepler GPUs have the ability allow connections from multiple CUDA streams, multiple MPI processes, or multiple threads within a process. This is accomplished though NVIDIA's HyperQ which provides 32 work queues between the host and the GPU, compared to Fermi's single work queue.

Each of Kepler SMXs has one cache of 64 KB on chip memory that can be configured as 48 KB of shared memory with 16 KB of L1 cache, or as 16 KB of shared memory with 48 KB of L1 cache or as a 32KB/32KB split between shared memory and L1 Cache. In addition, each SMX has a 48KB read-only data cache. The Kepler GPU also has 1536 KB of dedicated L2 cache memory.

Interconnect. One of the key differences between the Cray XK6 and prior generation XT systems is the Gemini interconnect ([20]) . Instead of a SeaStar ASIC for each node, each Gemini custom ASIC connects two nodes to the 3-D torus interconnect. All of the cables and backplane interconnections between node boards are the same for SeaStar and Gemini based system. The only difference is the mezzanine card on the node boards. The mezzanine card is a separate printed circuit board that attaches to the base XK6 node board and contains either the SeaStar or Gemini ASIC along with any support circuitry and the interconnections between the SeaStar or Gemini chips. This feature allowed ORNL to upgrade from an XT5/SeaStar system to an XK6/Gemini system while reusing the cabinets, cables, and backplanes.

2.2 Computational Cost of Nuclear Kinetics

Extending the nuclear network approximation from a simplified 14-species α-network to one including 150 species substantially improves upon prior treatments within CHIMERA by extending the capability of the network to track a broad variety of particle captures. Complete nucleosynthesis calculation – e.g. of the r-process – can then be obtained using many thousands of species via post-processing, where a thermodynamic profile is generated using tracer particles throughout the star.

Simplified networks fail to accurately describe both the composition and energy distribution of supernovae ejecta as directly observed. This deficiency has been recognized for some time, leading to the development of post-processing schemes to obtain detailed abundances. In post-processing, a thermodynamic profile generated by tracer particles from an earlier full simulation (including a reduced nuclear network and the associated neutrino transport) is used to evolve a larger nuclear network. A major limitation of this approach is the accuracy of the rate of nuclear energy released by the smaller in-situ network within the hydrodynamics. Since the nucleosynthesis depends on the thermodynamic conditions, and consequently the nuclear energy generation, a feedback exists that cannot be captured with post-processing, significantly affecting the abundances of species such as ^{44}Ti ^{57}Fe, ^{58}Ni and ^{60}Zn ([21]). Another principal limitation of the α-network model is the inability to follow the effects of electron or neutrino capture in neutronization, wherein an electron and proton combine to form a neutron and release neutrinos. This causes the electron fraction ($Y_e = \frac{Z}{A}$), and, concomitantly, the electron pressure, to be miscalculated. Lastly, post-processing is not capable of capturing the observed mixing of the chemical elements due to the lack of coupling to the hydrodynamics.

A network size of approximately 150 is the next logical step in nucleosynthesis calculations (see Figure 2), as it encompasses a significant fraction of elemental abundances and energy-producing reactions important to the core-collapse problem, allowing proper neutronization and a much more accurate rate of nuclear energy generation. Post-processing can then be used to analyze the nucleosynthesis of many-thousand species nuclear networks with post-processing.

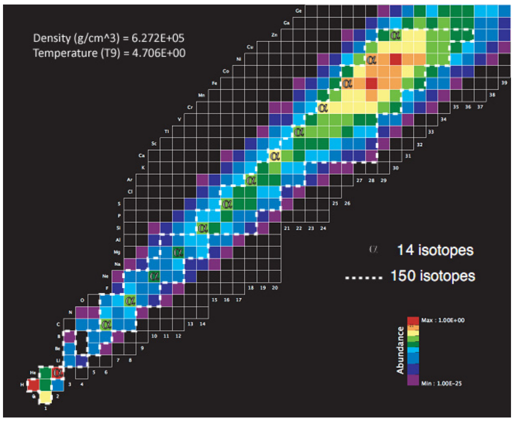


Fig. 2. α-network isotope abundances are represented in the figure, along with the expanded 150-species nuclear network resulting from a constant thermodynamic profile. One can see that this larger network encompasses a large portion of the more heavily populated species. Of particular note is the exclusion of the three most abundant species in the α-network, protons, ^{54}Fe, and ^{58}Ni.

The fully implicit nature of XNet (the nuclear burning module of CHIMERA) necessitates the choice of a suitable integration scheme. Previous work [22] has shown the simple first-order backward Euler method to be most efficient in advancing nuclear abundances within the constraints of the CCSN problem. With this scheme, nuclear abundances, $\mathbf{y}$, are evolved by some change, $\Delta\mathbf{y}$, of the system over a timestep, Δt, according to

$$\mathbf{y_{n+1}} = \mathbf{y_n} + \Delta\mathbf{y}. \quad (1)$$

This is done using the Newton-Raphson method, based on the Taylor series expansion of $\mathbf{y_{n+1}} = \mathbf{y_n} + f(\mathbf{y_{n+1}})$ about a known $f(\mathbf{y_n})$. This reduces to iteratively solving the N^2 dense matrix equation $\tilde{\mathbf{A}}\mathbf{x} = \mathbf{b}$ in the form

$$\left(\frac{\tilde{\mathbf{I}}}{\Delta t} - \tilde{\mathbf{J}}\right) \Delta \mathbf{y} = f(\mathbf{y_n}), \tag{2}$$

where $\tilde{\mathbf{J}}$ is the Jacobian of $f(\mathbf{y_n})$. Iteration continues until the solution converges according to mass conservation or some more stringent abundance conservation test, the choice of which depends upon the desired accuracy. Each iteration requires computing the full set of abundance derivatives, calculating all reaction rates, evaluating the Jacobian, evaluating the right-hand side, and then performing one matrix decomposition and backsubstitution. Complicating the computation itself is indirect memory addressing and loop carried dependencies associated with building the Jacobian. Double-precision REALS are required for the calculation, as the approach to equilibrium at various stages of the burning can lead to the near-cancellation of large fluxes.

With the computational time to evolve the network using a dense matrix solution with packages such as LAPACK being $\mathcal{O}(N^3)$ ([23]), moving from an α-network to a more realistic 150-species network can make the nucleosynthesis computation more expensive than the neutrino transport. Initial analysis reveals that increasing the number of species from 14 to 150 more than doubles the cost of a CHIMERA simulation without further development. In order to prevent the computational cost from limiting the scope of our studies, we must incorporate recent advancements in programming interfaces and computational architectures, specifically shared-memory parallelism (OpenMP) and general purpose graphical accelerators (GPGPUs). All of the comparisons made below are to a strictly serial, but fairly highly vectorized version of XNET used in CHIMERA to this point. Compiler-generated SSE instructions have been shown (via PAPI) to increase the the speed of XNET from 21% of theoretical peak to more than 67% of peak. This is the baseline from which we measure subsequent performance below. We note that the availability of new AVX instructions might lead to even better vectorization on the CPU, but, at present, require some recoding for the Bulldozer architecture (i.e. on Titan).

2.3 Thread Scheduling

The advent of multicore processors presents an opportunity for another layer of parallelism in CHIMERA. By taking advantage of the shared-memory property of multiple processing cores on a single chip, we can extend the level of parallel computation within CHIMERA. To do this, we assign parallelizable loops over radial zones being executed by a single MPI task to multiple threads with the OpenMP API. Load balance is achieved by specifying a proper scheduling policy for the manner in which work is divided among the threads. The options available for this policy in OpenMP are *static*, *dynamic*, or *guided*. Additional options allow for the compiler or runtime system to delegate work among the threads.

2.4 Core Affinity

It is also important to consider how the parallelization scheme maps to the computer architecture (i.e. core affinity). With the CRAY XK6, we have found that the optimal scheme for nuclear burning is to place four threads per MPI task, with these threads spread across two Bulldozer modules on two NUMA domains. One crucial aspect associated with choosing the number of OpenMP threads per MPI task is the relative "threadedness" of CHIMERA outside of nuclear burning. Here we investigate an optimal parallelization scheme based on the performance of one CHIMERA module–the nuclear burning network–which is subject to change in response to future updates to OpenMP implementation in CHIMERA.

2.5 OpenMP Summary

We consider the maximum nuclear burning time over a typical portion of a 1D simulation using a 150-species nuclear reaction network. The simulation time is chosen such that the nuclear burning is typical, i.e., after core bounce when the temperature is high. This allows us to make a simple, but effective, comparison between different scheduling schemes, the number of threads per task, and their mapping to the hardware architecture. We studied four- and eight-thread

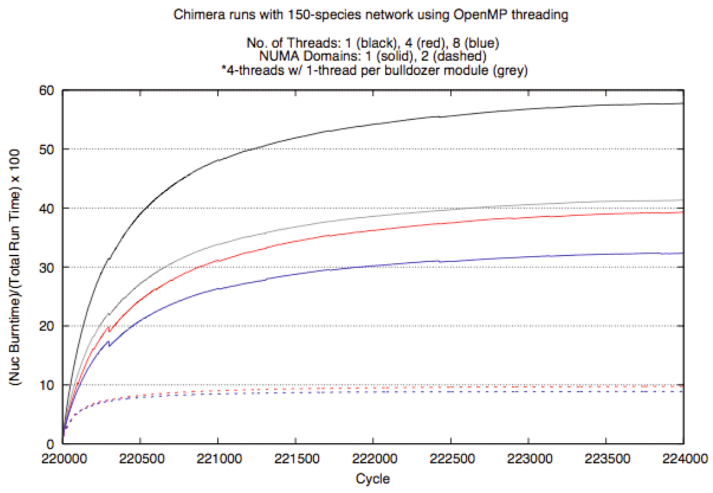


Fig. 3. Time spent in nuclear burning computation is plotted as a percentage of total CHIMERA time for a 150-species realistic network. The x-axis is cycle number (i.e. hydrodynamic time step number) from the benchmark simulation. The percentage of time dedicated to nuclear burning increases from zero as the simulation is restarted from an intermediate state. The final, asymptotic values for each computation would be typical of a long-running simulation.

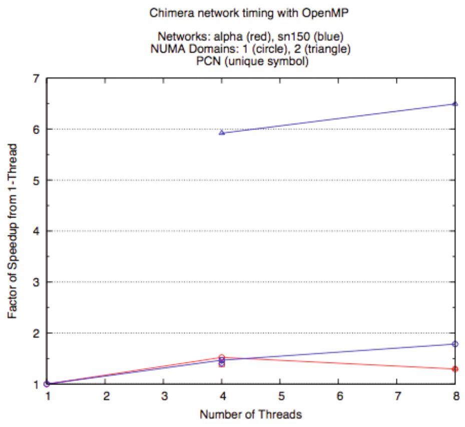


Fig. 4. Speedup of a CHIMERA simulation computation is plotted for an α-network and 150-species realistic network for different OpenMP configurations. The product of the number of MPI tasks and the number of OpenMP threads is held constant in each case.

performance across different NUMA domains and Bulldozer module affinities. The results are summarized in Figures 3 and 4.

This comparison clearly shows that the optimal core affinity is to schedule four threads per MPI task on two Bulldozer modules from separate NUMA domains. This configuration reduces time spent in nuclear burning from $\sim$ 60 percent of the total CHIMERA runtime to $\sim$ 8 percent of the total CHIMERA runtime. This equates to a factor of speedup from single thread performance of $\sim$ 6 in runtime for an entire CHIMERA simulation.

3 GPU Parallelization

3.1 XNet

In addition to OpenMP, GPUs offer another avenue to exploit parallelism in CHIMERA. In an effort to remove the false serialization produced by launching GPU kernels from separate CPU threads (present in CUDA versions earlier than 5.0), we investigate GPU performance from a single thread per MPI task.

We surveyed the FORTRAN compilers of CRAY and PGI using the default level-two optimization settings. More aggressive optimization did not improve the computation time for any of the tested compilers. Three different accelerated linear algebra packages were tested and compared for the decomposition and backsubstitution: CULA, LibSci Accelerator (LibSciACC), and MAGMA. When possible, a host-interface was compared against device-interface implementation. The host-interface requires no modification to existing code. In this scheme, device memory is managed by the library routine. With the device-interface approach, code must be adapted to manage the device memory manually, either directly with CUDA or by a high-level compiler directives API. In this study, we tested OpenACC and CUDA to build the matrix and right-hand-side for the device-interface decomposition and backsubstitution. Nuclear reaction networks of 14, 150, 365, 1072, and 2184 species were tested for each setup. All configurations were compared to the fastest vectorized implementation of XNet using CRAY LibSci LAPACK routines and the Cray Compiling Environment.

In all test cases, the nucleosynthesis was performed in a post-processing fashion with a standalone version of XNet using a representative CCSN thermodynamic profile ([24]) with typical initial abundances found in CCSN conditions.

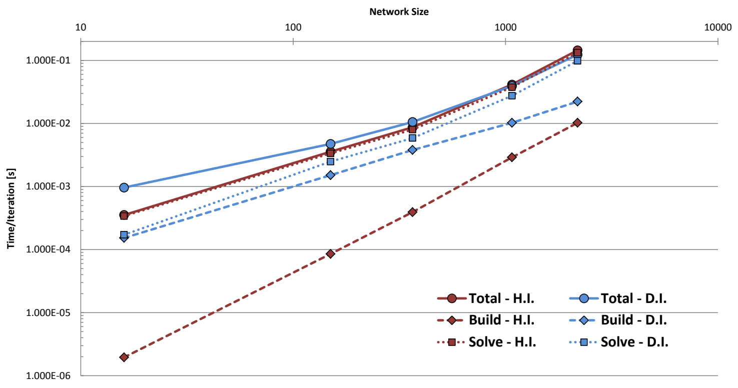


Fig. 5. Host-interface (H.I.) to device-interface (D.I.) comparison for CULA. The total computation time per N-R iteration is plotted and subdivided into time spent building the Jacobian (Build) and solving the matrix equation $\tilde{\mathbf{A}}\mathbf{x} = \mathbf{b}$ (Solve).

The first GPU-optimized linear algebra package surveyed was EM Photonics' CULA with the PGI programming environment. The host-interface was tested by calling the function *CULA_DGESV* and involved no modification to the code itself. The device-interface was also implemented using the PGI compiler with the *CULA_DEVICE_DGESV* routine. This implementation required managing the device memory with a combination of compiler directives and CUDA Fortran. The CULA package did not recognize device memory addresses passed with the OpenACC directive *host_data use_device*, so we were unable to test its capability with the Cray compiler. Instead, the Jacobian and right-hand side were declared as CUDA *device* type variables and assigned directly to the GPU, allowing for more efficient memory management via fewer host-device data transfers. Figure 5 summarizes these results.

The second set of linear algebra packages tested was the Cray XK6 optimized cuBLAS and LAPACK routines–collectively referred to as LibSci Accelerator (LibSciACC). These results are shown in Figure 6. The routines in this library package do not contain an explicit host-interface, but for comparison, we can

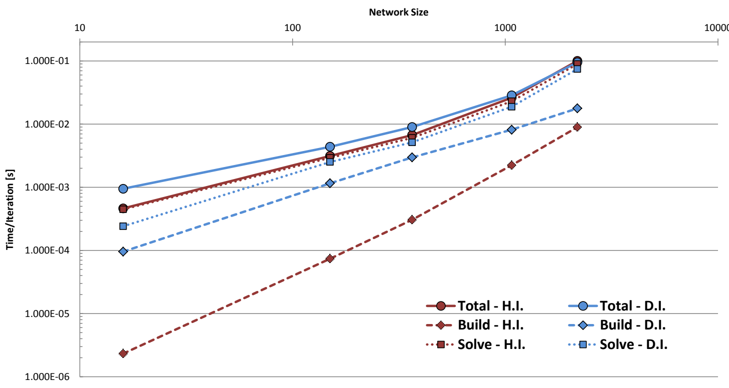


Fig. 6. Host-interface (H.I.) to device-interface (D.I.) comparison for LibSciACC. The total computation time per N-R iteration is plotted and subdivided into time spent building the Jacobian (Build) and solving the matrix equation $\tilde{\mathbf{A}}\mathbf{x} = \mathbf{b}$ (Solve).

emulate the behavior with a device-interface implementation in which we copy the Jacobian and right-hand side to the device immediately prior to the decomposition with OpenACC. The decomposition is forced to be computed on the accelerator with *DGETRF_ACC* and *DGETRS_ACC*. We extend the directives to compute abundance derivatives, evaluate the Jacobian, and build the right-hand side for the device-interface comparison. PGI's current adaptation of OpenACC does not include a directive clause to pass device memory addresses to library routines, so only the Cray compiler was tested for LibSciACC.

By comparing Figures 5 and 6, we see that the method of emulating a host-interface with LibSci Accelerator matches the behavior of the CULA host-interface. In both cases, the matrix solve dominates the time for the computation, especially for larger networks. For smaller networks, we see that building the Jacobian on the device has a non-negligible impact on the computation. As the network size grows, the performance of the GPU in building the Jacobian approaches that of the serial code in the host-interface. For CULA dense and LibSci Accelerator, the host-interface approach performs slightly better than the device-interface approach with the exception of very large networks (i.e. greater than 2000 species), at least with current Fermi GPU kernel queuing.

MAGMA–Matrix Algebra on GPU and Multicore Architectures–is a collection of linear algebra libraries developed by the Innovative Computing Laboratory at the University of Tennessee. MAGMA was developed with NVIDIA's native CUDA C programming language, which presents some complications when interfacing with the Fortran-coded XNet. We were unable to perform a successful test case using the device-interface MAGMA routines, but did have some success with the host-interface for both the Cray and PGI compilers. These results are shown in Figure 7. We note that there is little difference in the compilers themselves when directives are not used.

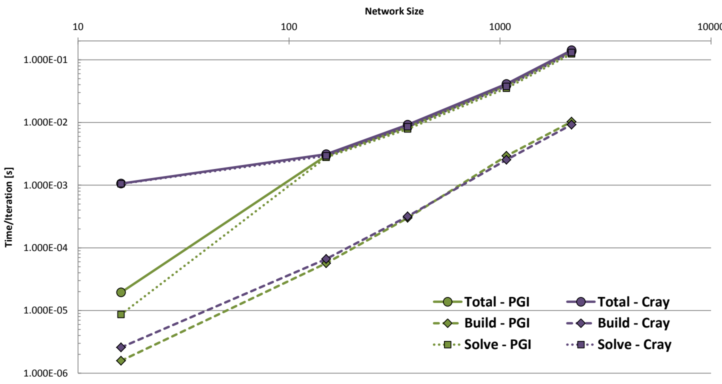


Fig. 7. Compiler comparison between implementations of the MAGMA host-interface routines for the matrix solve.

3.2 GPU Summary

We achieve the best performance when we use the LibSciACC routines with a host-interface memory management scheme, though there appears to be only a small amount of variability between all configurations. In this implementation, device memory is managed with OpenACC immediately prior to the matrix solve by copying the Jacobian and right-hand side to the GPU. Even in the best-case scenario, the small problem size prevents us from seeing a gain in performance. With the upcoming capability associated with CUDA 5.0 and the Kepler GK110 GPU, this constraint will be somewhat ameliorated by Hyper-Q technology. This will allow us to launch multiple nuclear module kernels simultaneously on the GPU, each corresponding to a different spatial zone in the CHIMERA domain decomposition.

Acknowledgments. O.E.B.M., J. A. H., and S. P.-K. received support from the Department of Energy Office of Advanced Scientific Computing Research. M. A. C. received support from the NSF PetaApps Program (grant number OCI-0749242). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Joubert, W., Su, S.Q.: An analysis of computational workloads for the ORNL Jaguar system. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, pp. 247–256. ACM, New York (2012)

[2] Matheson, T., Garnavich, P.M., Stanek, K.Z., Bersier, D., Holland, S.T., Krisciunas, K., Caldwell, N., Berlind, P., Bloom, J.S., Bolte, M., Bonanos, A.Z., Brown, M.J.I., Brown, W.R., Calkins, M.L., Challis, P., Chornock, R., Echevarria, L., Eisenstein, D.J., Everett, M.E., Filippenko, A.V., Flint, K., Foley, R.J., Freedman, D.L., Hamuy, M., Harding, P., Hathi, N.P., Hicken, M., Hoopes, C., Impey, C., Jannuzi, B.T., Jansen, R.A., Jha, S., Kaluzny, J., Kannappan, S., Kirshner, R.P., Latham, D.W., Lee, J.C., Leonard, D.C., Li, W., Luhman, K.L., Martini, P., Mathis, H., Maza, J., Megeath, S.T., Miller, L.R., Minniti, D., Olszewski, E.W., Papenkova, M., Phillips, M.M., Pindor, B., Sasselov, D.D., Schild, R., Schweiker, H., Spahr, T., Thomas-Osip, J., Thompson, I., Weisz, D., Windhorst, R., Zaritsky, D.: Photometry and Spectroscopy of GRB 030329 and Its Associated Supernova 2003dh: The First Two Months. ApJ 599, 394–407 (2003)

[3] Galama, T.J., Vreeswijk, P.M., van Paradijs, J., Kouveliotou, C., Augusteijn, T., Bohnhardt, H., Brewer, J.P., Doublier, V., Gonzalez, J.F., Leibundgut, B., Lidman, C., Hainaut, O.R., Patat, F., Heise, J., in 't Zand, J., Hurley, K., Groot, P.J., Strom, R.G., Mazzali, P.A., Iwamoto, K., Nomoto, K., Umeda, H., Nakamura, T., Young, T.R., Suzuki, T., Shigeyama, T., Koshut, T., Kippen, M., Robinson, C., de Wildt, P., Wijers, R.A.M.J., Tanvir, N., Greiner, J., Pian, E., Palazzi, E., Frontera, F., Masetti, N., Nicastro, L., Feroci, M., Costa, E., Piro, L., Peterson, B.A., Tinney, C., Boyle, B., Cannon, R., Stathakis, R., Sadler, E., Begam, M.C., Ianna, P.: An unusual supernova in the error box of the gamma-ray burst of 25 April 1998. Nature, 670–672 (1998)

[4] Gehrels, N., Sarazin, C.L., O'Brien, P.T., Zhang, B., Barbier, L., Barthelmy, S.D., Blustin, A., Burrows, D.N., Cannizzo, J., Cummings, J.R., Goad, M., Holland, S.T., Hurkett, C.P., Kennea, J.A., Levan, A., Markwardt, C.B., Mason, K.O., Meszaros, P., Page, M., Palmer, D.M., Rol, E., Sakamoto, T., Willingale, R., Angelini, L., Beardmore, A., Boyd, P.T., Breeveld, A., Campana, S., Chester, M.M., Chincarini, G., Cominsky, L.R., Cusumano, G., de Pasquale, M., Fenimore, E.E., Giommi, P., Gronwall, C., Grupe, D., Hill, J.E., Hinshaw, D., Hjorth, J., Hullinger, D., Hurley, K.C., Klose, S., Kobayashi, S., Kouveliotou, C., Krimm, H.A., Mangano, V., Marshall, F.E., McGowan, K., Moretti, A., Mushotzky, R.F., Nakazawa, K., Norris, J.P., Nousek, J.A., Osborne, J.P., Page, K., Parsons, A.M., Patel, S., Perri, M., Poole, T., Romano, P., Roming, P.W.A., Rosen, S., Sato, G., Schady, P., Smale, A.P., Sollerman, J., Starling, R., Still, M., Suzuki, M., Tagliaferri, G., Takahashi, T., Tashiro, M., Tueller, J., Wells, A.A., White, N.E., Wijers, R.A.M.J.: A short γ-ray burst apparently associated with an elliptical galaxy at redshift z = 0.225. Nature 437, 851–854 (2005)

[5] Arnett, W.D., Bahcall, J.N., Kirshner, R.P., Woosley, S.E.: Supernova 1987a. ARA&A 27, 701–756 (1987)

[6] McCray, R.: Supernova 1987A revisited. ARA&A 31, 175–216 (1993)

[7] Bruenn, S.W., De Nisco, K.R., Mezzacappa, A.: General Relativistic Effects in the Core Collapse Supernova Mechanism. ApJ 560, 326–338 (2001)

[8] Messer, O.E.B., Bruenn, S.W., Blondin, J.M., Hix, W.R., Mezzacappa, A.: Multidimensional, multiphysics simulations of core-collapse supernovae. J. Phy. Conf. Series 125(1), 012010 (2008)

[9] Lattimer, J., Swesty, F.D.: A Generalized Equation of State for Hot, Dense Matter. Nucl. Phys. A 535, 331–376 (1991)

[10] Colella, P., Woodward, P.: The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations. J. Comp. Phys. 54, 174–201 (1984)

[11] Blondin, J.M., Lundqvist, P.: Formation of the circumstellar shell around SN 1987A. ApJ 405, 337–352 (1993)
[12] Buras, R., Rampp, M., Janka, H.T., Kifonidis, K.: Two-dimensional hydrodynamic core-collapse supernova simulations with spectral neutrino transport. I. Numerical method and results for a 15 $M_\odot$ star. A&A 447, 1049–1092 (2006)
[13] Bruenn, S.W.: Stellar core collapse - Numerical model and infall epoch. ApJS 58, 771–841 (1985)
[14] Reddy, S., Prakash, M., Lattimer, J.M.: Neutrino interactions in hot and dense matter. Phy. Rev. D 58, 013009 (1998)
[15] Hannestad, S., Raffelt, G.: Supernova Neutrino Opacity from Nucleon-Nucleon Bremsstrahlung and Related Processes. ApJ 507, 339–352 (1998)
[16] Hanhart, C., Phillips, D.R., Reddy, S.: Neutrino and axion emissivities of neutron stars from nucleon-nucleon scattering data. Phys. Lett. B 499, 9 (2001)
[17] Hix, W.R., Thielemann, F.K.: Silicon Burning II: Quasi-Equilibrium and Explosive Burning. ApJ 511, 862–875 (1999)
[18] Timmes, F.X., Hoffman, R.D., Woosley, S.E.: An Inexpensive Nuclear Energy Generation Network for Stellar Hydrodynamics. ApJS 129, 377–398 (2000)
[19] Numerical Algorithms Group Ltd.: How to make best use of the AMD Interlagos processor (November 2011), http://www.hector.ac.uk/cse/reports/interlagos_whitepaper.pdf
[20] Cray, Inc.: Cray Gemini interconnect (May 2012), http://www.cray.com/Products/XE/Technology.aspx
[21] Woosley, S.E., Weaver, T.A.: The Evolution and Explosion of Massive Stars. II. Explosive Hydrodynamics and Nucleosynthesis. ApJS 101, 181–235 (1995)
[22] Thielemann, F.K.: private communication (1993)
[23] Timmes, F.X.: Integration of Nuclear Reaction Networks for Stellar Hydrodynamics. ApJS 124, 241–263 (1999)
[24] Fröhlich, C., Martínez-Pinedo, G., Liebendörfer, M., Thielemann, F.K., Bravo, E., Hix, W.R., Langanke, K., Zinner, N.T.: Neutrino-induced nucleosynthesis of A>64 nuclei: The νp process. Physical Review Letters 96(14), 142502 (2006)

Steering and In-situ Visualization for Simulation of Seismic Wave Propagation on Graphics Cards

David Michéa, Joachim Pouderoux, Fabrice Dupros, and Hideo Aochi

BRGM
BP 6009, 45060 Orléans Cedex 2, France
{d.michea,j.pouderoux,f.dupros,h.aochi}@brgm.fr

Abstract. Simulation of large scale seismic wave propagation is an important tool in seismology for efficient strong motion analysis and risk mitigation. Being particularly CPU-consuming, this three-dimensional problem has been early ported on graphics cards to improve the performance by several order of magnitude. Scientific visualization of data produced by these simulations is essential for a good comprehension of the physical phenomena involved. In the same time, post-petascale architectures demonstrates that the I/O turn to become a major performance bottleneck. This situation is worsened with GPU-based systems because of the gap between I/O bandwidth and computational capabilities. In this paper, we introduce a prototype of computational steering and in-situ visualization suitable for seismic wave propagation on hybrid architecture. We detail the overall architecture of the system we set up and comment on the parallel performance measured.

Keywords: computational steering, in-situ visualization, GPU, seismic wave propagation.

1 Introduction

One of the most widely used techniques for the numerical simulation of seismic wave propagation is the finite-difference method because of its simplicity and numerical efficiency. Most of the parallel implementations routinely used in the scientific community are based on cluster architectures and rely on the MPI library with a good parallel efficiency on tens of thousands cores. Other approaches are focused on the exploitation of Graphics Processing Unit (GPU) with significant acceleration in comparison with classical CPU implementation [1]. These improvements lead to an inflation of the amount of data produced and prevent from the use of naive approach to dump the data on storage systems. In these cases, I/O turn to become the major bottleneck which limits the performance of the simulation. A classical strategy is to restrict the amount of data that would be saved. A sub-sampling is generally preformed in time or in space with the limitation for the scientist of missing possible localized phenomena.

As [2], we believe that in-situ visualization and computational steering provide a valuable solution for the scientist in order to interact with the simulation and explore the three-dimensional domain of computation. Unfortunately current existing solutions like [3,4] are not suitable for GPU applications as they assume that data are directly accessible on the host. One of the key of the efficiency of GPU codes is that all data have to remain in the graphic card memory in order to maintain their performance as the memory transfers between host (CPU) and device (GPU) are in general very expensive.

We therefore introduce a simple strategy suitable for hybrid applications that allows us to interact with the *Ondes3D* software package developed for earthquake modeling [5].

2 Seismic Wave Modeling on Hybrid Architectures

2.1 Governing Equations

The seismic wave equation in the case of an elastic linear material is given in three dimensions by

$$\begin{cases} \rho\frac{\partial}{\partial t}v_x &= \frac{\partial}{\partial x}\sigma_{xx} + \frac{\partial}{\partial y}\sigma_{xy} + \frac{\partial}{\partial z}\sigma_{xz} + f_x \\ \rho\frac{\partial}{\partial t}v_y &= \frac{\partial}{\partial x}\sigma_{yx} + \frac{\partial}{\partial y}\sigma_{yy} + \frac{\partial}{\partial z}\sigma_{yz} + f_y \\ \rho\frac{\partial}{\partial t}v_z &= \frac{\partial}{\partial x}\sigma_{zx} + \frac{\partial}{\partial y}\sigma_{zy} + \frac{\partial}{\partial z}\sigma_{zz} + f_z \end{cases} \tag{1}$$

$$\begin{cases} \frac{\partial}{\partial t}\sigma_{xx} &= \lambda\left(\frac{\partial}{\partial x}v_x + \frac{\partial}{\partial y}v_y + \frac{\partial}{\partial z}v_z\right) + 2\mu\frac{\partial}{\partial x}v_x \\ \frac{\partial}{\partial t}\sigma_{yy} &= \lambda\left(\frac{\partial}{\partial x}v_x + \frac{\partial}{\partial y}v_y + \frac{\partial}{\partial z}v_z\right) + 2\mu\frac{\partial}{\partial y}v_y \\ \frac{\partial}{\partial t}\sigma_{zz} &= \lambda\left(\frac{\partial}{\partial x}v_x + \frac{\partial}{\partial y}v_y + \frac{\partial}{\partial z}v_z\right) + 2\mu\frac{\partial}{\partial z}v_z \\ \frac{\partial}{\partial t}\sigma_{xy} &= \mu\left(\frac{\partial}{\partial y}v_x + \frac{\partial}{\partial x}v_y\right) \\ \frac{\partial}{\partial t}\sigma_{xz} &= \mu\left(\frac{\partial}{\partial z}v_x + \frac{\partial}{\partial x}v_z\right) \\ \frac{\partial}{\partial t}\sigma_{yz} &= \mu\left(\frac{\partial}{\partial z}v_y + \frac{\partial}{\partial y}v_z\right). \end{cases} \tag{2}$$

v and σ represent the velocity and stress field respectively and f denotes a known external source force. ρ is the material density, λ and μ are the elastic coefficients known as Lamé parameters.

2.2 Description of the Numerical Scheme

The dominant numerical scheme to solve the above equations is certainly the explicit finite-difference method. It has been introduced in [6] for a second-order spatial approximation and has been extended in [7] to consider a fourth-order accurate stencil in space and a second-order stencil in time. One of the key features

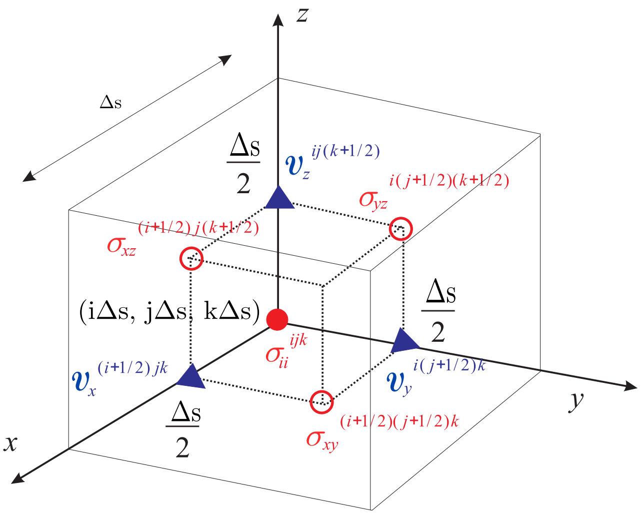


Fig. 1. Elementary 3D cell of the staggered grid and distribution of the stress (σ) and the velocity (v) components

of this scheme is the introduction of a staggered-grid [8] for the discretization of the seismic wave equation.

For classical collocated methods over a regular Cartesian grid, all the unknowns are evaluated at the same location, whereas the staggered grid leads to a shift of the derivatives by half a grid cell. The use of a staggered grid improves the overall quality of the scheme (in terms of numerical dissipation and stability), especially in the case of strong material heterogeneity. Figure 1 shows the elementary 3D cell and the distribution of the stress and velocity components. Exponents i, j, k indicate the spatial direction with ($\sigma^{ijk} = \sigma(i\Delta s, j\Delta s, k\Delta s)$, Δs corresponds to the space step and Δt to the time step. The off-diagonal stress tensor components are shifted in space by half an interval in two directions and the velocity components are shifted both in time and in space by a time step and by half a space time.

2.3 Implementation on Hybrid Architectures

In recent years, GPU computing has been extensively used to accelerate applications in various computational domains and the scientific literatures on this topic is abundant. Regarding the finite-differences method, several applications have been ported to GPUs as early as 2004 [9,10], particularly for seismic wave modelling [11,1].

These seismic applications were mainly based on the acoustic case that is less complex in terms of number of unknowns. The *Ondes3D* code implements the full three-dimensional elastic wave equation with CPML absorbing conditions [12]. The high reading redundancy (13/1) coming from the fourth-order stencil used in space makes GPU a very efficient architecture for such memory-bound applications. An average speedup of a factor 20 [13] has been measured in our case. The implementation relies on algorithms introduced in [1], with the additional use of texture fetching in CUDA to compensate for the lack of shared memory on the graphics card, and with the use of message passing (MPI) when several GPUs are used in parallel.

3 Steering Environment

3.1 Server Implementation

The server is composed of the hybrid simulation code with an additional MPI process called master node. This master node is synchronized with the time loop of simulation nodes, but does not compute the simulation. It launches a TCP server in a thread to communicate with the client, broadcast the requests and events received from the client to the computing nodes. Finally, it receives the data from the computing nodes, assembles them, and notifies the TCP server of their availability. Figure 2 described the server communications. The master

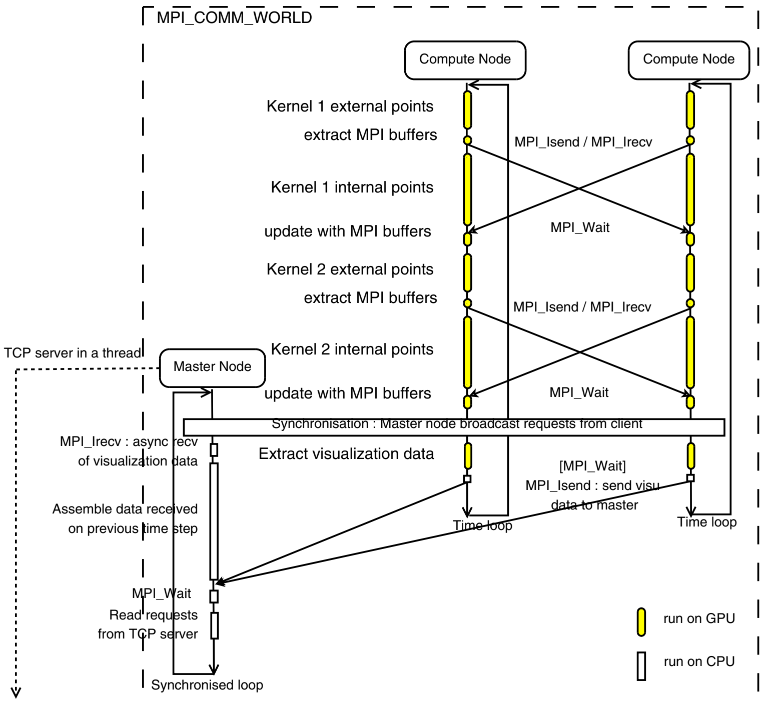


Fig. 2. Sequence diagram of the simulation server

node is synchronized with the time loop to ensure coherency between requests and responses. The use of the MPI library greatly facilitates synchronization and fast communication with the computation nodes. The use of a specialized thread for the TCP server allows to run asynchronously with the simulation. This way it can answer to the client's request without delay. To limit the amount of data transferred, we extract the data of interest at each time step on a cutting plane. This plane can be oriented and moved by the user with 6 degrees of freedom. Data are extracted on the computing nodes with CUDA kernels and sent to the master node at each time step. Data transmission from the computing nodes to the master node is done via MPI asynchronous communications.

3.2 Client Implementation

The visualization client connects on the server using TCP and transmits the events and user requests like flow control instructions (pause, play, record, rewind, restart) or the coordinates of the visualization plan. It enquires for new available data based on polling mechanisms at each render loop iteration (see Figure 3). Therefore the client receives the assembled raw data (eg. speed, displacement or acceleration), the timestep and visualization plan coordinates. The client copies the raw data in its graphic memory, computes it with a CUDA kernel to fill a Vertex Buffer Object (*VBO*) and then renders it using OpenGL library. In order to guarantee the coherency of the workflow, the visualization plane displayed corresponds to the data received and not to the place sent to the server. This causes a slight time shift between the moment the user starts to move the plane and the moment it actually moves.

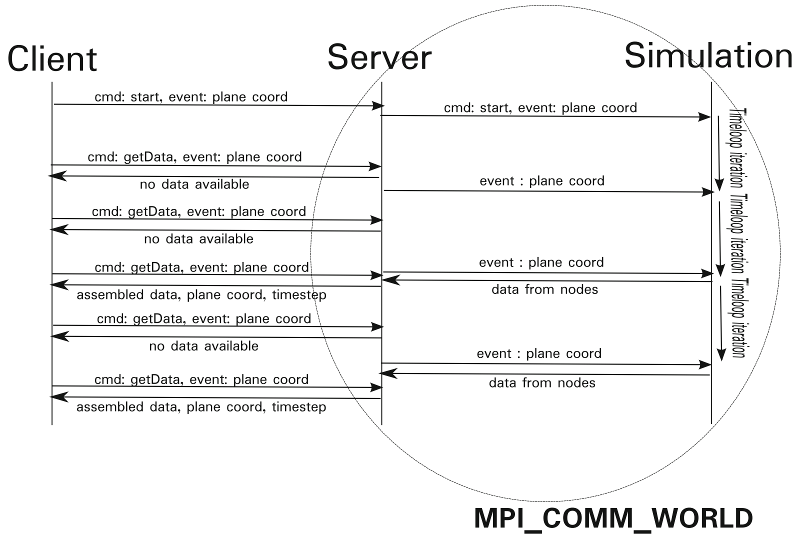

Client
Server
Simulation
cmd: start, event: plane coord
cmd: start, event: plane coord
cmd: getData, event: plane coord
no data available
event : plane coord
cmd: getData, event: plane coord
no data available
cmd: getData, event: plane coord
event : plane coord
assembled data, plane coord, timestep
data from nodes
cmd: getData, event: plane coord
no data available
event : plane coord
cmd: getData, event: plane coord
data from nodes
assembled data, plane coord, timestep
Timeloop iteration
Timeloop iteration
Timeloop iteration
MPI_COMM_WORLD

Fig. 3. The client continuously poll the server for new data to display

3.3 Experiments

Each computing node of our experimental setup is composed of a video card NVIDIA GeForce 8800 GTX installed in a quad-processor dual-core 64-bit AMD Opteron 2.8 GHz with 8 gigabytes of RAM memory. The 8800 GTX card has 16 multiprocessors, i.e., 128 cores, and 768 megabytes of memory. The bandwidth for the memory transfers is 86.4 gigabytes per second, with a memory bus width of 384 bits. This configuration is not up-to-date but it corresponds to a six nodes computing cluster that drives a virtual reality workbench (i.e Holobench from Barco). The nodes are interconnected with a Myrinet network. This architecture provides a good opportunity to evaluate our client/server steering environment along with a large display facility. Standard experiments have also been carried out on most recent configurations, particularly on workstation, in order to evaluate the performance of our methodology in a 1x1 configuration. We wrote two versions of the client depending on the targeted architecture (see Figure 4). The first one is based on QT and is suitable for standard Workstation. The other one is designed for the virtual reality workbench. We use VR Juggler [14] to manage multiple screens and interaction devices (6DOF wand, keyboard).

We use a 3D geological model of size 24000 m $\times$ 24000 m $\times$ 21000 m discretized using a grid of 240 $\times$ 240 $\times$ 210 points, i.e., with grid cells of size 100 m in the three spatial directions. The *Ondes3D* code is running on four graphics cards. The model is heterogeneous and composed of two horizontal elastic layers in contact. We take a time step of 8 ms and we simulate a total of 1000 time steps, i.e., a total duration of 8 s. CPML absorbing layers of a thickness of ten grid points are implemented on all the edges of the grid except the free surface.

First remark is that the communications are perfectly overlapped for large enough models (a few hundred of megabytes). This is coming from the ratio between the data exchanged (two-dimensional) compared with the computational domain (three-dimensional). This overlap also comes from the assembly of data received at the previous iteration for the master node. Thus, during the simulation, we visualize the data calculated at the previous time step. When the simulation is paused by the user at time step T, the computation actually stops at time step T+1.

We have measured the additional cost coming from in-situ visualization and the overhead is less than 3%. This is mainly due to the extraction kernels that interpolate the data with the cutting plane. Another limitation for the performance is coming from the MPI broadcasts from the master node. This extra-cost of 3% represents the upper-bound of the overhead when the size of the computational domain is varying. Considering our experimental architecture, any extrapolation on large scale configuration is difficult as the number of servers, the number of clients and the size of the computational domain will have a significant impact on the performance.

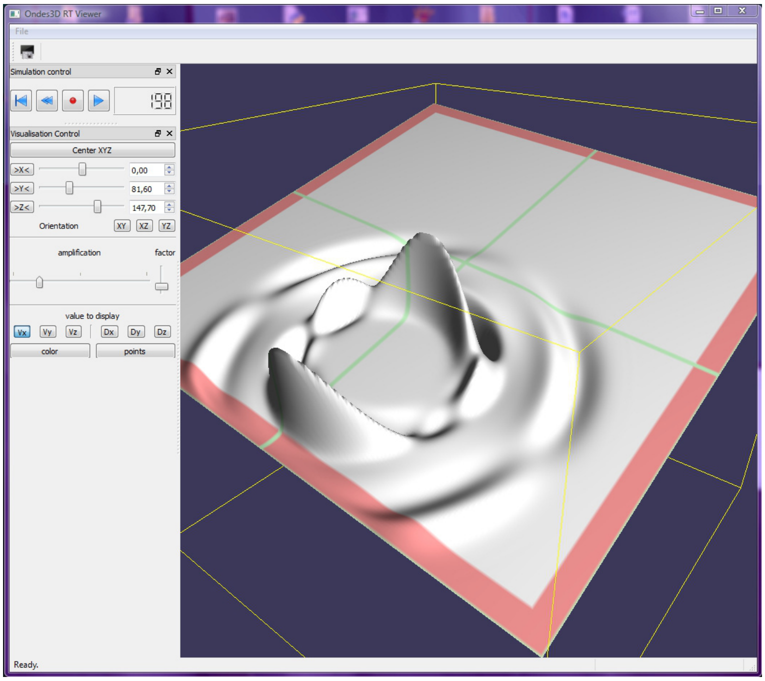


Fig. 4. Screenshot of our client implementation. Visualization of a run on twelve millions of grid points on four graphics cards. In green, the MPI border and in red the absorbing boundary conditions (CPML).

4 Conclusion

A steering environment suitable for GPU applications has been proposed. We have demontrated the efficiency of our methodology based on the standard elastodynamic equations on a cluster with four computing GPU. The overhead coming from the steering is an average of 3% mainly coming from the extraction of the relevant data. We believe that this kind of approach is a possible way to consider future exascale simulations with huge amount of data available for post-processing.

Significant efforts should be provided in the near future to enhance the scalability of our methodology, for instance by introducing more asynchronism in the workflow. Hierarchical patterns could also be considered for the gathering of the data. The relevant filters required to analyze the data should also be adapted on graphics card (i.e isosurface extraction).

References

1. Micikevicius, P.: 3D finite-difference computation on GPUs using CUDA. In: GPGPU-2: Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, DC, USA, pp. 79–84 (2009)

2. Ma, K.-L.: In situ visualization at extreme scale: Challenges and opportunities. IEEE Comput. Graph. Appl. 29, 14–19 (2009)
3. Eickermann, T., Frings, W., Gibbon, P., Kirtchakova, L., Mallmann, D., Visser, A.: Steering unicore applications with visit. Philosophical Transactions of the Royal Society of London Series A 363, 1855–1865 (2005)
4. James, G.I., Geist, G.A., James, I., Kohl, A., Papadopoulos, P.M.: Cumulvs: Providing fault-tolerance, visualization and steering of parallel applications. International Journal of High Performance Computing Applications 11, 224–236 (1996)
5. Aochi, H., Ducellier, A., Dupros, F., Delatre, M., Ulrich, T., de Martin, F., Yoshimi, M.: Finite Difference Simulations of Seismic Wave Propagation for the 2007 Mw 6.6 Niigata-ken Chuetsu-Oki Earthquake: Validity of Models and Reliable Input Ground Motion in the Near-Field. Pure Appl. Geophys., 1–22 (2007)
6. Virieux, J.: P-SV wave propagation in heterogeneous media: velocity-stress finite-difference method. Geophysics 51, 889–901 (1986)
7. Levander, A.R.: Fourth-order finite-difference *P-SV* seismograms. Geophysics 53, 1425–1436 (1988)
8. Madariaga, R.: Dynamics of an expanding circular fault. Bull. Seismol. Soc. Am. 66(3), 639–666 (1976)
9. Krakiwsky, S.E., Turner, L.E., Okoniewski, M.M.: Graphics processor unit (GPU) acceleration of finite-difference time-domain (FDTD) algorithm. In: IEEE International Symposium on Circuits and Systems, pp. 265–268 (2004)
10. Abdelkhalek, R.: Évaluation des accélérateurs de calcul GPGPU pour la modlisation sismique. Master's thesis, ENSEIRB, Bordeaux, France (2007)
11. Abdelkhalek, R., Calendra, H., Coulaud, O., Roman, J., Latu, G.: Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster. In: The 2009 High Performance Computing & Simulation - HPCS 2009, Leipzig Allemagne, pp. 36–43 (2009)
12. Martin, R., Komatitsch, D.: An unsplit convolutional perfectly matched layer technique improved at grazing incidence for the viscoelastic wave equation. Geophys. J. Int. 179(1), 333–344 (2009)
13. Michéa, D., Komatitsch, D.: Accelerating a 3D finite-difference wave propagation code using GPU graphics cards. Geophys. J. Int. 182(1), 389–402 (2010)
14. Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., Cruz-Neira, C.: VR Juggler: A Virtual Platform for Virtual Reality Application Development. IEEE Virtual Reality, 89–96 (2001)

PCJ - New Approach for Parallel Computations in Java

Marek Nowicki[1,2] and Piotr Bała[1,2]

[1] Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University,
Chopina 12/18, 87-100 Toruń, Poland
[2] Interdisciplinary Centre for Mathematical and Computational Modelling,
University of Warsaw,
Pawińskiego 5a, 02-106 Warsaw, Poland
{faramir,bala}@mat.umk.pl

Abstract. In this paper we present PCJ - a new library for parallel computations in Java. The PCJ library implements partitioned global address space approach. It hides communication details and therefore it is easy to use and allows for fast development of parallel programs. With the PCJ user can focus on implementation of the algorithm rather than on thread or network programming. The design details with examples of usage for basic operations are described. We also present evaluation of the performance of the PCJ communication on the state of art hardware such as cluster with gigabit interconnect. The results show good performance and scalability when compared to native MPI implementations.

1 Introduction

Changes in hardware are associated with interest in new programming languages which have not been traditionally considered for use in high performance computing. A good example is Java with its increasing performance and parallelization tools such as Java Concurrency which has been introduced in Java SE 5 and improved in Java SE 6 [1]. The parallelization tools available for Java do not limit to threads and include solutions based on various implementations of the MPI library [2], distributed Java Virtual Machine [3] and solutions based on Remote Method Invocation (RMI) [4]. Such solutions are based on the external communication libraries written in other languages. This causes number of problems in terms of usability, scalability and performance.

In our work, we present a new approach motivated by the partitioned global address space (PGAS) approach [5] represented by Co-Array Fortran [6], Unified Parallel C [7] or Titanium (a scientific computing dialect of Java) [8]. PGAS languages are becoming popular because they offer programming abstractions similar to shared memory.

Titanium defines new language constructs and has to use dedicated compiler. Other solutions developed for Java are mostly wrappers to the communications libraries such as MPI and depend on the libraries written in C. Such solutions have significant disadvantages for the users, in particular they are not easy to use.

Our approach has been designed and implemented as the Java library called PCJ (Parallel Computations in Java) [11,9]. Compared to Titanium, PCJ is not using extensions to the language which would require to use dedicated compiler to preprocess code. It much betters suits needs of the Java programmers. The PCJ offers methods for partitioning work, synchronizing tasks, getting and putting values in means of asynchronous one-sided communication. The library provides methods for broadcasting, creating groups of tasks, and monitoring changes of the variables. The PCJ library is created to help develop parallel applications which require significant amounts of memory, bandwidth or processing power.

In this paper we evaluate the performance of the PCJ using a relevant subset of Java Grande Forum Benchmark Suite tests [10] executed on the cluster with gigabit interconnection. The results are compared with analogous tests using the MPI library written in C.

2 Library Description

The PCJ [11] has been developed from scratch using the newest version of Java SE 7. Usage of a newest version of Java increases the performance, prolongs the library life and, in the future, helps to move to more recent versions of Java. Java SE 7 implements Sockets Direct Protocol (SDP) which can increase network performance over Infiniband connections.

In the design of the PCJ we stress compliance to the Java standards. The PCJ has the form of Java library which can be used without any modification of the language. The programmer does not have to use extensions and libraries which are not part of the standard Java distribution.

The PCJ library is built based on some fundamental assumptions presented below.

2.1 PCJ Fundamentals

In the PCJ library each task runs its own calculations and has its own local memory. By default, the variables are stored and accessed locally. Some variables can be shared between tasks, so in the PCJ they are called shared variables.

One task is intended to be the *Manager* which starts calculations on other tasks. It takes responsibility for setting unique identification to tasks, creating and assigning tasks into groups and synchronizing tasks within a group. The *Manager* is running on the main JVM – one which starts the PCJ. The remaining tasks are used for calculations. Since the manager is not CPU intensive, it can be run on the same physical node as one of the tasks used in the calculations.

All variables, which are shareable, are stored in a special *Storage* class. Each task has one and only one *Storage* instance. Each *shared variable* should have a special annotation `@Shared` with *share-name* of that variable. The class can become the *Storage* by extending `pl.umk.mat.pcj.storage.StorageAbstract` class. An example of the *Storage* class definition is viewable in the Listing 1.

```
public class BcastStorage extends StorageAbstract {

    @Shared("array")          // variable identifier
    private double[] array;
}
```

Listing 1. Example *Storage* class

There is also a *start point* class. This class should implement the `pl.umk-.mat.pcj.StartPoint` interface. It indicates, that the class should contain the `public void main()` method. This method is executed after initializing the PCJ, as a starting point like `public static void main(String[] args)` method in the normal execution. Listing 2 shows an example definition of *Start-Point* class.

```
public class Bcast implements StartPoint {

    @Override
    public void main() {
        System.out.println("My task id is: "
                + PCJ.myNode());
    }
}
```

Listing 2. Example *StartPoint* class

2.2 Protocol

The network communication is performed using New IO classes (`java.nio.*`). Sockets are represented as channels, communication is nonblocking and uses 32 MB buffers (`ByteBuffer`). The size of the buffer has been experimentally optimized to that value (see Figure 1). Network requests (e.g. read, connection) are processed by the `Selector` thread running in the loop. There is a dedicated queue used to store data to be transmitted. If data is available the Selector is notified to write data.

All tasks are connected to each other and to the *Manager*. Every *get* or *put* request can be accomplished in a direct connection without using any other task.

Transmitting message to all tasks (eg. *sync*, *broadcast*) uses the binary tree structure of the tasks. The *Manager* sends a message to the first task (with $nodeId = global\ node\ id = 0$). Then this node sends information to its two children ($nodeId * 2 + 1$ and $nodeId * 2 + 2$). Those nodes are sending message to its own children, and so on. That allows to achieve communication complexity of $O(n \log n)$.

In the PCJ protocol each message consists of four integers (`int`) and serialized data (`byte[]`). The integers are: type of message, its identifier, handshake number and number of objects to be transmitted. The data to be transmitted depends on the type of the message. For example for synchronization command it is a single integer - an identifier of the group to be synchronized.

During initialization, the newly created task connects to the *Manager* to inform about the successful start and to receive its unique *global node id*. Other, already connected tasks get information about the new one. The task that in the calculations receives information about a new task *welcomes* it by connecting to its listening address and obtaining its *global node id*. Information about new client is sent using tree structure presented before.

Every task that has finished initialization waits for all other tasks to connect to the *Manager*. When all tasks are connected, the *Manager* sends the signal to start the calculations. This is performed by broadcasting a dedicated message over the tree structure of tasks. The task that receives the message, runs the `public void main()` method from the *start point* class.

The tasks can be grouped to simplify the code and optimize data exchange. Working with all tasks and with a group of tasks is identical from the user point of view.

Joining the group works similarly to the initialization. The task sends the message to join a specified group, using its distinguished name, to the *Manager*. The *Manager* checks if the group already exists and then the task receives its *group node id*. All tasks in that group are notified, using the tree structure of tasks, about the new task in the group. Then the group members *welcome* the new task by sending their *global node id* and associated *group node id*. If a task sends request to join a non-existing group the *Manager* simply creates it. Of course, tasks can be members of many groups.

Synchronization, also known as barrier, works in a similar way to the procedure used to start calculations. Each task in the group is supposed to call the *sync* method. Upon calling this method the task sends an appropriate message to the *Manager* and pauses the current thread until the *Manager* receives messages from all tasks. Then the *Manager* broadcast a message to continue calculations using previously described tree structure of tasks.

There are methods for synchronizing all tasks, group of tasks or to synchronize tasks not associated to the groups. The synchronization of tasks, even without creating a group, is a way to get synchronous put and get methods.

Data exchange between tasks works in asynchronous way. For sending a value to other task the *put* method is used. While receiving a value from another task, the *get* method is used. The other task does not interrupt its calculations when one task performs *put* or *get* operation.

A source task, task that *puts* value, creates a message with a variable name and a new variable value and sends it to a destination task. The destination task puts the new value to its *Storage* space. In *put* method the receiver can monitor attempts of modification of a variable using the *monitor* and the *waitFor* methods.

The *get* method is analogous to the *put*. First of all, a receiving task, creates a message with a variable name to get, sends it to a sending task and waits until value is received. The sending task creates a reply message with desired variable name and value from its *Storage* space and sends it to the receiving task. After that, the receiving task returns received value and continue calculations.

Broadcast is performed in asynchronous way by putting a value to all tasks in a group. The message broadcast uses the tree structure of tasks.

3 Examples

3.1 Deploying PCJ

Starting up the calculations using the PCJ is displayed in Listing 3. A configuration of nodes and manager bind addresses are read from XML file. Then deployment of calculation on nodes is performed.

```java
public static void main(String[] args) throws Throwable {
    Configuration conf = Configuration
                                 .parse(new File(args[1]));

    NodeInfo[] nodes = conf.getNodes()
                                 .toArray(new NodeInfo[0]);
    ManagerInfo[] managers = conf.getManagers()
                                 .toArray(new ManagerInfo[0]);

    PCJ.deploy(Bcast.class,      // StartPoint
        BcastStorage.class,      // storage
        managers,                // managers
        nodes);                  // nodes info
}
```

Listing 3. Example of deploying PCJ

3.2 Join to Groups

Listing 4 shows how to create groups. In the example, tasks are divided into two groups - *group0* contains tasks with even *global node id*, *group1* contains tasks with odd *global node id*.

```java
    /*
     * All nodes in calculations joins to:
     *   - group0 -- when node id for task is even
     *   - group1 -- when node id for task is odd
     */
    Group g = PCJ.join("group" + (PCJ.myNode() % 2));
    PCJ.sync();
```

Listing 4. Example of joining to groups

3.3 Getting Value

When one task needs value from another task, the PCJ offers a mechanism to synchronize two or more tasks without implicit group creation. After synchronization, the first task can asynchronously get value of variable calling `get()` method. A sample source code for that problem is shown on Listing 5.

```
if (PCJ.myNode() == 3) {
    /* place calculated value to Storage */
    PCJ.getStorage().put("impact", impact);

    /* synchronize tasks: current and with id = 2 */
    PCJ.syncWith(2);
} else if (PCJ.myNode() == 2) {
    /* the same synchronizing as above */
    PCJ.syncWith(new int[]{2, 3});

    /* asynchronous get value of variable */
    impact = PCJ.get("impact", 3);
}
```

Listing 5. Example of synchronizing two tasks and getting value

3.4 Synchronize and Broadcast

The example for synchronizing all tasks, broadcasting a new value of array *arr* from task `0` to all tasks and monitoring of a variable by all nodes is available in Listing 6.

```
    PCJ.monitor("arr");         // tell to monitor
                                //   variable "arr"

    PCJ.sync();                 // synchronize all tasks

    if (PCJ.myNode() == 0) {    // if node id equals 0
        PCJ.broadcast("arr",    // broadcast new value
            new double[]{2.71828, 3.14159});
    }

    PCJ.waitFor("arr");         // wait for modification
                                //   of variable "arr"
```

Listing 6. Example of synchronizing, broadcasting value from node `0` and monitoring a variable

4 Scaling and Performance

In order to evaluate the PCJ we have run selected Java Grande Forum Benchmark Suite tests [10]. They address communication efficiency: *PingPong*, *Bcast*, *Barrier* and *RayTracer* benchmark. The selected codes use constructions that could be found in real life applications.

Tests were run in the time limit (10 seconds) and the limit of main loop repetitions (1000000 repetitions), whatever comes first.

We have compared the results for the PCJ (running on 64-bit Java Virtual Machine, Oracle version 1.7.0_03) with the results collected using the benchmarks written in C or C++ (Raytracing). The codes used for comparison were based on the same algorithm as used in Java implementation and prepared by the authors. The arrays and tables definitions and allocations were optimized to obtain efficient code. We have used different compilers and MPI libraries as presented in the Table 1. Compilations were performed with `-O2` parameter. All tests have been run on a cluster built of 64-bit Intel Xeon X5660 Processors with six cores at 2800 MHz. The nodes are equipped with 2 such processors, 24 GB RAM and Gigabit Ethernet.

Table 1. Versions of compilers and MPI libraries used in the benchmarks

MPICH	gcc (GCC) 4.1.2 20080704 (Red Hat 4.1.2-52)	MPICH2 1.4.1p1
OpenMPI	gcc (GCC) 4.1.2 20080704 (Red Hat 4.1.2-52)	OpenMPI 1.4.2
PGI	pgcc 11.3-0 64-bit target on x86-64 Linux	built-in

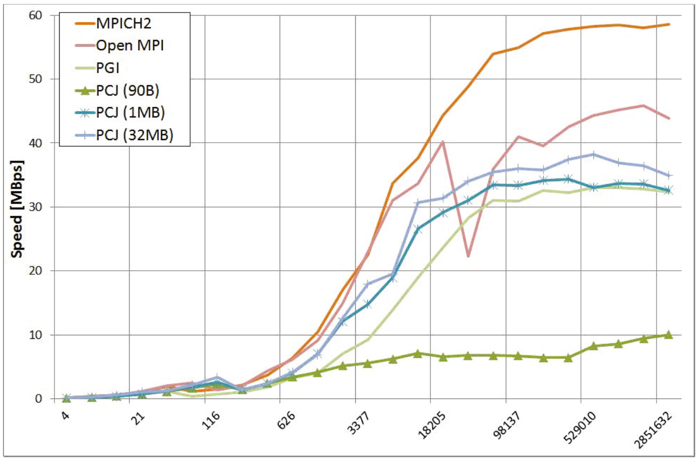


Fig. 1. Speed of PingPong with various size of array of double. The data for PCJ is presented for different size of the buffer.

The first test performed was *PingPong*. It is based on sending an array of doubles between two tasks, counting all the data sent. The results are presented in the Figure 1 for the different sizes of the buffer used in the PCJ. For the buffers larger that 1MB, the performance is similar and can compete with the solutions written in C that have been optimized for many years. Smaller buffer sizes can decrease performance.

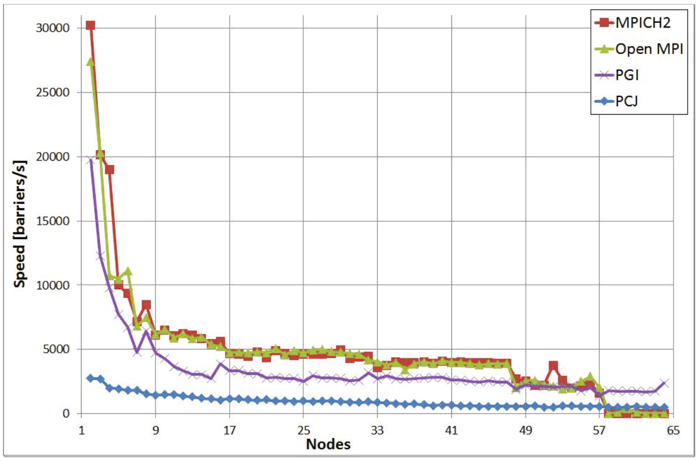
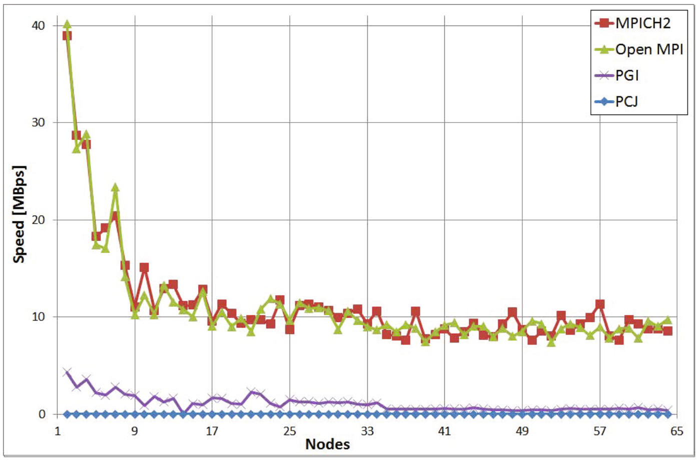


Fig. 2. Speed of performing barrier operation depending on the number of tasks

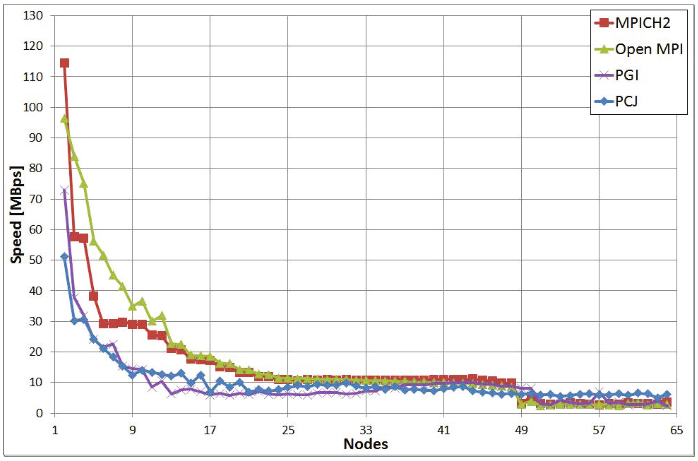
The *Barrier* test counts the number of barrier operations between all tasks used. Figure 2 shows results for the test. The PCJ speed compared to the Open-MPI results is low and it should be improved. Although scaling of the PCJ is good which is promising and shows room for improvements.

In the Figure 3 we present the result of the *Bcast* test. It relays on broadcasting to all tasks messages that consist of array of doubles of the specified size and counting data sent by the first one. We present tests for the different array sizes: 21 and 172072 double elements because there is high correlation between the array size and maximum speed. The results for larger array sizes are very competitive in comparison to the MPI results. For small data – the PCJ speed oscillates around the 4600 B/s and this part of the PCJ should be improved. This effect has the same origin as low barrier efficiency. For the large arrays results are improving.

RayTracer is the final test presented here. It measures the performance of 3D ray tracing of the scene rendered at a resolution of NxN pixels. The *Reduce* operation has not been implemented yet. The *Reduce* can be known from the MPI and is one of the required operation for this benchmark. The PCJ version of the *RayTracer* test contains a simple, naive method to do the *Reduce* operation. Because of the complex structure types used in original Java source code, the MPI version of *RayTracer* was rewritten in C++.



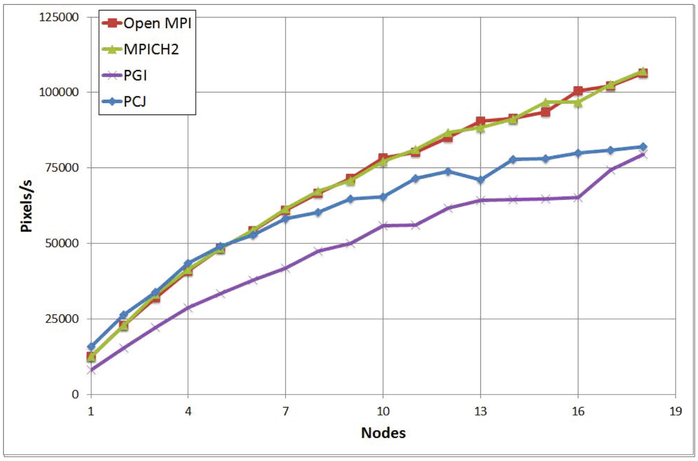
(a) 21 elements



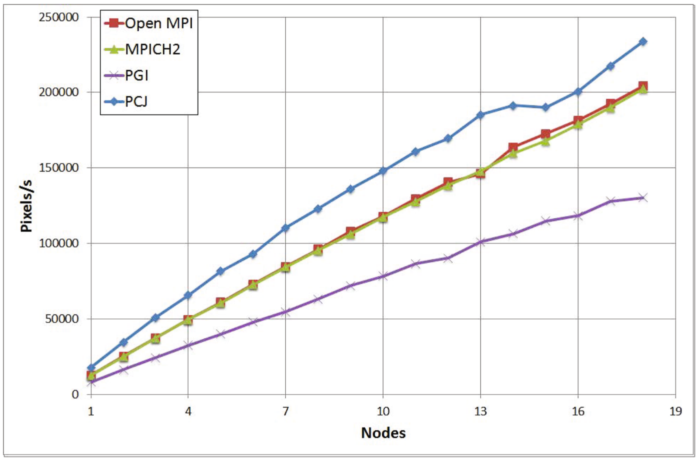
(b) 172072 elements

Fig. 3. Speed of broadcasting array of doubles

The results of the *RayTracer* benchmark are presented on the Figure 4 for the different sizes of data. Figure 4a presents achieved speed (number of pixels processed in unit time) for the scene of size 150x150 pixels and the Figure 4b data for scene of 500x500 pixels. The speed is competitive for the MPI and the PCJ versions of the benchmark. What is interesting, the run time for only one task is shortest for the Java code. For the larger scene the gained speed for the Java solution is the highest one.



(a) Scene size 150x150



(b) Scene size 500x500

Fig. 4. The performance of the 3D ray tracing of the scene for different sizes: 150x150 (upper) and 500x500 pixels (lower).

5 Conclusions and Future Work

The PCJ library offers a new approach for the development of parallel, distributed application in Java language. It uses the newest advantages of Java and therefore can be a good base for new parallel applications. In contrast to other available solutions, it does not require any additional libraries, applications or modified version of JVM. It is noteworthy that the PCJ library has great promise to be successful in scientific applications.

However, the presented tests show that there are still some areas for improvements. The efficiency of sending small data and speed of the task synchronization can be increased. Additionally, there are no advanced techniques for the breakdown recovery and node failure handling. Such mechanisms should be also implemented in order to make the PCJ a widely used library for distributed and parallel application for Java language. There is also need of features known from other libraries like *scatter*, *gather* and *reduce* data over tasks.

Acknowledgements. This work has been performed using PL-Grid infrastructure.

References

1. Java Platform, Standard Edition 6, Features and Enhancements, http://www.oracle.com/technetwork/java/javase/features-141434.html
2. Carpenter, B., Getov, V., Judd, G., Skjellum, T., Fox, G.: MPJ: MPI-like Message Passing for Java. Concurrency: Practice and Experience 12(11) (September 2000)
3. Boner, J., Kuleshov, E.: Clustering the Java virtual machine using aspect-oriented programming. In: AOSD 2007: Proceedings of the 6th International Conference on Aspect-Oriented Software Development (2007)
4. Nester, C., Philippsen, M., Haumacher, B.: A more efficient RMI for Java. In: Proceedings of the ACM 1999 Conference on Java Grande (JAVA 1999), pp. 152–159. ACM, New York (1999)
5. Mallón, D.A., Taboada, G.L., Teijeiro, C., Touriño, J., Fraguela, B.B., Gómez, A., Doallo, R., Mouriño, J.C.: Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) EuroPVM/MPI. LNCS, vol. 5759, pp. 174–184. Springer, Heidelberg (2009)
6. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. ACM SIGPLAN Fortran Forum 17(2), 1–31 (1998)
7. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. IDA Center for Computing (1999)
8. Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P., Aiken, A.: Titanium: A High-Performance Java Dialect. Concurrency: Practice and Experience 10(11-13) (September-November 1998)
9. PCJ library is avaliable from the authors upon request
10. Java Grande Project: benchmark suite, http://www.epcc.ed.ac.uk/research/java-grande/
11. Nowicki, M., Bała, P.: Parallel computations in Java with PCJ library. In: Smari, W.W., Zeljkovic, V. (eds.) 2012 International Conference on High Performance Computing and Simulation (HPCS), pp. 381–387. IEEE (2012)

Porting Production Level Quantum Chromodynamics Code to Graphics Processing Units – A Case Study

Teemu Rantalaiho[1,2]

[1] Department of Physics, University of Helsinki, Helsinki, Finland
[2] Helsinki Institute of Physics, Helsinki, Finland
teemu.rantalaiho@helsinki.fi

Abstract. We present our findings and results of a project to port an existing large lattice QCD codebase to run on GPUs and clusters of GPUs. Our design principles from the start were to strive for both productivity and performance, while tackling the problems presented by a large constantly moving codebase. The resulting simulator reproduces the original results while running up to 11 times faster than our highly optimized CPU-code and meeting productivity requirements. Multi-GPU support was implemented using MPI and scaling across nodes shows good weak scaling. We also contemplate the consequences of the dawning of the parallel computing era from a lattice QCD point of view and analyze where state-of-the art contemporary parallel computing architecture could be improved.

Keywords: graphics processing units, computational physics, lattice QCD.

1 Introduction

Lattice QCD (see for example [1], [2] or [3]) – short for quantum chromodynamics[1] – has long been one of the few heavy computational tasks that theoretical physicists have undertaken, and its importance as a tool for modelling the most fundamental laws of physics has grown over the years with advances in algorithms and increase of available computational power. At the moment it is the only known and reliable method to study *quantum field theories* at *strong coupling*.

It is evident that relentless advances in microchip technology, coupled with modest increase in their operating frequency, are pushing us – and in fact has already pushed us – quite far into the world of parallel computing. Simply put, the industry can output transistors in higher and higher densities, while not being able to run them significantly faster. This means that even consumer devices are seeing processors with more and more processor cores, as well as cores

[1] When talking about lattice QCD physicists can often refer also to other models of quantum field theory.

specialized in doing certain types of operations, such as video playback, graphics processing, network functions and many others. Parallelism in high performance computing is, of course, nothing new in itself, but as the fine-grained parallelism seeks its "optimal form" on the consumer side, high performance computing is getting ready for the change.

Formulating the quantum field theory on a space-time lattice provides an opportunity to study the model non-perturbatively and use computer simulations to get results for a wide range of phenomena – it enables, for example, one to compute the hadronic spectrum of QCD [4] from first principles and provides solutions for many vital gaps left by the perturbation theory, such as structure functions of composite particles [5], form-factors [6] and decay-constants [7]. It also enables one to study and test models for new physics, such as technicolor theories [8] and quantum field theories at finite temperature [9].

Currently the most widely adopted massively parallel programming model is NVIDIA's CUDA-architecture, because it provides through its CUDA-C programming language the most robust and productive solution. We also chose CUDA-C as our target, and there are various other groups doing lattice QCD with CUDA as well [10,11,12]. Recently also OpenCL [13] Lattice QCD codes have emerged [14,15], where the latter group actually have studied both CUDA and OpenCL.

1.1 Problem Statement

The idea behind lattice QCD is that one divides space-time into discrete boxes, called the lattice, and places the fields onto the lattice sites and onto the links between the sites, as shown in Fig. 1. Then one can simulate nature by creating a set of multiple field configurations, called an *ensemble* and calculate values of physical observables by computing ensemble averages over these states.

The set of states is normally produced with the help of a Markov chain, by combining a *molecular dynamics* algorithm together with a *Metropolis* acceptance test, and therefore the typical computational tasks in lattice QCD are:

1. Refresh generalized momentum variables from a heat bath, once per *trajectory*[2]
2. Compute generalized forces for fields for each step
3. Integrate classical equations of motion for the fields[3]
4. Perform a Metropolis acceptance test (at end of each trajectory) in order to achieve the correct limiting distribution.

The force calculation normally involves a matrix inversion, where the matrix indices run over the entire lattice. It is therefore the heaviest part of the computation – the matrix arises in simulations with dynamical fermions (normal

[2] A trajectory ranges normally from about ten to a few hundred classical steps.

[3] The integration is *not* done with respect to normal time variable, but through the Markov chain index-"time".

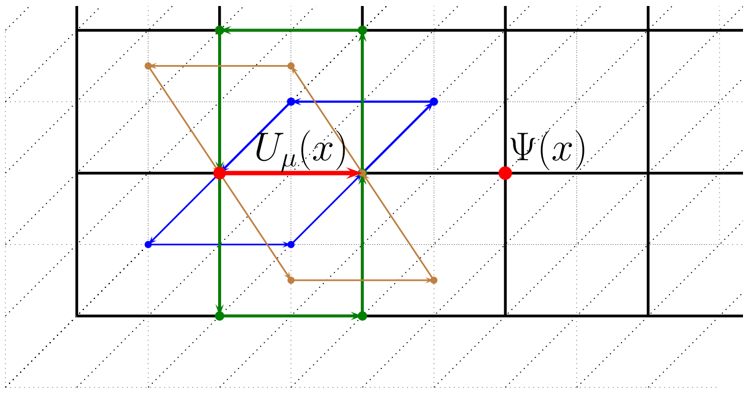


Fig. 1. The matter fields $\Psi(x)$ live on lattice sites, whereas gauge fields $U_\mu(x)$ live on links connecting the sites. Also depicted staples connecting to a single link variable, that are needed in computation of the gauge field forces.

propagating matter particles) and the simplest form for the Fermion matrix is[4]

$$A_{x,y} = [Q^\dagger Q]_{x,y} \qquad \text{where}$$

$$Q_{x,y} = \delta_{x,y} - \kappa \sum_{\mu=\pm1}^{\pm4} \delta_{y+\hat{\mu},x}(1+\gamma_\mu)U_\mu(x). \tag{1}$$

Here κ is a constant related to the mass(es) of the quark(s), $\delta_{x,y}$ is the *Kronecker delta function* (unit matrix elements), the sum goes over the space-time dimensions μ, γ_μ are 4-by-4 constant matrices and $U_\mu(x)$ are the link variable-matrices that carry the force (gluons for example), from one lattice site to the neighbouring one and in normal QCD they are 3-by-3 complex matrices.

Therefore the matrix A in the equation $Ar = z$, where we are solving for the vector r with given z, is an almost diagonal sparse matrix with *predefined sparsity pattern.* This fact makes lattice QCD ideal for parallelization, as the amount work done by each site is constant. The actual algorithm used in the matrix inversion is normally some variant of the conjugate gradient algorithm, and therefore we need fast code to handle the multiplication of a fermion vector, by the fermion matrix.

2 Our Solution Using CUDA

The fundamental problem in massively parallel programming is how to express the parallelism inherent in the algorithm. After brief analysis of our existing

[4] There are multiple different algorithms for simulating fermions, here we present the simplest one for illustrative purposes.

code (which is based on the original MILC collaboration's code [16]) it became apparent that there is a very clear pattern of parallelism in this kind of code: logic code, which is essentially sequential in nature, was interleaved with code that could be run in parallel throughout the entire lattice – in fact, these sections of the code had already been defined through compile time macros, which would simply iterate in a for-loop over the entire lattice, or part of it. At times larger parallel code sections were broken up due to the need to do MPI communication or because results had to be accumulated throughout the lattice to decide how to continue. Therefore the typical code was something like the following:

```
Logic code;
For all sites(i){ Do something for site i (i+1,...)}
Logic code;
sum = 0;
For all sites(i){
  sum +=  Some function of site i; }
If (|sum| < tolerance) } {break out of outer-loop;}
Logic code;
```

At first we thought about breaking up large parallel code sections into simpler vectorized operations over the lattice and parallelizing these vector operations to run on GPUs, but it soon became clear, that in this approach we would be sacrificing performance due to breaking up *kernel fusion*, which means that we can save memory bandwidth (the primary bottleneck in lattice QCD) by combining as many operations as possible for the local field values.[5]

Further analysis revealed that almost the entire codebase was dependent on only three fundamentally different parallel algorithms: *Map*, *Reduce* and *Bin*. Here *Map* just performs some operation at each lattice site, *Reduce* does the same, except at the end it accumulates a value over the lattice sites and *Bin* or *Histogram* does the same as *Reduce*, except that the values are accumulated to different bins (normally depending on lattice site coordinate). As a result we decided to go with the "trivial level of parallelization" – meaning one thread per site – and to try to re-use the existing code for the parallel sections. Afterwards we augmented this strategy by allowing multiple threads to run on one site and we use this feature to extract additional parallelism in those cases where the intra-site parallelism is trivial in nature, relieving register pressure and slightly improving scaling to smaller local volumes.

2.1 Parallel Call Framework

In order to reach maximal productivity, while giving us a migration path to the GPUs and being able to retain most of the codebase, we decided to use only features present in modern C and C++ compilers. This meant wrapping the parallel sections of the code with preprocessor directives that would translate the code inside into function objects for CUDA and normal functions for

[5] Allows us to fetch once, use multiple times and store once again.

ANSI C compilers. This way we can actually run the same code at near optimal performance with both CPUs and GPUs.[6]

To *user code*[7] this causes three changes. First the parallel calls need to be taken out of normal function scope, as unfortunately the C++ standard does not allow local types as template variables (this is fixed in C++11), and dressed with proper preprocessor directives, which translates the code inside into functions or function-objects:

```
forallsites(i) { <code> };
```

becomes

```
PARALLEL_CALL_BEGIN(<name>, <input>, i)
{ <code> }
PARALLEL_CALL_END();
```

for the *Map* algorithm,

```
PARALLEL_REDUCE_BEGIN(<name>, <in>, i, <out>)
 { <code>; <out> = ...  }
PARALLEL_REDUCE_SUM(...)
 { <sum code> }
PARALLEL_REDUCE_END(...);
```

for the *Reduce* algorithm and

```
PARALLEL_HISTOGRAM_BEGIN (<name>, <input>, i, <out>, <out_index>)
   { <code>; <out> = ... ; <out_index> = ... }
PARALLEL_HISTOGRAM_SUM(...)
   { <sum code> }
PARALLEL_HISTOGRAM_END(...);
```

for the *Bin*, or *Histogram*, algorithm. This way we can support arbitrary reduction operations both with normal reductions as well as with histograms. The change above is normally easy for an arbitrary parallel call with a few copy-paste operations. On top of this the inputs needed in the algorithm are packed inside a structure which is passed on as a parameter to the user code. The solution then for CUDA takes in the code inside the preprocessor macros, makes function objects out of them and passes these function objects as parameters to templatized versions for the solutions of the three parallel algorithm types. In this way we achieve:

- Good performance (code correctly inlined)
- Parallel algorithms coded once, used everywhere
- Complete abstraction of underlying parallel hardware

[6] We considered also OpenCL, but it provides little support for algorithm abstraction, normally achieved with function objects.

[7] By user code we mean the architecture independent part of the code that contains the actual lattice QCD algorithms.

The most important algorithm is *Map*, the implementation of which we wrote ourselves to get best performance; For a long time we used the thrust library [17] for CUDA for *Reduction*, but later implemented our own reduction routine in order to gain more flexibility to the code and to better cater for the use case where we want multiple threads per lattice site. The implementation of the *Histogram* or *Bin* algorithm was also developed by us, as no good pre-existing solutions were found.

The largest change to the user code was caused by the fact that we had to change layout of the fields in the memory from the typical *array of structures* to *structure of arrays* for CUDA code in order to enable *coalescing* in memory fetches and stores. Therefore on the CPUs the access pattern to read one field of three components r_i in three sites is:

Thread0:									
Site 0:			Site 1:			Site 2:			Site 3: ...
r_1	r_2	r_3	r_1	r_2	r_3	r_1	r_2	r_3	...

and the access pattern on GPUs for the same operation is:

Thread0:									
r_1		...	r_2		...	r_3		...	
Thread1:									
	r_1		...	r_2		...	r_3		...
Thread3:									
		r_1		...	r_2		...	r_3	

,

where the distance between r_i and r_{i+1} (called the *stride*) is number of sites allocated for the field. In effect this means storing each component of a field in a separate array, which places always the same components of a field of subsequent lattice sites next to each other. We also noticed that best performance can be reached when the values are stored as double2's, which means a tuple of two double precision (64-bit) floating point numbers (and float4's in case we have built single precision version of our code).

Changing the layout of the fields in memory causes two major changes to the code, as well as several minor ones. In parallel code sections we can no longer apply the normal array dereferencing to access the fields, but have to resort to accessor-functions that jump through the memory with the correct stride. A typical pattern has been something like:

```
forallsites(i){
  a = DoSomething(x[i], z[i]);
  x[i] = z[i] + a*y[i];
}
```

Now it becomes

```
PARALLEL_CALL_BEGIN(name, input, i){
  x = getField(input.x, i);
  y = getField(input.y, i);
```

```
   z = getField(input.z, i);
   a = DoSomething(x, z);
   x = z + a*y;
   setField(input.x, i, x);
 } PARALLEL_CALL_END();
```

Also due to memory layout change we had to adapt our MPI code to cope with it. It turns out that the best solution was to do an intermediate pass on the GPU, which writes the buffer to be sent through MPI into a single linear block, sends the block to the CPU, which performs a normal MPI send/receive, receives the block on the CPU side, transfers it back to the GPU as is and scatters the linear block into proper places in the memory block allocated for the field. We suspect that this is due to increased call overheads, caused by an increased amount of memory copy calls. Here it should be noted that scaling across multiple MPI nodes should be improved by taking advantage of direct GPU-to-GPU communications provided by the GPUDirect v.2 [18] implementation, where present, but we have not had the time to try it out ourselves yet. We do, however, take advantage of GPUDirect v.1 when present, which allows us to share the pinned memory buffers between the MPI-Implementation (typically Infiniband) and the GPU, eliminating the need for unnecessary copies due to DMA-transfers – in our experience this results in a minor overall improvement, in the order of five percent increase in performance.

Another optimization that we implemented was to take advantage of CUDA *streams* [19], which can be thought of as task-parallel threads within the GPU. Operations on different streams can be run in parallel to each other and an event-system can be used to synchronize the streams to each other. The operations within a single stream are of course completed sequentially. The authors in [20] take advantage of this system to concurrently perform memory transfers between the CPU and the GPU, while running the bulk of the matrix-vector multiplication for those sites that have no out-of-node neighbors and they report good scaling on relatively large lattices. Due to the fact that our solution already does a good job of overlapping MPI-communications with computational tasks even without multiple streams, we see an increase in performance only with small lattice-sizes, and even in these situations the improvement is of the order of five percent, but this feature may become more relevant with GPUDirect v.2 [18] enabled and with next generation GPUs, that have improved support for multiple MPI clients [21].

2.2 Performance Results

Our performance benchmarks were run on Vuori, a HP CP4000 BL ProLiant supercluster at CSC [22]. The tested hardware consisted of Tesla M2050 GPUs, as well as hexacore Opterons with 12.8 GB/s of memory bandwidth. The GPUs are about 10 times as fast as the CPUs, when measured by memory bandwidth, as the GPUs have about 129.5 GB/s of bandwidth with ECC turned on, and we always compare a GPU against a whole CPU running all six cores. The network is 4x quad-rate Infiniband with 3.2 GB/s of maximal throughput.

In Fig. 2 we present performance comparison results for two different models: One is based on the SU(2) symmetry where the link matrices are 2-by-2 complex matrices with 4 parameters; The other is the normal QCD symmetry group, the fundamental SU(3), where we store all 9 complex numbers of the 3-by-3 matrix. An easy opportunity for optimization here would be to use a smaller representation of the SU(3) matrix – the almost standard optimization is to use 12 real parameters and the remaining 6 parameters can be found by requiring unitarity of the matrix (see for example [23], page 5). The runs have 1 flavor of 2 pseudofermions and the SU(3) run implements the so-called clover improvement to the fermion action [24]. The results are what can be expected by looking at performance differences between the processor types and it can seen that our GPU version of the simulator is running at near-optimal speed, provided that the CPU code is running at near-optimal speed. In order to measure the quality of our GPU-porting work we decided to use as a measure the relative speed-up over the CPU code, as we know that our old CPU code performs well.[8]

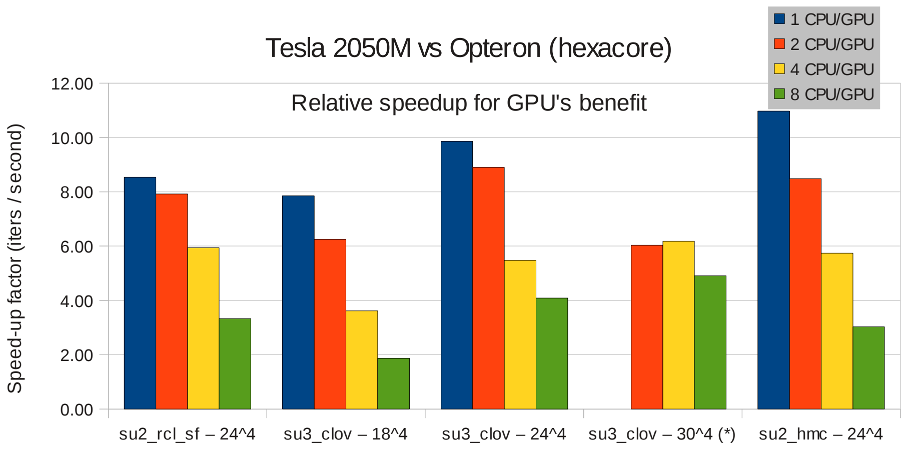


Fig. 2. GPU Performance against optimized CPU-versions

Multi-GPU scaling was studied with various models as well. In Fig. 3 we present results for SU(2) runs on a lattice of 24^4 sites and we can observe here that while the scaling from 1 to 2 GPUs is good and from 2 to 4 GPUs is decent, the relatively easy symmetry group of the model means less local work at each site and hence scaling from 4 to 8 GPUs starts hit the MPI-limitations. The performance is given as number of conjugate gradient (CG) iterations per second and the timings include force computations between steps – the number is therefore obtained by computing the total number of CG iterations done in one trajectory divided by the amount of time it took to compute the trajectory.

[8] Often authors give results in amount of GFlops/s (See [11,23]).

An interesting fact in this particular case is the superlinear scaling of the CPU performance; This is most likely caused by better utilization of the caches of the CPUs, starting to take advantage of the temporal locality in the algorithm.

Scaling of clover improved SU(3) theory is presented in figures 4, 5 and 6. Here we can see the effect of the local lattice volume clearly: the 30^4 lattice scales quite well up to 8 GPUs but in the 18^4 case the scaling from 4 to 8 GPUs is already quite bad, indicating that either the MPI implementation cannot keep up with the GPUs anymore, or that the local lattice size for each GPU is starting to become too small to fill the GPU completely – we suspect both, based on simulating a system of the size of the local volume with a single GPU, but have not studied further the balance of the two effects.

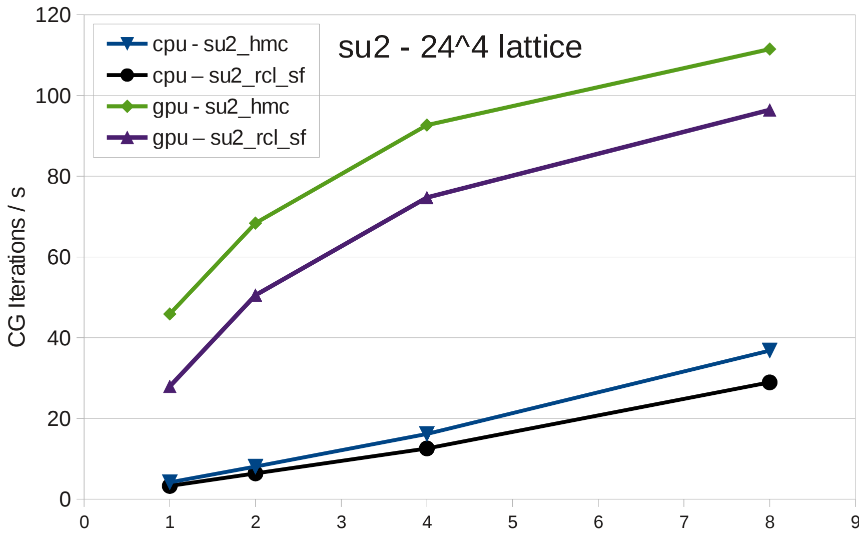


Fig. 3. Scaling of the SU(2) model at 24^4 sites from 1 to 8 GPUs and CPUs (6 to 48 cores)

We ran out of memory running the 30^4-sized lattice with just one GPU so the result here is absent. The scaling on the GPUs is good at large lattice-sizes and exhibits typical *weak scaling* – we can scale to 8 GPUs with 30^4-sized lattices and probably beyond, but could not yet test it as we did not have access to a system with more GPUs. This means that we can employ GPUs to get the same performance with smaller resource usage when studying small lattices and they enable us to study large lattices with significantly higher performance. Our GPU scaling with large enough lattices cuts execution time by 35 to 45 percent for every doubling of processor resources and CPU scaling cuts 50% for each doubling almost without exception – in this study we did not try how far we can keep on scaling CPUs.

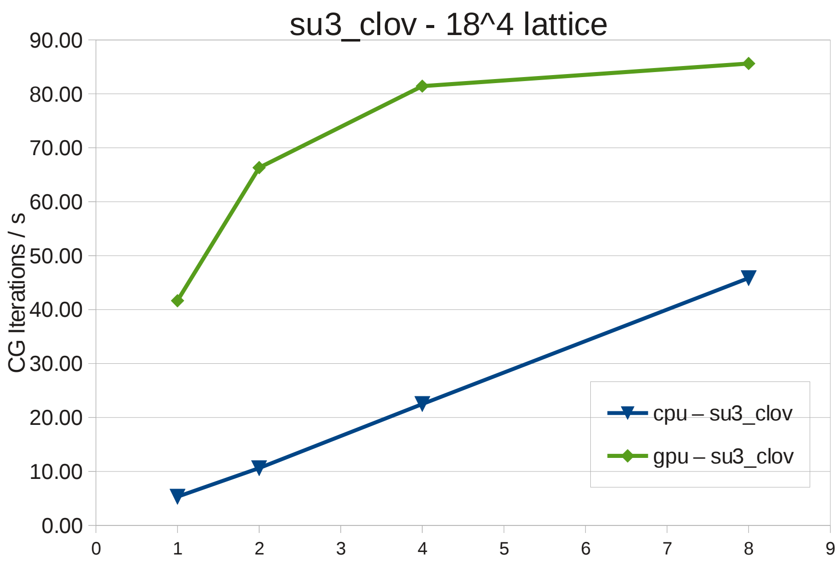


Fig. 4. Scaling of the SU(3) model at 18^4 sites from 1 to 8 GPUs and CPUs (6 to 48 cores)

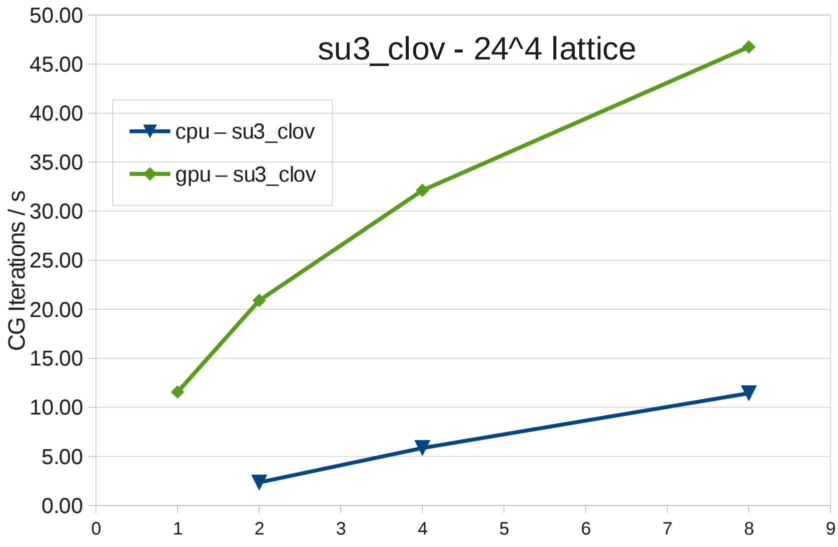


Fig. 5. Scaling of the SU(3) model at 24^4 sites from 1 to 8 GPUs and CPUs (6 to 48 cores)

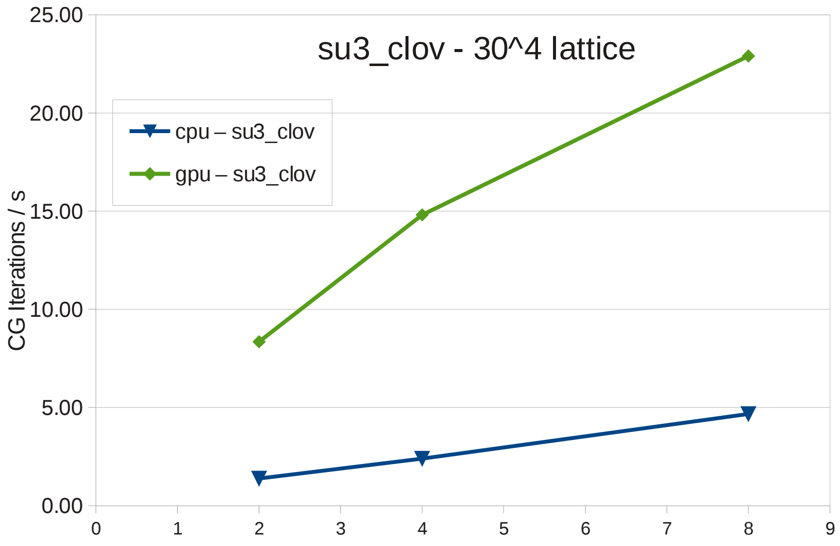


Fig. 6. Scaling of the SU(3) model at 30^4 sites from 1 to 8 GPUs and CPUs (6 to 48 cores)

It should be noted also, that as our results are measured from entire trajectories, the number of iterations per trajectory vary slightly, resulting in small variations in the ratio of CG iterations to force computations between different number of MPI processes. These variations may skew the comparisons between CPUs and GPUs by a few percent and in order to fight this effect we have tried to select trajectories with similar amounts of CG-iterations per force computations.

2.3 Novel Aspects of Our Solution

We feel that our solution for the parallelization is unique in many respects. The major difference to other large projects, such as the ones discussed in [11] and [23] is that we do not hide the code running on the GPUs behind libraries, but make it more accessible to developers and easier to make changes – only the difficult things, such as memory layout handling, MPI tweaks and implementation of the parallel algorithms, have been abstracted away and even those are a few function or macro calls away.

Bringing the GPU code to the developer also enables us to avoid memory copies between the CPU and GPUs as the fields can be kept on the GPU memory just by implementing the necessary functionality with GPU code. There are already groups that implement the entire HMC trajectory, or a large portion of it, using the GPU [25,15], but extending the application of GPUs even to measurements, like in our code, is not common yet. The QDP++ Library [11] seems to provide such tools. Also writing new code with our system is a breeze: it is as easy as programming for the CPUs and one can debug the code using the CPU-compiled version of the same code. Our only regrets are the memory layout issue and the fact that we had to move the parallel code outside the logic functions (see Section 2.1).

3 Discussion

Thanks to the fact that we were able to find such an easy way to port the existing parallel sections of the code to the GPUs, we ended up porting almost all of them. The conventional thinking in parallel computing has long been that one should look at the profiler, see where more than 90 percent of the time is spent, and parallelize that.

As it turns out in our case, 80 to 90 percent of the time (in typical runs) is pretty nicely contained within the conjugate gradient algorithms, but there are several of those, and the remaining 10 to 20 percent is already quite a large part. Let us assume that we can reach the tenfold performance improvement by applying GPU acceleration to the parallel sections of the code and let us quickly check an example of how the performance scales:

	T_{CG}	T_{Rest}	T_{tot}	Improvement
Baseline	8	2	10	1x
Optimize CG	0.8	2	2.8	3.6x
Optimize All	0.8	0.2	1.0	10x

Here we see that optimizing only the conjugate gradient will result in a mere 3.6-fold overall performance gain. While in itself it is not that bad, even more impressive is, if we can achieve the full tenfold improvement promised by the architecture we are using.

We believe that one should choose the right tool for the job, and therefore parallel sections of the code should by default be run on parallel architectures. If this is not feasible for some legacy code, then we should at least try to pave the way for the correct solution when we write new code. We believe that most algorithms involved in high performance computing have a notion of trivial level of parallelization, let it be a 'site' or 'cell', as in our case, or a node, a particle or just an entry in an array, and that this parallelism should be exploited to the fullest in order to reduce wasted processor time and energy.

On the other hand productivity is also a major concern and here there is still a lot of work to do: programming on massively parallel architectures should be made as simple as possible and libraries, such as *Thrust* [17], should provide solutions for all common, abstractable parallel algorithms. The programming language used should be expressive enough to enable hiding different levels of parallelism and yet provide low-level access to code needed in micro-optimizations. As an example of a modern effort to reach these goals see for example [26].

In our simulation code, we achieve good performance on a relatively small amount of parallel threads per site, which means that there are always multiple independent instructions for the compiler to use in hiding the pipeline and memory latencies. We have noticed though that in some kernels register pressure is starting to limit performance, and extending our strategy to deploy multiple threads per site to those situations where the threads collaboratively produce a result, would help both with register pressure and with strong scaling, as then a smaller local lattice volume is needed to fill the GPU with work. The abstraction of such interfaces might be non-trivial though and may require some sort of adjustable libraries to be implemented for the most time-critical kernels. It is made even further difficult by the fact, that in different kernels an optimal amount of threads per site might vary, creating even more pressure to get a permanent solution for the memory-layout problem, since the number of threads per site, dictates the optimal layout of fields in the memory, which of course cannot vary within one simulation.

This issue is related to the memory layout issue and the ideal solution probably would deal with both. One possibility for such a solution would be that once the compiler detects portions of code that have excessive register pressure, it would automatically issue multiple threads to handle the auto-vectorizable portions of the code by taking advantage of the unused processing units. Another option would be to simply have the hardware support strided access directly would ease the situation considerably and would probably pave way for a good software-based solution for the parallelism-problem. In the mean time we shall explore the various ideas we have on the subject.

Another option would be to run more threads in one site on the kernels that experience high register pressure to exploit the vector-level parallelism inherent in the small matrix-vector operations, but it is not a trivial task to hide this parallelization from the user with current programming languages. Here a nice solution could be something like a *device-side BLAS-library*, that would provide optimized code for small linear-algebra operations inside the kernels, although it might very well be, that the crude C-programming language does not provide good way to abstract away the number of threads needed for each operation.

Note. During the review process of the present paper we have implemented a collaborative intra-site threading technique for CUDA capable GPUs in two of the heavy computational loops that do improve performance by relieving register pressure, allowing more thread-level parallelism. The implementation uses on-chip *shared memory* to share the different *color-components* of a *Wilson vector* between the collaborating GPU threads in the colormatrix-vector multiplication needed in Eq.1 and therefore each of the threads only have to fetch one row of the matrix. The sharing of the color-components is done between strictly different *thread-warps* in order to avoid costly shared-memory *bank-conflicts* (apart from the ones necessarily caused by 64-bit accesses inherent in double precision calculations) between threads of the same warp (see [19]) – this requires us to synchronize the different warps a few times in order to ensure correct ordering of operations, but the performance impact of the synchronizations seems minor.

4 Conclusions

We have presented our solution for a lattice QCD simulator for CUDA-capable GPUs that has been implemented by porting an existing C+MPI-based simulator, derived from the MILC-collaboration's original code[16]. Our code includes a large number of lattice QCD related algorithms and is aimed mostly at exploring new physics and new models.

Our algorithm provides weak scaling over MPI-nodes and a clear performance benefit over optimized CPU-versions, with 6 to 11 fold increase in performance over the hexacore Opterons.[9] The scaling across MPI processes with GPUs is not as strong as with CPUs due to overheads caused by memory layout changes, increased PCI Express traffic and also because GPUs simply require more network bandwidth since they run faster, yet multi-GPU performance is competitive with large lattice sizes and should be even more competitive, once GPUDirect v.2 is taken advantage of.

The extreme ease of use of our framework coupled with very real performance benefits lead us to believe that we will be ready to face the challenges of the new massively parallel computing era. We feel that our way of expressing parallelism can stand the test of time, as the level of abstraction is high enough to enable the developer to forget underlying architecture, while still being low enough to allow removal of bottlenecks in performance critical sections of the code.

Acknowledgements. This research is supported by the Academy of Finland project 1134018 and the simulations were run on CSC – IT Center for Science, Finland. We wish to thank David Weir and Kari Rummukainen for valuable discussions.

References

1. Montvay, I., Münster, G.: Quantum Fields on a Lattice. Cambridge Monographs on Mathematical Physics. Cambridge University Press, The Edinburgh Building (1994)
2. Rothe, H.J.: Lattice Gauge Theories: An Introduction, 3rd edn. World Scientific Publishing Company, Hackendsack (2005)
3. Gupta, R.: Introduction to Lattice QCD. ArXiv High Energy Physics - Lattice e-prints (July 1998)
4. Fodor, Z., Hoelbling, C.: Light Hadron Masses from lattice QCD. Reviews of Modern Physics 84, 449–495 (2012)
5. Göckeler, M., Hägler, P., Horsley, R., Pleiter, D., Rakow, P.E.L., Schäfer, A., Schierholz, G., Zanotti, J.M.: Generalized parton distributions and structure functions from full lattice QCD. Nuclear Physics B Proceedings Supplements 140, 399–404 (2005)
6. Renner, D.B.: Form factors from lattice QCD. ArXiv e-prints (July 2012)
7. McNeile, C., Davies, C.T.H., Follana, E., Hornbostel, K., Lepage, G.P.: Heavy meson masses and decay constants from relativistic heavy quarks in full lattice QCD. ArXiv e-prints (July 2012)

[9] Tesla 2050M is about 10x faster than the hexacore Opteron.

8. Rummukainen, K.: QCD-like technicolor on the lattice. In: Llanes-Estrada, F.J., Peláez, J.R. (eds.). American Institute of Physics Conference Series, vol. 1343, pp. 51–56 (May 2011)
9. Petreczky, P.: Recent progress in lattice QCD at finite temperature. ArXiv e-prints (June 2009)
10. Alexandrou, C., Brinet, M., Carbonell, J., Constantinou, M., Guichon, P., et al.: Nucleon form factors and moments of parton distributions in twisted mass lattice QCD. In: Proceedings of The XXIst International Europhysics Conference on High Energy Physics, EPS-HEP 2011, Grenoble, Rhones Alpes France, July 21-27, vol. 308 (2011)
11. Winter, F.: Accelerating QDP++ using GPUs. In: Proceedings of the XXIX International Symposium on Lattice Field Theory (Lattice 2011), Squaw Valley, Lake Tahoe, California, July 10-16 (2011)
12. Babich, R., Clark, M.A., Joó, B., Shi, G., Brower, R.C., Gottlieb, S.: Scaling lattice qcd beyond 100 gpus. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 70:1–70:11. ACM, New York (2011)
13. Munshi, A.: The OpenCL specification, Version 1.2 (2011)
14. Bach, M., Lindenstruth, V., Philipsen, O., Pinke, C.: Lattice QCD based on OpenCL. ArXiv e-prints (September 2012)
15. Bonati, C., Cossu, G., D'Elia, M., Incardona, P.: QCD simulations with staggered fermions on GPUs. Computer Physics Communications 183, 853–863 (2012)
16. MILC: MIMD Lattice Computation (MILC) Collaboration, http://physics.indiana.edu/~sg/milc.html
17. Hoberock, J., Bell, N.: Thrust: A parallel template library (2010)
18. NVIDIA Corporation: NVIDIA GPUDirect$^{\text{TM}}$ Technology (2012)
19. NVIDIA Corporation: NVIDIA CUDA C programming guide, Version 4.2 (2012)
20. Alexandru, A., Lujan, M., Pelissier, C., Gamari, B., Lee, F.: Efficient Implementation of the Overlap Operator on Multi-GPUs. In: Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing, SAAHPC 2011, pp. 123–130. IEEE Computer Society, Washington, DC (2011)
21. NVIDIA Corporation: NVIDIA's Next Generation CUDA(TM) Compute Architecture: Kepler(TM) GK110 – Whitepaper (2012)
22. CSC: IT Center for Science, http://www.csc.fi
23. Babich, R., Clark, M.A., Joó, B.: Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, IEEE Computer Society, Washington, DC (2010)
24. Sheikholeslami, B., Wohlert, R.: Improved Continuum Limit Lattice Action for QCD with Wilson Fermions. Nucl. Phys. B259, 572 (1985)
25. Chiu, T.W., Hsieh, T.H., Mao, Y.Y.: Pseudoscalar Meson in Two Flavors QCD with the Optimal Domain-Wall Fermion. Physics Letters B B717, 420 (2012)
26. UPC Consortium: UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab (2005)

Time Propagation of Many-Body Quantum States on Graphics Processing Units

Topi Siro[1,2] and Ari Harju[1,2]

[1] Department of Applied Physics, Aalto University School of Science, Helsinki, Finland
[2] Helsinki Institute of Physics, Helsinki, Finland

Abstract. We demonstrate the effectiveness of graphics processing units (GPU) in computing the time evolution of a many-body quantum state. We study the Hubbard model with exact diagonalization. We discretize the time into small steps and expand the time evolution operator into a Taylor series. An algorithm for calculating the time evolution on a GPU is given and tested on a 1D lattice to observe spin-charge separation.

1 Introduction

In quantum physics, the goal is to solve the Schrödinger equation,

$$H\left|\psi\right\rangle = i\hbar\frac{\partial}{\partial t}\left|\psi\right\rangle, \tag{1}$$

where H is the Hamiltonian that characterizes the energy of the system in question. Formally, given some initial state $|\psi_0\rangle$, we can solve the above equation to give the state of the system at time t as

$$|\psi(t)\rangle = \hat{T}e^{-\frac{i}{\hbar}\int_0^t H(\tau)d\tau}\left|\psi_0\right\rangle, \tag{2}$$

where $\hat{T}$ is the time-ordering operator. In computations, the integral is discretized into N small timesteps Δt:

$$|\psi(t)\rangle = \left(\prod_{j=N}^{0} e^{-\frac{i}{\hbar}H(j\Delta t)}\right)\left|\psi_0\right\rangle, \tag{3}$$

where the operators are time-ordered such that the earliest Hamiltonian operates first. If we assume that the Hamiltonian is time-independent, then we get the simple result,

$$|\psi(t)\rangle = e^{-iHt/\hbar}\left|\psi_0\right\rangle, \tag{4}$$

where we see that the time evolution of a state is given by the operator e^{-iHt} when we choose $\hbar = 1$.

In condensed matter physics, one popular choice for the Hamiltonian H is the Hubbard model, which was introduced in the 1960s to describe interacting electrons in a solid[1,2,3]. It has since been the subject of extensive study and is still a source of interesting physics[4]. It is perhaps the simplest model to display many of the essential features of electron correlations, such as ferromagnetism and conductor-Mott-insulator transition.

In the Hubbard model, the solid is described by a fixed lattice, where electrons can hop from one lattice site to another. The electrons are always bound to an atom, such that their wave functions are vectors whose components squared are the probabilities of finding the electron at the corresponding lattice site. Interactions take place only between electrons that are residing on the same site.

The Hamiltonian can be written as

$$\begin{aligned} H &= H_{hop} + H_{int} && (5) \\ &= -t \sum_{<ij>} \sum_{\sigma=\uparrow,\downarrow} (c^{\dagger}_{i,\sigma} c_{j,\sigma} + h.c) + U \sum_{i} n_{i,\uparrow} n_{i,\downarrow}, && (6) \end{aligned}$$

where $< ij >$ denotes a sum over neighboring lattice sites, $c^{\dagger}_{i,\sigma}$ and $c_{i,\sigma}$ are the creation and annihilation operators which respectively create and annihilate an electron at site i with spin σ, and $n_{i,\sigma} = c^{\dagger}_{i,\sigma} c_{i,\sigma}$ counts the number of such electrons. U is the interaction energy and t is the hopping amplitude. The creation and annihilation operators obey the typical anticommutation rules for fermions,

$$\left\{ c^{\dagger}_{i\sigma}, c_{j\tau} \right\} = \delta_{ij}\delta_{\sigma\tau} \quad \text{and} \quad \left\{ c^{\dagger}_{i\sigma}, c^{\dagger}_{j\tau} \right\} = \{c_{i\sigma}, c_{j\tau}\} = 0, \tag{7}$$

which means that there are four possible occupations for a lattice site: either it is empty, has one up electron, one down electron or one of each.

An important property of the Hamiltonian is that the numbers of both up and down electrons are separately conserved. This is convenient because it allows one to fix the number of up and down electrons and thus restrict to a subspace of the whole Hilbert space.

Despite the model's simplicity, an analytic solution is only available in one dimension, and it was found by Lieb and Wu in 1968[5]. In general, computational methods are required. While both terms in the Hamiltonian are easy to diagonalize separately, their sum is highly nontrivial. One method to numerically solve the Hubbard model is exact diagonalization. The idea is to simply calculate the matrix elements in a suitable basis and then diagonalize the resulting matrix. The obvious downside of this approach is that the number of lattice sites and particles that can be considered is quite low due to the very rapid growth of the dimension of the Hamiltonian matrix as a function of the system size. However, the major advantage is that the results are exact up to numerical accuracy, which makes exact diagonalization well-suited to situations where a perturbative solution is not possible. It can also be used to test the reliability of other, approximative methods by comparing their results with the exact ones.

2 Exact Diagonalization

We will now present our scheme for labeling the basis states and forming the Hamiltonian matrix for the Hubbard model. The approach is very intuitive, and similar schemes can be found in the literature, for example in Ref. [6].

To calculate the matrix elements of the Hubbard Hamiltonian in Equation (6), we choose a simple basis where the lattice sites are numbered from 0 upward and the basis states correspond to all the ways of distributing the electrons in the lattice. For example, if we have $N_s = 4$ lattice sites, $N_\uparrow = 2$ spin up electrons and $N_\downarrow = 3$ spin down electrons, then

$$c^\dagger_{0\uparrow}c^\dagger_{2\uparrow}c^\dagger_{0\downarrow}c^\dagger_{2\downarrow}c^\dagger_{3\downarrow}\,|\mathcal{O}\rangle \tag{8}$$

is one basis state. In state (8), the up electrons reside on sites 0 and 2 and the down electrons on sites 0, 2 and 3. The empty lattice into which the electrons are created is denoted by $|\mathcal{O}\rangle$. To resolve any ambiguities arising from different orderings of the creation operators in Equation (8), we define that in the basis states all spin up operators are to the left of all spin down operators, and site indices are in ascending order.

The dimension of the Hamiltonian matrix is equal to the number of ways of distributing $N_\uparrow$ spin up electrons and $N_\downarrow$ spin down electrons into N_s lattice sites, i.e.

$$\dim H = \binom{N_s}{N_\uparrow}\binom{N_s}{N_\downarrow}. \tag{9}$$

The size of the basis grows extremely fast. For example, in the half-filled case where $N_\uparrow = N_\downarrow = N_s/2$, for 12 sites $\dim H = 853776$, for 14 sites $\dim H \approx 11.8 \times 10^6$ and for 16 sites $\dim H \approx 166 \times 10^6$. In addition, the matrices are very sparse, because the number of available hops, and thus the number of nonzero elements in a row, grows only linearly while the size of the matrix grows exponentially.

To form the Hamiltonian, we need to label and order the basis states. A convenient way to do this is to represent each state with binary numbers such that occupied and unoccupied sites are denoted by 1 and 0, respectively. For example, the state in (8) becomes

$$c^\dagger_{0\uparrow}c^\dagger_{2\uparrow}c^\dagger_{0\downarrow}c^\dagger_{2\downarrow}c^\dagger_{3\downarrow}\,|\mathcal{O}\rangle \to \underbrace{(0101)}_{up} \times \underbrace{(1101)}_{down}. \tag{10}$$

Note that in our convention site indices run from right to left in the binary number representation.

A simple way to order the basis states is to do it according to the size of the binary number that represents the state. Using this scheme, if we index the states by J, the conversion from the binary representation is given by

$$J = i_\uparrow \binom{N_s}{N_\downarrow} + i_\downarrow, \tag{11}$$

Table 1. A scheme for labelling the basis states for $N_s = 4$, $N_\uparrow = 2$, $N_\downarrow = 3$. States are ordered first according to the up spin configuration (first column) and then according to the down spin configuration (second column), in ascending order.

$\uparrow$	$\downarrow$	$i_\uparrow$	$i_\downarrow$	J
0011	0111	0	0	0
0011	1011	0	1	1
0011	1101	0	2	2
0011	1110	0	3	3
0101	0111	1	0	4
0101	1011	1	1	5
0101	1101	1	2	6
0101	1110	1	3	7
0110	0111	2	0	8
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
1100	1101	5	2	22
1100	1110	5	3	23

where $i_\uparrow$ and $i_\downarrow$ are the positions of the up and down configurations in an ordered list, starting from 0, of all N_s-bit numbers with $N_\uparrow$ and $N_\downarrow$ bits set, respectively. To clarify, for example in (10), the possible up configurations, in order, are 0011, 0101, 0110, 1001, 1010 and 1100, so 0101 is the second configuration, and $i_\uparrow = 1$. Similarly, 1101 is the third 4-bit number with 3 bits set, so $i_\downarrow = 2$. Thus, we get

$$J = 1 \times \binom{4}{3} + 2 = 6, \tag{12}$$

which is confirmed by Table 1.

Forming the Hamiltonian matrix is now straightforward. The interaction part H_{int} is diagonal, and it essentially just counts the number of doubly occupied lattice sites and increases the energy by U for each instance. The matrix elements of the hopping part, H_{hop}, are $\pm t$ between basis states that can be reached from each other by a hop of a single electron, and vanish otherwise.

For example, if we have a one-dimensional lattice with periodic boundaries, from Table 1, we see that

$$|\langle 2| H_{hop} |6\rangle| = t \quad \text{and} \quad \langle 3| H_{hop} |6\rangle = 0 \tag{13}$$

because $|6\rangle$ can be reached from $|2\rangle$ when the up electron at site 1 hops to site 2. From $|3\rangle$ to $|6\rangle$ it takes two hops so the matrix element vanishes. Because of the binary number representation of the basis states, in the computer these calculations can be conveniently done with integers and bitshift operations.

The signs of the nonzero matrix elements are determined by the anticommutation relations of the creation and annihilation operators. An extra minus sign is picked up for each electron of the same spin that is hopped over. So in the one-dimensional case, if the hop is over the periodic boundary and the total number of electrons of the same spin is even, the matrix element changes sign. For example,

$$\langle 6| H_{hop} |22\rangle = t \quad \text{and} \quad \langle 0| H_{hop} |3\rangle = -t \tag{14}$$

because there is an even number of up spins and an odd number of down spins (note the minus in the Hamiltonian). Note that the method is completely general and applies to any kind of lattice and any number of electrons. In a general lattice, the sign is determined by the number of electrons of the same spin residing in lattice sites whose labels are between the labels of the origin and the destination of the hop.

3 GPU Computing

3.1 Introduction

Graphics processing units (GPU), originally developed to aid the central processing unit (CPU) in rendering graphics, have evolved into powerful computational engines, capable of general purpose calculations. In recent years, they have been increasingly used in a variety of scientific disciplines, including physics, to speed up computations that benefit from the architecture of the GPU.

The GPU is quite different than the CPU. Simply put, while the CPU performs few concurrent tasks quickly, the GPU executes a very large number of slower tasks simultaneously, i.e. in parallel. Although modern CPUs typically consist of multiple cores, allowing parallel computation to some extent, the scale of parallelization in the GPU is orders of magnitude larger, up to tens of thousands of simultaneous threads in modern cards.

To benefit from GPUs, the problem has to be suitable for large scale parallelization. In addition, specifics of the GPU architecture need to be taken into account. For example, to get a performance gain, the program should have high arithmetic intensity, defined as the number of arithmetic operations divided by the number of memory operations. This is because accesses to the memory have a high latency and it is therefore desirable to hide this latency with calculations. Also, data transfer between the CPU and the GPU through the PCI Express bus is slow, and should therefore be minimized.

3.2 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing architecture by the GPU manufacturer NVIDIA. It allows programmers to utilize GPUs with CUDA capability in general purpose computation through the use of an extension of the C programming language.

CUDA programs consist of the main program that runs on the CPU (*the host*), and special functions called kernels that run on the GPU (*the device*). Since the GPU has a SIMD architecture, kernels are written from the viewpoint of a single thread. Threads are organized into groups that are called blocks.

When a kernel is launched from the host code, the programmer specifies the number of blocks and the number of threads per block in the kernel call. Each thread has its identification number within the block stored in an internal variable threadIdx, which can be one-, two- or three-dimensional. Similarly, the block ID is stored in a variable called blockIdx. With these ID numbers the programmer can assign the threads to different data.

Threads can be synchronized within a block, but blocks have to be able to execute independently. Within a block, threads have a common fast memory space called shared memory that can be used as a user-managed cache. Optimally, one would like to read the relevant data from the slow global memory into the fast shared memory, perform the calculation there and write the result back to the global memory. In the latest generation of NVIDIA GPUs, called Fermi, there are also automatic L1 and L2 caches for global memory accesses. For a more thorough overview of CUDA, we refer to Ref. [7]

4 Time Propagation on the GPU

Previously, GPU implementations relating to time-propagation in quantum mechanics have been reported in References [8] and [9]. In the former, the time-dependent Schrödinger equation is solved with a CUDA program based on Chebyshev polynomials and and the FFT algorithm. The latter introduces a generic solver for a system of first-order ordinary differential equations, which is applied to the time-dependent Schrödinger equation. We will focus on a straightforward series expansion of the time evolution operator, which to our knowledge has not been discussed before in the literature.

4.1 Taylor Series

Although there are numerous ways to compute the exponential of a matrix[10], the computation of the matrix for e^{-iHt} would be very complicated and time-consuming. Fortunately, we are not interested in the matrix itself, but in the state after the time evolution, i.e. the product $e^{-iHt}x$, where x is the initial state vector. The simplest way to obtain the resulting vector y is to expand the exponential in a Taylor series:

$$y = e^{-iHt}x = \left(1 - iHt - \frac{H^2t^2}{2} + \frac{iH^3t^3}{6} + \dots\right)x. \tag{15}$$

There is a simple recursion relation between consecutive terms in the series: if we denote the nth term by y_n, we have

$$y_n = \frac{-iHt}{n}y_{n-1}, \qquad y_0 = x. \tag{16}$$

So all we need to calculate the time evolved state y is to repeatedly operate with H and accumulate the sum to desired order. Our GPU implementation of the Hx

```
1 y = x
2 For n=1,2,...,k, Do
3     temp = Hx
4     temp = temp*(i*dt/n)
5     y = y + temp
6     x = temp
7 End Do
```

Fig. 1. The algorithm for the time evolution of the initial state x_0 by a timestep of lenght dt. The Taylor series is computed to the kth order, and the result is stored in the vector y.

operation has been reported in Ref. [11], where we calculate the ground state of the Hubbard model with the Lanczos algorithm, where the Hx operation is the most time-consuming part of the algorithm with about 99.9% of the total time. We also present the details of our Hx kernel and benchmark results against a single-core CPU implementation. For example, we obtain speedups of over 100 and 60 for the Lanczos algorithm in the case of the 1D Hubbard model at half-filling. Against implementations on N cores, the speedup is divided by approximately N, since the relevant linear algebra operations can be parallelized easily.

To compute the time evolution, we just have to program the recursion defined by Equation 16. The algorithm for a single timestep of length dt can be seen in Figure 4.1. First, we initialize the result vector y with the initial vector x (the first term in Eq. 16). Then, we use the vector *temp* to compute the next term in the series by the recursion rule, and add the result to y. Finally, we copy *temp* to x for the next iteration. The loop runs up to k, which determines the order of the Taylor expansion. Numerical accuracy of the time evolution is controlled by k and the length of the time step, dt. For small dt, the series (15) converges rapidly, and we can take fewer terms in the expansion.

In practice, the multiplication on line 4 has been incorporated into the end of our Hx kernel reported in Ref. [11] to avoid unnecessary global memory accesses. The sum on line 5 is done with the axpy function from the CUBLAS library. Finally, on line 6, instead of moving the actual data, we only switch the pointers x and *temp* for the next iteration.

To reach the state at some time t, we just repeat the loop in Figure 4.1 to propagate by dt each time. If the Hamiltonian is time-dependent, we update the Hamiltonian between iterations. The accuracy should be checked by verifying that the norm of the vector stays at unity. In the time-dependent case, the rate of change in the Hamiltonian also affects the required timestep length. A good way to ensure unitarity is to split a timestep in two and check that the results agree to an adequate degree.

4.2 Imaginary Time Propagation

The algorithm from the previous section also offers an alternative way to find the ground state of the system with the so called imaginary time propagation method (ITP)[12]. In ITP, we study the time evolution of the system under imaginary time, $t \to -it$, which transforms the time evolution operator to e^{-Ht}. To see what this does, consider a state $|\psi\rangle$, decomposed into a superposition of the eigenstates of H, denoted by $|\phi_n\rangle$, with amplitudes α_n:

$$|\psi\rangle = \sum_n \alpha_n |\phi_n\rangle . \tag{17}$$

Now, applying the time evolution operator with imaginary time on this state results in

$$e^{-Ht} |\psi\rangle = \sum_n \alpha_n e^{-Ht} |\phi_n\rangle = \sum_n \alpha_n e^{-E_n t} |\phi_n\rangle , \tag{18}$$

where E_n is the energy of the nth eigenstate.

From (18) we see that as time goes on, the eigenstates decay with a rate that is determined by the corresponding energy. The eigenstate with the lowest energy, i.e. the ground state, decays slowest, so the time evolution tends toward the ground state. Note that the time evolution is no longer unitary because of the missing i in the exponent, meaning that we have to normalize the vector after each step.

Compared to the Lanczos algorithm, finding the ground state with ITP is much slower: typically thousands of steps are required in ITP to reach convergence, while the Lanczos algorithm usually converges in under a hundred steps. Additionally, depending on the order of the Taylor series, there are multiple costly operations with H per step in ITP, while Lanczos only has one per step. However, one use for ITP is to check that the Lanczos implementation is working properly by comparing the results of both methods.

5 Example: Spin-Charge Separation

A curious phenomenon in interacting many-body systems is the decoupling of the charge and spin degrees of freedom. The low-energy excitations are spinons, carrying only spin, and holons, carrying only charge. The spinons and holons move with different velocities, which leads to spin and charge becoming spatially separated. This phenomenon is called spin-charge separation.[13] Its existence in one-dimensional systems is clear because exact solutions to various models are available, and there is also experimental evidence [14].

Jagla et al.[15] studied spin-charge separation in the Hubbard model with exact diagonalization and found that, indeed, at least in one dimension, the Hubbard model exhibits this phenomenon. They considered a one-dimensional chain of 16 sites with periodic boundary conditions. They first calculated the

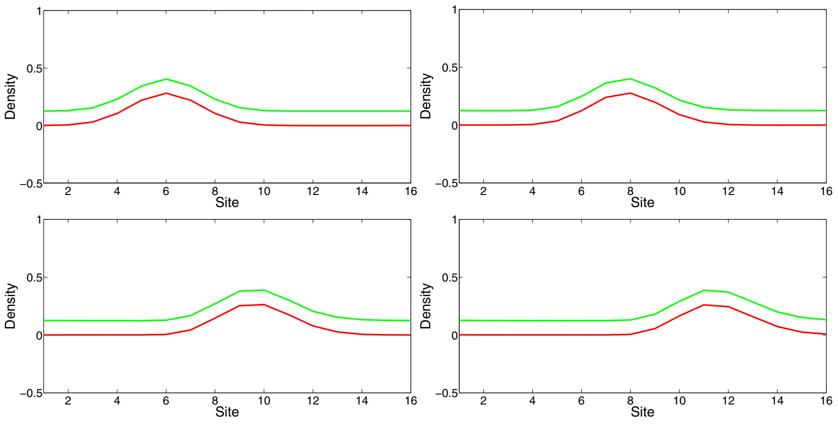


Fig. 2. Time evolution of charge (green) and spin (red) densities, when $U = 0$. The pictures correspond to times $t = 0, 1$ (top row) and $t = 2, 3$ (bottom row).

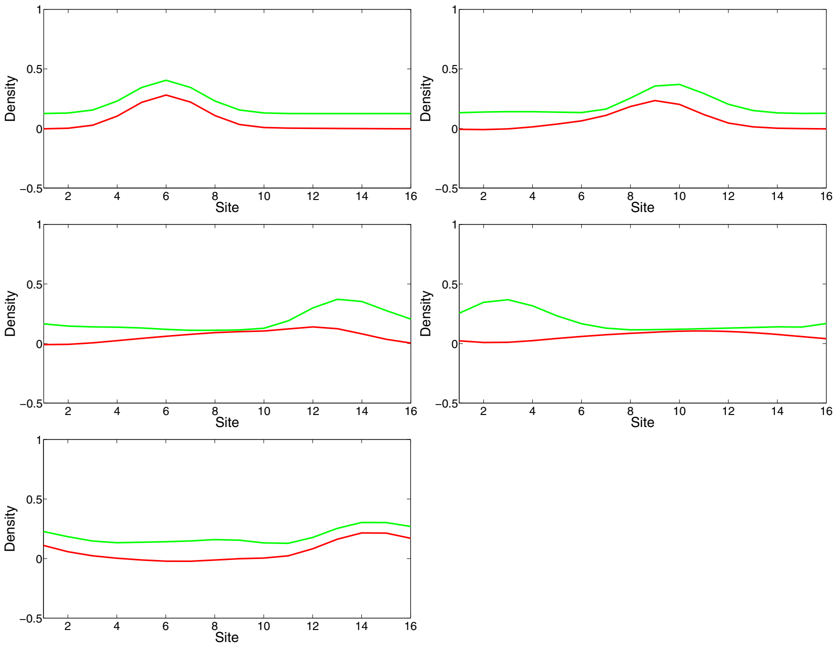


Fig. 3. Time evolution with $U = 10$. The pictures correspond to times $t = 0, 2, 4, 7$ and 13.

two-electron ground state, $|\phi_0\rangle$, and then created another up electron as a wave packet at time $t = 0$, such that the initial state became

$$|\psi(t = 0)\rangle = \sum_j \alpha_j c^\dagger_{j\uparrow} |\phi_0\rangle . \tag{19}$$

The coefficients α generate a Gaussian wave packet,

$$\alpha_j = e^{ik_0(x_j - x_0) - \beta|x_j - x_0|^2}, \tag{20}$$

where k_0 is the momentum and x_0 is the position of the wave packet. The width of the packet is $1/\sqrt{\beta}$.

The quantities of interest are the charge and spin densities, which are respectively defined by $\rho_c(j,t)$ where $n_{j\sigma}$ are the number operators that count the number of electrons with spin σ at site j. Jagla et al. then studied the time evolution of the charge and spin densities, starting from the state (19) with momentum $k_0 = \frac{\pi}{2}$. In the non-interacting case, $U = 0$, they found that the charge and spin wave packets move with the same velocity, as expected. But when the interaction is on, the charge packet moves faster than the spin packet, and the two become spatially separated, indicating spin-charge separation. Results of the duplication of this study with our GPU implementation can be seen in Figures 2 and 3.

In the non-interacting case, the charge and spin densities remain identical, apart from a constant, which is due to the background from the two-electron ground state. In the case U=10, the spin density wave packet starts to lag behind the charge density packet (the second picture in Figure 3). The charge wave packet retains its form, but the spin wave packet spreads wide, and it becomes almost stationary as the charge is on the opposite side of the ring (the fourth picture). When the charge wave packet has completed the loop and arrives at the spin density center of mass, the spin density narrows again (the fifth picture), which Jagla et al. interpret to imply interaction between the charge and spin excitations.

6 Conclusions

We have presented a straightforward way to propagate many-body wave functions in time on a GPU by using the exact diagonalization method and a Taylor series expansion of the time evolution operator e^{iHt}. The crucial advantage in using a GPU is the speedup of the Hx operation, for which an efficient implementation has been detailed in Ref. [11].

To demonstrate our program, we applied it to a 1D Hubbard chain to observe the spin-charge separation when an electron was created as a wave packet in the ground state. Our results were consistent with the ones in Ref. [15].

Our approach could be very useful, for example, in the context of quantum dots, which could serve as realizations of the qubit in a working quantum computer[16]. For instance, our GPU implementation could be used to speed up spin dynamics calculations, which involve multiple time propagations with different parameters[17,18]. Furthermore, dynamics can be used in studying electronic structure and transport properties[19]. For example, efficient time propagation allows the computation of large ensembles over different scatterer configurations, needed for realistic modeling[20].

References

1. Kanamori, J.: Electron correlation and ferromagnetism of transition metals. Prog. Theor. Phys. 30(3), 275–289 (1963)
2. Gutzwiller, M.C.: Effect of correlation on the ferromagnetism of transition metals. Phys. Rev. Lett. 10, 159–162 (1963)
3. Hubbard, J.: Electron correlations in narrow energy bands. Proc. R. Soc. Lond. A 276, 238–257 (1963)
4. Meng, Z.Y., Lang, T.C., Wessel, S., Assaad, F.F., Muramatsu, A.: Quantum spin liquid emerging in two-dimensional correlated Dirac fermions. Nature 464, 847–851 (2010)
5. Lieb, E.H., Wu, F.Y.: Absence of Mott transition in an exact solution of the short-range, one-band model in one dimension. Phys. Rev. Lett. 20(25), 1445–1448 (1968)
6. Lin, H.Q., Gubernatis, J.E.: Exact diagonalization methods for quantum systems. Comput. Phys. 7, 400 (1993)
7. NVIDIA: CUDA C Programming Guide, Version 4.0 (2011), `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`
8. Dziubak, T., Matulewski, J.: An object-oriented implementation of a solver of the time-dependent Schrödinger equation using the CUDA technology. Computer Physics Communications 183(3), 800–812 (2012)
9. Broin, C.O., Nikolopoulos, L.A.A.: An OpenCL implementation for the solution of the time-dependent Schrödinger equation on GPUs and CPUs. Computer Physics Communications 183(10), 2071–2080 (2012)
10. Moler, C., Van Loan, C.: Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. SIAM Rev. 45, 3–49 (2006)
11. Siro, T., Harju, A.: Exact diagonalization of the Hubbard model on graphics processing units. Comp. Phys. Comm. 183, 1884–1889 (2012)
12. Lipparini, E.: Modern Many-Particle Physics: Atomic Gases, Quantum Dots and Quantum Fluids. World Scientific (2003)
13. Kollath, C., Schollwöck, U.: Cold fermi gases: a new perspective on spin-charge separation. New J. Phys. 8, 220 (2006)
14. Kim, C., Matsuura, A.Y., Shen, Z.-X., Motoyama, N., Eisaki, H., Uchida, S., Tohyama, T., Maekawa, S.: Observation of spin-charge separation in one-dimensional $SrCuO_2$. Phys. Rev. Lett. 77, 4054 (1996)
15. Jagla, E.A., Hallberg, K., Balseiro, C.A.: Numerical study of charge and spin separation in low-dimensional systems. Phys. Rev. B 47, 5849 (1993)
16. Hiltunen, T., Harju, A.: Y-junction splitting spin states of a moving quantum dot. Phys. Rev. B 86, 121301 (2012)
17. Särkkä, J., Harju, A.: Spin dynamics at the singlet-triplet crossings in a double quantum dot. New Journal of Physics 13(4), 043010 (2011)

18. Särkkä, J., Harju, A.: Spin dynamics in a double quantum dot: Exact diagonalization study. Phys. Rev. B 77, 245315 (2008)
19. Yuan, S., De Raedt, H., Katsnelson, M.I.: Modeling electronic structure and transport properties of graphene with resonant scattering centers. Phys. Rev. B 82, 115448 (2010)
20. Uppstu, A., Saloriutta, K., Harju, A., Puska, M., Jauho, A.-P.: Electronic transport in graphene-based structures: An effective cross-section approach. Phys. Rev. B 85, 041401 (2012)

Parallel Numerical Simulation of Seismic Waves Propagation with Intel Math Kernel Library

Mikhail A. Belonosov[1], Clement Kostov[2], Galina V. Reshetova[3], Sergey A. Soloviev[1], and Vladimir A. Tcheverda[1]

[1] Institute of Petroleum Geology and Geophysics SB RAS, Novosibirsk, Russia
[2] Schlumberger Moscow Research
[3] Institute of Computational Mathematics and Mathematical geophysics SB RAS, Novosibirsk, Russia

Abstract. This paper describes the implementation of parallel computing to model seismic waves in heterogeneous media based on Laguerre transform with respect to time. The main advantages of the transform are a definite sign of the spatial part of the operator and its independence of the parameter of separation. This property allows one to efficiently organize parallel computations by means of decomposition of the computational domain with successive application of the additive Schwarz method. At each step of the Schwarz alternations, a system of linear algebraic equations in each subdomain is resolved independently of all the others. A proper choice of Domain Decomposition reduces the size of matrices and ensures the use of direct solvers, in particular, the ones based on LU decomposition. Thanks to the independence of the matrix of the parameter of Laguerre transform with respect to time, LU decomposition for each subdomain is done only once, saved in the memory and used afterwards for different right-hand sides.

A software is being developed for a cluster using hybrid OpenMP and MPI parallelization. At each cluster node, a system of linear algebraic equations with different right-hand sides is solved by the direct sparse solver PARDISO from Intel Math Kernel Library (Intel MKL). The solver is extensively parallelized and optimized for the high performance on many core systems with shared memory. A high performance parallel algorithm to solve the problem has been developed. The algorithm scalability and efficiency is investigated. For a two-dimensional heterogeneous medium, describing a realistic geological structure, which is typical of the North Sea, the results of numerical modeling are presented.

1 Introduction

The large-scale numerical simulation of elastic wave propagation in realistic 3D heterogeneous media is impossible without parallel computations based on domain decomposition. So far, the most popular approach here is to use explicit finite-difference schemes based on staggered grids, despite drawbacks such as the necessity to perform data send/receive at each time step, full re-simulation of the wavefield for each new source, and hard disk data storage for implementation of a reverse-time migration. In this regard, considerable attention has recently been given to the development of alternative techniques for simulation of seismic waves, especially, the ones working in the temporal frequency domain [1]. However, the use of such methods for general heterogeneous media also faces a range of significant issues. The main issue is a consequence of the indefiniteness of the impedance matrix. This property brings about a very slow rate of convergence for the iterative procedures solving the linear algebraic equations resulting from the finite-dimensional approximation of the elastic wave equations in the temporal frequency domain.

Our motivation is to overcome this difficulty and to do that, we apply the approach based on the Laguerre transform with respect to time. This leads to a uniformly elliptic system of linear equations [2] and, so, ensures the convergence of the Schwarz alternations [3], based on a suitable domain decomposition with overlapping [4]. We choose the domain decomposition providing the possibility of applying in each of elementary subdomains the LU factorization of the corresponding matrix. Once LU decomposition is performed, it is stored and later used for each component of the Laguerre decomposition and each source position. It should be stressed that it is a consequence of the main advantage of the Laguerre transform: the matrix of the corresponding system does not depend on the separation parameter and, hence, the LU factorization in each subdomain is performed only once.

2 Statement of the Problem: Separation of Time

Let us consider a 2D system of second order elastic equations for a volumetric source with zero initial conditions:

$$\left.\begin{aligned}
&\rho\tfrac{\partial^2 u_1}{\partial t^2} = \tfrac{\partial}{\partial x}\left[(\lambda+2\mu)\tfrac{\partial u_1}{\partial x} + \lambda\tfrac{\partial u_2}{\partial z}\right] + \tfrac{\partial}{\partial z}\left[\mu\tfrac{\partial u_2}{\partial x} + \mu\tfrac{\partial u_1}{\partial z}\right] + g(x,z)f'(t);\\
&\rho\tfrac{\partial^2 u_2}{\partial t^2} = \tfrac{\partial}{\partial x}\left[\mu\tfrac{\partial u_2}{\partial x} + \mu\tfrac{\partial u_1}{\partial z}\right] + \tfrac{\partial}{\partial z}\left[\lambda\tfrac{\partial u_1}{\partial x} + (\lambda+2\mu)\tfrac{\partial u_2}{\partial z}\right] + h(x,z)f'(t);\\
&u_j\big|_{t=0} = \tfrac{\partial u_j}{\partial t}\big|_{t=0} = 0, \quad j = 1,2.
\end{aligned}\right\} \qquad (1)$$

Here ρ is density, λ, μ are Lamet coefficients (P, S-velocities is defined in the following way: $V_p = \sqrt{\frac{\lambda+2\mu}{\rho}}$ and $V_s = \sqrt{\frac{\mu}{\rho}}$). The functions $g(x,z)$ and $h(x,z)$ reflect spacial distribution of the source, $f(t)$ is the source function. Further as a function $f(t)$ we will take the Richer impulse with dominant frequency ν_0:

$$f(t) = \left[1 - 2\pi^2\nu_0^2\left(t - \frac{t_0}{\nu_0}\right)^2\right]\cdot e^{-\pi^2\nu_0^2\left(t-\frac{t_0}{\nu_0}\right)^2}.$$

2.1 Laguerre Transform and Additive Schwarz Method

The integral Laguerre transform for the function $F(t) \in L_2(0,\infty)$ is given by the following relation:

$$F_n = \int_0^\infty F(t)(ht)^{-\frac{\alpha}{2}} l_n^\alpha(ht)dt, \tag{2}$$

with the inversion formula

$$F(t) = \sum_{n=0}^{\infty} F_n \cdot (ht)^{\frac{\alpha}{2}} l_n^\alpha(ht). \tag{3}$$

Here $l_n^\alpha(ht)$ are orthonormal Laguerre functions

$$l_n^\alpha(ht) = \sqrt{\frac{n!}{(n+\alpha)!}}(ht)^{\frac{\alpha}{2}} e^{-\frac{ht}{2}} L_n^\alpha(ht),$$

with $h \in R_+$, $\alpha \in Z_+$ and $L_n^\alpha(ht)$ being classical Laguerre polynomials [5]:

$$L_n^\alpha(y) = \frac{1}{n!} e^y y^{-\alpha} \frac{d^n}{dy^n}\left(y^{\alpha+n} e^{-y}\right). \tag{4}$$

h is the scaling parameter and it is responsible for dilation/compression of the Laguerre functions. The parameter α reflects the attenuation rate i.e. for a smaller size of α – the Laguerre function attenuate faster (see Figure 1).

Application of the integral Laguerre transform (2) to the system of elastic equations (1) transforms it to the system of eliptic second order partial differential equations with a negative definite operator:

$$\left.\begin{array}{l}
\frac{\partial}{\partial x}\left[(\lambda+2\mu)\frac{\partial u_1^n}{\partial x} + \lambda\frac{\partial u_2^n}{\partial z}\right] + \frac{\partial}{\partial z}\left[\mu\frac{\partial u_2^n}{\partial x} + \mu\frac{\partial u_1^n}{\partial z}\right] - \rho\frac{h^2}{4}u_1^n = (\hat{u}_1)_{n-1}; \\
\frac{\partial}{\partial x}\left[\mu\frac{\partial u_2^n}{\partial x} + \mu\frac{\partial u_1^n}{\partial z}\right] + \frac{\partial}{\partial z}\left[\lambda\frac{\partial u_1^n}{\partial x} + (\lambda+2\mu)\frac{\partial u_2^n}{\partial z}\right] - \rho\frac{h^2}{4}u_2^n = (\hat{u}_2)_{n-1}.
\end{array}\right\}$$

Its right-hand side is defined by the recurrence formulas

$$(\hat{u}_1)_{n-1} = \rho h^2 \sqrt{\tfrac{n!}{(n+\alpha)!}} \sum_{k=0}^{n-1} (n-k) \sqrt{\tfrac{(k+\alpha)!}{k!}} u_1^k - g(x,z) f_n,$$

$$(\hat{u}_2)_{n-1} = \rho h^2 \sqrt{\tfrac{n!}{(n+\alpha)!}} \sum_{k=0}^{n-1} (n-k) \sqrt{\tfrac{(k+\alpha)!}{k!}} u_2^k - h(x,z) f_n.$$

In order to define how many Laguerre functions should be used in expansion (3) an empirical criterion is applyed. It is based on the fact that the waveform of volumetric source in a 3D homogeneous medium coincides with the first derivation of the source function. That is why the number of Laguerre functions N is chosen from the condition:

$$\int_0^T \left[f'(t-T) - \sum_{n=0}^{N(T)} f_n (ht)^{-\alpha/2} l_n^\alpha(ht) \right]^2 dt \le \varepsilon^2$$

that ensures the prescribed accuracy of the root-mean-square deviation of an initial impulse from its expansion by the Laguerre functions on the time interval $(0, T)$. An example of such a choice, but for an insufficient number of Laguerre functions is shown in Figure 2.

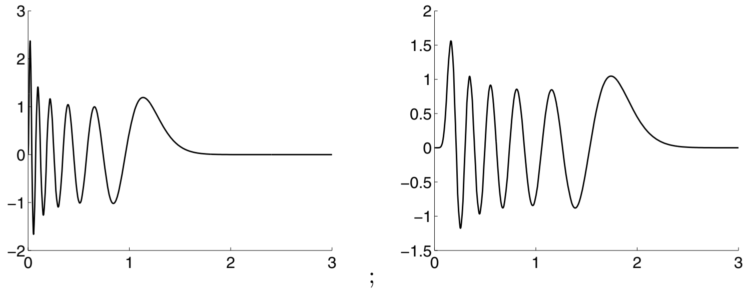


Fig. 1. The Laguerre functions $l_n^\alpha(ht)$ with $n = 10$, $h = 100$ for different values of α: $\alpha = 5$ on the left; $\alpha = 20$ on the right

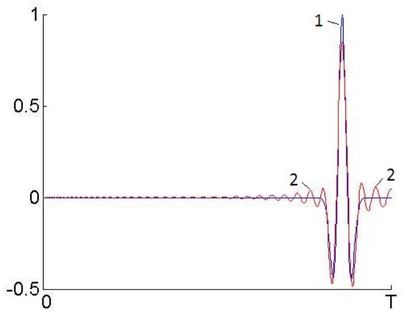


Fig. 2. Line 1 – the Richer impulse, line 2 – its reconstruction for some value of N

3 Numerical Approximation and Organization of Parallel Calculations

Parallelization of the algorithm is implemented on the base of the domain decomposition and the additive Schwarz method.

3.1 Additive Schwarz Method

A full description of the additive Schwarz method can be found in [3,6]. The basic idea of this method is to search for the solution not in the original computational domain, if it is too large, but to decompose it to elementary subdomains of an appropriate size and to resolve the problem in each of these subdomains. In particular, to resolve the boundary value problem in the domain D with the boundary S, it is decomposed to two overlapping subdomains D_1 and D_2 (Figure 3), so two new boundaries S_1 and S_2 are introduced. The Schwarz alternations start with computation of solutions within subdomains D_1 and D_2 with arbitrary boundary conditions on S_1 and S_2, respectively. For each subsequent iteration (m+1), the solution in D_1 is constructed using as boundary conditions on S_1 the trace of a solution in D_2 computed by the previous iteration (m). The same procedure is used to update the solution in D_2. The convergence of iterations for this version of the additive Schwarz method is ensured by the negative definiteness of the operator and overlapping of the neighboring subdomains [3,6].

As a stopping criterion for the Schwarz alternations, we should attain the desired level of threshold of the following value:

$$E_{rr} = \max\left(\frac{\|u_1^n - u_1^{n-1}\|_\Gamma}{\|u_1^{n-1}\|_\Gamma}, \frac{\|u_2^n - u_2^{n-1}\|_\Gamma}{\|u_2^{n-1}\|_\Gamma}\right), \tag{5}$$

where E_{rr} characterizes a relative correlation of the solution on two sequential time steps, Γ is the unification of all the boundaries introduced by domain decomposition on overlapping interfaces. In our numerical simulation, the threshold $E_{rr} \leq 10^{-5}$.

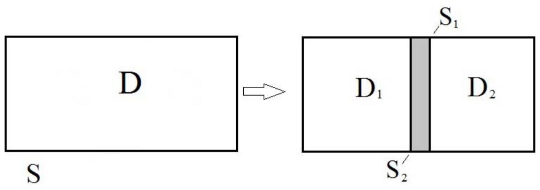


Fig. 3. The overlapping domain decomposition and Schwarz iterations

3.2 Restriction of the Computational Domain

In order to restrict the target area, we use a certain modification of the elastic Perfectly Matched Layer (PML) presented in [7]. Such a modification was proposed and implemented by G.V. Reshetova and V.A. Tcheverda [8]. The main idea is to introduce the PML for a system of first order elastic equations and then to implement the Laguerre transform. As a result, we obtain the following system of equations:

$$\left.\begin{aligned}
&\rho\left(\tfrac{h}{2}+d_x(x)\right)u_{1,1}^n=\tfrac{\partial\sigma_1^n}{\partial x}-\rho(\bar{u}_{1,1})_{n-1},\\
&\rho\left(\tfrac{h}{2}+d_z(z)\right)u_{1,2}^n=\tfrac{\partial\sigma_3^n}{\partial z}-\rho(\bar{u}_{1,2})_{n-1},\\
&\rho\left(\tfrac{h}{2}+d_x(x)\right)u_{2,1}^n=\tfrac{\partial\sigma_3^n}{\partial x}-\rho(\bar{u}_{2,1})_{n-1},\\
&\rho\left(\tfrac{h}{2}+d_z(z)\right)u_{2,2}^n=\tfrac{\partial\sigma_2^n}{\partial z}-\rho(\bar{u}_{2,2})_{n-1},\\
&\left(\tfrac{h}{2}+d_x(x)\right)\sigma_{1,1}^n=(\lambda+2\mu)\tfrac{\partial u_1^n}{\partial x}-(\bar{\sigma}_{1,1})_{n-1}+G(x,z)f_n,\\
&\left(\tfrac{h}{2}+d_z(z)\right)\sigma_{1,2}^n=\lambda\tfrac{\partial u_2^n}{\partial z}-(\bar{\sigma}_{1,2})_{n-1},\\
&\left(\tfrac{h}{2}+d_x(x)\right)\sigma_{2,1}^n=\lambda\tfrac{\partial u_1^n}{\partial x}-(\bar{\sigma}_{2,1})_{n-1}+H(x,z)f_n,\\
&\left(\tfrac{h}{2}+d_z(z)\right)\sigma_{2,2}^n=(\lambda+2\mu)\tfrac{\partial u_2^n}{\partial z}-(\bar{\sigma}_{2,2})_{n-1},\\
&\left(\tfrac{h}{2}+d_x(x)\right)\sigma_{3,1}^n=\mu\tfrac{\partial u_2^n}{\partial x}-(\bar{\sigma}_{3,1})_{n-1},\\
&\left(\tfrac{h}{2}+d_z(z)\right)\sigma_{3,2}^n=\mu\tfrac{\partial u_1^n}{\partial z}-(\bar{\sigma}_{3,2})_{n-1},
\end{aligned}\right\}, \tag{6}$$

where $(\bar{u}_{1,1})_{n-1}$, $(\bar{u}_{1,2})_{n-1}$, $(\bar{u}_{2,1})_{n-1}$, $(\bar{u}_{2,2})_{n-1}$, $(\bar{\sigma}_{1,1})_{n-1}$, $(\bar{\sigma}_{1,2})_{n-1}$, $(\bar{\sigma}_{2,1})_{n-1}$, $(\bar{\sigma}_{2,2})_{n-1}$, $(\bar{\sigma}_{3,1})_{n-1}$, $(\bar{\sigma}_{3,2})_{n-1}$ are calculated by the following recurrent relation:

$$(\bar{w})_{n-1}=h\sqrt{\tfrac{n!}{(n+\alpha)!}}\sum_{k=0}^{n-1}\sqrt{\tfrac{(k+\alpha)!}{k!}}w^k.$$

Recall that unknown functions inside the PML are split to the two components

$$u_1=u_{1,1}+u_{1,2},\quad u_2=u_{2,1}+u_{2,2};$$

$$\sigma_{xx}=\sigma_{xx,1}+\sigma_{xx,2},\quad \sigma_{zz}=\sigma_{zz,1}+\sigma_{zz,2},\quad \sigma_{xz}=\sigma_{xz,1}+\sigma_{xz,2};$$

and seismic wave absorption is provided by special damping functions along the axes x and z:

$$\left.\begin{aligned}
&\begin{pmatrix}d_x(x)\\ d_z(z)\end{pmatrix}=0,\text{if }\begin{pmatrix}x\le a\\ z\le a\end{pmatrix};\\
&\begin{pmatrix}d_x(x)\\ d_z(z)\end{pmatrix}=\left.\begin{matrix}d_0\left(\frac{x-a}{\delta}\right)^4\\ d_0\left(\frac{z-a}{\delta}\right)^4\end{matrix}\right\},\text{otherwise}
\end{aligned}\right\},$$

with a numerical value

$$d_0 = \left| \ln\left(\frac{1}{R}\right) \right| \frac{2 \max\limits_{x,z}(v_p(x,z))}{\delta}.$$

Here a is the beginning of the PML, δ is its width, R is the value of a desired level of artificial reflection from the PML (in our experiments, it was taken $R = 10^{-5}$).

Next let us multiply each odd equation of the system of equations (6) by $d_2 = d_z(z)$, each even equation by $d_1 = d_x(x)$ and after several substitutions we obtain the following system of second order partial differential equations:

$$\left.\begin{aligned}
&\left(\tfrac{h}{2}+d_2\right)\tfrac{\partial}{\partial x}\left[\tfrac{(\lambda+2\mu)}{\frac{h}{2}+d_1}\tfrac{\partial u_x^n}{\partial x}\right] + \tfrac{\partial}{\partial x}\left[\lambda\tfrac{\partial u_z^n}{\partial z}\right] + \tfrac{\partial}{\partial z}\left[\mu\tfrac{\partial u_z^n}{\partial x}\right] + \\
&+\left(\tfrac{h}{2}+d_1\right)\tfrac{\partial}{\partial z}\left[\tfrac{\mu}{\frac{h}{2}+d_2}\tfrac{\partial u_x^n}{\partial z}\right] - \rho\left(\tfrac{h}{2}+d_1\right)\left(\tfrac{h}{2}+d_2\right)u_x^n = \\
&= \rho\left[\left(\tfrac{h}{2}+d_2\right)(\bar{u}_{x,1})_{n-1} + \left(\tfrac{h}{2}+d_1\right)(\bar{u}_{1,2})_{n-1}\right] + \\
&+\left(\tfrac{h}{2}+d_2\right)\tfrac{\partial}{\partial x}\left[\tfrac{(\bar{\sigma}_{1,1})_{n-1}}{\frac{h}{2}+d_1} + \tfrac{(\bar{\sigma}_{1,2})_{n-1}}{\frac{h}{2}+d_2}\right] + \\
&+\left(\tfrac{h}{2}+d_1\right)\tfrac{\partial}{\partial z}\left[\tfrac{(\bar{\sigma}_{3,1})_{n-1}}{\frac{h}{2}+d_1} + \tfrac{(\bar{\sigma}_{3,2})_{n-1}}{\frac{h}{2}+d_2}\right] - \\
&-\tfrac{d_2' f_n\left(\frac{h}{2}+d_2\right)}{\frac{h}{2}+d_1} + \tfrac{d_1'\delta(x-x_0,z-z_0) f_n\left(\frac{h}{2}+d_2\right)}{\left(\frac{h}{2}+d_1\right)^2},
\end{aligned}\right\} \tag{7}$$

$$\left.\begin{aligned}
&\left(\tfrac{h}{2}+d_2\right)\tfrac{\partial}{\partial x}\left(\tfrac{\mu}{\frac{h}{2}+d_1}\tfrac{\partial u_2^n}{\partial x}\right) + \tfrac{\partial}{\partial x}\left[\mu\tfrac{\partial u_1^n}{\partial z}\right] + \tfrac{\partial}{\partial z}\left[\lambda\tfrac{\partial u_1^n}{\partial x}\right] + \\
&+\left(\tfrac{h}{2}+d_1\right)\tfrac{\partial}{\partial z}\left[\tfrac{(\lambda+2\mu)}{\frac{h}{2}+d_2}\tfrac{\partial u_2^n}{\partial z}\right] - \rho\left(\tfrac{h}{2}+d_1\right)\left(\tfrac{h}{2}+d_2\right)u_2^n = \\
&= \rho\left[\left(\tfrac{h}{2}+d_2\right)(\bar{u}_{2,1})_{n-1} + \left(\tfrac{h}{2}+d_1\right)(\bar{u}_{2,2})_{n-1}\right] + \\
&+\left(\tfrac{h}{2}+d_2\right)\tfrac{\partial}{\partial x}\left[\tfrac{(\bar{\sigma}_{3,1})_{n-1}}{\frac{h}{2}+d_1} + \tfrac{(\bar{\sigma}_{3,2})_{n-1}}{\frac{h}{2}+d_2}\right] + \\
&+\left(\tfrac{h}{2}+d_1\right)\tfrac{\partial}{\partial z}\left[\tfrac{(\bar{\sigma}_{2,1})_{n-1}}{\frac{h}{2}+d_1} + \tfrac{(\bar{\sigma}_{2,2})_{n-1}}{\frac{h}{2}+d_2}\right] - \delta_2' f_n + \tfrac{d_2'\delta(x-x_0,z-z_0) f_n}{\frac{h}{2}+d_1}.
\end{aligned}\right\} \tag{8}$$

It is worth mentioning that system (7) – (8) introduces an unsplit PML (see [8]).

3.3 Numerical Approximation

The finite-difference approximation of system (7) – (8) is done by the standard staggered grid scheme [9] that was modified for second order systems [10]. Its stencil is presented in Figure 4. Formally, we need u_1 and u_2 only, all other variables are complement. But their knowledge is necessary for simulation, and so they are computed in the corresponding nodes.

This approximation gives a system of linear algebraic equations with a sparse nine-diagonal matrix. It is worth mentioning that the matrix of this system does not depend on the separation parameter n.

For each value of the separation parameter n, we have a system of linear algebraic equations (SLAE) with the same sparse negative definite matrix but with different right-hand sides. The negative definiteness of the matrix ensures convergence of the additive Schwarz method (see [6]). Since it does not depend on the separation parameter, it is reasonable to use direct solver on the base of LU decomposition: in each subdomain it can be done only once, saved in the RAM and subsequently be used for all right-hand sides.

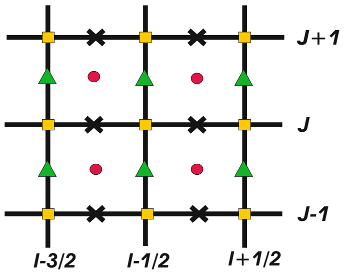


Fig. 4. Stencil of the finite-difference scheme that is used for approximation of system (7) – (8). Squares and circles are for u_1^n and u_2^n respectively, while $\overline{\sigma}$ is calculated in triangles.

3.4 LU FACTORIZATION

In order to perform the LU factorization and to solve a SLAE for a large number of right-hand sides we use Intel Math Kernel Library (Intel MKL) PARDISO direct solver (http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mklman/index.htm) that is parallelized via OpenMP. The PARDISO package is a shared-memory multiprocessing parallel direct solver, designed to solve sparse SLAEs. It is based on row-column reordering of an initial matrix, effective parallelization of a factorization and solving steps. To perform the row-column permutation, Intel MKL PARDISO uses a nested dissection algorithm from the METIS package [11] that decreases the size of a required RAM to store LU factors. In Table 1 we present an amount of RAM required for LU decomposition for the problem we deal with. One can conclude that

1. Because of a sparse structure of a matrix we need only a few megabytes of RAM to store a matrix of finite-difference approximation.
2. The main amount of RAM is used to store the LU factorization.

Table 1. Properties of the LU factorization of a nine-diagonal matrix obtained after finite-difference approximation of system (7) – (8)

Domain size, $n_x = n_z$	Matrix size	Nonzero elements of the matrix	Nonzero elements of LU factors	LU (MB)
100	20 200	180 196	1 620 174	12
200	80 400	720 396	8 108 008	62
400	320 800	2 880 796	39 251 440	299
800	1 281 600	11 521 596	187 492 542	1 430
1 600	5 123 200	46 083 196	858 718 476	6 552

A factorization step of the Intel MKL PARDISO solver is extensively parallelized and optimized for providing a high performance on multi-core systems with shared memory. In order to improve the factorization performance, algorithms of Intel MKL PARDISO are based on a Level-3 BLAS update. Moreover, there are additional features in PARDISO, which can improve the performance, in particular, left-looking [12] and two-level [13] factorization algorithms. The first algorithm improves the scalability on a small number of threads while the second – on many threads (more than eight). The computational cost of solving SLAE with many right-hand sides (RHS) is the same or higher than those needed for factorization. A solving step in Intel MKL PARDISO is optimized both for one RHS and for many RHS. The comparison of PARDISO vs. SuperLU has been made on the cluster of Siberian Supercomputer Center (nodes based on X5675 3.00 GHz Westmere). The results obtained are presented in the chart:

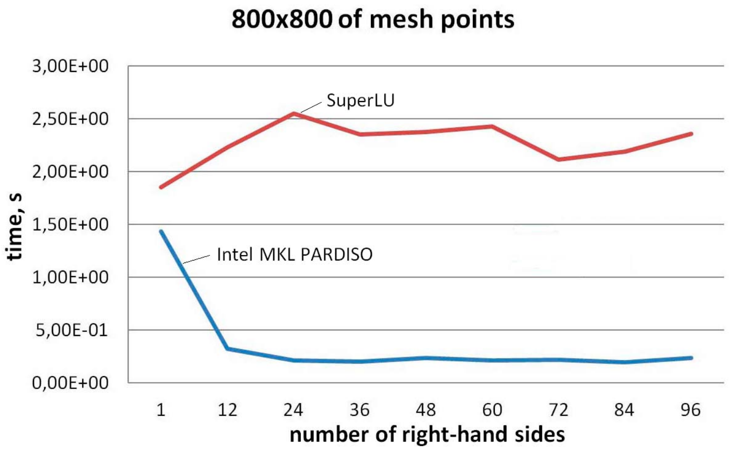


Fig. 5. Dependence the of normalized solving time on the number of RHS

3.5 Parallel Computations

High performance computations are usually performed on computational systems with distributed memory or with MPP (Massive Parallel Processing) architecture. Such systems consist of several nodes with several processors. Each of them has access to RAM of this node. Each processor is usually multi-core. That is why modern computational systems have a hybrid architecture and organized as a set of nodes with distributed memory (MPP architecture), each of them is also computational system with shared memory. Proposed numerical algorithm is orientated onto such architectures and can be effectively loaded at any cluster and consists of the following steps:

1. domain decomposition on "elementary" subdomains providing possibility to store the LU factorization in shared memory of a node (see Table 1);
2. usage of PARDISO MKL for effective parallel computations on the node with OpenMP (SMP architecture);
3. exchanges between subdomains in the process of the Schwarz iterations via MPI (MPP architecture).

The scheme of parallel computations is presented in Figure 6. Numerical experiments were carried out on computational systems with a hybrid parallel architecture. At each node, there are 8 GB RAM, so we can decompose our computational domain to squares of 800×800 mesh points.

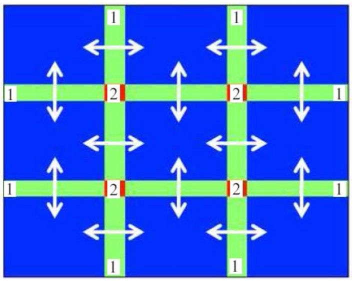


Fig. 6. Parallel computations. Strips 1 correspond to overlapping of two neighbors, while rectangles 2 match overlapping of four subdomains. Arrows correspond to the direction of exchanges between the nodes (MPI).

4 Numerical Experiments

Numerical experiments were carried out on the high performance computer of the Moscow State University with a hybrid parallel architecture: 519 nodes, each of them consists of two quad-core processors and has 8 GB RAM.

The first experiments were performed to understand the main properties of the method. We have considered a simplest situation: a homogeneous elastic medium with the wave propagation velocities $V_p = 2500$ m/s, $V_s = 2000$ m/s and the density $\rho = 2000$ kg/m^3. As the source function, we chose Ricker impulse with dominant frequency 30 Hz. Parameters of the Laguerre transform: $h = 300$, $\alpha = 5$. For the total simulation time $T = 3$ s, 550 polynomials were used. The size of the computational domain was 1000×1000 m with PML.

4.1 Dependence of the Number of Iterations on the Width of Overlapping

First of all, the dependence of the number of iterations on the width of overlapping is analyzed. These results are presented in Table 2. As one can see, the optimal overlapping is equal to 25 points.

Table 2. Dependence of number of iterations on width of the overlapping

Width of the overlapping (m)	Width of the overlapping (number of mesh points)	Number of iterations
30	15	7
40	20	6
50	25	5
70	35	5

Next both weak and strong scalabilities (see [14]) of the algorithm were studied.

4.2 Weak Scalability

Schematically, the way to estimate weak scaling is presented in Figure 7. We fix the load of a node that is equal to the size of the subdomain and enlarge the size of the global computational domain. Thereby we enlarge the number of nodes. As the measure for weak scaling, we use the following function:

$$eff_{weak}(N) = \frac{T(N)}{T(N_0)}, \tag{9}$$

where $T(N)$ is the calculation time for N nodes as long as the size of the problem N times increases (Figure 7). The ideal weak scalability corresponds to $eff_{weak}(n) \equiv 1$.

For the numerical experiment, we took 800×800 mesh points on each node. In Figure 8 the curve $T(N)/T(9)$ is presented. As can be seen, it has a low variations around the one that reflects good weak scalability of the algorithm.

4.3 Strong Scalability

Schematically the way to estimate strong scaling is presented in Figure 9. In contrast to weak scaling, here we fix the size of the global computational domain

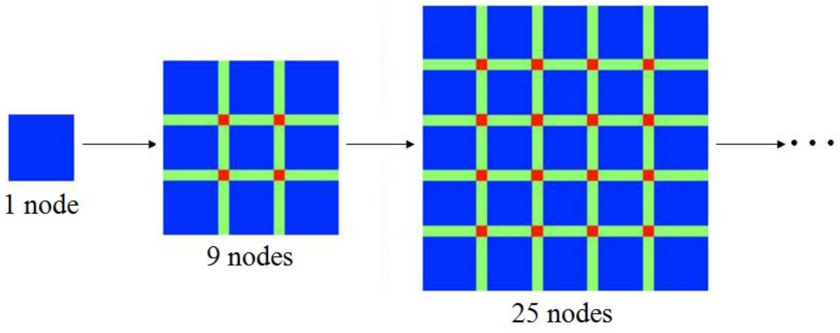


Fig. 7. Weak scaling comuputation

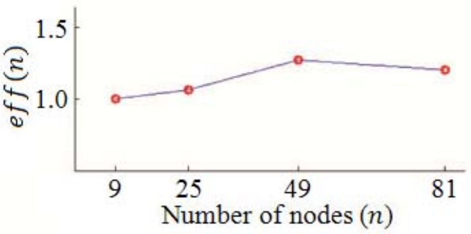


Fig. 8. Weak scalability of the algorithm

and enlarge the number of subdomains. To estimate strong scalability, we use the following function:

$$eff_{strong}(N) = \frac{T(N)}{N_0 \cdot T(N_0)}, \tag{10}$$

where N_0 is the initial number of processes. The ideal strong scalability should coincide with the linear dependence of calculation time on the number of processes, that is $T(N) = \alpha N^{-1}$, where α is certain coefficient. That is why for the ideal strong scalability $eff_{strong}(N) = N^{-1}$.

Numerical experiments to estimate strong scalability were carried out in the same conditions as previously: a homogeneous medium, the Richer impulse with dominant frequency 30 Hz, 550 Laguerre functions, a global computational domain of 2400×2400 mesh points, that is, nine initial subdomains of 800× 800 points, each of them being loaded on its own node.

The result (function 10) is presented in Figure 10 (line 2). Line 1 is the ideal scalability. Thus, one can conclude that the algorithm possesses satisfactory strong scalability as well.

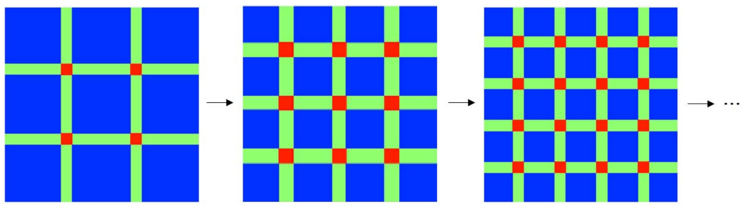


Fig. 9. Strong scaling comuputation

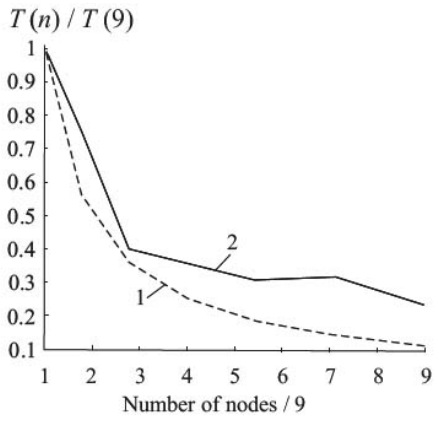


Fig. 10. Strong scalability of the algorithm: line 1 is the ideal scalability, line 2 is the scalability obtained in the numerical experiment

5 Numerical Experiment for Realistic Model

Finally, we would like to present the results of the numerical experiment for presented in Fig.11 (on the left) the realistic Gullfaks model, describing some geological area of the North Sea [15].

The volumetric point source with coordinates (1620, 20) radiates a Ricker impulse with the dominant frequency 30 Hz. The model was discretized on the uniform grid $h_x = h_z = 2$ m that corresponds to 25 mesh points on a wavelength. The integral Laguerre transform with 550 harmonics is computed with $h = 300$ and $\alpha = 3$. The total simulation time is $T = 3$ s. The computational domain is decomposed to (3×3) identical subdomains with overlapping of 50 m (25 points). To obtain solution with the residual $E_{rr} \leq 10^{-5}$ (see (5)) it took 10 Schwarz iterations. The result of this experiment is presented in Figure 11 (on the right).

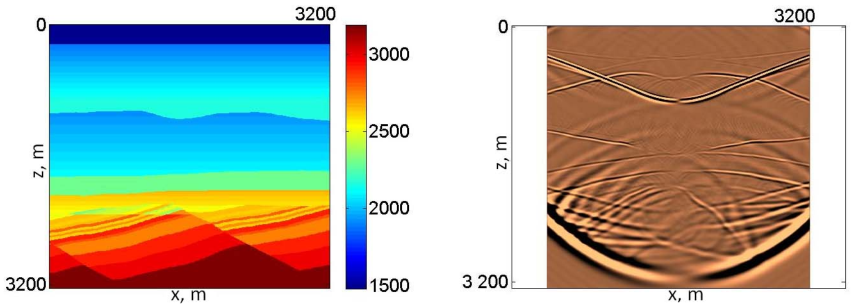


Fig. 11. The 2D Gullfaks model: P-wave velocity (left image); a snapshot for the Gullfaks model (right image)

6 Conclusions

This paper presents the algorithm for the numerical simulation of elastic waves in an inhomogeneous medium based on decomposition of a computational domain, implementation of the integral Laguerre transform and the additive Schwarz method. This algorithm is ideally suited to parallel high-performance computers with a hybrid architecture, representing a set of nodes that combine several multi-core processors with shared memory that are unified by InfiniBand to exchange the data between parallel processes at different nodes. The system of linear algebraic equations for each subdomain is solved with the use of PARDISO from Intel Math Kernel Library (Intel MKL), which is parallelized via OpenMP. The revealed scalability of the algorithm confirms the prospects of the numerical simulation on the base of this algorithm in 3D inhomogeneous media.

To conclude, let us point out that the LU factorization is not the only way to solve the system obtained after numerical approximation. In particular we can use the Cholesky factorization for interior subdomains (outside the PML) and the LU factorization for boundary subdomains. Also one can use an approximation of sparse matrices by matrices of a lower rank (see e.g. [16])).

Acknowledgements. The research described was performed in cooperation with Schlumberger Moscow Research and partially supported by RFBR grants 11-05-00947, 12-05-31008 and President of Russia grant for young scientists MK-77.2013.5. We are also grateful to the R&D site of Intel at Novosibirsk (MKL group) for assistance while application of PARDISO from Intel Math Kernel Library.

References

1. Plessix, R.E.: A Helmholtz iterative solver for 3D seismic-imaging problems. Geophysics 72(5), 185–194 (2007)
2. Mikhailenko, B.G., Mikhailov, A.A., Reshetova, G.V.: Numerical viscoelastic modeling by the spectral Laguerre method. Geophysical Prospecting 51, 37–48 (2003)

3. Chan, T., Mathew, T.P.: Domain decomposition. Acta Numerica 3, 61–143 (1994)
4. Gander, M., Halpern, L., Nataf, F.: Optimized Schwarz Methods. In: 12th International Conference on Domain Decomposition Methods, pp. 15–27 (2001)
5. Suetin, P.K.: Classical orthogonal polynomials. Nauka, M. 203–243 (1974)
6. Nepomnyashchikh, S.V.: Domain decomposition methods. Radon Series Comput. Appl. Math. 1, 81–159 (2007)
7. Collino, F., Tsogka, C.: Application of the PML absorbing layer model to the linear elastodynamic problem in anisotropic heterogeneous media. Geophysics 66(1), 294–307 (2001)
8. Reshetova, G.V., Tcheverda, V.A.: Implementation of Laguerre transform to construct perfectly matched layer without splitting. Mathematical Modelling 18(1), 91–101 (2006)
9. Virieux, J.: P-SV wave propagation in heterogeneous media: Velocity - stress finite-difference method. Geophysics 51(4), 889–901 (1986)
10. Zahradnik, J., Priolo, E.: Heterogeneous formulations of elastodynamic equations and finite-difference schemes. Geophysical Journal International 120(3), 663–676 (1995)
11. Karypis, G., Kumar, V.: A fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing 20(1), 359–392 (1998)
12. Schenk, O., Gartner, K., Fichtner, W.: Efficient Sparce LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors. BIT 240(1), 158–176 (2000)
13. Schenk, O., Gartner, K.: Two-level scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems. Parallel Computing 28, 187–197 (2002)
14. Colella, P., Bell, J., Keen, N., Ligocki, T., Lijewski, M., van Straalen, B.: Performance and scaling of locally-structured grid methods for partial differential equations. Journal of Physics: Conference Series 78, 012013 (2007)
15. Fossen, H., Hesthammer, J.: Structural geology of the Gullfaks Field, northern North Sea, vol. 127, pp. 231–261. Geological Society, London (1998) (Special Publications)
16. Zhang, Z., Zha, H., Simon, H.: Low-Rank Approximation with Sparce Factors: Basic Algorithms and Error Analysis. SIAM J. of Matrix Analysis and Applications (3), 706–727 (1999)

Part III

Parallel Algorithms

Blocked Schur Algorithms for Computing the Matrix Square Root

Edvin Deadman[1], Nicholas J. Higham[2], and Rui Ralha[3]

[1] Numerical Algorithms Group
edvin.deadman@nag.co.uk
[2] University of Manchester
higham@maths.manchester.ac.uk
[3] University of Minho, Portugal
r_ralha@math.minho.pt

Abstract. The Schur method for computing a matrix square root reduces the matrix to the Schur triangular form and then computes a square root of the triangular matrix. We show that by using either standard blocking or recursive blocking the computation of the square root of the triangular matrix can be made rich in matrix multiplication. Numerical experiments making appropriate use of level 3 BLAS show significant speedups over the point algorithm, both in the square root phase and in the algorithm as a whole. In parallel implementations, recursive blocking is found to provide better performance than standard blocking when the parallelism comes only from threaded BLAS, but the reverse is true when parallelism is explicitly expressed using OpenMP. The excellent numerical stability of the point algorithm is shown to be preserved by blocking. These results are extended to the real Schur method. Blocking is also shown to be effective for multiplying triangular matrices.

1 Introduction

A square root of a matrix $A \in \mathbb{C}^{n\times n}$ is any matrix satisfying $X^2 = A$. Matrix square roots have many applications, including in Markov models of finance, the solution of differential equations and the computation of the polar decomposition and the matrix sign function [12].

A square root of a matrix (if one exists) is not unique. However, if A has no eigenvalues on the closed negative real line then there is a unique *principal square root* $A^{1/2}$ whose eigenvalues all lie in the open right half-plane. This is the square root usually needed in practice. If A is real, then so is $A^{1/2}$. For proofs of these facts and more on the theory of matrix square roots see [12].

The most numerically stable way of computing matrix square roots is via the Schur method of Björck and Hammarling [6]. The matrix A is reduced to upper triangular form and a recurrence relation enables the square root of the triangular matrix to be computed a column or superdiagonal at a time. In §2 we show that the recurrence can be reorganized using a standard blocking scheme or recursive blocking in order to make it rich in matrix multiplications. We show experimentally that significant speedups result when level 3 BLAS are exploited in the implementation, with recursive blocking providing the best performance. In §3 we show that the blocked methods maintain the excellent backward stability of the non-blocked method. In §4 we discuss the use of the new approach within the Schur method and explain how it can be extended to the real Schur method of Higham [10]. We compare our serial implementations with existing MATLAB functions. In §5 we compare parallel implementations of the Schur method, finding that standard blocking offers the greatest speedups when the code is explicitly parallelized with OpenMP. In §6 we discuss some further applications of recursive blocking to multiplication and inversion of triangular matrices. Finally, conclusions are given in §7.

2 The Use of Blocking in the Schur Method

To compute $A^{1/2}$, a Schur decomposition $A = QTQ^*$ is obtained, where T is upper triangular and Q is unitary. Then $A^{1/2} = QT^{1/2}Q^*$. For the remainder of this section we will focus on upper triangular matrices only. The equation

$$U^2 = T \tag{1}$$

can be solved by noting that U is also upper triangular, so that by equating elements,

$$U_{ii}^2 = T_{ii}, \tag{2}$$

$$U_{ii}U_{ij} + U_{ij}U_{jj} = T_{ij} - \sum_{k=i+1}^{j-1} U_{ik}U_{kj}. \tag{3}$$

These equations can be solved either a column or a superdiagonal at a time, but solving a column at a time is preferable since it allows more efficient use of cache memory. Different choices of sign in the scalar square roots of (2) lead to different matrix square roots. This method will be referred to hereafter as the "point" method.

The algorithm can be blocked by letting the U_{ij} and T_{ij} in (2) and (3) refer to $m \times m$ blocks, where $m \ll n$ (we assume, for simplicity, that m divides n). The diagonal blocks U_{ii} are then obtained using the point method and the off-diagonal blocks are obtained by solving the Sylvester equations (3) using LAPACK routine xTRSYL (where 'x' denotes D or Z according to whether real or complex arithmetic is used) [4]. Level 3 BLAS can be used in computing the right-hand side of (3) so significant improvements in efficiency are expected. This approach is referred to as the (standard) block method.

To test this approach, a Fortran implementation was written and compiled with gfortran on a 64 bit Intel Xeon machine, using the ACML Library for LAPACK and BLAS calls. Complex upper triangular matrices were generated, with random elements whose real and imaginary parts were chosen from the uniform distribution on $[0, 1)$. Figure 1 shows the run times for the methods, for values of n up to 8000. A block size of 64 was chosen, although the speed did not appear to be particularly sensitive to the block size—similar results were obtained with blocks of size 16, 32, and 128. The block method was found to be up to 6 times faster than the point method. The residuals $\|\widehat{U}^2 - T\|/\|T\|$, where $\widehat{U}$ is the computed value of U, were similar for both methods. Table 1 shows that, for $n = 4000$, approximately 85% of the run time is spent in ZGEMM calls.

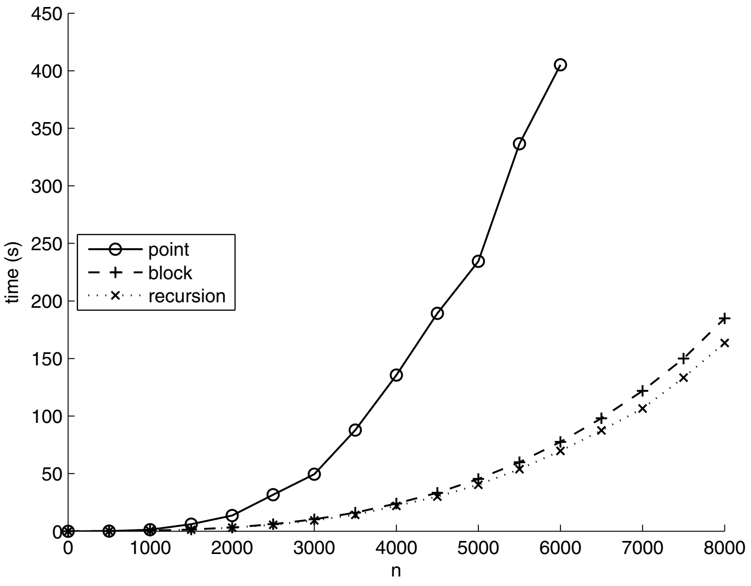


Fig. 1. Run times for the point, block, and recursion methods for computing the square root of a complex $n \times n$ triangular matrix for $n \in [0, 8000]$

A larger block size enables larger GEMM calls to be made. However, it leads to larger calls to the point algorithm and to xTRSYL (which only uses level 2 BLAS). A recursive approach may allow increased use of level 3 BLAS.

Equation (1) can be rewritten as

$$\begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}^2 = \begin{pmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{pmatrix}, \tag{4}$$

where the submatrices are of size $n/2$ or $(n \pm 1)/2$ depending on the parity of n. Then $U_{11}^2 = T_{11}$ and $U_{22}^2 = T_{22}$ can be solved recursively, until some base level is reached, at which point the point algorithm is used. The Sylvester equation $U_{11}U_{12} + U_{12}U_{22} = T_{12}$ can then be solved using a recursive algorithm devised by Jonsson and Kågström [14]. In this algorithm, the Sylvester equation $AX + XB = C$, with A and B triangular, is written as

$$\begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} + \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

where each submatrix is of size $n/2$ or $(n \pm 1)/2$. Then

$$A_{11}X_{11} + X_{11}B_{11} = C_{11} - A_{12}X_{21}, \tag{5}$$

$$A_{11}X_{12} + X_{12}B_{22} = C_{12} - A_{12}X_{22} - X_{11}B_{12}, \tag{6}$$

$$A_{22}X_{21} + X_{21}B_{11} = C_{21}, \tag{7}$$

$$A_{22}X_{22} + X_{22}B_{22} = C_{22} - X_{21}B_{12}. \tag{8}$$

Equation (7) is solved recursively, followed by (5) and (8), and finally (6). At the base level a routine such as xTRSYL is used.

The run times for a Fortran implementation of the recursion method in complex arithmetic, with a base level of size 64, are shown in Figure 1. The approach was found to be consistently 10% faster than the block method, and up to 8 times faster than the point method, with similar residuals in each case. The precise choice of base level made little difference to the run time.

Table 2 shows that the run time is dominated by GEMM calls and that the time spent in ZTRSYL and the point algorithm is similar to the block method. The largest GEMM call uses a submatrix of size $n/4$.

Table 1. Profiling of the block method for computing the square root of a triangular matrix, with $n = 4000$. Format: *time in seconds (number of calls).*

Total time taken:	24.03
Calls to point algorithm:	0.019 (63)
Calls to ZTRSYL	3.47 (1953)
Calls to ZGEMM:	20.54 (39711)

3 Stability of the Blocked Algorithms

We use the standard model of floating point arithmetic [11, §2.2] in which the result of a floating point operation, op, on two scalars x and y is written as

$$fl(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \qquad |\delta| \le u,$$

where u is the unit roundoff. In analyzing a sequence of floating point operations it is useful to write [11, §3.4]

$$\prod_{i=1}^{n} (1 + \delta_i)^{\rho_i} = 1 + \theta_n, \qquad \rho_i = \pm 1,$$

Table 2. Profiling of the recursive method for computing the square root of a triangular matrix, with $n = 4000$. Format: *time in seconds (number of calls)*.

Total time taken:	22.04
Calls to point algorithm:	0.002 (64)
Calls to ZTRSYL	3.37 (2016)
Calls to ZGEMM total:	18.64 (2604)
Calls to ZGEMM with $n = 1000$	7.40 (4)
Calls to ZGEMM with $n = 500$	5.34 (24)
Calls to ZGEMM with $n = 250$	3.16 (112)
Calls to ZGEMM with $n = 125$	1.81 (480)
Calls to ZGEMM with $n <= 63$	0.94 (1984)

where

$$|\theta_n| \le \frac{nu}{1-nu} =: \gamma_n.$$

It is also convenient to define $\widetilde{\gamma}_n = \gamma_{cn}$ for some small integer c whose precise value is unimportant. We use a hat denote a computed quantity and write $|A|$ for the matrix whose elements are the absolute values of the elements of A.

Björck and Hammarling [6] obtained a normwise backward error bound for the Schur method. The computed square root $\widehat{X}$ of the full matrix A satisfies $\widehat{X}^2 = A + \Delta A$, where

$$\|\Delta A\|_F \le \widetilde{\gamma}_{n^3} \|\widehat{X}\|_F^2. \tag{9}$$

Higham [12, §6.2] obtained a componentwise bound for the triangular phase of the algorithm. The computed square root $\widehat{U}$ of the triangular matrix T satisfies $\widehat{U}^2 = T + \Delta T$, where

$$|\Delta T| \le \widetilde{\gamma}_n |\widehat{U}|^2. \tag{10}$$

This bound implies (9). We now investigate whether the bound (10) still holds when the triangular phase of the algorithm is blocked.

Consider the Sylvester equation $AX + XB = C$ in $n \times n$ matrices with triangular A and B. When it is solved in the standard way by the solution of n triangular systems the residual of the computed $\hat{X}$ satisfies [11, §16.1]

$$|C - (A\widehat{X} + \widehat{X}B)| \le \widetilde{\gamma}_n (|A||\widehat{X}| + |\widehat{X}||B|). \tag{11}$$

In the (non-recursive) block method, to bound ΔT_{ij} we must account for the error in performing the matrix multiplications on the right-hand side of (3). Standard error analysis for matrix multiplication yields, for blocks of size m,

$$\left| fl\left(\sum_{k=i+1}^{j-1} \widehat{U}_{ik}\widehat{U}_{kj}\right) - \sum_{k=i+1}^{j-1} \widehat{U}_{ik}\widehat{U}_{kj} \right| \le \widetilde{\gamma}_n |\widehat{U}|^2_{ij}.$$

Substituting this into the residual for the Sylvester equation in the off-diagonal blocks, we obtain the componentwise bound (10).

To obtain a bound for the recursive blocked method we must first check if (11) holds when the Sylvester equation is solved using Jonsson and Kågström's recursive algorithm. This can be done by induction, assuming that (11) holds at the base level. For the inductive step, if suffices to incorporate the error estimates for the matrix multiplications in the right hand sides of (5)–(8) into the residual bound.

Induction can then be applied to the recursive blocked method for the square root. The bounds (10) and (11) are assumed to hold at the base level. The inductive step is similar to the analysis for the block method. Overall, (10) is obtained.

We conclude that both our blocked algorithms for computing the matrix square root satisfy backward error bounds of the same forms (9) and (10) as the point algorithm.

4 Serial Implementations

When used with full (non-triangular) matrices, more modest speedups are expected because of the significant overhead in computing the Schur decomposition. Figure 2 compares run times of the MATLAB function `sqrtm` (which does not use any blocking) and Fortran implementations of the the point method (`fort_point`) and the recursive blocked method (`fort_recurse`), called from within MATLAB using a mex interface, on a 64 bit Intel i3 machine. The matrices have elements whose real and imaginary parts are chosen from the uniform random distribution on the interval $[0, 1)$. The recursive routine is found to be up to 2.5 times faster than `sqrtm` and 2 times faster than `fort_point`.

An extension of the Schur method due to Higham [10] enables the square root of a real matrix to be computed without using complex arithmetic. A real Schur decomposition of A is computed. Square roots of the 2×2 diagonal blocks of the upper quasi-triangular factor are computed using an explicit formula. The recurrence (3) now proceeds either a block column or a block superdiagonal at a time, where the blocks are of size 1×1, 1×2, 2×1, or 2×2 depending on the diagonal block structure. A MATLAB implementation of this algorithm `sqrtm_real` is available in the Matrix Function Toolbox [9]. The algorithm can also be implemented in a recursive manner, the only subtlety being that the "splitting point" for the recursion must be chosen to avoid splitting any 2×2 diagonal blocks. A similar error analysis to §3 applies to the real recursive method, though since only a normwise bound is available for the point algorithm applied to the quasi-triangular matrix the backward error bound (10) holds in the Frobenius norm rather than elementwise.

Figure 3 compares the run times of `sqrtm` and `sqrtm_real` with Fortran implementations of the real point method (`fort_point_real`) and the real recursive method (`fort_recurse_real`), also called from within MATLAB. The matrix elements are chosen from the uniform random distribution on $[0, 1)$. The recursive routine is found to be up to 6 times faster than `sqrtm` and `sqrtm_real` and 2 times faster than `fort_point_real`.

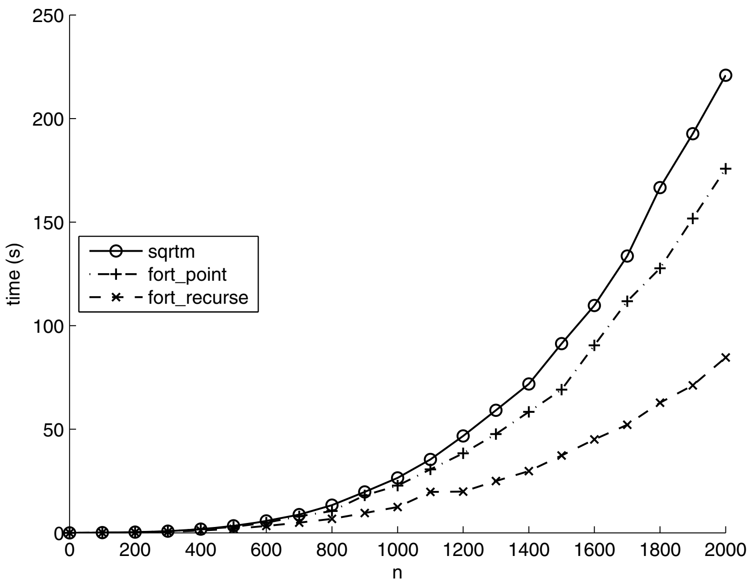


Fig. 2. Run times for `sqrtm`, `fort_recurse`, and `fort_point` for computing the square root of a full $n \times n$ matrix for $n \in [0, 2000]$

Both the real and complex recursive blocked routines spend over 90% of their run time in computing the Schur decomposition, compared with 44% for `fort_point`, 46% for `fort_point_real`, 25% for `sqrtm`, and 16% for `sqrtm_real`. The latter two percentages reflect the overhead of the MATLAB interpreter in executing the recurrences for the (quasi-) triangular square root phase. The 90% figure is consistent with the flop counts of $28n^3$ flops for computing the Schur decomposition and transforming back from Schur form and $n^3/3$ flops for the square root of the triangular matrix.

5 Parallel Implementations

The blocked and recursive algorithms allow parallel architectures to be exploited simply by using threaded BLAS. Further performance gains might be extracted by explicitly parallelizing the triangular phase using OpenMP.

In (3), the (i, j) element of U can be computed only after the elements to its left in the ith row and below it in the jth column have been found. Computing U by column therefore offers no opportunity for parallelism within the column computation. Instead we will compute U by superdiagonal, which allows the elements on each superdiagonal to be computed in parallel. Parallelization of the blocked algorithm is analogous.

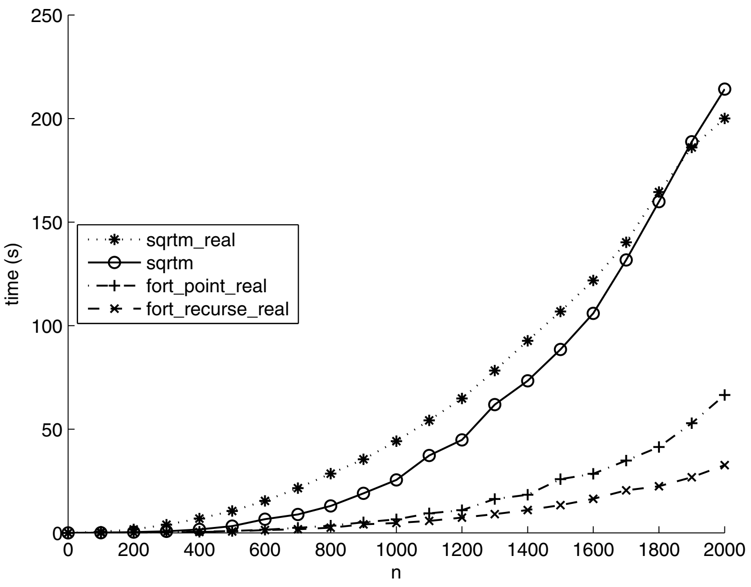


Fig. 3. Run times for `sqrtm`, `sqrtm_real`, `fort_recurse_real` and `fort_point_real` for computing the square root of a full $n \times n$ matrix for $n \in [0, 2000]$

The recursive block method can be parallelized using OpenMP tasks. Each recursive call generates a new task. Synchronization points are required to ensure that data dependencies are preserved. Hence, in equation (4), U_{11} and U_{22} can be computed in parallel, and only then can U_{12} be found. When solving the Sylvester equation recursively, only (5) and (8) can be solved in parallel.

When sufficient threads are available (for example when computing the Schur decomposition) threaded BLAS should be used. When all threads are busy (for example during the triangular phase of the algorithm), serial BLAS should be used, to avoid the overhead of creating threads unnecessarily. Unfortunately, it is not possible to control the number of threads available to individual BLAS calls in this way. In the implementations described below threaded BLAS are used throughout, despite this "overparallelization" overhead.

The parallelized Fortran test codes were compiled on a machine containing 4 Intel Xeon CPUs, with 8 available threads, linking to ACML threaded BLAS [1]. Figure 4 compares run times for the triangular phase of the algorithm, with triangular test matrices generated with elements having real and imaginary parts chosen from the uniform random distribution on the interval $[0, 1)$.

The point algorithm does not use BLAS, but 2-fold speedups on eight cores are obtained using OpenMP. With standard blocking, threaded BLAS alone gives a 2-fold speed up, but using OpenMP gives a 5.5 times speedup. With recursive blocking, a 3-fold speedup is obtained by using threaded BLAS,

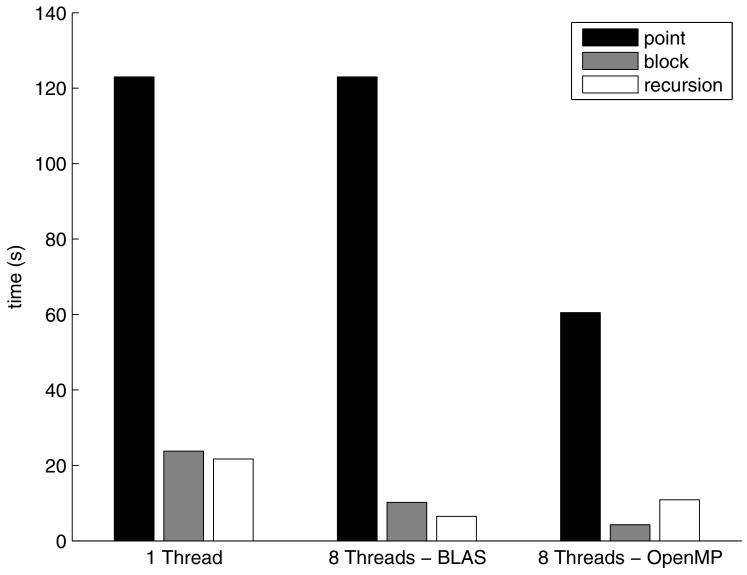


Fig. 4. Run times for parallel implementations of the point, block, and recursion methods for computing the square root of a 4000×4000 triangular matrix

but using OpenMP then decreases the performance because of the multiple synchronization points at each level of the recursion. Overall, if the only parallelization available is from threaded BLAS, then the recursive algorithm is the fastest. However, if OpenMP is used then shorter run times are obtained using the standard blocking method.

Figure 5 compares run times for computing the square root of a full square matrix. Here, the run times are dominated by the Schur decomposition, so the most significant gains are obtained by simply using threaded BLAS and the gains due to the new triangular algorithms are less apparent.

6 Further Applications of Recursive Blocking

We briefly mention two further applications of recursive blocking schemes.

Currently there are no LAPACK or BLAS routines designed specifically for multiplying two triangular matrices, $T = UV$ (the closest is the BLAS routine xTRMM which multiplies a triangular matrix by a full matrix). However, a block algorithm is easily derived by partitioning the matrices into blocks. The product of two off-diagonal blocks is computed using xGEMM. The product of an off-diagonal block and a diagonal block is computed using xTRMM. Finally the point method is used when multiplying two diagonal blocks.

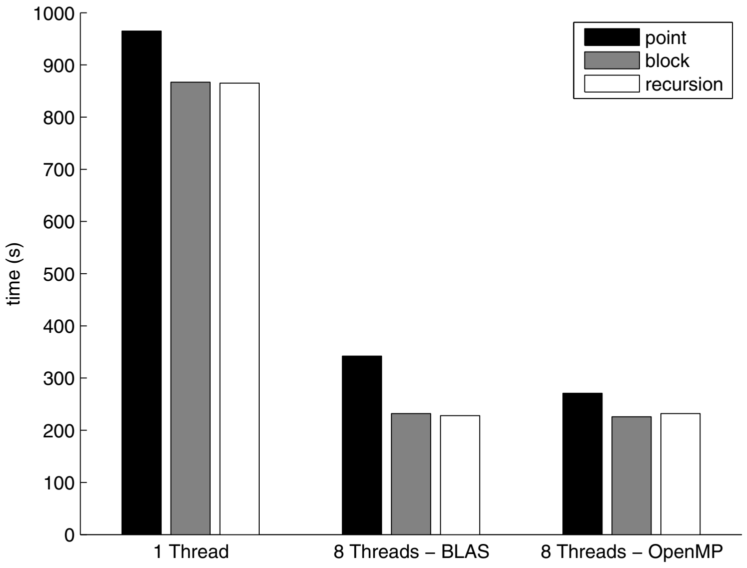


Fig. 5. Run times for parallel implementations of the point, block, and recursion methods for computing the square root of a 4000 × 4000 full matrix

In the recursive approach, $T = UV$ is rewritten as

$$\begin{pmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \begin{pmatrix} V_{11} & V_{12} \\ 0 & V_{22} \end{pmatrix}.$$

Then $T_{11} = U_{11}V_{11}$ and $T_{22} = U_{22}V_{22}$ are computed recursively and $T_{12} = U_{11}V_{12} + U_{12}V_{22}$ is computed using two calls to xTRMM.

Figure 6 shows run times for some triangular matrix multiplications using serial Fortran implementations of the point method, standard blocking, and recursive blocking on a single Intel Xeon CPU (the block size and base levels were both 64 in this case, although the results were not too sensitive to the precise choice of these parameters). As for the matrix square root, the block algorithms significantly outperform the point algorithm, with the recursive approach outperforming the standard blocking approach by approximately 5%. However, if the result of the multiplication is required to overwrite one of the matrices (so that $U \leftarrow UV$, as is the case in xTRMM) then standard blocking may be preferable because less workspace is required.

The inverse of a triangular matrix can be computed recursively, by expanding $UU^{-1} = I$ as

$$\begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \begin{pmatrix} (U^{-1})_{11} & (U^{-1})_{12} \\ 0 & (U^{-1})_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix}.$$

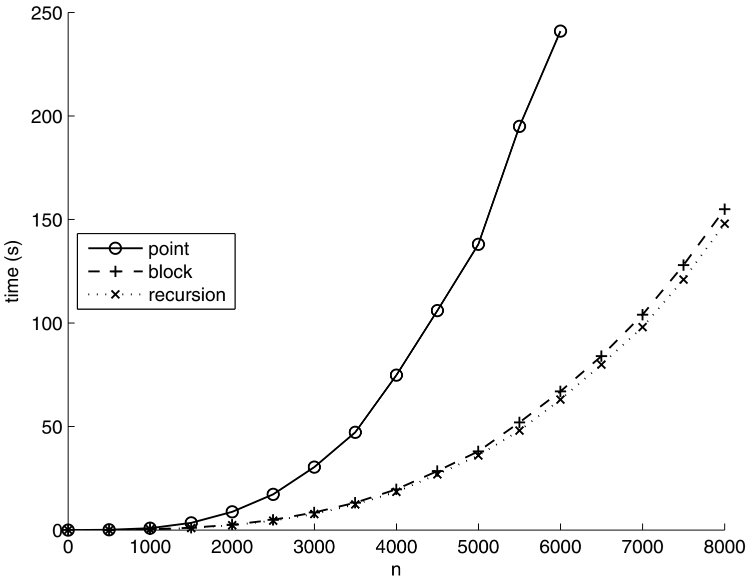


Fig. 6. Run times for the point, block, and recursion methods for multiplying randomly generated $n \times n$ triangular matrices for $n \in [0, 8000]$

Then $(\widehat{U}^{-1})_{11}$ and $(\widehat{U}^{-1})_{22}$ are computed and $(\widehat{U}^{-1})_{12}$ is obtained by solving $U_{11}(\widehat{U}^{-1})_{12} + U_{12}(\widehat{U}^{-1})_{22} = 0$. Provided that forward substitution is used, the right (or left) recursive inversion method can be shown inductively to satisfy the same right (or left) elementwise residual bound as the point method [7]. A Fortran implementation of this idea was found to perform similarly to LAPACK code xTRTRI, so no real benefit was derived from recursive blocking.

7 Conclusions

We investigated two different blocking techniques within Björck and Hammarling's recurrence for computing a square root of a triangular matrix, finding that in serial implementations recursive blocking gives the best performance. Neither approach entails any loss of backward stability. We implemented the recursive blocking with both the Schur method and the real Schur method (which works entirely in real arithmetic) and found the new codes to be significantly faster than corresponding point codes, which include the MATLAB functions `sqrtm` (built-in) and `sqrtm_real` (from [9]). Parallel implementations were investigated using a combination of threaded BLAS and explicit parallelization via OpenMP. When the only parallelization comes from threaded BLAS recursive blocking still gives the best performance. However, when OpenMP is used better performance is obtained using standard blocking. The new codes will appear in a future mark of the NAG Library [15]. Since future marks of the NAG Library will be implemented explicitly in parallel with OpenMP, the standard blocking algorithm will be used. Recursive blocking is also fruitful for multiplying triangular matrices.

Because of the importance of the (quasi-) triangular square root, which arises in algorithms for computing the matrix logarithm [2], [3], matrix pth roots [5], [8], and arbitrary matrix powers [13], this computational kernel is a strong contender for inclusion in any future extensions of the BLAS.

References

1. Advanced Micro Devices, Inc., Numerical Algorithms Group Ltd. AMD Core Math Library (ACML), 4.1.0 edn. (2008)
2. Al-Mohy, A.H., Higham, N.J.: Improved inverse scaling and squaring algorithms for the matrix logarithm. SIAM J. Sci. Comput. 34(4), C153–C169 (2012)
3. Al-Mohy, A.H., Higham, N.J., Relton, S.D.: Computing the Fréchet derivative of the matrix logarithm and estimating the condition number. MIMS EPrint 2012.72. Manchester Institute for Mathematical Sciences. The University of Manchester, UK (2012)
4. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. Society for Industrial and Applied Mathematics. Philadelphia, PA (1999)
5. Bini, D.A., Higham, N.J., Meini, B.: Algorithms for the matrix pth root. Numer. Algorithms 39(4), 349–378 (2005)
6. Björck, Å., Hammarling, S.: A Schur method for the square root of a matrix. Linear Algebra Appl. 52/53, 127–140 (1983)
7. Du Croz, J.J., Higham, N.J.: Stability methods for matrix inversion. IMA J. Numer. Anal. 12(1), 1–19 (1992)
8. Guo, C.-H., Higham, N.J.: A Schur–Newton method for the matrix pth root and its inverse. SIAM J. Matrix Anal. Appl. 28(3), 788–804 (2006)
9. Higham, N.J.: The Matrix Function Toolbox, `http://www.ma.man.ac.uk/~higham/mftoolbox`
10. Higham, N.J.: Computing real square roots of a real matrix. Linear Algebra Appl. 88/89, 405–430 (1987)
11. Higham, N.J.: Accuracy and Stability of Numerical Algorithms, 2nd edn. SIAM (2002)
12. Higham, N.J.: Functions of Matrices: Theory and Computation. SIAM (2008)
13. Higham, N.J., Lin, L.: A Schur–Padé algorithm for fractional powers of a matrix. SIAM J. Matrix Anal. Appl. 32(3), 1056–1078 (2011)
14. Jonsson, I., Kågström, B.: Recursive blocked algorithms for solving triangular systems - part I: One-sided and coupled Sylvester-type matrix equations. ACM Trans. Math. Software 28(4), 392–415 (2002)
15. Numerical Algorithms Group. The NAG Fortran Library, `http://www.nag.co.uk`

Distributed Evolutionary Computing System Based on Web Browsers with JavaScript

Jerzy Duda and Wojciech Dłubacz

AGH University of Science and Technology, Krakow, Poland
{jduda,wdlubacz}@zarz.agh.edu.pl

Abstract. The paper presents a distributed computing system that is based on evolutionary algorithms and utilizing a web browser on a client's side. Evolutionary algorithm is coded in JavaScript language embedded in a web page sent to the client. The code is optimized with regards to the memory usage and communication efficiency between the server and the clients. The server side is also based on JavaScript language, as node.js server was applied. The proposed system has been tested on the basis of permutation flowshop scheduling problem, one of the most popular optimization benchmarks for heuristics studied in the literature. The results have shown, that the system scales quite smoothly, taking additional advantage of local search algorithm executed by some clients.

Keywords: distributed computing, browser-based computation, evolutionary algorithm, JavaScript, flowshop scheduling.

1 Introduction

A lot of problems that can be found in the real world are hard to solve, especially the ones that are proven to be NP-hard. For large instances of such problems it is impossible to find an optimal solution within a reasonable period of time, so various heuristics are usually used in order to find suboptimal solutions in an acceptable time. Among them the most popular and widely studied in the world literature are metaheuristics based on phenomena and laws taken from nature, e.g. evolutionary algorithms (EA), simulated annealing (SA), or most recently, ant colony optimization (ACO), particle swarm optimization (PSO) and artificial bee colonies (ABC). Besides these metaheuristics there are many others like tabu search (TS), iterative local search (ILS) or variable neighbourhood search (VNS). All of them can be divided into two groups: operating on a single solution in each iteration (SA, TS, ILS, VNS) and operating on a set of solutions, often referred to as a population (EA, ACO, PSO, ABC).

Further decrease of computation time for hard-to-solve problems can be achieved by the parallelization of heuristic algorithms, which can be introduced in a simple and almost natural way for the metaheuristics, which deal with the population of solutions. There are three general types of such parallelization that are usually exploited in distributed systems. In the global parallelization model there is one population, but calculations of objective function for the population's members are performed in a parallel way by slave units. This approach is particularly useful in a multicore or a multiprocessor architecture, where communication time for large problem instances is almost negligible. In the island model the whole population is divided into subpopulations that can be run even on different heterogeneous machines. In this case communication time is significant, so subpopulations are run independently and occasionally exchange solutions among each other. Finally, in the master-slave model there is a central population that communicates with subpopulations to collect the solutions [1].

Distributed computing usually utilizes the island model, as it is the most universal, machine and system independent and does not require symmetric or synchronised communication. This paper presents such system that is primary intended for computing large instances of NP-hard problems. The system is based on evolutionary algorithm and on the client's side it utilizes currently probably the most popular and widely used computer tool – a web browser.

In the next section the idea of distributed computing over the Internet will be discussed. The third section describes architectural details of the proposed system, while in the fourth section some computational experiments in finding solution to the permutation flowshop scheduling problem are provided.

2 Distributed Computing Based on Web Browsers

Distributed computing based on the Internet clients for solving large scientific problems has become popular in the recent decade, since contemporary ordinary personal computers that can be found at our homes or offices have significant computing power that, if combined, can outperform not one contemporary supercomputer. Currently (as of October 2012), the world's fastest supercomputer IBM Sequoia has a computing power of 16.3 PetaFLOPS, the second K computer has 10.5 PetaFLOPS and the third Tianhe-IA has 'only' 2.5 PetaFLOPS [2], while the most popular distributed computing network BOINC consisted of more than 300 thousand computers is estimated to have a 24-hour average computing power of more than 7.2 PetaFLOPS [3].

The most frequently used platforms for distributed computing over the Internet are the aforementioned Berkeley Open Infrastructure for Network Computing (BOINC) and distributed.net. They both require the participants in a computational project to install a special application dedicated to a particular operating system. Such application can be then run in a background as a demon or as a service, utilizing some of the client's CPU or GPU resources to perform a particular task. The task that is calculated depends on the project, e.g. participants compute some mathematical problems in ABC@home for, biochemical problems in Rosetta@home or astronomical challenges in the most popular project SETI@home.

Despite the growing popularity of distributed computing over the Internet, systems that do not require any additional application on the client's side, e.g. based entirely on a web browser and JavaScript are very uncommon. Only few examples of such systems can be found in the world literature. One of the first was a browser-based distributed evolutionary system proposed by Merelo et. al. [4]. The system utilized the master-slave model of distributed computing and used AJAX technology for sending a request from a client to a server in order to receive a task to compute. The server for each request generated an individual and sent it back to the client for evaluation. After evaluation the value of objective function for that individual was sent back to the server. As only the server was responsible for generation of new populations, the system scaled poorly. Moreover, the clients' web browsers were permanently occupied with the calculations of objective functions so it was virtually impossible to simultaneously use them for other purposes like browsing the Internet.

More efficient distributed system called Parasitic JavaScript was proposed by Jenkin in [5]. The system utilized JavaSript Object Notation (JSON) instead of XML for exchanging information between the web browsers and the server, and it was possible to limit the CPU load on the clients machines thanks to timer based programming. The author described that his system could be used e.g. for genetic algorithm solving n-queens problem, but did not provide any detailed results.

The idea of parasitic computing was first described by Barabasi et al in [6]. Their system consisted of a collection of target nodes (web servers) connected to a network and a single home parasite node that initiated the computation by sending messages to target computers directing them to perform particular computations. The construction of the message was such that an invalid solution failed the TCP checksum and was dropped, thus only valid solutions were carried back to the server. In that way the authors solved a 2-SAT problem. Although the proposed method was not very practical for solving the problem, it showed that user computers could be applied for computational purpose even without the awareness of their owners.

The system proposed in this paper can be also used in parasitic computing, however, the users that would like to participate in some computations have to intentionally go to a given web address. Moreover, they can look at the JavaScript code that is embedded in a web page to see what kind of computations are performed.

3 System Architecture

Distributed evolutionary system proposed in this paper uses Internet technologies similar to the ones used in Jenkin's Parasitic JavaScript system, but first of all, it is based on the island model, so the best solutions found by the clients are exchanged and the system uses collective intelligence of the independent agents running in web browsers. The second difference is that our system is based almost solely on the JavaScript language, as also the server side is built around node.js system [15]. Node.js is a framework for developing high-performance, concurrent programs that do not rely on the mainstream multithreading approach, but use asynchronous I/O with an event-driven programming model [16]. The prototype version of the system uses node.js version 0.6.10 with socket.IO library version 0.8.7 [17].

One of the advantages of the node.js server is its ability to serve a huge number of concurrent requests. In the preliminary tests presented in Section 4 there were only 16 clients used, however, the tests performed by some developers indicate that node.js is able to handle even 100 000 concurrent connections on a single core [18]. Moreover, the current version of node.js framework has an experimental support for easy clustering. Such cluster implements a typical prefork server, in which child processes (called workers) are spawned per CPU. Cluster implementation in node.js also provides crucial functionality like zero-downtime restarts and worker resuscitation [19].

The node.js server has been chosen to build a prototype of the system, because it is relatively easy to implement, as it uses JavaScript language like on the clients' side. To make the system even more scalable, C++ or some modern functional language will be considered on the server's side in the future.

The clients use JavaScript code embedded in a web page provided by the server. JavaScript code is precompiled by all contemporary web browsers, so computing potential of the language is comparable to other programming languages that use JIT compilation like Java or C#. Very fast interpretation of JavaScript code was first introduced by Google programmers in their V8 engine project [19]. For some popular benchmarks it has been shown that JavaScript was only two times slower than Fortran language that usually had the best performance [20]. The authors of the paper have used different web browsers (Mozilla FireFox, Google Chrome and Internet Explorer 8) for the clients of the system and did not found any significant differences in the time of JS code execution.

The server and the clients communicate with each other by sending messages in JSON format. The size of a single message depends on the size of the problem, e.g. for the flowshop problem with 20 jobs it used about 215 bytes per one message – it was then shorter than an average HTTP header.

Concerning the memory usage on the clients' side, the JavaScript code that is run in web browsers is designed in the way that ensures efficient work of a built-in garbage collector. For example the code does not have any cycle references, so the garbage collector can freely remove unused objects from memory.

General architecture of the system is outlined in Figure 1.

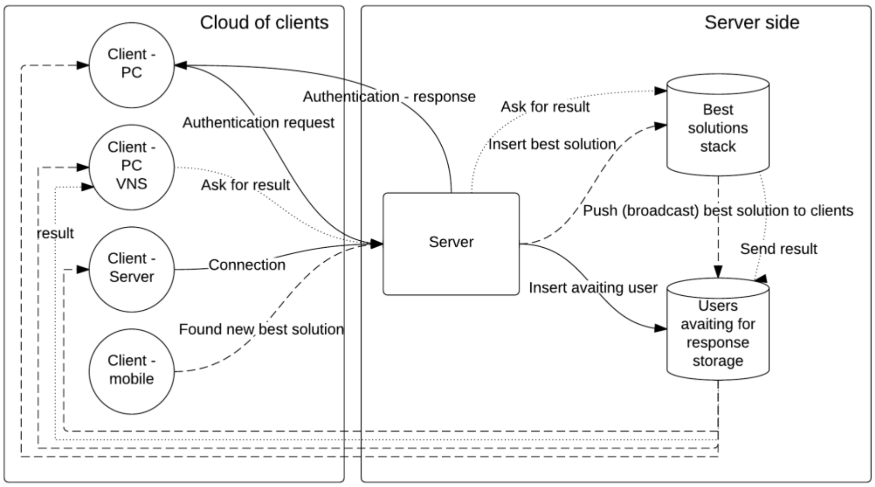


Fig. 1. The overall system architecture

A client (a desktop computer, a mobile phone or any other device able to interpret JavaScript code) connects to the server by opening a web page that includes embedded JavaScript code executing a genetic algorithm or a local search algorithm (VNS). Following scenarios are possible:

1. A client is authorized by the server and this request is put on the stack of the users awaiting for the data from the server (solid line).
2. A client finds better solution than the best solution found so far and this solution is put on the stack of the best solutions found by all genetic algorithms executed by the clients. The stack created in this way will be used later for local search algorithms in order to further improve the results. Simultaneously this solution is broadcasted to the remaining clients, informing them about the most current best solution (dashed line).
3. The client that received the request to execute a local search algorithm first requests the server for the solution to be improved (from the stack). The server sends a random solution, and the client executes the local search. If the solution is improved it is sent to the server, but it is not put on the stack of the best solutions (dotted line).

As it was mentioned earlier, the framework proposed by Merelo et al. used AJAX technology to maintain client-server communication, what caused the server to be overloaded when too many users were connected. Jenkins used asynchronous AJAX that did not require permanent communication with the server, thus his system scaled more smoothly. The system proposed in this paper uses another, more efficient technology, so called push technology, which can be perceived as a reverse of AJAX. The data is pushed from the server to the clients even if the browser did not request it explicitly. This technology is not natively supported by web browsers, so it was necessary to implement it as AJAX long polling.

4 Computational Experiments

In order to evaluate the performance of the system the authors took a permutation flowshop scheduling problem (PFSP), one of the most studied optimization problems in the literature, and commonly used to test new heuristics. The most common objective function for this problem is to find the same order of n jobs on m machines that minimizes the completion time of the last job on the last machine, so called makespan. Finding the makespan for more than three machines was proved to be NP-complete and is equivalent to travelling salesmen problem (TSP) [7]. As there is $n!$ possible solutions the PFSP becomes really hard to solve, especially for instances with more than 100 jobs. Thus many heuristics have been proposed for solving this problem, including the ones based on metaheuristics like tabu search, genetic algorithms, simulated annealing, and recently also on swarm intelligence, such as ant colony optimization, particle swarm optimization or artificial bee colony. Unfortunately even in the most current publications the authors of the proposed algorithms usually performed their experiments with maximum 100 jobs as the computation time for larger instances of the problem sometimes exceeded 500 minutes [8].

To overcome this problem, parallel computing can be applied. Parallel methods for the PFSP based on metaheuristics presented in the literature usually utilize either many threads on a single computer or POSIX threads and MPI communication on a dedicated system [9][10].

In our distributed system the web browsers contributing in the calculations together in order to obtain the lowest possible value of makespan. As it was described previously, each client runs its own population and exchanges its best solutions with other clients through the server. The only tasks for the server is then to provide the instance of the problem to be solved by a client, and to receive the best solutions and further broadcast them to other clients.

In the preliminary experiments the clients computed a standard version of genetic algorithm (part of evolutionary algorithms family). An outline of this algorithm is presented in Fig. 2.

```
Initialise P = {$s_1$, $s_2$, … $s_{pop\_size}$}
Repeat
   P' ← Selection(P)
   For i=1 To pop_size Step 2
    If rand<p_crs {$s'_i$,$s'_{i+1}$} ← Crossover($s'_i$,$s'_{i+1}$)
    If rand<p_mut $s'_i$ ← Mutation($s'_i$)
   Next i
Until stop_condition = true
```

Fig. 2. An outline of genetic algorithm executed by the clients

One exception from the standard genetic algorithm was the application of a OX2 crossover instead of standard one- or mutli-point crossovers. The OX2 crossover was proposed by Syswerda [11] and it was found to be quite efficient for the sequential representation of solutions. As a mutation operator, a simple insertion mutation was used. Population size was set to 100 individuals and 250 generations were computed by each client.

The computational experiments with the makespan minimization for the flowshop scheduling problem have been conducted for the instances with 50, 100 and 200 jobs and constant number of 10 machines. The problems have been chosen from the standard Taillard's benchmark set [12]. The results of 20 independent runs with different number of clients in the form of average increase over the best-known solution for a given instance published on Taillard's website are provided in Fig. 3.

Contrary to the expectations, quality of the solutions achieved for the problems with smaller number of jobs was worse than for the problems with higher number of jobs. This is due to the fact that our system intentionally utilized only standard genetic algorithm without any problem-specific local improvement dedicated to the PFSP, as the system was intended to be universal and problem-independent. Special local search algorithms are usually very effective for small problems, as they can found optimal solutions in a relatively small solution space.

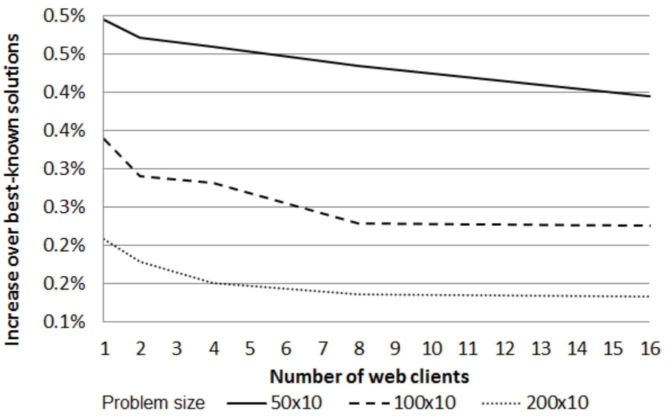


Fig. 3. Average increase over best-known solutions for the flowshop problem for different instances and different number of clients

Regarding the scalability of the system, it scaled quite well for the relatively small number of web clients (up to 8-10), while the gain from larger number of web clients, especially for the problems with 100 and 200 jobs, was not so impressive.

To further improve the quality of solutions a local search algorithm based on variable neighbourhood search (VNS) method has been applied. VNS was proposed by Mladenovic and Hansen [13] as a metaheuristic for solving combinatorial and global optimisation problems, but it often successfully used as a local search algorithm for other metaheuristics in hybrid systems [14]. The overall schema of VNS algorithm used by some system's clients is shown in Fig. 4.

```
Initialise s*
Repeat
  s' ← RandomSolution($N_k$(s*))
  s*' ← LocalSearch(s')
  If f(s*') < f(s*) Then
    s* ← s*';  k ← 1
  Else
    k ← k + 1
Until stop_condition = true
```

Fig. 4. An outline of variable neighbourhood search algorithm

The main idea of VNS is a systematic change of neighbourhood within a local search. VNS changes the neighbourhood N_k of the current solution $s*$ in two stages. Firstly, to find a local optimum, and then to change the neighbourhood (perturbation stage), if no further improvement can be achieved within the current neighbourhood.

In the proposed distributed system some clients receive a code that executes variable neighbourhood search algorithm instead of evolutionary algorithm. In order to evaluate the impact of local search on the quality of solutions in the next phase of the experiments one client always executed VNS algorithm. The role of this client was to improve the solutions that were placed on the stack of the best solutions found by genetic algorithm. Fig. 5 presents the results of the experiments with VNS client.

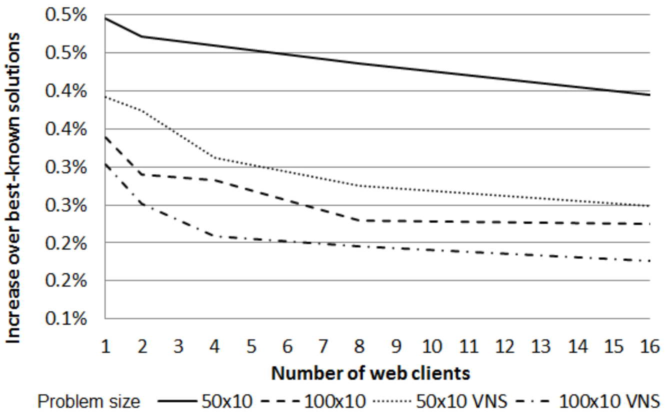


Fig. 5. Comparison of the distributed evolutionary system with and without a local search algorithm (VNS) for different number of clients

As it was expected, the application of VNS algorithm in the system allowed to improve the solutions obtained by genetic algorithms, but it also caused that the system scaled a little bit better. It is worth to notice that VNS local search still does not exploit any specific future of PFSP, so the system remains universal. These experiments showed primarily that the introduction of other metaheuristics to the system may bring significant advantages.

5 Conclusions and Future Work

Distributed computing based on JavaScript is cheap, relatively easy to implement, while simultaneously may be quite efficient. The code embedded in a web page is precompiled by contemporary web browser engines so it is executed almost as fast as the code written in other languages. The main advantage of the proposed system over the popular distributed systems like BOINC is that the participants do not have to install any additional applications in order to join some computational project. For these reasons the authors believe that web-based distributed systems will become more and more popular in the near future.

The system proposed by the authors requires further testing, especially with larger amount of clients distributed all over the Internet and to solve problems other than the flowshop problem. Genetic algorithm used in the system as a main solving method should also be improved. Other operators will be introduced as well as some mechanism for auto-adjustment of the algorithm parameters will be developed (currently all parameters are fixed). Additionally, some other local search algorithms like iterative local search (ILS) will be introduced to the system as well for further improvement of solutions. After the necessary improvements the proposed system is planned to be published for public use.

Acknowledgements. This work was partly financed within the statutory activity no 11/11.200.232.

References

1. Talbi, E.-G.: Parallel combinatorial optimization. John Wiley and Sons (2006)
2. TOP500 Supercomputer (June 2012), `http://www.top500.org/` (accessed October 10, 2012)
3. BOINC website, `http://boinc.berkeley.edu/` (accessed October 10, 2012)
4. Merelo, J.J., García, A.M., Laredo, J.L.J., Lupión, J., Tricas, F.: Browser-based distributed evolutionary computation: performance and scaling behaviour. In: Proceedings of the 2007 GECCO, pp. 2851–2858. ACM, New York (2007)
5. Jenkin, N.: Parasitic JavaScript, COMP520-08 Report, University of Waikato, New Zealand (2008)
6. Barabási, A.-L., Freeh, V.W., Jeong, H., Brockman, J.: Parasitic computing. Nature 412, 894–897 (2001)
7. Pensini, M.P., Mauri, G., Gardin, F.: Flowshop and TSP. In: Mündemann, F.W., Becker, J.D., Eisele, I. (eds.) Parallelism, Learning, Evolution. LNCS, vol. 565, pp. 157–182. Springer, Heidelberg (1991)
8. Ponnambalam, S.G., Jawahar, N., Chandrasekaran, S.: Discrete Particle Swarm Optimization Algorithm for Flowshop Scheduling. In: Lazinica, A. (ed.) Particle Swarm Optimization. InTech (2009)
9. Kouki, S., Ladhari, T., Jemni, M.: A Parallel Distributed Algorithm for the Permutation Flow Shop Scheduling Problem. In: Hsu, C.-H., Yang, L.T., Park, J.H., Yeo, S.-S. (eds.) ICA3PP 2010, Part I. LNCS, vol. 6082, pp. 328–337. Springer, Heidelberg (2010)
10. Cung, V.D., Martins, S.L., Ribeiro, C.C., Roucairol, C.: Strategies for the parallel implementation of metaheuristics, Essays and Surveys in Metaheuristics, pp. 263–308. Kluwer Academic Publishers (2002)
11. Syswerda, G.: Schedule optimization using genetic algorithms. In: Davis, L. (ed.) Handbook of Genetic Algorithms, pp. 332–349. Van Nostrand Reinhol, New York (1991)
12. Taillard, E.D.: Benchmarks for basic scheduling problems. European Journal of Operational Research 64, 278–285 (1993)
13. Mladenovic, N., Hansen, P.: Variable neighborhood search. Computers and Operations Research 24(11), 1097–1100 (1997)
14. Zobolas, G.I., Tarantilis, C.D., Ioannou, G.: Minimizing makespan in permutation flow shop scheduling problems using a hybrid metaheuristic algorithm. Comput. Oper. Res. 36(4), 1249–1267 (2009)
15. `http://nodejs.org/` (accessed November 20, 2012)
16. Tilkov, S., Vinoski, S.: Node.js: Using JavaScript to Build High-Performance Network Programs. IEEE Internet Computing, 80–83 (2010)
17. Introducing Socket.IO, `http://socket.io/` (accessed November 20, 2012)
18. Scaling node.js to 100k concurrent connections, `http://blog.caustik.com/2012/04/08/scaling-node-js-to-100k-concurrent-connections/` (accessed November 20, 2012)
19. Multi-Process Node.js: Motivations, Challenges and Solutions, `http://www.infoq.com/articles/multi-core-node-js/` (accessed November 20, 2012)
20. V8 JavaScript Engine, `http://code.google.com/p/v8/` (accessed November 20, 2012)
21. `http://shootout.alioth.debian.org/` (accessed November 20, 2012)

Use of Direct Solvers in TFETI Massively Parallel Implementation

Václav Hapla[1,2], David Horák[1,2], and Michal Merta[1,2]

[1] Dep. of Applied Math., VSB-Tech. Univ. of Ostrava, 17. listopadu 15, CZ 70833 Ostrava, Czech Republic
vaclav.hapla@vsb.cz

[2] Centre of Excellence IT4Innovations, VSB-Tech. Univ. of Ostrava, 17. listopadu 15, CZ 70833 Ostrava, Czech Republic

Abstract. The FETI methods blend iterative and direct solvers. The dual problem is solved iteratively using e.g. CG method; in each iteration, the auxiliary problems related to the application of an unassembled system matrix (subdomain problems' solutions and projector application in dual operator) are solved directly. The paper deals with the comparison of the direct solvers available in PETSc on the Cray XE6 machine HECToR (PETSc, MUMPS, SuperLU) regarding their performance in the two most time consuming actions in TFETI – the pseudoinverse application and the coarse problem solution. For the numerical experiments, our novel TFETI implementation in FLLOP (FETI Light Layer on top of PETSc) library was used.

Keywords: Domain decomposition, FETI, Total FETI, TFETI, FLLOP, PETSc, parallel direct solver, pseudoinverse, natural coarse space matrix, coarse problem.

1 Introduction

The class of methods called FETI (Finite Element Tearing and Interconnecting) turned out to be one of the most successful for parallel solution of elliptic partial differential equations arising from many engineering problems. They blend iterative and direct solvers. The dual problem is solved iteratively using e.g. CG method; in each iteration, the auxiliary problems related to the application of an unassembled system matrix (subdomain problems' solutions and projector application in dual operator) are solved directly.

The first auxiliary problem is the stiffness matrix's pseudoinverse application. It is parallelizable without any data transfers because of a nice block-diagonal structure, each core then factorizes the regularized local block (subdomain stiffness matrix). The second one is the coarse problem (CP) solution appearing in the application of the projector onto the kernel of so called natural coarse space matrix. However, this problem does not possess such a nice structure suitable for parallel processing; some communication is needed in this case.

Natural effort using the massively parallel computers is to maximize the number of subdomains so that sizes of subdomain stiffness matrices are reduced which accelerates their factorization and subsequent forward/back solves carrying out the pseudoinverse applications. On the other hand, negative effect of that is an increase of the null space dimension and the number of Lagrange multipliers on the subdomains' interfaces (dual dimension), which decelerate the CP solution. It can hardly be solved sequentially on the master core for large scale problems so that it becomes a bottleneck.

In this paper, we compare an effect of a choice of the LU direct solver from a set available in PETSc on HECToR (PETSc, MUMPS, SuperLU) on performance of the TFETI massively parallel implementation in our FLLOP library (FETI Light Layer on top of PETSc) [15]. As a benchmark, a model 3D linear elasticity problem was chosen. The best choice was then used to solve an engineering problem. The LU factorization was used because of objectivity to the compared libraries (SuperLU is not able to compute Cholesky factorization) and also due to better observed performance.

2 TFETI

The FETI-1 method is based on the decomposition of the spatial domain into non-overlapping subdomains that are "glued" by Lagrange multipliers, enforcing arising equality constraints by special projectors. The original FETI-1 method assumes that the boundary subdomains inherit the Dirichlet conditions from the original problem, so that the dimensions of subdomains' stiffness matrices' kernels may vary from zero (corresponding to the boundary subdomains with sufficient Dirichlet data to fix rigid body motions) to a certain positive maximum (corresponding to the floating subdomains).

The basic idea of Total-FETI (TFETI) [2] is to keep all the subdomains floating and enforce the Dirichlet boundary conditions by means of the matrix of constraints and the Lagrange multipliers, similarly to the gluing conditions along the subdomains' interfaces. Use of this concept simplifies implementation of the subdomain stiffness matrix pseudoinverse needed in FETI methods. The key point is that the kernels of local stiffness matrices are known a priori, have the same dimension, can be formed directly in parallel, and enable effective regularization of the subdomain stiffness matrices [3].

Let us consider a partitioning of a global domain Ω into N_s subdomains Ω^s and denote by K^s the subdomain stiffness matrix and by R^s a matrix whose columns span the kernel of K^s. Let B^s be a matrix with values $-1, 0, 1$ describing the gluing of the subdomains and

$$K = \begin{bmatrix} K^1 & & \\ & \ddots & \\ & & K^{N_s} \end{bmatrix}, \quad R = \begin{bmatrix} R^1 & & \\ & \ddots & \\ & & R^{N_s} \end{bmatrix}, \quad B = [B^1, \ldots, B^{N_s}]. \tag{1}$$

Let N_p denote the primal dimension, N_d the dual dimension, N_n the null space dimension and N_c the number of cores being at disposal for our computation.

Let us apply the duality theory to the primal problem

$$\min \frac{1}{2}u^T K u - u^T f \quad \text{s.t.} \quad Bu = 0 \tag{2}$$

and let us establish the following notation

$$F = BK^\dagger B^T,\ G = R^T B^T,\ d = BK^\dagger f,\ e = R^T f,$$

where $K^\dagger$ denotes a generalized inverse matrix satisfying $KK^\dagger K = K$, G the natural coarse space matrix. We obtain a new minimization problem

$$\min \frac{1}{2}\lambda^T F\lambda - \lambda^T d \quad \text{s.t.} \quad G\lambda = e. \tag{3}$$

Further the equality constraints $G\lambda = e$ can be homogenized to $G\lambda = 0$ by splitting λ into $\mu + \tilde{\lambda}$ where $\tilde{\lambda}$ satisfies $G\tilde{\lambda} = e$ (e.g. $\tilde{\lambda} = G^T(GG^T)^{-1}e$) which implies $\mu \in \operatorname{Ker} G$. We then substite $\lambda = \mu + \tilde{\lambda}$, omit terms without μ, minimize over μ and add $\tilde{\lambda}$ to μ.

Finally, the equality constraints $G\lambda = 0$ can be enforced by the projector $P = I - Q$ onto the null space of G, where $Q = G^T(GG^T)^{-1}G$ is the projector onto the image space of G^T ($\operatorname{Im} Q = \operatorname{Im} G^T$ and $\operatorname{Im} P = \operatorname{Ker} G$). It holds that $P\mu = \mu$ because $\mu \in \operatorname{Ker} G$, so the final problem reads

$$PF\mu = P(d - F\tilde{\lambda}). \tag{4}$$

The problem (4) may be solved effectively by the conjugate gradient method thanks to the classical estimate by Farhat, Mandel and Roux of the spectral condition number:

$$\kappa(PFP|\text{Im}P) \le C\frac{H}{h}$$

where h is the discretization parameter and H is the decomposition parameter. This estimate remains valid for TFETI.

3 Direct Solvers Available in PETSc

PETSc (Portable, Extensible Toolkit for Scientific Computation) [10] is a suite of data structures and routines for the parallel solution of scientific applications modeled by partial differential equations. It supports MPI, shared memory pthreads, and NVIDIA GPUs, as well as hybrid MPI-shared memory pthreads or MPI-GPU parallelism. PETSc includes an expanding suite of parallel linear, nonlinear equation solvers and time integrators that may be used in application codes written in Fortran, C, C++, Python, and MATLAB (sequential). PETSc provides many of the mechanisms needed within parallel application codes, such as parallel matrix and vector assembly routines. The library is organized hierarchically, enabling users to employ the level of abstraction that is most appropriate for a particular problem. By using techniques of object-oriented programming, PETSc provides enormous flexibility for users. PETSc contains built-in direct solvers for in-place, symbolic, numeric LU and Cholesky factorizations of matrix. It also supports many external packages; we will be particularly interested in direct solvers MUMPS and SuperLU that are shipped with PETSc on Cray [14].

MUMPS (MUltifrontal Massively Parallel sparse direct Solver) [11] is a package for solving systems of linear equations with square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric. MUMPS employs a multifrontal method for LU and LDL^T factorization [9]. MUMPS exploits both parallelism arising from sparsity in the system matrix and from dense factorizations kernels. The main features of the MUMPS package include the solution of the transposed system, input of the matrix in assembled format (distributed or centralized) or elemental format, error analysis, iterative refinement, scaling of the original matrix, out-of-core capability, parallel analysis, detection of null pivots, basic estimate of rank deficiency and null space basis for symmetric matrices, and computation of a Schur complement matrix. MUMPS offers several built-in ordering algorithms and a tight interface to some external ordering packages. MUMPS is available in various arithmetics (real or complex, single or double precision). The software is mainly written in Fortran 90 although a C interface is available. The parallel version of MUMPS requires MPI, BLAS, BLACS, and ScaLAPACK libraries.

SuperLU [12] is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on high-end computers, based on the supernodal LU factorization method. The library is written in C and is callable from either C or Fortran. The library routines perform an LU decomposition with numerical pivoting and triangular system solves through forward and backward substitution. The LU factorization routines can handle non-square matrices but the triangular solves are performed only for square matrices. The matrix columns may be preordered. Working precision iterative refinement subroutines are provided for improved backward stability. Routines are also provided to equilibrate the system, estimate the condition number, calculate the relative backward error, and estimate error bounds for the refined solutions. The routines have been carefully designed for optimal performance in solving large systems on modern computer architectures. The factorization algorithm uses a graph reduction technique to reduce graph traversal time in the symbolic analysis, and data movement between levels of the memory hierarchy is reduced through loop ordering and the use of dense matrix operations in the numerical kernel. For the distributed memory implementation, a two-dimensional block cyclic matrix distribution is used to enhance scalability. SuperLU contains a collection of three related subroutine libraries: sequential SuperLU for uniprocessors, the multithreaded version for medium-size SMPs, and the MPI version for large distributed memory machines.

4 Parallel Implementation and Numerical Experiments

Parallelization of FETI/TFETI can be implemented mostly using SPMD technique – distributing matrix portions among processing units. This allows algorithms to be almost the same for sequential and parallel case; only data structure implementation differs. Distribution of primal matrices is quite straightforward as every subblock reflects a subdomain. K and R possess nice block-diagonal layout and can be implemented using a block-diagonal matrix composite type where subblocks are ordinary sequential matrices and every core holds an array of subblocks associated with its subdomains. Let us specify the current core by $rank$ ($rank = 0, ..., N_c - 1$) and its associated X-object's portion by $X_{[rank]}$.

As it was mentioned, natural effort using the massively parallel computers is to maximize number of subdomains so that sizes of subdomain stiffness matrices are reduced; this accelerates their factorization and subsequent pseudoinverse application. Negative effect of that is an increase of dual and null space dimension, which decelerate the CP solution. In the following two subsections we discuss and compare various approaches for these crucial phases realized in our FLLOP (FETI Light Layer on top of PETSc) library implementing the TFETI method [15].

Matrices and vectors for numerical experiments were obtained from a regular decomposition and discretization of a model problem of an elastic cube with edge length of 1 mm, Young modulus 2.0e+5 MPa and Poisson ratio 0.3 (see Figure 9). Dirichlet boundary conditions are prescribed on three sides (for each side there are zero displacements in normal direction). On one of the free sides there is a Neumann condition with pressure 10 MPa in the normal direction. To illustrate the efficiency of various direct solvers we used a discretization of the cube into 4,096,000 elements and a decomposition into 512; 1,000; 4,096; 8,000 subdomains. The patterns of matrices to be factorized are for the decomposition into 8 subdomains illustrated in Figure 1 and Figure 2 (not to scale). Let us mention here that the regular discretization and decomposition was chosen for the detailed analysis to ensure an equal workload of all cores.

The numerical experiments were run on HECToR supercomputer [13] operated by EPCC. It's the UK's front-line national supercomputing service. For our experiments, we used the current Phase 3 system (Cray XE6). Regarding supplied software, the Cray compiler suite and Cray-supplied optimized PETSc 3.2-p7 (`petsc/3.2.01` module) together with the TPSL (Third Party Scientific Libraries) (`tpsl/1.2.01` module) were used. The Cray's TPSL suite provides optimized versions of numerical libraries like MUMPS, SuperLU(_DIST), (Par)METIS, HYPRE and other. The solvers in Cray PETSc are heavily optimized using the Cray Adaptive Sparse Kernels (CASK) library. CASK is an auto-tuned library within the Cray PETSc package that is transparent to the application developer, but improves the performance of most PETSc iterative solvers.

About hardware, the Phase 3 system is contained in 30 cabinets and comprise of a total of 704 compute blades. Each blade contains four compute nodes giving a total of 2816 compute nodes, each with two 16-core AMD Opteron 2.3GHz Interlagos processors. This amounts to a total of 90,112 cores. Each 16-core socket is coupled with a Cray Gemini routing and communications chip. Each 16-core processor shares 16 GB of memory. The theoretical peak performance of the phase 3 system is over 800 Tflops.

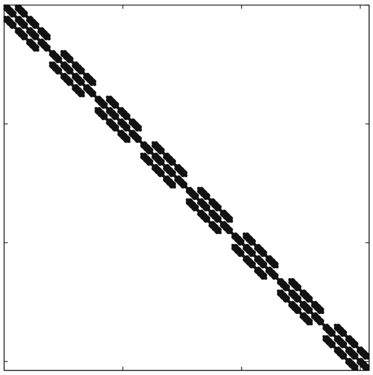


Fig. 1. Pattern of K^{reg} matrix

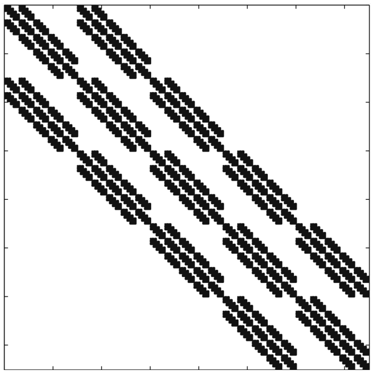


Fig. 2. Pattern of GG^T matrix

4.1 Pseudoinverse Action

In the case of the pseudoinverse, each core regularizes subdomain stiffness matrix $K_{[rank]}$ using fixing nodes [3,4,5] and factorizes it in the preprocessing phase

$$factorize(K^{reg}_{[rank]}) -> L_{K,[rank]}, U_{K,[rank]}.$$

The application of $K^\dagger$, i.e. the matrix-vector multiplication $K^\dagger v$ then consists of purely local backward and forward substitutions once in each CG iteration

$$K^\dagger_{[rank]} v_{[rank]} = U_{K,[rank]} \backslash (L_{K,[rank]} \backslash v_{[rank]}).$$

The time for the preprocessing and the average time for one application for PETSc, MUMPS and SuperLU are shown in Table 1 and graphically illustrated in Figures 3 and 4 (log. scale); there is a nice time reduction due to the decomposition of the domain into more and more subdomains, decreasing the local matrix dimension. The best results were obtained with MUMPS library; SuperLU was for the factorization of the problem with the largest sudomain dimension 10 times worse and PETSc built-in LU even 25 times worse. It is obvious that the multifrontal approach is very suitable for the subdomain stiffness matrix's structure.

4.2 Coarse Problem Solution

The natural coarse space matrix G is computed in a way where each of cores owns sparse sequential matrices $R_{[rank]}$ and $B_{[rank]}$, so that this core computes local block $G_{[rank]} = R^T_{[rank]} B^T_{[rank]}$ of G matrix without any communication. We then redistribute horizontal sequential sparse blocks $G_{[rank]}$ into vertical ones (i.e. horizontal $G^T_{[rank]}$).

According to the observations, the actions Gw, $G^T w$ take approximately the same time for different G matrix distributions (assembled G distributed into

Table 1. Performance of K^{reg} factorization / $K^{\dagger}$ action / factorization + all actions for varying decompositions in sec

	$N_s = N_c$	512	1,000	4,096	8,000
	N_p/N_c	27,783	14,739	3,993	2,187
solver	iters.	30	26	18	15
PETSc	K^{reg} fact.	2.03e+02	7.44e+01	3.35e+00	8.06e-01
	$K^{\dagger}$ act.	2.64e-01	1.17e-01	1.90e-02	9.05e-03
	fact. + all act.	210.8	77.4	3.7	0.9
MUMPS	K^{reg} fact.	7.93e+00	3.05e+00	5.72e-01	2.11e-01
	$K^{\dagger}$ act.	1.50e-01	6.34e-02	1.76e-02	9.11e-03
	fact. + all act.	12.4	4.7	0.9	0.3
SuperLU	K^{reg} fact.	8.55e+01	1.88e+01	1.49e+00	5.32e-01
	$K^{\dagger}$ act.	6.38e-01	2.11e-01	3.04e-02	1.35e-02
	fact. + all act.	104.6	24.3	2.0	0.7

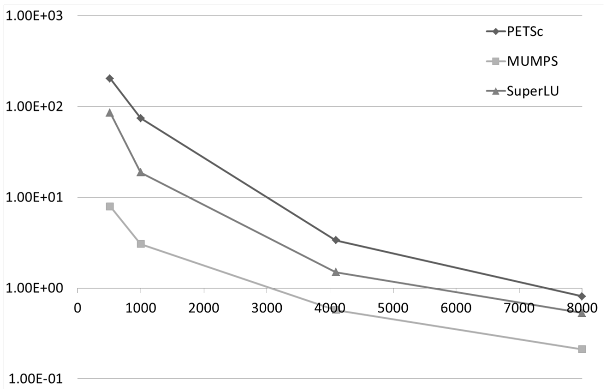


Fig. 3. Times for K^{reg} factorization (log. scale)

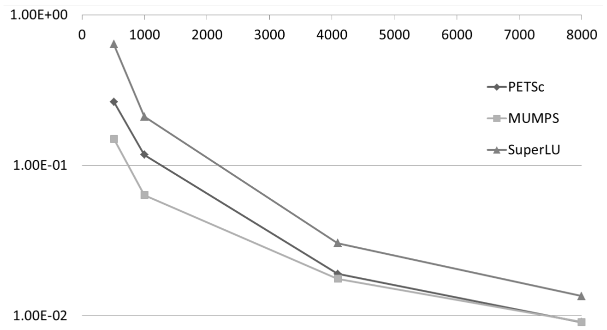


Fig. 4. Times for $K^{\dagger}$ action (log. scale)

horizontal blocks, assembled G distributed into vertical blocks, unassembled G kept in the form $R^T B^T$). So the action time and level of communication depend primarily on the implementation of the CP solution

$$GG^T x = y,$$

which can be hardly solved sequentially on the master core for large scale problems.

We have suggested and compared several strategies of CP solution (iteratively using PCG, directly using LU factorization, applying explicit inverse of GG^T, orthonormalizing rows of G so that the CP is eliminated) in [1,8]. Here we compare the use of sequential and parallel direct solvers using the LU factorization.

The groups of cores (so called subcommunicators) arise from splitting all cores in the global "world" communicator using PETSc built-in "pseudopreconditioner" PCREDUNDANT; the number of these subcommunicators is N_r (number of cores doing redundant work), i.e. the number of cores in each subcommunicator is equal to N_c/N_r. We have to transfer whole G matrix to the zeroth core or to all subcommunicators. Master core or subcommunicators' cores then compute product GG^T using matrix-matrix multiplication. This CP matrix is factorized

$$factorize(GG^T) -> L_{GG^T}, U_{GG^T}$$

sequentially on master core (by in PETSc built-in LU) or in parallel using MUMPS [11] or SuperLU_DIST [12] in subcommunicators. The application of $(GG^T)^{-1}$, i.e. the matrix-vector multiplication $(GG^T)^{-1}w$ then consists of backward and forward substitutions, which are not local and not negligible amount of the communication is needed once in each CG iteration

$$(GG^T)^{-1}w = U_{GG^T} \backslash (L_{GG^T} \backslash w).$$

Parallel approach has a big advantage consisting in the reduction of memory requirements for the CP solution; there are practically no memory limits as more and more cores can be engaged into the subcommunicators.

The performance results of sequential direct solvers for varying decomposition are depicted in Table 2 and illustrated in Figures 5 and 6 and the performance of parallel direct solvers for the decomposition into 8,000 subdomains in Table 3 and in Figures 7 and 8 (log. scale).

Figures 3, 4, 5 and 6 illustrate the so called communicating vessels effect: the computational savings for K^{reg} factorization and $K^\dagger$ action reached by the decomposition into more subdomains are eliminated by an increasing computational and communication requirements for the CP solution – GG^T factorization and $(GG^T)^{-1}$ action. Decomposition into more subdomains for the fixed discretization leads to the reduction of the condition number of the dual operator and the number of iterations, but on the other hand it increases the CP dimension, so that there is no doubt about the parallel solution of the CP.

However, the CP dimension is not large enough to justify the fully parallel approach; in this case, communication takes over computation. The significant

Table 2. Performance of sequential GG^T factorization / $(GG^T)^{-1}$ action / factorization + all actions on the master core for varying decompositions in sec

	$N_s = N_c$	512	1,000	4,096	8,000
solver	N_n	3,072	6,000	24,576	48,000
PETSc	GG^T fact.	1.07e+00	4.55e+00	8.50e+01	3.37e+02
	$(GG^T)^{-1}$ act.	1.28e-02	3.24e-02	2.36e-01	6.21e-01
	fact. + all act.	1.5	5.4	89.2	346.7
MUMPS	GG^T fact.	2.73e-01	7.34e-01	7.94e+00	2.65e+01
	$(GG^T)^{-1}$ act.	1.12e-02	2.32e-02	1.34e-01	2.99e-01
	fact. + all act.	0.6	1.3	10.4	31.0
SuperLU	GG^T fact.	1.14e+00	3.67e+00	4.28e+01	1.78e+02
	$(GG^T)^{-1}$ act.	4.49e-02	1.24e-01	8.63e-01	2.42e+00
	fact. + all act.	2.5	6.9	58.4	214.2

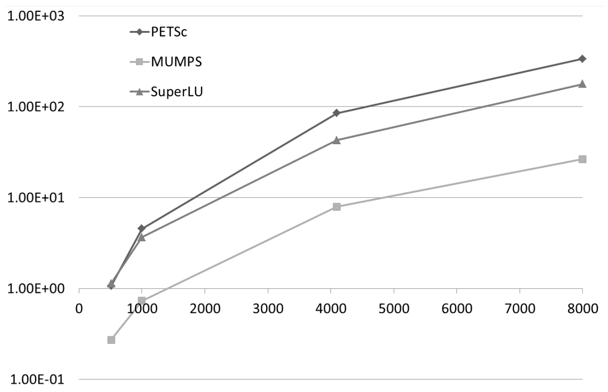


Fig. 5. Times for GG^T factorization in seq. case (log. scale)

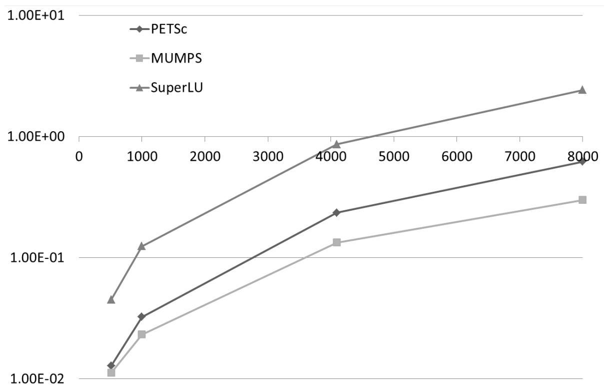


Fig. 6. Times for $(GG^T)^{-1}$ action in seq. case (log. scale)

Table 3. Performance of parallel GG^T factorization / $(GG^T)^{-1}$ action / factorization + all actions depending on the subcommunicator's size for the decomposition into 8,000 subdomains in sec

	N_r	2,000	1,000	500	20	16	10
solver	N_c/N_r	4	8	16	400	500	800
MUMPS	GG^T fact.	2.33e+01	1.54e+01	1.61e+01	1.18e+01	1.10e+01	1.22e+01
	$(GG^T)^{-1}$ act.	9.03e-01	4.07e-01	2.60e-01	1.36e-01	1.48e-01	1.75e-01
	fact. + all act.	36.9	21.5	20.0	13.8	13.2	14.8
SuperLU	GG^T fact.	2.94e+01	2.04e+01	1.62e+01	6.38e+00	6.73e+00	failed
	$(GG^T)^{-1}$ act.	1.03e+00	4.99e-01	4.07e-01	1.44e-01	8.35e-02	failed
	fact. + all act.	44.9	27.8	22.3	8.5	8.0	failed

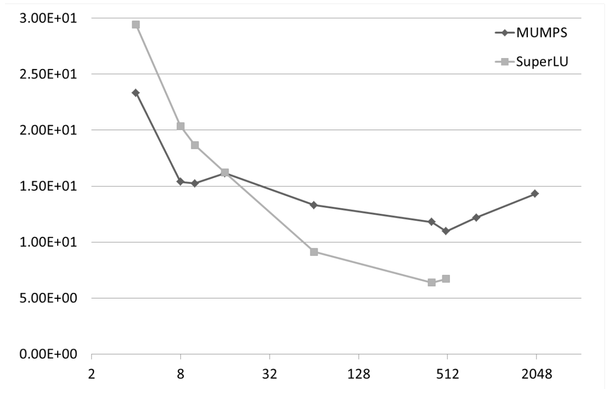


Fig. 7. Times for GG^T factorization in parallel depending on the subcommunicator's size for the decomposition into 8,000 subd

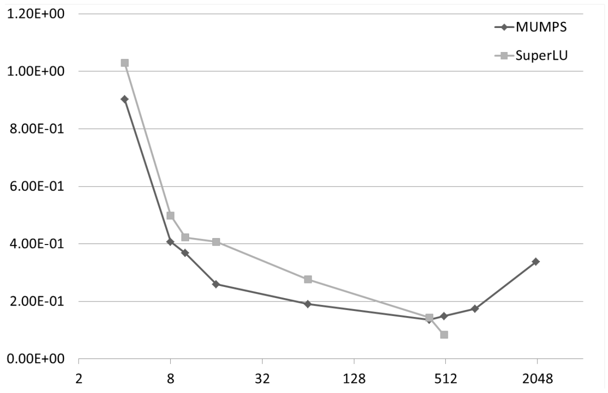


Fig. 8. Times for $(GG^T)^{-1}$ action in parallel depending on the subcommunicator's size for the decomposition into 8,000 subd

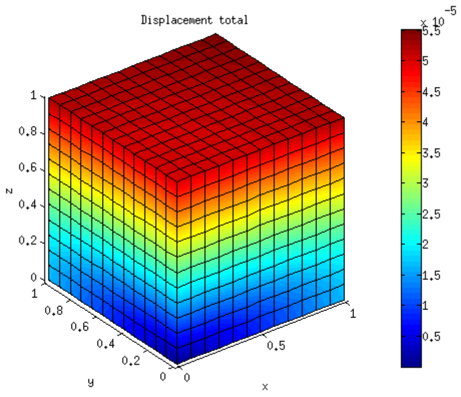


Fig. 9. Model cube benchmark

Table 4. Total performance of FLLOP TFETI with the best direct solvers for the cube problem in sec, N_p=17,496,000; N_d=5,053,920; N_n=48,000; $N_c = N_s$=8,000; CG iterations: 15, MUMPS/SuperLU(N_r=16)

K^{reg} fact.	all $K^\dagger$ act.	GG^T fact.	all $(GG^T)^{-1}$ act.	total prep.	total sol.
0.2	0.1	6.7	1.25	11.4	1.9

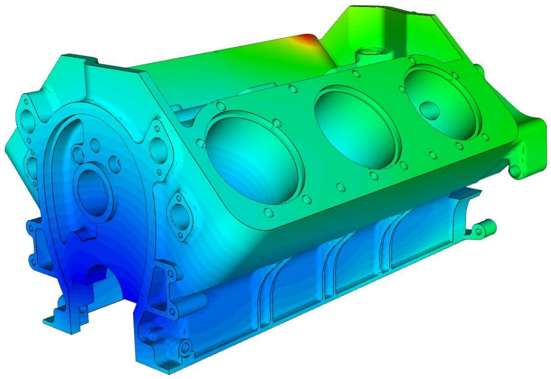


Fig. 10. Engine benchmark

Table 5. Total performance of FLLOP TFETI with the best direct solvers for the engine problem in sec, N_p=98,214,558; N_d=13,395,882; N_n=30,072; $N_c = N_s$=5,012; CG iterations: 181, MUMPS/SuperLU(N_r=16)

K^{reg} fact.	all $K^\dagger$ act.	GG^T fact.	all $(GG^T)^{-1}$ act.	total prep.	total sol.
3.89	15.0	18.0	74.8	28.2	233.0

efficiency improvement can be achieved by means of the partial parallelization of this CP solution; see Table 3 and Figures 7 and 8. The optimal number of cores per subcommunicator for our problems is 500 which corresponds to N_r=16.

5 Results for the Best Choice of Solvers

Previous sections have shown that the best combination of direct solvers for TFETI is following. Regarding the direct solver realizing actions of the stiffness matrix pseudoinverse $K^\dagger$, which is a fully local problem, MUMPS appeared to be the best from the PETSc / MUMPS / SuperLU trio; the multifrontal method employed in MUMPS is really very efficient for the matrices arising from the FEM discretizations. For the CP, parallel solution in subcommunicators should always be considered, and the supernodal approach represented by the SuperLU_DIST library proved to be more suitable than the multifrontal approach.

To support these conclusions, we ran two benchmarks, now using the above-mentioned combined approach. MUMPS was used for the pseudoinverse, and for the CP SuperLU_DIST on 500 cores (N_r=16). The first benchmark was the cube case from previous sections, decomposed into 8,000 subdomains; the second one was an engineering problem of a car engine block (see Figure 10), decomposed into 5,012 subdomains by METIS. In Tables 4, 5, we report the computational times for K^{reg} factorization, all $K^\dagger$ actions, GG^T factorization, all $(GG^T)^{-1}$ actions, total preprocessing time and total solution time for the best direct solvers.

6 Conclusions

Without CP parallelization we are not able to solve large problems because the whole CP resides in the master process' memory which is of course limited. Furhermore, the master performs the sequential computation while all other cores have to wait; this breaks the scalability of the whole method. So there is no other way than using some parallel direct solver. On the other hand, engaging all processes in "world" communicator in the CP solution leads to an enormous communication overhead. Therefore, we conclude that the CP should be solved only in the subcommunicators of appropriate size. For the used Cray XE6 architecture, its vendor supplied libraries and our PETSc-based implementation (FLLOP [15]) we recommend to use MUMPS for the pseudoinverse application and SuperLU_DIST for the coarse problem solution in parallel on 500 cores (N_r=16).

Acknowledgements. Special thanks to our colleagues Alexandros Markopoulos and Tomáš Brzobohatý who generated data for numerical experiments, and also to Elmer team at CSC in Helsinki for great inspiring conditions during our research stay.

This publication was supported by the Projects of major infrastructures for research, development and innovation of Ministry of Education, Youth and Sports

(LM2011033), the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by the project SPOMECH - Creating a multidisciplinary R&D team for reliable solution of mechanical problems (CZ.1.07/2.3.00/20.0070) within Operational Programme 'Education for competitiveness' funded by Structural Funds of the European Union and state budget of the Czech Republic and by the project Cooperation for Future (CZ.1.07/2.4.00/31.0035).

The research has also been supported by the grants: HPC-Europa2 project funded by the European Commission - DG Research in the Seventh Framework Programme under grant agreement No. 228398, PRACE 2IP project receiving funding from the EU's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. RI-283493, by the Czech Science Foundation under grant agreement No. GACR 101/09/P601 and by the Ministry of Education, Youth and Sports under grant agreement No. MSM 6198910027.

References

1. Hapla, V., Horak, D.: TFETI Coarse Space Projectors Parallelization Strategies. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 152–162. Springer, Heidelberg (2012), doi:10.1007/978-3-642-31464-3_16
2. Dostál, Z., Horák, D., Kučera, R.: Total FETI – an easier implementable variant of the FETI method for numerical solution of elliptic PDE. Commun. in Numerical Methods in Engineering 22, 1155–1162 (2006)
3. Kozubek, T., Vondrák, V., Menšík, M., Horák, D., Dostál, Z., Hapla, V., Kabelíková, P., Čermák, M.: Total FETI domain decomposition method and its massively parallel implementation. Advances in Engineering Software (2011)
4. Brzobohatý, T., Dostál, Z., Kozubek, T., Kovář, P., Markopoulos, A.: Cholesky decomposition with fixing nodes to stable computation of a generalized inverse of the stiffness matrix of a floating structure. International Journal for Numerical Methods in Enginering 88(5), 493–509 (2011), doi:10.1002/nme.3187
5. Dostál, Z., Kozubek, T., Markopoulos, A., Menšík, M.: Cholesky decomposition of a positive semidefinite matrix with known kernel. Applied Mathematics and Computation 217(13), 6067–6077 (2011), doi:10.1016/j.amc.2010.12.069
6. Balay, S., Brown, J., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.2, Argonne National Laboratory (2011)
7. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: Efficient management of parallelism in object oriented numerical software libraries. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds.) Modern Software Tools in Scientific Computing, pp. 163–202. Birkhäuser Press (1997)
8. PRACE-2IP WP9 Deliverable: Support for Industrial Applications Year 1 (2012), https://bscw.zam.kfa-juelich.de/bscw/bscw.cgi/d799956/D9.1.1.pdf
9. Liu, J.: The multifrontal method for sparse matrix solution: theory and practice. SIAM Review 34, 82–109 (1992)
10. PETSc Web page, http://www.mcs.anl.gov/petsc/

11. MUMPS Web page, http://graal.ens-lyon.fr/MUMPS/
12. SuperLU Web page, http://acts.nersc.gov/superlu/
13. HECToR Web Page, http://www.hector.ac.uk/
14. Cray Application Developer's Environment User's Guide, http://docs.cray.com/books/S-2396-601/S-2396-601.pdf
15. FLLOP Web Page, http://spomech.vsb.cz/feti/

Parallel Implementation of the Sherman-Morrison Matrix Inverse Algorithm

Xin He, Marcus Holm, and Maya Neytcheva

Department of Information Technology, Uppsala University, Sweden
{he.xin,marcus.holm,maya.neytcheva}@it.uu.se

Abstract. We present two parallel strategies to compute the inverse of a dense matrix, based on the so-called Sherman-Morrison algorithm and demonstrate their efficiency in memory and runtime on multicore CPU and GPU-equipped computers. Our methods are shown to be much more efficient than the direct method to compute the inverse of a nonsingular dense matrix, yielding up to 12 times faster performance on the CPU.

1 Introduction

The task to compute explicitly the inverse of a given nonsingular matrix is among the heaviest computational kernels in matrix linear algebra. We consider a real square nonsingular matrix A of size $n \times n$ and pose the task to compute A^{-1}. We restrict ourselves to the class of dense matrices.

First, we briefly mention some well-known algorithms, and then we present our contribution. One straightforward approach to compute A^{-1} could be to determine its columns as solutions of the linear systems $LU\mathbf{x}_i = I_n$, where LU is the factorization of A and I_n is the identity matrix of size n. The so-obtained matrix $X = \{\mathbf{x}_i\}$, $i = 1, \cdots n$ is then the inverse of A. As is well-known, the computational cost to factorize A is $O(n^3)$ and each solution with L and U costs $O(n^2)$ operations. The total cost to compute A^{-1} is then $O(n^3)$. In this paper, the above approach is refereed to as the direct method to compute the inverse of a non-singular matrix. Another often used way to compute A^{-1} is via the Gauss-Jordan method. Its computational cost is also $O(n^3)$. Both of these algorithms are numerically unstable and require permutations. There exist parallel blocked versions of both the LU factorization with partial pivoting and the Gauss-Jordan method, e.g., [3,7,6]. The parallel performance of these algorithms is out of the scope of this paper.

As we aim at handling large problems, it becomes critical to achieve full utilization of the available complex computer hardware resources. It is well-known that, for instance, the solution of systems with triangular matrices is inherently serial and it is, therefore, not likely that such an algorithm could be efficiently implemented on multicore and GPU-equipped computers. Applying permutations is another matrix manipulation, which is not easily parallelizable.

Instead, we consider here an approach based on the so-called Sherman-Morrison (SM) algorithm, explained below. The major difference is that although the computational complexity of the SM algorithm is still $O(n^3)$, the algorithm can be written in a block form that uses only BLAS3 operations. Provided that we possess highly efficient BLAS operations, tuned for multicore architectures or GPUs, we can expect that for large enough problems, the SM implementation would make a much better utilization of all the computational resources and, thus, could outperform other methods for computing the inverse of a matrix, such as the above mentioned ones.

To apply SM, we present A in a particular form, namely, as

$$A = A_0 + XY^T \tag{1}$$

where $A_0 \in \mathbb{R}^{n\times n}$ is a matrix, whose inverse is easy to compute (e.g., A_0 could be diagonal, or even the identity matrix) and $X, Y \in \mathbb{R}^{n\times m}$.

Note, that any matrix can be represented in the form (1), for instance by taking A_0 to be the diagonal of A, $X = A - A_0$ and $Y = I_n$. Clearly, the representation of A in the form (1) is not unique. There are numerous application areas, however, where we need to compute the inverses of matrices, which arise directly in the form (1), as in some statistical problems, for example, related to seismic, genetic studies, in certain flow problems etc. Moreover, due to the treatment of large data sets, n can often be of order 10^6 or more, and m can be of order 10^4 to 10^6. In some applications X and Y can be sparse, however taking advantage of the sparsity falls out of the scope of this work.

In certain cases, as in some statistical applications, the inverse of A has to be computed explicitly and is needed in further analysis. In other cases, it suffices to compute only an approximation of A^{-1}, to be used as a high quality multiplicative preconditioner to A, when applying iterative solution methods in large scale scientific computations. Dropping relatively small-valued entries, related to some given tolerance, is the usual technique to obtain a sparse approximation of A^{-1}, however this is also left out of the scope of this paper. The interested reader is referred to [1].

The SM algorithm and a block version of it have been derived and studied earlier in [1,2,4]. We focus here on the parallel implementation of the SM algorithm on two computer architectures - shared memory multicore CPU and GPU. The paper is organized as follows. In Section 2 we present a recursive form (referred here to as the 'single vector' form) and two block forms of the SM algorithm. The implementation of the two block SM algorithms is presented in Section 3. Performance results for the CPU and GPU implementations are reported in Section 4 and some discussion points are found in Section 5.

2 The SM Algorithm

2.1 The Single Vector form of the SM Algorithm

Let A be of the form (1) and $I_m \in \mathbb{R}^{m\times m}$ be the identity matrix of size m. Provided that the matrix $I_m + Y^T A_0^{-1} X$ is nonsingular, Sherman-Morrison-Woodbury's formula gives an explicit form of $(A_0 + XY^T)^{-1}$, expressed as

$$(A_0 + XY^T)^{-1} = A_0^{-1} - A_0^{-1} X (I_m + Y^T A_0^{-1} X)^{-1} Y^T A_0^{-1}. \tag{2}$$

Applying formula (2) on the columns of X and Y, in [1,2] an algorithm is derived to compute A^{-1} in the following form

$$A^{-1} = A_0^{-1} - A_0^{-1} U R^{-1} V^T A_0^{-1}, \tag{3}$$

where $R \in \mathbb{R}^{m\times m}$ is a diagonal matrix and $U, V \in \mathbb{R}^{n\times m}$. The computational procedure and the matrices U, V and R are explicitly presented in the SVSM algorithm. We use Matlab-type vector notations and `IA, IA0` denote A^{-1} and A_0^{-1}, respectively.

Algorithm. SVSM-Single Vector SM

```
for k = 1:m
    U(:,k) = X(:,k)
    V(:,k) = Y(:,k)
    for l = 1:k-1
        U(:,k) = U(:,k) - (V(:,l)'*IA0*X(:,k)) * R(l,l) * U(:,l)
        V(:,k) = V(:,k) - (Y(:,k)'*IA0*U(:,l)) * R(l,l) * V(:,l)
    end
    R(k,k) = 1/(1+V(:,k)'*IA0*X(:,k))
end
IA = IA0 - IA0*U*R*V'*IA0
```

As we can see, SVSM consists of vector and matrix-vector (BLAS1, BLAS2) operations only, which are less efficient than BLAS3. A block implementation consisting of more efficient matrix-matrix operations can be expected to achieve better performance.

2.2 Block Version of the SM Algorithm

A block version of the SM algorithm has already been suggested in [4], similar to the method we now propose. Consider X and Y to be of block-column form. Thus, let $X = \{X_k\}_{k=1,\ldots,p}$, $Y = \{Y_k\}_{k=1,\ldots,p}$ and $X_k, Y_k \in \mathbb{R}^{n\times s_k}$ with $\sum_{k=1}^{p} s_k = m$. Then, clearly, there holds that

$$A = A_0 + \sum_{k=1}^{p} X_k Y_k^T. \tag{4}$$

Define $A_k = A_{k-1} + X_k Y_k^T$ and assume that the matrices $R_k = I_{s_k} + Y_k^T A_{k-1}^{-1} X_k$ are nonsingular for $k = 1, \ldots, p$. Applying formula (2), we express the inverses of the matrices A_k as follows:

$$A_k^{-1} = A_{k-1}^{-1} - A_{k-1}^{-1} X_k R_k^{-1} Y_k^T A_{k-1}^{-1},\ k = 1, \ldots, p. \tag{5}$$

Since $A_p^{-1} = A^{-1}$ then, applying formula (5) recursively, we obtain

$$A^{-1} = A_0^{-1} - \sum_{k=1}^{p} A_{k-1}^{-1} X_k R_k^{-1} Y_k^T A_{k-1}^{-1}. \tag{6}$$

Then, another sequence of factors $\{U_k, V_k\}_{k=1,\ldots,p} \in \mathbb{R}^{n\times s_k}$, where

$$U_k = X_k - \sum_{i=1}^{k-1} U_i R_i^{-1} V_i^T A_0^{-1} X_k, \tag{7}$$

$$V_k = Y_k - \sum_{i=1}^{k-1} V_i R_i^{-T} U_i^T A_0^{-T} Y_k, \tag{8}$$

are well defined. In addition, the following relations hold:

$$A_{k-1}^{-1} X_k = A_0^{-1} U_k, \quad Y_k^T A_{k-1}^{-1} = V_k^T A_0^{-1},$$

and

$$R_k = I_{s_k} + Y_k^T A_0^{-1} U_k = I_{s_k} + V_k^T A_0^{-1} X_k. \tag{9}$$

The above relations enable us to compute the factors U, V and R blockwise. Namely, for $U = [U_1, U_2, \ldots, U_p]$ and $V = [V_1, V_2, \ldots, V_p]$ with matrices U_k and V_k as block columns, the inverse of A can be rewritten as

$$A^{-1} = A_0^{-1} - A_0^{-1} U R^{-1} V^T A_0^{-1}, \tag{10}$$

where $R^{-1} = \mathrm{diag}(R_1^{-1}, R_2^{-1}, \ldots, R_p^{-1})$.

We see that in this case we have to invert matrix blocks R_k of certain sizes s_k. However, we can tune the block sizes in a suitable way so that the time to compute the block inverses does not dominate the overall run time. Clearly, the block size parameter s_k may vary between the different steps of the recursion. Without loss of generality, from now on we consider $s_k = s$ to be constant. Algorithm BSM presents the pseudo-code of the latter block algorithm.

The BSM algorithm requires four block-column matrices of size $n \times s$, P_k, Q_k, U_k, V_k and one square matrix $R0$ of size $s \times s$. For $s = 1$, BSM reduces to the single vector SM algorithm. Due to the same reason mentioned already, the BSM algorithm can be expected to be more efficient compared to the single vector SM algorithm especially for large matrices (see the numerical experiments in [4]). Here we choose to use the direct method to compute the inverse of $R0$. The description of this method is presented in Section 1. As mentioned already, the total cost to compute $R0^{-1}$ via the direct method is $O(s^3)$. The reason why we choose the direct method here is that in practice the size of $R0$ is relatively small and the efficiency of the direct method for small matrices is very acceptable. A detailed discussion about the choice of the block size s is presented in Section 2.4. The computational complexity of BSM is shown in Table 1.

Algorithm. BSM (Block SM)

```
p = m/s; % p - number of blocks, s - block size, Is - identity of size s
U(:,1:s) = X(:,1:s);  V(:,1:s) = Y(:,1:s)
R0 = Is + Y(:,1:s)' * IA0 * U(:,1:s); R(1:s,1:s) = inv(R0)
for k = 2:p
    X_{k} = X(:, (k-1)*s+1:k*s)
    Y_{k} = Y(:, (k-1)*s+1:k*s)
    W = IA0 * X_{k}
    P_{k}(1:(k-1)*s,:) = V(:,1:(k-1)*s)' * W
    Q_{k}(1:(k-1)*s,:) = R(1:(k-1)*s,1:(k-1)*s) * P_{k}
    U_{k} = X_{k} - U(:,1:(k-1)*s) * Q_{k}(1:(k-1)*s,s)

    W = IA0' * Y_{k}
    P_{k}(1:(k-1)*s,:) = U(:,1:(k-1)*s)'*W
    Q_{k}(1:(k-1)*s,:) = R(1:(k-1)*s,1:(k-1)*s)' * P_{k}
    V_{k} = Y_{k} - V(:,1:(k-1)*s) * Q_{k}(1:(k-1)*s,s)
    R0 = Is + Y_{k}' * IA0*U_{k}
    R((k-1)*s+1:k*s,(k-1)*s+1:k*s) = inv(R0)
    U(:,(k-1)*s+1:k*s) = U_{k}
    V(:,(k-1)*s+1:k*s) = V_{k}
end
IA=IA0 - IA0*U*R*V'*IA0
```

Table 1. The computational complexity of the BSM algorithm, $\sigma = m/n$

the computational work for P_k, Q_k, U_k, V_k and $R0^{-1}$			
at the kth step $k = 1, 2, \cdots, p$			
P_k	Q_k	U_k, V_k	$R0^{-1}$
$2[2(k-1)ns^2]$	$2[2(k-1)s^3]$	$2[2(k-1)ns^2 + ns]$	$5s^3 + 2ns^2$
total work by summing up k			
P_k	Q_k	U_k, V_k	$R0^{-1}$
$2nm(m-s)$	$2m(m-s)s$	$2nm(m-s) + 2nm$	$5ms^2 + 2nms$
total computational complexity of BSM			
$4\sigma^2 n^3 + 2\sigma(1 + \sigma s - s)n^2 + 3\sigma s^2 n$			

Remark 1. As is seen from the assumptions, the BSM algorithm may break down - either when a zero scalar entry ($R_{k,k}$) is encountered in SVSM or a singular block ($R0$) is produced in BSM. We refer to [1] for a discussion on that issue and some techniques how to handle such a situation. In our numerical simulations, a breakdown of the BSM algorithm has never been encountered.

Remark 2. Even though A^{-1} is unique, the factors U, V and R in the resulting form of the inverse in ([10]) are not. These depend on the choice of A_0, the order the columns of X and Y are used, the block factors s_k etc.

2.3 Block SM with Reduced Memory Footprint

BSM stores the factors U and V separately, each of them being a matrix of size $n \times m$. This can be a problem when n is large and m is close to n. To simplify our discussion regarding the shape of these factor matrices, we introduce $\sigma \equiv m/n$. We present a variant on the above algorithm that stores the product of $UR^{-1}V$ as a whole matrix H instead of separately storing the matrix factors, in this way significantly reducing memory requirements when m is a large fraction of n ($\sigma \sim 1$). For small m, however, the storage of H requires more space than the individual factors. The trade-off for the decreased memory footprint is a higher computational complexity.

Algorithm. RMBSM (Reduced Memory BSM)

```
p = m/s; % p - number of blocks, s - block size, Is - identity of size s
U=X(:,1:s);  V=Y(:,1:s);
R0 = Is+Y(:,1:s)'*IA0*U; IR0=inv(R0);
H=U*IR0*V';
for k=2:p,
    X_{k} = X(:,(k-1)*s+1:k*s);    Y_{k} = Y(:,(k-1)*s+1:k*s);
    U = X_{k}-H*IA0*X_{k};    V = Y_{k} -H'*IA0'*Y_{k};
    R0 = Is+Y_{k}'*IA0*U; IR0=inv(R0);
    H = H + U*IR0*V';
end
IA=IA0 - IA0*H*IA0;
```

The total computational complexity of RMBSM is found to be the following:

$$\left(6 + \frac{1}{s}\right) \sigma n^3 + \left[4s\left(\sigma - 1\right) + 2\sigma - 1\right] n^2 + \left(5\sigma s - 2\right) sn,$$

where $m = \sigma n$.

2.4 Tuning the Block Size s

The block size parameter s could be chosen arbitrarily in the range $[1, m]$, however the choice affects both performance and memory consumption. By varying s, each of the two block SM algorithms is affected differently. Here we present some theoretical reasoning regarding the choice of the block size, and in the next section we present experimental results for confirmation.

While the total computational complexity for BSM is minimized for a small s, the algorithm is reduced to the single vector SM for $s = 1$, forcing BLAS to use inefficient BLAS1 routines. BLAS libraries tend to work most efficiently with large matrices, i.e. for $s \gg 1$. Then, the optimal point of the tradeoff between computational complexity and BLAS library efficiency must be determined by numerical experiments and will depend on platform and implementation specifics.

The total computational complexity for RMBSM is minimized for $s = m$ for all cases where $\sigma \equiv m/n \leq 0.5$. This is the largest possible value of s, so in this case there is no performance tradeoff as above. The total computational complexity using the optimal block size is therefore $(5\sigma^3 + 4\sigma^2 + 2\sigma)n^3$. The existence of a linear term in σ dramatically decreases the effectiveness of this algorithm compared to BSM. For the case when $\sigma = 1$, the complexity of RMBSM is minimized for $s = 1$, leading to the same tradeoff as for the BSM algorithm. When $0.5 \leq \sigma \leq 1$, the block size that minimizes the computational complexity varies in a somewhat complicated way in the range $[1, m]$. This is explored by numerical experiments.

As it would be expected, both algorithms require more memory for larger block sizes. The memory demand also strongly depends on m. As the experiments presented in Section 4 show, the memory demand of the BSM algorithm increases faster than that of RMBSM.

3 Implementing Block SM for Multicore Computers

Effective programs for multicore systems must support sufficient parallelism to fully exploit the available hardware. The BSM and RMBSM algorithms both feature very low level of parallelism on the algorithm description level. To compute the block matrices U_k, V_k in BSM or H in RMBSM, the previous ones, i.e., U_{k-1} and V_{k-1}, must be computed first, which limits available parallelism.

This problem can be addressed by recognizing that the two algorithms consist almost exclusively of matrix products (GEMM) and the computation of a matrix inverse (GESV), operations which provide a high degree of parallelism. The available parallelism of GEMM depends on the shape of the matrices and the GEMM algorithm [5], but for large enough matrices it is always much greater than the number of cores on our machine. As long as we use reasonably large blocks, the GEMM operations in our algorithms can be efficiently executed in parallel by a parallel BLAS library routine. A similar argument can be made for the GESV operation, meaning the bulk of the work of our algorithm can be executed on a multicore machine without writing explicitly parallel code.

One goal with this work is to identify differences in the behavior of the BSM and RMBSM algorithms running on the GPU compared to the CPU in order to determine the utilization of a possible heterogeneous multicore version to be implemented in the future. We write a straightforward implementation, by simply replacing GEMM calls with calls to CUBLAS (Nvidia SDK 3.2) and compare its performance with that of the CPU codes. Algorithmically, the available parallelism on the GPU is the same as discussed, but the GPU has many more cores and therefore a more stringent requirement on the size of matrices. The effect of this and other properties of GPU hardware are observed and discussed in Section 4.3.

4 Numerical Experiments

In this section we use the SM algorithms (SVSM, BSM and RMBSM) to compute the inverse of a nonsingular matrix $A \in \mathbb{R}^{n\times n}$. We assume that the matrix A is already available in the form $A = A_0 + XY^T$ with $A_0 = I_n$ (I_n denoting the identity matrix of size n). In all experiments X and Y are randomly generated dense matrices of size $n \times m$, i.e., each element is a random number from a uniform distribution between 0 and 1. Also, in all experiments the matrices X and Y are equally partitioned into column sets, i.e., $X = [X_1, X_2, \cdots, X_p]$, $Y = [Y_1, Y_2, \cdots, Y_p]$, $\{X_k, Y_k\}_{k=1}^{p} \in \mathbb{R}^{n\times s}$, where $s = m/p$.

4.1 Speed Optimization

Based on the discussion in Section 2.4, we expect to see that the performance of BSM and RMBSM vary with the block size s. The following experiments are performed with Fortran implementations of the algorithms on a system with eight-core Intel Xeon X6550 processors. The BLAS routines are from the Sun Performance Library (see e.g., [9]).

In Figures 1 and 2(a) we plot the runtime for some test problems and see that our expectations are aligned with the obtained performance results. For mid-to-low values of σ, the impact of the block size on library efficiency is so significant that $s = m$ is the optimal choice of block size regardless of algorithm and problem. For $\sigma > 0.5$, the optimal block size appears to be in the range $(0.02 \times m, 0.2 \times m)$.

We also see that the RMBSM algorithm, while improving on SVSM at large s, is less efficient than the BSM algorithm. We are convinced that RMBSM is more appropriate for larger problems.

In Figure 2(a)we observe that both BSM and RMBSM show better performance than the direct method. The BSM algorithm with the optimal block size performs about 12 times faster than the direct method.

The parallel speedup of the two block algorithms is plotted in Figure 2(b). We see that both the two block algorithms achieve close to linear speedup, which means that they fully benefit from the parallelism, inherent in BLAS3 operations.

4.2 Memory Optimization

As Figure 3 shows, the memory consumption of the BSM algorithm grows rapidly with m, which motivated the usage of the RMBSM algorithm, designed to consume less memory when m is relatively large. The memory footprint of RMBSM varies more slowly than that of BSM, yielding a memory savings of up to about

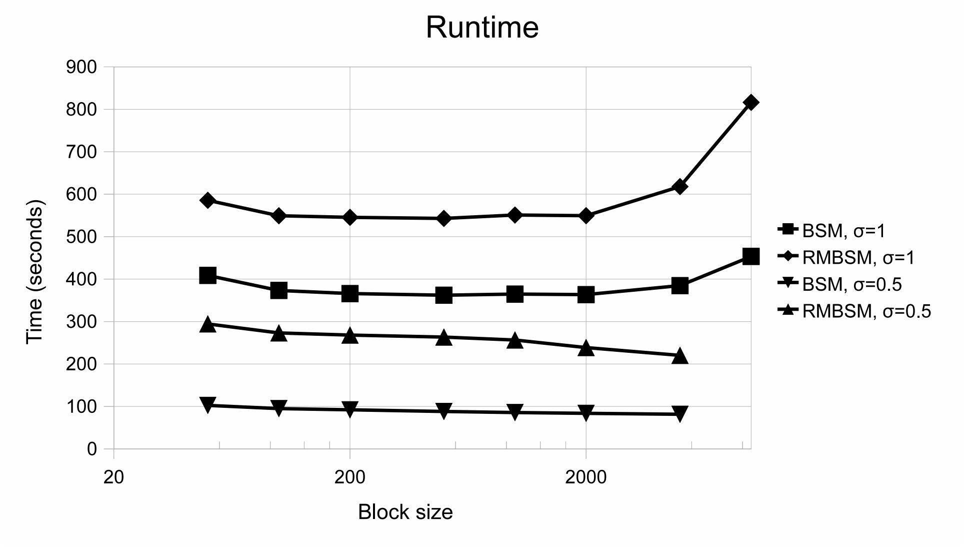


Fig. 1. Performance of BSM and RMBSM with varying blocksize for two sizes of factor matrices U and V. Matrix size is $n = 10k$, $\sigma \equiv m/n$.

50% when $m = n$. With smaller m, however, the usefulness of RMBSM is more limited (see Figure 4).

For a given n and m, block size selection not only affects the performance but also the memory footprint. Our experimental results here can be illustrated by the following three cases:

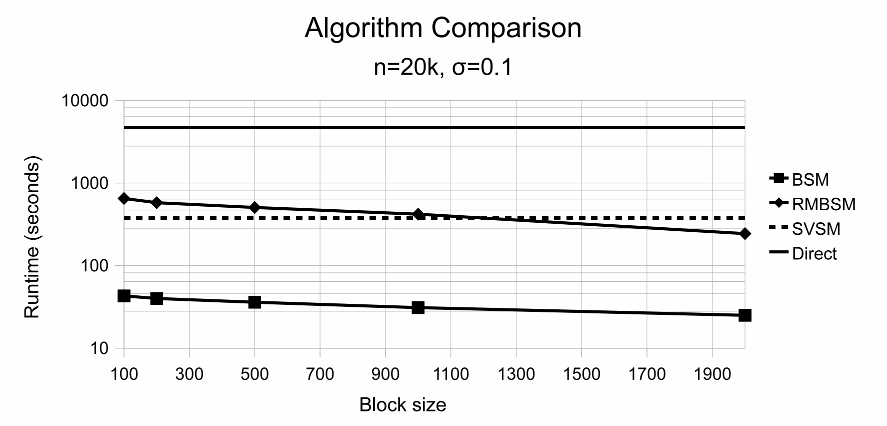
If $m \sim n$, then memory may be a problem. If a small block size does not sufficiently shrink the memory consumption of BSM, RMBSM will further reduce memory consumption.

If $m \sim n/2$, then memory can still be a significant obstacle, but RMBSM is not very effective. Choosing a small block size can reduce the memory consumption of BSM by up to a factor of 3.

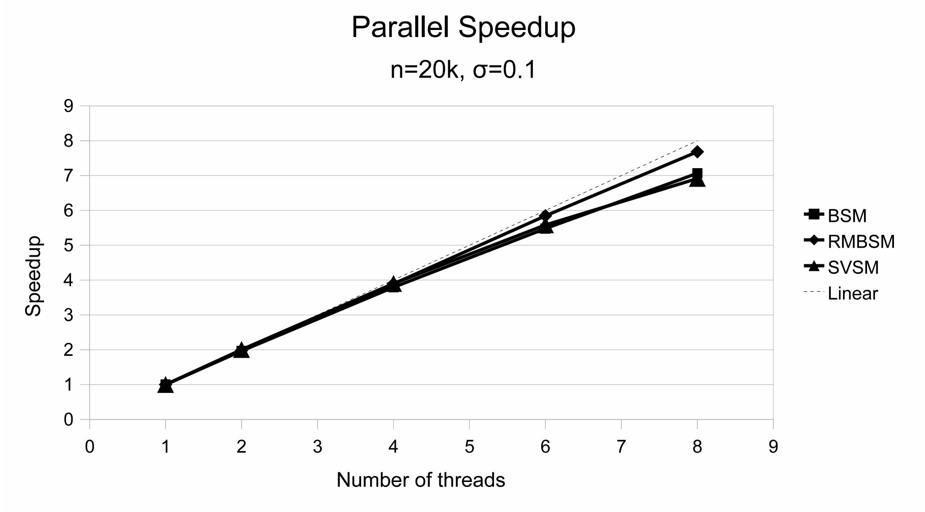
If $m \ll n/2$, then memory consumption may not pose any problems, and RMBSM actually consumes more memory than BSM.

4.3 Numerical Experiments Using GPU

The GPU experiments are performed on a compute node consisting of two 8-core AMD Opteron 6220 (Bulldozer) processors at 3 GHz and a Nvidia Tesla M2050 GPU. Since the Bulldozer processors are configured so that two cores share a single FPU, using eight threads yields full hardware utilization and optimal performance for our CPU codes. These codes rely on the AMD Core Math Library (ACML) for their BLAS routines. The GPU codes use Nvidia CUBLAS, which lack the GESV routine. We therefore perform only the GEMM calls on the GPU, but these operations dominate the runtime to such an extent that the results remain meaningful, especially when $m \ll n$.



(a) Single-threaded runtimes of the algorithms with varying block size for a sample problem (logarithmic y scale)



(b) Scaling behavior on a multicore system. Matrix size is n, $\sigma \equiv m/n$.

Fig. 2. Performance on a multicore machine

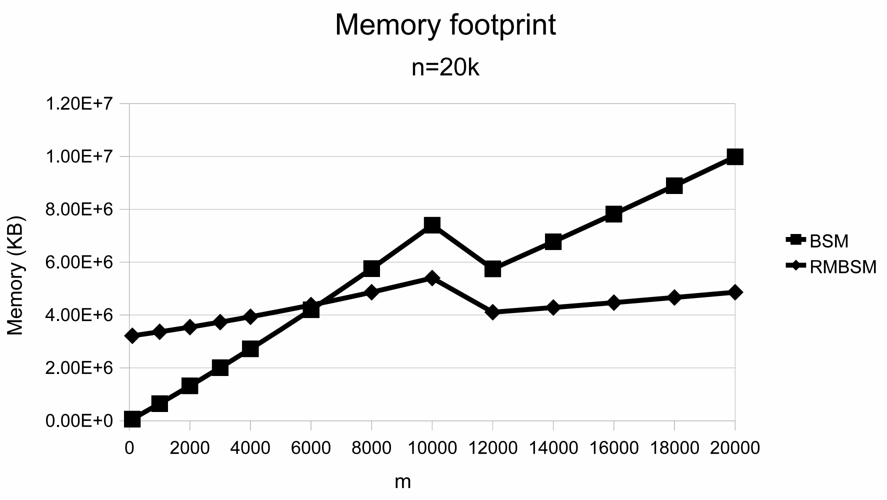


Fig. 3. Memory usage when varying m, the width of the matrix factors U and V. Optimized block size (Section 2.4) is equal to m for $m \leq n/2$, and equal to $m/5$ when $m > n/2$.

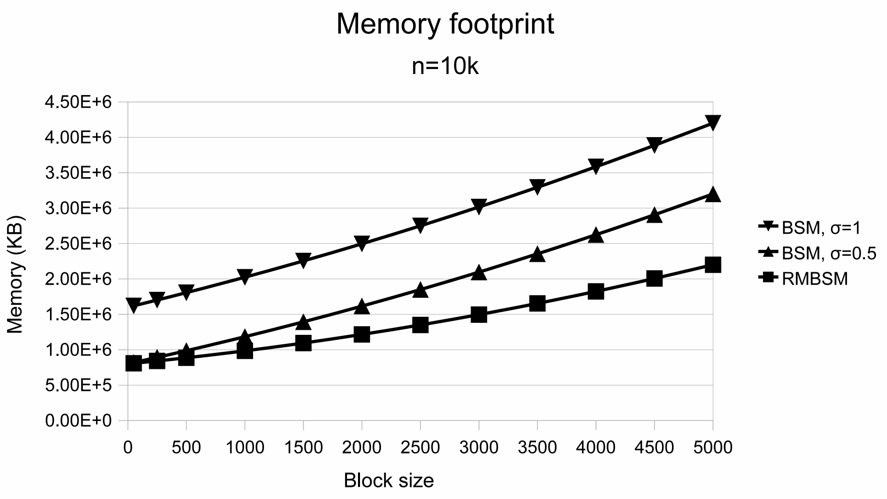


Fig. 4. Memory usage of algorithms for varying block sizes and two values of σ, the ratio of m to n. Note that the memory footprint of RMBSM varies with blocksize, not m, so only one curve is shown (double precision data).

The Nvidia Tesla M2050 has a peak theoretical double precision performance of 515 GFLOPS, but CUBLAS performs at only about 200 GFLOPS [8], while the AMD processor has a peak theoretical double precision performance of almost 200 GFLOPS, which means that any speedup will be modest at best. As the results in Figure 5 show that our GPU implementation achieves up to 20% faster execution than the CPU. We can also see that the performance of the GPU codes appears to be more sensitive to the block size choice than that of the CPU codes. This is likely due to two factors: our straightforward implementation grossly ignores the cost of data transmission, and the profiler-reported device occupancy is only 33%. The effect of the former factor is that we perform unnecessary data movement between host memory and device memory, which can take up to 50% of the total runtime, according to profiling results. A proper implementation could decrease the data movement by up to a factor of five. The issue with low occupancy may or may not actually impact performance depending on whether the global memory accesses on the device are successfully hidden, but this is worth considering when designing a better optimized implementation with respect to the GPU architecture.

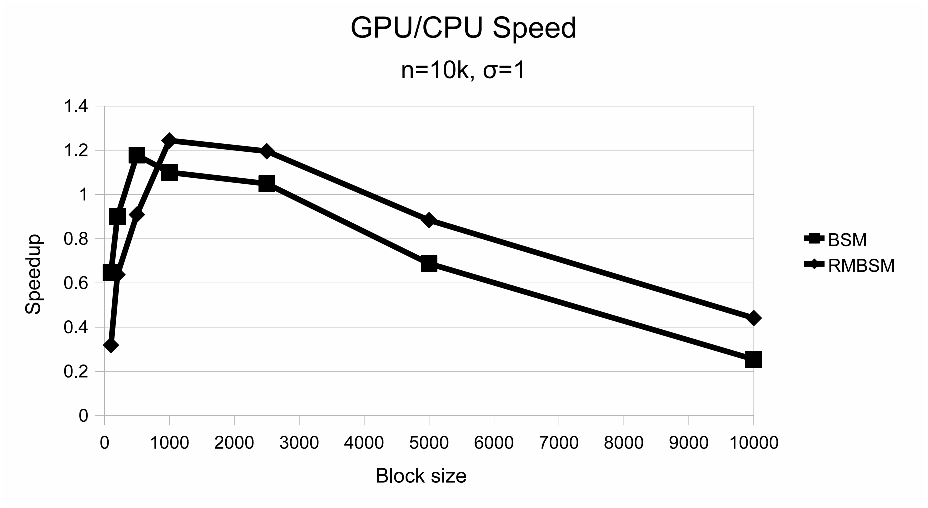


Fig. 5. Speedup of running both algorithms on the Nvidia Tesla M2050 GPU compared to the Opteron 6620@3GHz running eight threads

5 Conclusions

In this paper we consider the task to compute the inverse of a given dense nonsingular matrix and we present two parallelization strategies for the block SM algorithm. Even though, due to effects of floating point arithmetic the inverse, computed using this algorithm might in some cases be less accurate than the true inverse, the computational procedure is a useful benchmark test when developing and analyzing new algorithms and their implementations on CPU/GPU architectures. Other possible applications are, e.g., approximate inverses to be used as multiplicative preconditioners.

We present the parallel performance of the two block SM factorizations, i.e., BSM and RMBSM, using a multicore CPU as well as a GPU. The main conclusions are that effective parallelization is quite easily implemented and the speedup is almost linear.

These results are achieved over a wide range of choices of the block size. The preferred block size may be influenced by memory considerations. A small block size can save up to 66% of the memory usage of BSM. For a larger problem, the RMBSM algorithm can save up to 50% of that of BSM.

The results from our straightforward GPU implementation, though modest, warrant a more serious implementation effort in the future.

The effect of sparse matrices on the performance of the block SM algorithm, data structures and parallelization techniques, as well as obtaining an approximate inverse of a dense or sparse matrix, needs to be further considered and is still in progress.

A relevant consideration that we have not addressed in this paper is the numerical accuracy and stability of BSM and RMBSM. As already mentioned, there exist other alternatives to compute the inverse of a non-singular matrix, e.g., the various direct methods, iterative methods and the Gauss-Jordan method. In general, one could expect that the direct methods produce more accurate matrix inverses. The latter issue was tested separately. The performed numerical experiments (not included here) did not show loss of accuracy in the computed matrix inverses, however a more rigorous error analysis of the numerical stability of the SM algorithm is of definite practical relevance.

References

1. Bru, R., Cerdán, J., Marín, J., Mas, J.: Preconditioning sparse nonsymmetric linear systems with the Sherman-Morrison formula. SIAM J. Sci. Comput. 25, 701–715 (2003)
2. Bru, R., Marín, J., Mas, J., Tůma, M.: Balanced incomplete factorization. SIAM J. Sci. Comput. 30, 2302–2318 (2008)
3. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. 35, 38–53 (2009)
4. Cerdán, J., Faraj, T., Malla, N., Marín, J., Mas, J.: Block approximate inverse preconditioners for sparse nonsymmetric linear systems. Electron. Trans. Numer. Anal. 37, 23–40 (2010)
5. Gunnels, J., Lin, C., Morrow, G., van de Geijn, R.: Analysis of a Class of Parallel Matrix Multiplication Algorithms. In: First Merged Internatial Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998), pp. 110–116. IEEE Press, New York (1998)

6. Hellel, D.: A survery of parallel algorithms in numerical linear algebra. SIAM Rev. 20, 740–777 (1978)
7. Melab, N., Talbi, E.-G., Petiton, S.: A parallel adaptive Gauss-Jordan algorithm. J. Supercomput. 17, 167–185 (2000)
8. Nath, R., Tomov, S., Dongarra, J.: An improved MAGMA GEMM for Fermi GPUs. Int. J. High Perform. Comput. 24, 511–515 (2010)
9. Sun Performance Library Reference Manual, http://docs.sun.com/app/docs/doc/820-2171

Multi-threaded Nested Filtering Factorization Preconditioner

Pawan Kumar[⋆], Karl Meerbergen, and Dirk Roose

Department of Computer Science,
KU Leuven, Celestijnenlaan 200A, 3001 Heverlee-Leuven, Belgium
{pawan.kumar,karl.meerbergen,dirk.roose}@cs.kuleuven.be

Abstract. The scalability and robustness of a class of non-overlapping domain decomposition preconditioners using 2-way nested dissection reordering is studied. In particular, three methods are considered: a nested symmetric successive over-relaxation (NSSOR), a nested version of modified ILU with rowsum constraint (NMILUR), and nested filtering factorization (NFF). The NMILUR preconditioner satisfies the rowsum property i.e., a right filtering condition on the vector $(1, \ldots, 1)^T$. The NFF method is more general in the sense that it satisfies right filtering condition on any given vector. There is a subtle difference between NMILUR and NFF, but NFF is much more robust and converges faster than NSSOR and NMILUR. The test cases consist of a Poisson problem and convection-diffusion problems with jumping coefficients.

1 Introduction

We consider the problem of solving large sparse linear systems of the form

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

by an iterative method preconditioned by a nonoverlapping domain decomposition method. A preconditioner $\mathbf{B}$ is said to satisfy "rowsum" property when

$$\mathbf{B1} = \mathbf{A1} \tag{2}$$

where $\mathbf{1} = [1, 1, 1, ..., 1]^T$. In general, a preconditioner may satisfy a more general (right) filtering condition as follows

$$\mathbf{Bt} = \mathbf{At} \tag{3}$$

where $\mathbf{t}$ is a filter vector. The basic linear fixed point method for solving the linear system (1) above is given as follows

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \mathbf{B}^{-1}(\mathbf{b} - \mathbf{Ax}^n) = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{x}^n + \mathbf{B}^{-1}\mathbf{b}. \tag{4}$$

[⋆] This work was done when the first author had a visiting position at Université Libre de Bruxelles, Brussels and a postdoctoral position at KU Leuven, Leuven and Flanders Exascience Lab (Intel labs Europe), Leuven.

Now, subtracting (4) from the identity $\mathbf{x} = \mathbf{x} - \mathbf{B}^{-1}\mathbf{A}\mathbf{x} + \mathbf{B}^{-1}\mathbf{b}$, we obtain the following expression of the error at the $(n+1)$th step

$$\mathbf{e}^{n+1} = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{e}^{n} = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})^{2}\mathbf{e}^{n-1} = \cdots = (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})^{n+1}\mathbf{e}^{0}.$$

If $\mathbf{B}$ satisfies the filtering condition (3), we have $(\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})\mathbf{t} = \mathbf{0}$, thus by choosing a suitable $\mathbf{t}$, a desired component of the error vector could be removed. Good candidates for the filter vectors are approximations to the eigenvectors corresponding to the smallest eigenvalues of the preconditioned coefficient matrix: $\mathbf{B}^{-1}\mathbf{A}$ [17].

In the past, several approximate and exact filtering preconditioners were proposed for block tridiagonal matrices of the form $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$. The matrix $\mathbf{A}$ is partitioned as follows

$$\mathbf{A} = \begin{pmatrix} D_1 & U_1 & & \\ L_1 & D_2 & \ddots & \\ & \ddots & \ddots & U_{n-1} \\ & & L_{n-1} & D_n \end{pmatrix}, \tag{5}$$

where the matrices $\mathbf{L}$, $\mathbf{D}$, and $\mathbf{U}$ are sparse matrices as follows

$$\mathbf{L} = \begin{pmatrix} 0 & & & \\ L_1 & 0 & & \\ & \ddots & \ddots & \\ & & L_{n-1} & 0 \end{pmatrix}, \quad \mathbf{D} = \begin{pmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_n \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} 0 & U_1 & & \\ & \ddots & \ddots & \\ & & 0 & U_{n-1} \\ & & & 0 \end{pmatrix}.$$

The exact block $\mathbf{LU}$ factorization of the matrix $\mathbf{A}$ reads

$$\mathbf{A} = (\mathbf{T} + \mathbf{L})(\mathbf{I} + \mathbf{T}^{-1}\mathbf{U}),$$

where $\mathbf{T}$ is a block diagonal matrix obtained from the following recurrence

$$T_i = \begin{cases} D_1, & i = 1, \\ D_i - L_{i-1}T_{i-1}^{-1}U_{i-1}, & 1 < i \leq n. \end{cases} \tag{6}$$

The Schur complements T_i, $(i > 1)$ are costly to compute because the submatrices $T_i, i > 1$ usually become denser (even though T_1 is sparse). There exist many preconditioners that involve some approximation of the Schur complement. However, we are particularly interested in the approximations that retain the filtering condition (3). For some approximations, satisfying filtering condition with vector $\mathbf{1}$ is particularly useful for convection-diffusion problems as we will see later. One of the earliest known methods satisfying (3) is the modified incomplete LU [17]; this method is applicable to general matrices that may not have block tridiagonal form. For block tridiagonal matrices, in [18], a tridiagonal approximation to T_{i-1}^{-1} and $L_{i-1}T_{i-1}^{-1}U_{i-1}$ was proposed. Axelson and Polman [16] proposed an approximation that satisfies the following relation

$$\tilde{T}_i\mathbf{1} = (D_i - L_{i-1}\tilde{T}_{i-1}^{-1}U_{i-1})\mathbf{1},$$

$$\tilde{T}_i\mathbf{n} = (D_i - L_{i-1}\tilde{T}_{i-1}^{-1}U_{i-1})\mathbf{n},$$

where $\mathbf{1}$ and $\mathbf{n} = [1, 2, \ldots, n]^T$ are test vectors. In [15], a sequence of filtering decompositions is proposed where the choice of filter vectors are the sine and cosine functions that damp both the high and low frequency components of the error. Recently, improved filtering decompositions were proposed [13,14]. A parallel implementation of a filtering decomposition appeared in [12].

In [9] a class of parallel preconditioners based on a 2-way nested dissection (ND) ordering was proposed. In particular, a method named nested filtering factorization (NFF) was proposed. Given a matrix $\mathbf{A}$, the 2-way ND reordering leads to a permuted matrix $\mathbf{P^T A P}$ with the following structure

$$\mathbf{P^T A P} = \left(\begin{array}{cc|c} D_0 & & U_0 \\ & D_1 & U_1 \\ \hline L_0 & L_1 & S_0 \end{array}\right) = (\mathbf{T} + \mathbf{L})(\mathbf{I} + \mathbf{T}^{-1}\mathbf{U}), \tag{7}$$

where

$$\mathbf{L} = \left(\begin{array}{cc|c} & & \\ \hline L_0 & L_1 & \end{array}\right), \quad \mathbf{T} = \left(\begin{array}{cc|c} T_0 & & \\ & T_1 & \\ \hline & & T_2 \end{array}\right), \quad \mathbf{U} = \left(\begin{array}{cc|c} & & U_0 \\ & & U_1 \\ \hline & & \end{array}\right).$$

This leads to the following recursion for T_i

$$T_i = \begin{cases} D_i, & i = 0, 1, \\ S_0 - L_0 T_0^{-1} U_0 - L_1 T_1^{-1} U_1, & i = 2. \end{cases}$$

Unlike (6), where T_i depends on T_{i-1}, for the recursion above, T_0 and T_1 are independent but T_2 depends on both T_0 and T_1. To achieve more concurrency in the algorithm, we apply the 2-way ND reordering to the domains D_0 and D_1 and obtain a similar factorization. The factored forms of D_0 and D_1 could then be used to estimate T_2. The preconditioners are obtained by approximating the Schur complements as was done in the block tridiagonal case. In this paper, we consider three possible approximations (first introduced in [11]) as follows:

- **NSSOR**: Here $T_2 = S_0$, the terms $L_0 T_0^{-1} U_0$ and $L_1 T_1^{-1} U_1$ are dropped. We call this NSSOR because the method is a multilevel extension of the classical SSOR method.
- **NMILUR**: Here T_2 is approximated as follows

$$T_2 = S_0 - \mathrm{diag}(L_0 T_0^{-1} U_0 \mathbf{1}) - \mathrm{diag}(L_1 T_1^{-1} U_1 \mathbf{1}).$$

- **NFF**: Here T_2 is approximated as follows

$$T_2 = S_0 - L_0 \beta_1 U_0 - L_1 \beta_2 U_1$$

  where

$$\beta_1 = \mathrm{diag}(T_0^{-1}(U_0 t_0)./(L_0 t_0)),\ \beta_2 = \mathrm{diag}(T_1^{-1}(U_1 t_1)./(L_1 t_1)),$$

  here ./ is a point-wise vector division, diag is the MATLAB operator, t_0 and t_1 are given column vectors.

Table 1. Number of iterations for the Poisson problem on $40 \times 40 \times 40$ grid with zero boundary condition. Preconditioners used are AMG: BoomerAMG in Hypre, AGMG: unsmoothed aggregation based multigrid [19], CG: conjugate gradient, PARASAILS: sparse approximate preconditioner in Hypre [2], tolerance for the relative residual is 10^{-7}. Ndoms stands for number of subdomains.

Solvers	GMRES			CG			
Preconditioners/Ndoms	NSSOR	NMILUR	NFF	AMG	AGMG	PARASAILS	none
2	24	60	12	5	12	52	92
4	27	86	13	6	12	55	92

Notice the subtle difference in the approximations for NMILUR and NFF both satisfying a right filtering property; NMILUR satisfies right filtering on the vector of all ones and NFF satisfies filtering on any given filter vector $\mathbf{t}$. As we will see later, NFF is remarkably fast and robust compared to NMILUR and NSSOR and is a "potential" competitor of the algebraic multigrid methods (AMG). In Table 1, we compare the iteration count of GMRES preconditioned by NSSOR, NMILUR and NFF, and CG preconditioned by state-of-the-art preconditoners of Hypre [2] (PARASAILS and boomer AMG) and AGMG (aggregation based AMG) [19]. Unlike AMG methods which rely on smoothing and a coarse grid solve, the methods considered in this paper belong to a class of multilevel Schur complement based methods. In this paper, we present preliminary results on the scalability of these methods on a shared memory architecture. Our choice of using 2-way ND is justified as follows:

- The recursive 2-way ND reordering leads to a blindly cache-aware algorithm exhibiting both spatial and temporal locality as illustrated in Figure 3.
- The matrix-vector product, setup, and solution phases (forward and backward solves) can be executed concurrently as explained in the sections that follow.
- It is well known that ND is a fill-reducing reordering for direct methods. Such reorderings also reduce fill-ins for the preconditioners considered in this paper.

We show a straightforward parallelization of the methods using `cilk plus` with new vectorization features using elemental functions. We also identify the possible future work that may lead to better parallelization. Our numerical experiments consist of a model Poisson problem and convection-diffusion problem with jumping coefficients.

2 Nested Filtering Factorization

As mentioned above, effective parallelism and cache reuse can be achieved by the 2-way nested dissection reordering first proposed by Alan George [10] in 1973 for a regular finite element mesh. The method has been extended to tackle general problems where only the adjacency graph of the matrix is required [6–8].

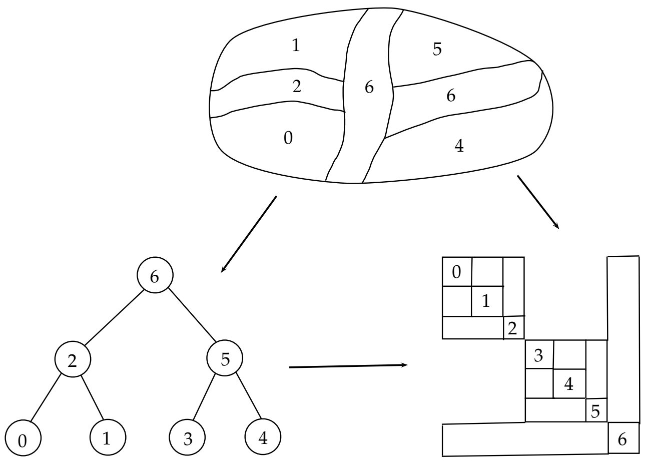


Fig. 1. Top: Graph partitioned into 4 parts, the four parts are first numbered, followed by the separator. Bottom Left: Corresponding tree, Bottom right: Corresponding matrix after reordering.

In Fig. 1, we illustrate the ND reordering, the corresponding tree graph, and the structure of the ND reordered matrix. In the 2-way ND method, a separator (vertices or edges of the graph) is selected which (when removed) separates the graph into two or more subgraphs. The process is continued recusively on the separated subgraphs. In Fig. 1, we show the typical block structure of a ND reordered matrix; such structure is obtained by renumbering the nodes in the separated graphs first, followed by numbering the nodes in the separator. In Fig. 1 we show a graph partitioned into four parts. Renumbering the nodes is equivalent to permuting the matrix. Assume that ND has been applied recursively $l+1$ times. We will refer to the top level as level $l+1$ (vertex 6 of Fig. 1 is at top level) and the bottom level as level 0 (vertices 0, 1, 3, and 4 in Fig. 1). Let $\mathbf{P} \in \mathbb{N}^{n \times n}$ be the required permutation matrix, then the permuted matrix corresponding to ND reordering is a bordered block diagonal matrix with the following structure

$$\mathbf{P^T A P} = \left(\begin{array}{cc|c} D_0^l & & U_0^l \\ & D_1^l & U_1^l \\ \hline L_0^l & L_1^l & S_0^{l+1} \end{array}\right) = (\mathbf{T_l} + \mathbf{L_l})\mathbf{T_l}^{-1}(\mathbf{T_l} + \mathbf{U_l}), \tag{8}$$

where

$$\mathbf{L_l} = \left(\begin{array}{cc|c} & & \\ \hline L_0^l & L_1^l & \end{array}\right), \quad \mathbf{T_l} = \left(\begin{array}{cc|c} D_0^l & & \\ & D_1^l & \\ \hline & & T_0^{l+1} \end{array}\right), \quad \mathbf{U_l} = \left(\begin{array}{cc|c} & & U_0^l \\ & & U_1^l \\ \hline & & \end{array}\right).$$

From (8), it is easy to see that T_0^{l+1} is

$$T_0^{l+1} = S_0^{l+1} - L_0^l (D_0^l)^{-1} U_0^l - L_1^l (D_1^l)^{-1} U_1^l. \tag{9}$$

Notice that T_0^{l+1} depends on D_0^l and D_1^l. Both D_0^l and D_1^l themselves have nested bordered block diagonal structure as follows

$$D_0^l = \left(\begin{array}{cc|c} D_0^{l-1} & & U_0^{l-1} \\ & D_1^{l-1} & U_1^{l-1} \\ \hline L_0^{l-1} & L_1^{l-1} & S_0^l \end{array}\right), \quad D_1^l = \left(\begin{array}{cc|c} D_2^{l-1} & & U_2^{l-1} \\ & D_3^{l-1} & U_3^{l-1} \\ \hline L_2^{l-1} & L_3^{l-1} & S_1^l \end{array}\right),$$

and both of them admit a block **LU** factorization as we had for the original matrix $\mathbf{P^T AP}$ in (8). The factored forms of D_0^l and D_1^l could then be used to estimate (9). This process of estimating the Schur complement by using the block **LU** factorization of the subdomain matrices is continued till the second last level (the last level 0 does not have ND structure). At the second last level $l = 1$, the domain matrices are represented by D_k^1, $k = 0, \ldots, 2^l - 1$ where

$$D_k^1 = \left(\begin{array}{cc|c} D_{2k}^0 & & U_{2k}^0 \\ & D_{2k+1}^0 & U_{2k+1}^0 \\ \hline L_{2k}^0 & L_{2k+1}^0 & S_k^1 \end{array}\right), \; k = 0, \ldots, 2^l - 1. \tag{10}$$

As above, to obtain a block **LU** factorization for D_k^1, $k = 0, \ldots, 2^l - 1$, we need a Schur complement computation

$$T_k^1 = S_k^1 - L_{2k}^0 (D_{2k}^0)^{-1} U_{2k}^0 - L_{2k+1}^0 (D_{2k+1}^0)^{-1} U_{2k+1}^0, \; k = 0, \ldots, 2^l - 1, \tag{11}$$

where D_{2k}^0 and D_{2k+1}^0 are assumed to be small enough to be factored cheaply and easily by a direct method. If the Schur complements T_k^j are not approximated, we obtain an exact nested factorization, but our objective here is to obtain preconditoners by avoiding the exact computation of the form $L_{2k}^s (D_{2k}^s)^{-1} U_{2k}^s$ and $L_{2k+1}^s (D_{2k+1}^s)^{-1} U_{2k+1}^s$, $s = 1, \ldots, l+1$ that appears during Schur complement computation. From here onwards, we denote an approximation of the Schur complements T_k^s by $\tilde{T}_k^s$ and an approximation of the domain matrix D_k^s by $\tilde{D}_k^s$. Thus, the block **LU** factors for the domain matrix $\tilde{D}_k^{s+1}$ are given as follows

$$\tilde{D}_k^{s+1} = \left(\begin{array}{cc|c} \tilde{D}_{2k}^s & & U_{2k}^s \\ & \tilde{D}_{2k+1}^s & U_{2k+1}^s \\ \hline L_{2k}^s & L_{2k+1}^s & S_k^{s+1} \end{array}\right) = \left(\begin{array}{cc|c} \tilde{D}_{2k}^s & & \\ & \tilde{D}_{2k+1}^s & \\ \hline L_{2k}^s & L_{2k+1}^s & \tilde{T}_k^{s+1} \end{array}\right) \left(\begin{array}{cc|c} I & & (\tilde{D}_{2k}^s)^{-1} U_{2k}^s \\ & I & (\tilde{D}_{2k+1}^s)^{-1} U_{2k+1}^s \\ \hline & & I \end{array}\right). \tag{12}$$

As already mentioned in the section 1, we shall consider three possible approximations as follows:

– **NSSOR**: Here we approximate the Schur complements by setting

$$T_k^j = S_k^j, \; j = 1, \ldots, l+1, \; k = 0, \ldots, 2^j - 1.$$

This can be implemented by calling Algorithm 1 with parameters $lev = l+1$ and $k = 0$. Since for NSSOR the Schur complements are approximated by the diagonal blocks, we only need to factor these diagonal block to be used during the solve phase. In Algorithm 1, first the top level separator block is factored, see step (6), then, the next level separator blocks are factored by recursive calls in steps (7) and step (8). Finally the domain matrices at the lowermost level 0 are kept in factored form in step (4).

Algorithm 1. BuildNSSOR(lev, k)

```
1: INPUT: $\mathbf{A}$, $lev = l + 1$
2: OUTPUT: $\tilde{T}$
3: **if** $lev$=0 **then**
4:    Factor($D_k^0$)
5: **else**
6:    Factor($S_k^{lev}$)
7:    **cilk_spawn** BuildNSSOR($lev - 1$, $2k$)
8:    BuildNSSOR($lev - 1$, $2k + 1$)
9: **end if**
```

– **NMILUR**: In this method, the Schur complements are approximated such that the preconditioner satisfies the so-called rowsum property (or right filtering on vector of all ones).

$$T_k^1 = S_k^1 - \mathrm{diag}(L_{2k}^0 (D_{2k}^0)^{-1} U_{2k}^0 \mathbf{1}) - \mathrm{diag}(L_{2k+1}^0 (D_{2k+1}^0)^{-1} U_{2k+1}^0 \mathbf{1}),$$
$$k = 0, \ldots, 2^l - 1$$
$$T_k^{l+1} = S_k^{l+1} - \mathrm{diag}(L_{2k}^l (\tilde{D}_{2k}^l)^{-1} U_{2k}^l \mathbf{1}) - \mathrm{diag}(L_{2k+1}^l (\tilde{D}_{2k+1}^l)^{-1} U_{2k+1}^l \mathbf{1}),$$

where $l > 0, k = 0, \ldots, 2^l - 1$. See Algorithm 2 for implementation details. The steps involved in building the nested preconditioners are outlined in Fig. 2. First of all, the partitioned domain matrices (D_k^0) are factored and are used to construct the Schur complements T_k^1. Afterwards, we have an approximate block **LU** factorization for the domain matrices $\tilde{D}_k^1$ given by (12). Then these approximate factors are used to build the Schur complements T_k^2 which then leads to an approximate block **LU** factorization for the domain matrices $\tilde{D}_k^2$ and so on. These are precisely the steps described in algorithm 2.

– **NFF**: Here the Schur complements are approximated such that the preconditioner satisfies the filtering property on a given filter vector $\mathbf{t}^T = (t_{2k}^l, t_{2k+1}^l)$ as follows

$$\tilde{T}_k^{l+1} = S_k^{l+1} - L_{2k}^l \beta_{2k}^l U_{2k}^l - L_{2k+1}^l \beta_{2k+1}^l U_{2k+1}^l$$
$$\beta_{2k}^l = \mathrm{diag}(\tilde{D}_{2k}^l)^{-1} (U_{2k}^l t_{2k}^l)./(U_{2k}^l t_{2k}^l) \quad (13)$$
$$\beta_{2k+1}^l = \mathrm{diag}(\tilde{D}_{2k+1}^l)^{-1} (U_{2k+1}^l t_{2k+1}^l)./(U_{2k+1}^l t_{2k+1}^l) \quad (14)$$

See Algorithm 3 for implementation details.

Once the Schur complements are found and stored in $\tilde{T}$, the solution procedure for solving with these nested preconditioners is identical. This involves calling the subsequent forward and backward sweep routines shown in Algorithms 4

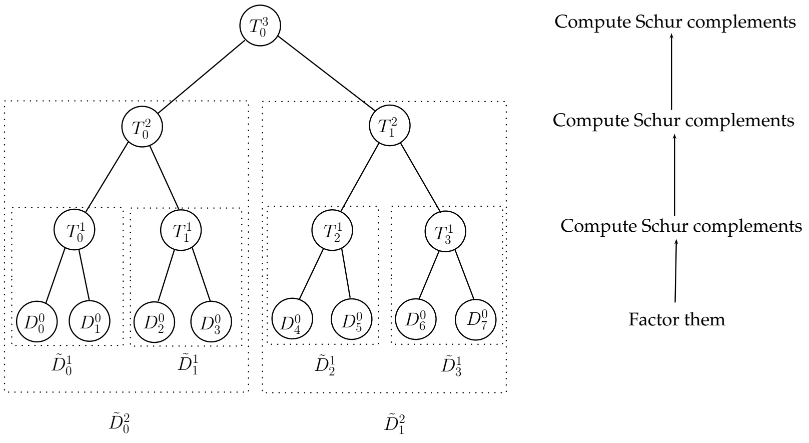


Fig. 2. Steps in building the nested preconditioners

and 5 respectively. Notice here that the solve with the domain matrices $\tilde{D}_{2k}^l$ and $\tilde{D}_{2k+1}^l$ uses the factored form as shown in equation (12). We illustrate the forward sweep procedure. A typical forward sweep is given as follows

$$\left(\begin{array}{cc|c} \tilde{D}_{2k}^l & & \\ & \tilde{D}_{2k+1}^l & \\ \hline L_{2k}^l & L_{2k+1}^l & \tilde{T}_k^{l+1} \end{array}\right) \begin{pmatrix} y_{2k}^l \\ y_{2k+1}^l \\ \overline{y}_{2k}^{l+1} \end{pmatrix} = \begin{pmatrix} b_{2k}^l \\ b_{2k+1}^l \\ \overline{b}_k^{l+1} \end{pmatrix}. \tag{15}$$

Notice that the bar notation is used (for example $\overline{b}_k^{l+1}$) to denote the part of the vector corresponding to the separator block. The forward sweep now corresponds to the following steps:

$$\begin{aligned} \tilde{D}_{2k}^l y_{2k}^l &= b_{2k}^l, \\ \tilde{D}_{2k+1}^l y_{2k+1}^l &= b_{2k+1}^l, \\ \tilde{T}_k^{l+1} \overline{y}_{2k}^{l+1} &= \overline{b}_k^{l+1} - L_{2k}^l y_{2k}^l - L_{2k+1}^l y_{2k+1}^l. \end{aligned}$$

The three steps of the forward sweep procedure correspond to the steps (6), (7), and (8) in Algorithm 4. The backward sweep procedure is similar and is shown in Algorithm 5.

The domain matrices $\tilde{D}_{2k}^l$ and $\tilde{D}_{2k+1}^l$ themselves allow ND representation for $l > 1$, thus, the steps (6) and (7) are recursive calls to the forward sweep and backward sweep algorithms.

Next we consider the sparse matrix vector (SpMV) operation implemented using `cilk plus`. A `cilk plus` version of the sparse SpMV is described in Algorithm 6. A typical matrix-vector product is given by

$$\begin{pmatrix} y_{2k}^l \\ y_{2k+1}^l \\ \overline{y}_{2k}^{l+1} \end{pmatrix} = \left(\begin{array}{cc|c} D_{2k}^l & & U_{2k}^l \\ & D_{2k+1}^l & U_{2k+1}^l \\ \hline L_{2k}^l & L_{2k+1}^l & S_k^{l+1} \end{array}\right) \begin{pmatrix} x_{2k}^l \\ x_{2k+1}^l \\ \overline{x}_{2k}^{l+1} \end{pmatrix} \tag{16}$$

Algorithm 2. BuildNMILUR(lev, k)

```
1: INPUT: $\mathbf{A}$, $lev = l + 1$
2: OUTPUT: $\tilde{T}$
3: if $lev = 0$ then
4:     Factor $(D_k^0)$
5: else
6:     cilk_spawn BuildNMILUR($lev - 1$, $2k$)
7:     BuildNMILUR($lev - 1$, $2k + 1$)
8:     cilk_sync
9:     Compute Schur
           $\tilde{T}_k^{lev} = S_k^{lev} - L_{2k}^{lev-1}(\tilde{D}_{2k}^{lev-1})^{-1}(U_{2k}^{lev-1}\mathbf{1}) - L_{2k+1}^{lev-1}(\tilde{D}_{2k+1}^{lev-1})^{-1}(U_{2k+1}^{lev-1}\mathbf{1})$
       where solve with $\tilde{D}_{2k}^{lev-1}$ and $\tilde{D}_{2k+1}^{lev-1}$ uses the factored form in (12)
10:    Factor($\tilde{T}_k^{lev}$)
11: end if
```

Algorithm 3. BuildNFF(lev, k)

```
1: INPUT: $\mathbf{A}$, $lev = l + 1$
2: OUTPUT: $\tilde{T}$
3: if $lev = 0$ then
4:     Factor($D_k^{lev}$)
5: else
6:     cilk_spawn BuildNFF($lev - 1$, $2k$)
7:     BuildNFF($lev - 1$, $2k + 1$)
8:     cilk_sync
9:     Compute Schur complements $\tilde{T}_k^{lev}$ for NFF as follows
           $\tilde{T}_k^{lev} = S_k^{lev} - L_{2k}^{lev-1}(\beta_{2k}^{lev-1})L_{2k+1}^{lev-1} - L_{2k+1}^{lev-1}(\beta_{2k+1}^{lev-1})L_{2k+1}^{lev-1}$
       where $\beta_{2k}^{lev-1}$ and $\beta_{2k+1}^{lev-1}$ are given by (13) and (14) respectively.
10:    Set Factor($\tilde{T}_k^{lev}$)
11: end if
```

The above computations correspond to steps (7), (8), and (9) in Algorithm 6. The computations of $D_{2k}^{l-1}x_{2k}^{l-1}$ and $D_{2k+1}^{l-1}x_{2k+1}^{l-1}$ are recursive calls to the SpMV routine. The algorithm presented above leads to spatial and temporal locality of data access while read phase of input vector and write phase of the output vector as illustrated in Figure 3. In this figure, we show that due to recursion a segment of the input and output vector fits in the L3 cache. In the subsequent phase, the data located in the L3 cache is utilized by L2 cache. Thus, we notice the data being used is nearby (spatial locality) and they are accessed frequently (temporal locality).

Algorithm 4. ForwardSweep(lev, k)

```
1: INPUT: $\tilde{T}, \mathbf{b}$, $lev = l + 1$
2: OUTPUT: $\mathbf{y}$
3: if $lev = 0$ then
4:     Solve for $y_k^0$ in $D_k^0 y_k^0 = b_k^0$
5: else
6:     cilk_spawn Solve for $y_{2k}^{lev-1}$ in $\tilde{D}_{2k}^{lev-1} y_{2k}^{lev-1} = b_{2k}^{lev-1}$
7:     Solve for $y_{2k+1}^{lev-1}$ in $\tilde{D}_{2k+1}^{lev-1} y_{2k+1}^{lev-1} = b_{2k+1}^{lev-1}$
       cilk_sync
8:     Solve for $\overline{y}_k^{lev}$ in
           $$\tilde{T}_k^{lev} \overline{y}_k^{lev} = \overline{b}_k^{lev} - L_{2k}^{lev-1} y_{2k}^{lev-1} - L_{2k+1}^{lev-1} y_{2k+1}^{lev-1}$$
9: end if
```

Algorithm 5. BackwardSweep(lev, k)

```
1: INPUT: $\tilde{T}, \mathbf{y}$, $lev = l + 1$
2: OUTPUT: $\mathbf{x}$
3: if $lev = 0$ then
4:     Set $\overline{x}_k^0 = \overline{y}_k^0$
5:     cilk_spawn Find $x_{2k+1}^0 = y_{2k+1}^0 - (D_{2k+1}^0)^{-1} U_{2k+1}^0 \overline{x}_k^0$
6:     Find $x_{2k}^0 = y_{2k}^0 - (D_{2k}^0)^{-1} U_{2k}^0 \overline{x}_k^0$
7: else
8:     Set $\overline{x}_k^{lev} = \overline{y}_k^{lev}$
9:     cilk_spawn Find $x_{2k+1}^{lev} = y_{2k+1}^{lev} - (\tilde{D}_{2k+1}^{lev})^{-1} U_{2k+1}^{lev} \overline{x}_k^{lev}$
10:    Find $x_{2k}^{lev} = y_{2k}^{lev} - (\tilde{D}_{2k}^{lev})^{-1} U_{2k}^{lev} \overline{x}_k^{lev}$
11: end if
```

3 Numerical Experiments

The numerical experiments were performed in double precision arithmetic on a quad core Intel processor i7-2820QM (SandyBridge) with 2.30GHz, 16 GB RAM and three levels of cache: L1 (32kB), L2 (256kB), and L3 (8MB). We used the Intel compiler version 12.1.4 which includes `cilk` keywords as extension to the native compiler for building the NSSOR, NMILUR, and NFF methods.

To solve the Poisson problem and the convection-diffusion problems, we used restarted GMRES with a inner subspace dimension of 500 which is kept rather high to ignore the side effects of restarts. The algorithm is stopped whenever the relative norm

$$\|\mathbf{b} - \mathbf{A}\mathbf{x}_k\| / \|\mathbf{b}\| < 10^{-7}.$$

The exact solution $\mathbf{x}^*$ is generated randomly and the right hand side $\mathbf{b}$ is set to $\mathbf{A}\mathbf{x}^*$. For partitioning and reordering, we use the 2-way ND reordering of METIS [5]. The local subdomains are factored and solved using the multifrontal sparse direct solver UMFPACK [4]. The matrix-vector products with submatrices are computed using the Sparse BLAS library [1]. The compressed row sparse (CRS)

Algorithm 6. SpMV(lev, k)

```
1: INPUT: $\mathbf{A}$, $\mathbf{x}$, $lev = l + 1$
2: OUTPUT: $\mathbf{y} = \mathbf{Ax}$
3: if $lev = 1$ then
4:     cilk_spawn $y_{2k}^{lev} = D_{2k}^{lev-1} x_{2k}^{lev-1} + U_{2k}^{lev-1} \overline{x}_k^{lev}$
5:     $y_{2k+1}^{lev} = D_{2k+1}^{lev-1} x_{2k+1}^{lev-1} + U_{2k+1}^{lev-1} \overline{x}_k^{lev}$
6: else
7:     $\overline{y}_k^{lev} = L_{2k}^{lev-1} x_{2k}^{lev-1} + L_{2k+1}^{lev-1} x_{2k+1}^{lev-1} + S_k^{lev} \overline{x}_k^{lev}$
8:     cilk_spawn $y_{2k}^{lev-1} = D_{2k}^{lev-1} x_{2k}^{lev-1} + U_{2k}^{lev-1} \overline{x}_k^{lev}$
9:     cilk_spawn $y_{2k+1}^{lev-1} = D_{2k+1}^{lev-1} x_{2k+1}^{lev-1} + U_{2k+1}^{lev-1} \overline{x}_k^{lev}$
10: end if
```

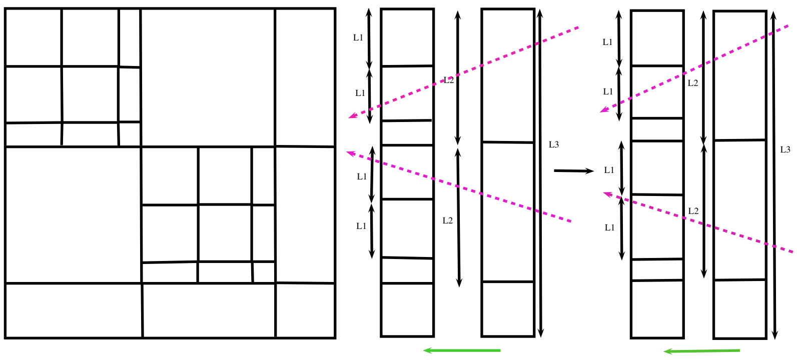


Fig. 3. Illustration of cache utilization during the read phase of the input vector and write phase of the output vector. The dotted lines indicate the data flow in cache hierarchy.

storage scheme is an efficient storage scheme for a matrix without sufficient block structure. For matrices reordered after nested dissection reordering, we have several blocks per block row. This requires maintaining separate row pointers for each of the blocks within the same row. The count of total row indices increases like $O(n\sqrt{P})$, P being the number of threads/partitions. On the other hand, for the coordinate storage format, the number of indices used to store the non-zero entries remain independent of the number of blocks. Hence, instead of the CRS, the coordinate storage format is used to store the **L**, **D**, **U**, and **T** matrices. These matrices are stored as an array of structs that encode the binary tree representation; the indices $2i$ and $2i + 1$ being the left and right subdomains of the domain indexed i. The scalar products are computed using the recently introduced elemental functions of `cilk plus` [3].

3.1 Convection Diffusion Problem

We consider the following boundary value problem

$$\begin{aligned} \operatorname{div}(\mathbf{a}(x)u) - \operatorname{div}(\kappa(x)\nabla u) &= f \ \text{ in } \ \Omega, \\ u &= 0 \ \text{ on } \ \partial\Omega_D, \\ \tfrac{\partial u}{\partial n} &= 0 \ \text{ on } \ \partial\Omega_N, \end{aligned} \tag{17}$$

where $\Omega = [0,1]^n$ ($n = 2$, or 3), $\partial\Omega_N = \partial\Omega \setminus \partial\Omega_D$. The velocity vector **a** and the tensor κ are the given coefficients of the partial differential operator. The domain is unit square in 2D and unit cube in 3D. The equation is discretized using the cell-centered finite volume scheme.

We consider following three test cases:

Poisson: Here $\kappa = 1$ and $\mathbf{a} = 0$. In Table 2 we observe that NSSOR is the most efficient preconditioner since it leads to the lowest execution time in comparison to both NMILUR and NFF. The sharp decrease in the setup time with increasing number of cores for all the methods is both due to parallelism and smaller subdomain sizes incurring lesser work with increasing number of subdomains. For one and two cores, we keep the number of subdomains equal to two, thus the iteration count remains the same, but with four subdomains using four threads, we see a significant increase in the iteration count for both NSSOR and NMILU; the iteration count for NFF increases slightly. We observe a speedup of roughly 1.5 for both NSSOR and NFF. The iteration count for NMILUR increases so much that the solve time increases for four subdomains compared to that for two subdomains.

Table 2. Number of iterations and execution times for Poisson problem on $60 \times 60 \times 60$ grid

cores	NSSOR				NMILUR				NFF			
	its	setup	solve	total	its	setup	solve	total	its	setup	solve	total
1	29	114s	15s	129s	75	116s	39s	155s	13	137s	6s	143s
2	29	65s	11s	76s	75	65s	31s	96s	13	77s	5s	82s
4	34	10s	10s	20s	135	11s	44s	55s	15	28s	4s	32s

Skyscraper Problems: For this case, the velocity $\mathbf{a} = 0$ and the tensor $\kappa(x)$ is isotropic and discontinuous as follows. The domain contains many zones of high permeability which are isolated from each other. Let $\lfloor x \rfloor$ denote the greatest integer less than x. In 3D, we have

$$\kappa(x) = \begin{cases} 10^3 * (\lfloor 10 * x_2 \rfloor + 1), & \text{if } \lfloor 10 * x_i \rfloor = 0 \bmod(2),\ i = 1, 2, 3, \\ 1, & \text{otherwise.} \end{cases}$$

In Table 3, the numerical experiments with this test case are shown. For this problem NFF converges faster than NSSOR and NMILUR. The iteration count

Table 3. Number of iterations and execution times for skyscraper problem on $60 \times 60 \times 60$ grid

cores	NSSOR				NMILUR				NFF			
	its	setup	solve	total	its	setup	solve	total	its	setup	solve	total
1	96	115s	50s	165s	75	116s	39s	155s	19	135s	10s	145s
2	106	63s	43s	76s	75	65s	31s	96s	19	77s	7s	84s
4	> 1000	9s	$> 344s$	20s	135	11s	44s	55s	44	28s	13s	41s

for NFF is lowest, however, we do see a sharp increase in the iteration count for NFF when the number of subdomains increase from 2 to 4.

Convective Skyscraper Problems: These problems are similar to the skyscraper problems, but now the velocity field is changed to $\mathbf{a} = (1000, 1000, 1000)^T$.

The numerical experiments for this test case are shown in Table (4). Once again NFF has the lowest iteration count followed by NSSOR and NMILUR. For this case, we do not see a sharp jump in the iteration count when the number of subdomains increases from 2 to 4. The setup phase of NFF is large due to the exact factorization of the subdomains. But as the number of subdomains increase, we see a sharp decrease in the setup cost. The setup times of the new methods are high because the subdomain size is large for 2-4 subdomains. The setup times are expected to decrease drastically as the numbers of subdomains increases.

Table 4. Number of iterations and execution times for convective skyscraper problem on $60 \times 60 \times 60$ grid

cores	NSSOR				NMILUR				NFF			
	its	setup	solve	total	its	setup	solve	total	its	setup	solve	total
1	81	113s	42s	155s	230	117s	131s	248s	18	134s	9s	143s
2	82	63s	33s	76s	241	65s	108s	173s	18	76s	7s	83s
4	152	9s	49s	58s	461	11s	189s	199s	21	27s	6s	33s

4 Conclusion

A class of non-overlapping domain decomposition preconditioners based on 2-way nested dissection reordering is implemented. For multithreading, we used `cilk plus` with new elemental function features. We have presented preliminary scalability results indicating that the nested filtering factorization preconditioner is a promising method. In particular, the NSSOR and NFF preconditioners can be applied to general symmetric positive definite problems. Note that for Poisson and convection-diffusion problems, geometric or algebraic multigrid could be used. However, for more general problems, the NFF preconditioner may be preferred especially if the following improvements are implemented:

– introduction of parallelism in the levels higher up the tree-for example, it is possible to spawn more threads during Schur complement computations and solve phases;
– better cache utilization by implementing a reordering based on a space filling curve for more efficient sparse matrix-vector computations.

Acknowledgement. The matrices are generated using the finite volume code written by Frédéric Nataf (University of Paris 6). We thank the referees for their careful reading and valuable remarks and suggestions that helped to improve this paper. This paper presents research results of the Belgian Network DYSCO (Dynamical Systems, Control, and Optimization), funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office.

References

1. NIST Sparse BLAS, `http://math.nist.gov/spblas/`
2. HYPRE, `http://acts.nersc.gov/hypre/`
3. Cilk plus, online documentation at `http://software.intel.com/file/40297`
4. Davis, T.A.: Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. ACM Transactions on Mathematical Software 30(2), 196–199 (2004)
5. METIS graph partitioner, `http://glaros.dtc.umn.edu/gkhome/metis/metis/download`
6. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. In: International Conference on Parallel Processing, pp. 113–122 (1995)
7. Karypis, G., Kumar, V.: Analysis of Multilevel Graph Partitioning. In: Supercomputing (1995)
8. Karypis, G., Kumar, V.: Multilevel k-way Partitioning Scheme for Irregular Graphs. J. Parallel Distrib. Comput. 48(1), 96–129 (1998)
9. Grigori, L., Kumar, P., Nataf, F., Wang, K.: A class of multilevel parallel preconditioning strategies. HAL Report RR-7410 (2010) (2010), `http://hal.inria.fr/inria-00524110_v1/`
10. George, J.A.: Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis 10(2), 345–363 (1973)
11. Kumar, P.: A class of preconditioning techniques suitable for partial differential equations of structured and unstructured mesh. PhD thesis (2010), `http://www.sudoc.fr/147701619`
12. Weiler, W., Wittum, G.: Parallel Frequency Filtering. Computing 58, 303–316 (1997)
13. Wagner, C.: Tangential frequency filtering decompositions for symmetric matrices. Numer. Math. 78(1), 119–142 (1997)
14. Wagner, C.: Tangential frequency filtering decompositions for unsymmetric matrices. Numer. Math. 78(1), 143–163 (1997)
15. Wittum, G.: Filternde Zerlegungen-Schnelle Loeser fur grosse Gleichungssysteme. Teubner Skripten zur Numerik Band 1. Teubner-Verlag, Stuttgart (1992)

16. Axelson, O., Polman, B.: A robust preconditioner based on algebraic substructuring and two level grids. In: Hackbusch, W. (ed.) Robust multi-grid methods, NNFM, Bd.23. Vieweg-Verlag, Braunschweig
17. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS publishing company, Boston (1996)
18. Kettler, R.: Analysis and comparison of relaxation schemes in robust multigrid and preconditioned conjugate gradient methods. In: Multigrid Methods. Lecture notes in mathematics, vol. 960, pp. 502–534 (1982)
19. Notay, Y.: AGMG: Agregation based AMG, `http://homepages.ulb.ac.be/~ynotay`

An Approach of the QR Factorization for Tall-and-Skinny Matrices on Multicore Platforms

Sergey V. Kuznetsov[*]

Intel Corporation, Software and Services Group, Novosibirsk, Russia
sergey.v.kuznetsov@intel.com

Abstract. In this paper we focus primarily on a technique used to parallelize the LAPACK QR factorization of tall-and-skinny matrices. The modifications of the panel QR factorization we suggest neither affect the accuracy nor increase memory consumption. Results for tall-and-skinny matrices on the Intel® Xeon® platforms, and comparisons between the Intel® Math Kernel Library (Intel MKL) QR, PLASMA QR and the method proposed are provided.

1 Introduction

Since the number of cores in modern high-performance computers has increased significantly, it is very important to develop algorithms that can be efficiently parallelized, and thus exploit the capabilities of modern computers. The most difficult problem for parallelizing linear algebra algorithms is to efficiently balance the load of all cores. Aiming to achieve scalability on multi-core platforms, in this paper we focus primarily on a technique used to parallelize the LAPACK QR factorization of tall-and-skinny matrices. The modifications of the panel QR factorization we suggest neither affect the accuracy nor increase memory consumption.

The standard LAPACK QR implementation is based on a block version of the algorithm [2] that can exploit parallelism coming from highly optimized BLAS available in Intel Math Kernel Library or its analogues. It is very efficient in a serial mode and shows some speed-up on multi-core computers when it exploits parallelism coming from BLAS. Due to the implementation, the panel QR factorization contains many synchronization points and the computation of Householder vector is one of the bottlenecks for the parallelization of the panel QR factorization.

In [6], a parallel tiled QR factorization algorithm has been proposed. The operations in the tiled QR algorithm are represented as a sequence of small tasks that operate on square blocks of data. However the storage format of output matrices in the tiled QR is different from the standard implementation and the memory for storing Householder transformations is two times higher than for the standard QR implementation [2]. In addition, the tiled QR factorization requires higher computational cost. A parallel tiled QR factorization algorithm has been recently implemented as a part of the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) project [3].

[*] Address: Intel A/O, 6/1 pr. ak. Lavrentieva, Novosibirsk, 630090, Russia.

In this paper, we describe a modification of the standard panel QR factorization that can be applied for any matrix. The algorithm allows us to keep the LAPACK standard interface, provide better performance in sequential mode and get scalability for tall-and-skinny matrices. Due to better performance characteristics the algorithm described in the paper can be used as a building block for further acceleration of other existing QR implementations.

The basic operations utilized by the standard panel QR factorization are the computation and application of Householder transformations. At each step of the panel QR factorization the following two tasks are performed: construction of a Householder vector and application of the Householder transformation to the trailing matrix. The trailing matrix update involves two operations: a matrix-vector multiplication with a Householder vector and a rank one update of the trailing matrix. The accumulation of the Householder reflections into a compact WY form in the standard panel QR factorization is deferred until they have all been generated. The accumulation is a recursive computation of columns for triangular matrix T which also involves a matrix vector multiplication with the Householder vector followed by a triangular matrix-vector multiplication. Since the accumulation of Householder transformations and the trailing matrix update use the same Householder vector, these two matrix vector multiplications can be merged into one matrix-vector multiplication.

Moreover, observe that a Householder vector v that reduces a vector x to a multiple of the first standard basis vector is just a scaling of x except for the first component. Thus, a matrix-vector multiplication involving v can be easily recovered from a matrix-vector multiplication with x. This makes it possible to perform the most computational work for a matrix-vector multiplication involving a Householder vector v before or simultaneously with the 2-norm of the Householder vector. The next important observation is that the 2-norm of the Householder vector can be also computed with the help of a matrix-vector multiplication. Combining these two observations we can state that the computation of the 2-norm of the Householder vector v computed with the help of a matrix-vector multiply can be overloaded with other useful computations needed for the trailing matrix update and computation of the next column of the matrix T by using components of the vector x.

A similar algorithm has been used by the author for accelerating ScaLAPACK reduction to bidiagonal form [5]. In [5], construction of a Householder vector and its application to the trailing matrix were only merged but accumulation of the Householder transformations into a compact WY form was computed by a separate routine such as a standard LAPACK bidiagonal reduction routine. The modifications of the algorithm described in this paper allow merging all three key steps of QR factorization, so it has better performance characteristics than the algorithm from [5]. In addition, the question of providing the same level of the accuracy as in the standard LAPACK QR factorization are considered in this paper.

In a parallel environment, our approach is based on a block row partitioning of a given matrix A of size n by $m (n \gg m)$. Our parallel QR factorization of A involves m stages. In stage i, each thread performs a matrix-vector multiply by using a submatrix. Let the vector z_j of the length m be the result of the local matrix-vector multiply for the j-th thread. The next step is the reduction $z := z + z_j$ over all threads.

When the master thread finishes computations of two key parameters of Householder transformations, all threads start their updates of the trailing submatrix independently.

The paper is organized as follows. In section 2 we consider a representation of Householder reflections that permits us to work directly with components of a vector. Section 2 describes how the representation along with the speculative computation of the 2-norm of the Householder transformation can be combined with useful computations needed for the panel update and forming the next column in the triangular matrix T in VTV^T representation. Results for tall-and-skinny matrices on Intel Xeon processors[1], and comparisons between the Intel MKL QR form MKL 11.0 Update 1 which extensively uses our sequential and parallel implementations of the panel QR factorization, PLASMA QR and our implementation of the recursive RGEQR3 algorithm from [9] combined with our parallel panel factorization are provided in section 6.

We follow the notational convention used in numerical analysis, In particular, $\|\cdot\|_2$ denotes the 2-norm for both vectors and matrices, and e_i denotes the i-th standard basis vector in R^n. Throughout this paper a_{ij} is the a_{ij} -element of the matrix A, the symbols $[A]_{*,j}$ and $[A]_{i,*}$ denote the j-th column and the i-th row of A, respectively. A submatrix composed of the i-th to the j-th columns of matrix A is denoted by $[A]_{*,i:j}$, and $[A]_{i:j,k}$ denotes a sub-vector composed of the i-th to j-th elements of the k-th column of A. Let $x = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$ be a vector. A sub-vector composed of the i-th to j-th components of x is denoted by $[x]_{i:j}$. The symbol $\{x\}_{i:k}$ denotes the vector of length $n,\ n \geq k$ defined as:

$$\{x\}_{i:k} = (0 \ \dots \ 0 \quad x_i \ \dots \ x_k \quad 0 \ \dots \ 0\ )^T.$$

In other words, the j-th component of $\{x\}_{i:k}$ is equal to the j-th element of x if $i \leq j \leq k$ and 0 otherwise.

2 Definitions and Preliminaries

The basic operations utilized by the QR factorization of an n-by-l matrix A are the computation and application of Householder transformations.

Definition 2.1 Given a vector $x \in R^n$, one can find an Householder transformation $H_k(x)$ of order n so that

$$H_k(x)x = (x_1 \quad x_2 \quad \cdots \quad x_{k-1} \quad \beta_k(x) \quad 0 \quad \dots \quad 0)^T \tag{1}$$

The transformation $H_k(x)$ is defined by the following formula:

$$H_k(x) = I - \tau_k(x) v_k(x) v_k^{\mathrm{T}}(x)$$

where the components of the Householder vector $v_k(x)$ and the scalar $\tau_k(x)$ are computed as follows:

[1] Intel, the Intel logo, Xeon and Xeon Inside are trademarks of Intel Corporation in the U.S. and/or other countries.

$$\begin{aligned} \beta_k(x) &= -sign(x_k)\|[x]_{k:n}\|_2 \\ r_k(x) &= 1/(x_k - \beta_k(x)) \\ \tau_k(x) &= 1 - x_k/\beta_k(x) \end{aligned} \tag{2}$$

$$\begin{aligned} v_k(x) = (0 \quad \dots \quad 0 \quad 1 \ r_k(x)x_{k+1} \quad r_k(x)x_{k+2} \quad \dots \quad r_k(x)x_n) \\ = \quad e_k + r_k(x)\{x\}_{(k+1):n} \end{aligned}$$

Remark 1

$$\|[x]_{k:n}\|_2 = \sqrt{[x]_{k:n}^{\mathrm{T}} x_{k:n}} = \sqrt{x_k^2 + [x]_{(k+1):n}^{\mathrm{T}} [x]_{(k+1):n}} \tag{3}$$

It should be stressed that the computation of $\|[x]_{k:n}\|_2$ is a bottleneck for parallelization of panel QR factorization. The main idea of the approach described in the paper is to compute the 2-norm of $[x]_{k:n}$ as a part of matrix-vector multiplication and perform some additional useful computations while computing this key parameter.

To enhance the performance, the standard QR factorization uses a technique called *blocking*. The technique means, having chosen a block size l, a given matrix A of size n-by-m is split into vertical panels $A_1, A_2, \dots, A_p$ where A_i is a panel of A such that $A_i = [A]_{*,(i-1)*l+1:\, i*l}$. For each panel we generate the Householder transformations $H_{(i-1)*l+1}, H_{(i-1)*l+2}, \dots, H_{i*l}$ such that

$$H_{(i-1)*l+1} \ H_{(i-1)*l+2} \cdots H_{i*l} A_i = R_i$$

where R_i is an upper trapezoidal matrix: $r_{kj}^{(i)} = 0$ if $k > j + (i-1)*l$.

Each Householder transformation is immediately applied to an unreduced part of a panel as follows:

$$H_k A_i = \left(I - \tau_k v_k v_k^{\mathrm{T}}\right) A_i = A_i - \tau_k v_k v_k^{\mathrm{T}} A_i \tag{4}$$

but applying these transformations to the rest of the panels is deferred until they have all been generated.

The product of a sequence of the Householder transformations $H_{(i-1)*l+1} \ H_{(i-1)*l+2} \cdots H_{i*l}$ of order n is accumulated within each panel and it is represented in the VTV^{T} form (see, for example, [4]) as:

$$H_{(i-1)*l+1} \ H_{(i-1)*l+2} \cdots H_{i*l} = I - V_i \ T_i \ V_i^{\mathrm{T}}$$

where T_i is an l-by-l upper triangular matrix. After the Householder transformations are put in VTV^{T} form, they are applied to the rest of the panels as:

$$(A_{i+1} \mid \ \dots \ \mid \ A_p) := (I - V_i \ T_i V_i^{\mathrm{T}})(A_{i+1} \mid \ \dots \ \mid \ A_p) \tag{5}$$

The standard QR factorization is constructed based on the three following elementary steps: DGEQR2, DLARFT and DLARFB.

DGEQR2: This LAPACK subroutine performs the unblocked QR factorization of an n-by-l panel A_i. The subroutine is based on the following algorithm:

Algorithm 1.

```
For $k = (i-1) * l + 1, i * l$
    Call LAPACK subroutine DLARFG to generate a real Householder
     transformation $H_k([A]_{*,k})$ such that

        $H_k([A]_{*,k})[A]_{*,k} = (r_{1k} \quad r_{2k} \quad \cdots \quad r_{kk} \quad 0 \quad \ldots \quad 0)^T$

    Call LAPACK subroutine DLARF to apply the Householder transformation
     $H_k([A]_{*,k})$ to the unreduced part of A:

        $[A]_{*,(k+1):i*l} := H_k([A]_{*,k})[A]_{*,(k+1):i*l}$

End for
```

DLARFT: This LAPACK subroutine forms a triangular matrix T_i and a rectangular matrix V_i of size n-by-l, which is defined as a product of l Householder transformations $H_{(i-1)*l+1}\ H_{(i-1)*l+2} \cdots H_{i*l}$. This subroutine is always called after DGEQR2.

Remark 2. In LAPACK's DGEQRF subroutine the arrays V and R defined as

$$V = (V_1 \mid V_2 \mid \ldots \mid V_p),$$

$$R = (R_1 \mid R_2 \mid \ldots \mid R_p)$$

do not need extra space to be stored since they overwrite the matrix A. More precisely, the elements on and above the diagonal of A contain the m-by-m upper triangular matrix R; the elements below the diagonal of A contain the components of vectors $v_1, v_2, \ldots, v_m$ where each the vector v_i represents, along with τ_i, the Householder transformation H_i.

DLARFB: This LAPACK subroutine uses matrices V_i and T_i computed by subroutines DGEQR2 and DLARFT for updating unreduced panels of A according to (5).

3 Panel QR Factorization

In this section we demonstrate how to combine generation of the Householder transformation with useful computations needed for a follow-up trailing matrix update and forming next column of a triangular matrix V in the VTV^T representation. This allows us to reduce the idle time at all steps of the panel QR factorization. Another advantage is that the number of cache misses is significantly reduced since we do not need to reload the Householder vectors for computing the VTV^T representation.

Assume that one needs to find a Householder transformation

$$H_k(x) = I - \tau_k(x) v_k(x) v_k^{\mathrm{T}}(x)$$

of order n so that (1) holds, apply the Householder transformation to an n-by-$(m-k)$ matrix C and compute the VTV^{T} representation of the product $H_1 H_2 \cdots H_{k-1} H_k(x)$, and that the product $H_1 H_2 \cdots H_{k-1}$ has been already defined by

$$H_1 H_2 \cdots H_{k-1} = I - V_{k-1} T_{k-1} V_{k-1}^{T}$$

where T_{k-1} is $(k-1)$-by-$(k-1)$ upper triangular and V_{k-1} is an n-by-$(k-1)$ matrix.

In the first instance we consider the rank one matrix update of C. Taking into account the representation (2) of the Householder vector $v_k(x)$

$$v_k(x) = e_k + r_k(x)\{x\}_{(k+1):n}^{\mathrm{T}} \tag{6}$$

the product $H_k(x)C$ can be written in the form:

$$\begin{aligned} C \coloneqq H_k(x)C &= \left(I - \tau_k(x) v_k(x) v_k^{\mathrm{T}}(x)\right) C \\ &= C - \tau_k(x) v_k(x) \left(e_k^{\mathrm{T}} C + r_k(x)\{x\}_{(k+1):n}^{\mathrm{T}} C\right) \\ &= C - \tau_k(x) v_k(x)\left([C]_{k,*} + r_k(x)\{x\}_{(k+1):n}^{\mathrm{T}} C\right) = C + v_k(x)\, w_k(x) \end{aligned} \tag{7}$$

where

$$w_k(x) = -\tau_k(x)([C]_{k,*} + r_k(x)\,\{x\}_{(k+1):n}^{\mathrm{T}} C) \tag{8}$$

We are now going to demonstrate that the two matrix-vector multiplications used for the accumulation of Householder transformations and the trailing matrix update can be merged into one matrix-vector multiplication. As it is proven in [4], the product

$$\begin{aligned} &H_1 H_2 \cdots H_{k-1} H_k(x) = \\ &= \left(I - V_{k-1} T_{k-1} V_{k-1}^{\mathrm{T}}\right)\left(I - \tau_k(x) v_k(x) v_k^{\mathrm{T}}(x)\right) \end{aligned}$$

can be written in the form

$$\begin{aligned} &\left(I - V_{k-1} T_{k-1} V_{k-1}^{\mathrm{T}}\right)\left(I - \tau_k(x) v_k(x) v_k^{\mathrm{T}}(x)\right) \\ &= I - \begin{pmatrix} V_{k-1} & v_k(x) \end{pmatrix} \begin{pmatrix} T_{k-1} & -\tau_k(x)\, T_{k-1} V_{k-1}^{\mathrm{T}}\, v_k(x) \\ 0 & \tau_k(x) \end{pmatrix} \begin{pmatrix} V_{k-1}^{\mathrm{T}} \\ v_k^{\mathrm{T}}(x) \end{pmatrix} \\ &= I - V_k T_k V_k^{\mathrm{T}} \end{aligned}$$

where the matrices T_k and V_k are defined by:

$$T_k = \begin{pmatrix} T_{k-1} & T_{k-1}\, z_k^{\mathrm{T}}(x) \\ 0 & \tau_k(x) \end{pmatrix}$$

$$V_k = (V_{k-1} \quad v_k(x)) \ ,$$

$$z_k(x) = \ -\tau_k(x) v_k^{\mathrm{T}}(x) V_{k-1}$$

In view of (2), we have:

$$\begin{aligned} z_k(x) &= \ -\tau_k(x)\, v_k^{\mathrm{T}}(x) V_{k-1} \\ &= -\tau_k(x) \left(e_k^{\mathrm{T}} + r_k(x)\, \{x\}_{(k+1):n}^{\mathrm{T}} \ \right) V_{k-1} \\ &= -\tau_k(x)\ (\ [V_{k-1}]_{k,*} + r_k(x)\, \{x\}_{(k+1):n}^{\mathrm{T}} V_{k-1}) \end{aligned} \tag{9}$$

It is easy to see that the two matrix-vector multiplications $\{x\}_{(k+1):n}^{\mathrm{T}} V_{k-1}$ and $\{x\}_{(k+1):n}^{\mathrm{T}}\, C$ can be merged into one matrix-vector multiplication defined by

$$\{x\}_{(k+1):n}^{\mathrm{T}} (V_{k-1} \quad C)$$

Let us show how computation of the 2-norm of the Householder vector can be combined with the latter matrix-vector multiplication. To demonstrate this we consider the matrix B of size n-by-m defined as follows:

$$B = (V_{k-1} \quad x \quad C) \tag{10}$$

Compute the vector y of length m by the formula

$$\begin{aligned} y &= \{x\}_{(k+1):n}^{\mathrm{T}} B \\ &= (\{x\}_{(k+1):n}^{\mathrm{T}} V_{k-1} \ \mid \ \{x\}_{(k+1):n}^{\mathrm{T}} \{x\}_{(k+1):n} \ \mid \ \{x\}_{(k+1):n}^{\mathrm{T}} C) \end{aligned} \tag{11}$$

It is easy to see that $\|[x]_{k:n}\|_2 = \sqrt{x_k^2 + y_k}$ due to (3). The 2-norm of the Householder vector gives us possibility to compute the key parameters $r_k(x)$ and $\tau_k(x)$ according to the equations (2).

Let us update the components of the vector y as

$$\begin{aligned} y &:= [B]_{k,*} + r_k(x) y \\ &= ([V]_{k,*} + r_k(x)\, \{x\}_{(k+1):n}^{\mathrm{T}} V_{k-1} \ \mid \ y_k \ \mid \ [C]_{k,*} + r_k(x)\, \{x\}_{(k+1):n}^{\mathrm{T}}\, C\ ) \end{aligned} \tag{12}$$

and then scale the components of y to multiply each component by $-\tau_k(x)$: $y := \ -\tau_k(x) y$.

Taking into account (8) and (9), we can state that

$$[y]_{1:(k-1)} = z_k(x)$$

$$[y]_{(k+1):n} = w_k(x)$$

Thus one matrix-vector multiply as in (11), one BLAS DAXPY operation defined by (12) and one call to DSCAL allow us to start computation of the k-th column in the triangular matrix T_k according to (9), and apply the Householder transformation $H_k(x)$ to the matrix C where C is updated by (7)-(8).

Combining all these observations we get sequential Algorithm 2 – a program named DGEQR2_RFT to perform all three tasks mentioned at the beginning of the section. We assume that the matrix B defined by $B = (V_{k-1} \quad x \quad C)$ has already been formed and is an input argument for the algorithm. The matrix T and a scalar integer k are other input parameters. The algorithm modifies matrices T and B.

Algorithm 2. DGEQR2_RFT$(k,\ B,\ T)$

```
*Set* $[x]_{k:n} = [B]_{k:n,k}$ , $[x]_{1:(k-1)} = 0$
*Call DGEMV to compute*

        $y = \{x\}^T_{(k+1):n}B = [x]^T_{(k+1):n}[B]_{(k+1):n,1:m}$

*Compute* $r_k(x)$, $\beta_k(x)$ *and* $\tau_k(x)$ *by (2)*
*Call DAXPY to update components of* $y$:

        $y := [B]_{k,*} + r_k(x)y$

*Call DSCAL to compute* $y := -\tau_k(x)y$
*If k > 1, call DTRMV to compute the* $k$*-th column of* $T$ *by*

        $[T]_{1:(k-1),k} = [T]_{1:(k-1),1:(k-1)}[y]^T_{1:(k-1)}$

*Set* $t_{kk} = \tau_k(x)$, $v_k = 1$
*Call DSCAL to compute components of the vector* $v$:

        $[v]_{(k+1):n} = r_k(k)[x]_{(k+1):n}$

*Overwrite* $[B]_{(k+1):n,*} = [x]_{(k+1):n}$ *by* $[v]_{(k+1):n}$.
*Call DGER to perform a rank 1 operation*

        $[B]_{k:n,(k+1):m} \coloneqq [B]_{k:n,(k+1):m} + [v]_{k:n}[y]_{(k+1):m}$

*Set* $b_{kk} = x_k = \beta_k(x)$
```

Let us now come back to the standard QR factorization of the panel A . Assume that the computation of the QR factorization of A has progressed to where column k is being reduced and we have determined Householder transformations $H_1, H_2, \dots, H_{k-1}$ such that

$$A^{(k)} = H_1\,H_2 \cdots H_{k-1}A = \begin{pmatrix} R_{k-1} & A^{(k)}_{12} \\ 0 & A^{(k)}_{22} \end{pmatrix}$$

where R_{k-1} is upper triangular matrix. In LAPACK's DGEQRF subroutine the elements below the diagonal of A contain the components of vectors $v_1, v_2, \ldots, v_{k-1}$ where each vector v_j, along with τ_j, represents the Householder transformation H_j. Since matrix $A^{(k)}$ has the same structure as the matrix B defined by (10) and the k-th step of the panel QR factorization of A requires to perform the same three tasks considered in the section, it is easy to prove that Algorithm 3 produces the QR factorization of a given n-by-m matrix.

Algorithm 3. DGEQRF_Panel_Factorization(A, T)

```
For k=1, m
    Call DGEQR2_RFT (k, A, T)
End for
```

4 Accuracy

The computation of the 2-norm of a subvector with the help of matrix-vector multiplication is not reliable since it may lead to an unnecessary overflow and underflow of floating point numbers. For example, if x_k and y_k are so small that their squares underflow, then $\beta_k(x) = \sqrt{x_k^2 + y_k}$ from Algorithm 2 will be zero even though the true value of $\beta_k(x)$ is not zero. Because of that we are going to modify Algorithm 2 to avoid unnecessary overflow and underflow.

The standard QR panel factorization uses the DLARFG LAPACK subroutine for computing elements of the Householder transformation. First of all, DLARFG calls the BLAS Level 1 DNRM2 function for computing the 2-norm of a vector. The value of DNRM2 is then checked whether it is in IEEE 754 double precision range, and if the computed value is out of the range, additional scaling is applied to a vector to avoid underflow. More precisely, the DLARFG LAPACK subroutine uses additional scaling of a vector if the DNRM2 BLAS function returns value less than $\varepsilon_{min} = \frac{sfmin}{the\ relative\ machine\ precision}$ where $sfmin$ is the safe minimum such that $\frac{1}{sfmin}$ does not overflow.

There are many ways to avoid overflow and underflow in Algorithm 2. For example, we can compute the infinity norm of a vector $\|x\|_\infty = \max |x_i|$ before the matrix-vector multiplication and then we can use matrix-vector multiplication or use the DLARFG routine. However, computation of the infinity norm increases the idle time in parallel mode. Another way is to compute y_k with the help of a matrix-vector multiplication, analyze its value and call the DLARFG LAPACK subroutine if y_k is out of the following predefined range: $[\varepsilon_{min}, \varepsilon_{max}]$ where ε_{max} is less or equal to the double precision overflow threshold. In other words, the 2-norm of a vector is speculatively computed. It is evident that this approach can increase computational cost, but it is preferable to performing obligatory computation of the 2-norm or infinity norm since additional computation is only needed in a few cases. In some cases the LAPACK computational drivers, such as least square drivers, perform additional scaling (see, for example, the DLASCL auxiliary subroutine) before calling QR factorization drivers, and this reduces the additional computational cost.

We give pseudo code below for the panel QR factorization which uses speculative computation of the 2-norm and avoids overflow and underflow.

Algorithm 4. DGEQR2_RFT(A, T)

```
For k=1, m
    Set $[x]_{k:n} = [A]_{k:n,k}$ , $[x]_{1:(k-1)} = 0$
    Call DGEMV to compute
            $y = \{x\}^T_{(k+1):n} A = [x]^T_{(k+1):n} [A]_{(k+1):n,1:m}$
    If $y_k \le \varepsilon_{min}$ or $y_k \ge \varepsilon_{max}$ then
            Call DLARFG to compute scalars $\tau_k(x)$, $r_k(x)$, $\beta_k(x)$ and
             the Householder vector $v$ for $H_k([A]_{*,k})$
            Call DGEMV to compute
                    $y = [v]^T_{k:n} [A]_{k:n,1:m}$
    Else
            Compute $r_k(x)$, $\beta_k(x)$ and $\tau_k(x)$ by (2)
            Call DAXPY to update components of $y$:
                    $y := [A]_{k,*} + r_k(x) y$
            Call DSCAL to compute components of the vector
                    $[A]_{(k+1):n,k} = r_k(x) [A]_{(k+1):n,k}, A_{kk} = 1$
    End if
        Call DSCAL to compute $y := -\tau_k(x) y$
    If $k > 1$, call DTRMV to compute the $k$-th column of $T$ by
            $[T]_{1:(k-1),k} = [T]_{1:(k-1),1:(k-1)} [y]^T_{1:(k-1)}$
    Set $t_{kk} = \tau_k(x)$
    Call DGER to perform the rank one operation
            $[A]_{k:n,(k+1):m} := [A]_{k:n,(k+1):m} + [v]_{k:n} [y]_{(k+1):m}$
    Set $a_{kk} = \beta_k(x)$ and $t_{kk} = \tau_k(x)$
End for
```

5 Parallel Implementation

In this section we discuss parallel implementation of Algorithm 4 for the panel QR factorization assuming that the number of threads is small.

There are three Level 2 BLAS basic operations in Algorithm 4: matrix-vector multiplication performed by DGEMV, triangular matrix vector multiplication performed by DTRMV and the rank 1 matrix update performed by DGER. It is well known that Level 2 BLAS computations are memory bandwidth limited and parallel implementations of DGEMV or DGER demonstrates poor scalability in parallel mode on platforms like Intel Pentium® processors or Intel Core™2 Quad processors with high ratios of peak performance vs. memory bandwidth. Architecture systems with the Intel Core™ i5 or i7 processors and their latest 2nd and 3rd generations bring a whole new level of memory bandwidth to high performance computing. As we demonstrate in the next section, the recent improvement of memory bandwidth makes it possible to develop a scalable implementation of the panel QR factorization for tall-and-skinny matrices.

Theoretically the panel QR factorization described in details in the previous section can be parallelized for any distribution of a matrix: by rows, by columns or using mixed distribution. In what follows a block row partitioning of a given panel A of dimension n-by-m $(n \gg m)$ is the only case considered. Let p be the number of threads assigned for the QR factorization of A. We consider rectangular blocking, where A is broken into an n_b-by-m block matrix with p blocks $A^{(i)}$ where $n_b = n/p$:

$$A = \begin{pmatrix} A^{(1)} \\ A^{(2)} \\ \vdots \\ A^{(p)} \end{pmatrix}$$

The panel QR factorization of A involves m stages. At stage i, each thread performs a matrix-vector multiplication using its own submatrix $A^{(i)}$. Let the vector z_j of length m be the result of the local matrix-vector multiply for the j-th thread. The next step is the reduction $z \coloneqq z + z_j$ over all threads. When the master thread finishes computation of the key parameters r_i and τ_i of Householder transformations, all threads start their updates of the trailing submatrix independently.

Thus the simplest algorithm one might try consists in the following steps:

Algorithm 5. Parallel panel factorization($A,\ T$)

```
Let $n_b$ be the number of rows in a block, $k_i$ be the first row in the block $A^{(i)}$, $l_i$
be the last row in the block $A^{(i)}$ and $p$ be the number of threads.
For $k = 1,\ m$
    Set $[x]_{(k+1):n} = [A]_{(k+1):n,k}$, $[x]_{1:(k-1)} = 0$
    $i = omp\_get\_thread\_num() + 1$
    $k_i = k$
    If $i > 1$ then $k_i = (i-1) * n_b + 1$
```

```
        $l_i = i * n_b$
        Call DGEMV to compute $z_i = \{x\}^T_{k_i:l_i} A^{(i)}$
        If $i == 1$ then
            Set $y = z_1$
            For $j = 2, p$
                $y := y + z_j$
            End for
            If $\varepsilon_{min} \le y_k \le \varepsilon_{max}$ then
                Compute $r_k(x)$, $\beta_k(x)$ and $\tau_k(x)$ by (2).
                Call DAXPY to update components of $y$:
                    $y := [A]_{k,*} + r_k(x)\, y$
                Call DSCAL to compute the vector $v$ :
                    $[v]_{(k+1):n} = r_k(x)[x]_{(k+1):n}$
                Overwrite $[A]_{(k+1):n,k} = [x]_{(k+1):n}$ by $[v]_{(k+1):n}$.
            Else
                Call DLARFG to compute the vector $[v]_{k:n}$ and scalars $r_k(x)$,
                $\beta_k(x)$ and $\tau_k(x)$
                Call DGEMV to compute $y = [v]^T_{k:n}[A]_{k:n,1:m}$
            End if
            Call DSCAL to compute $y := -\tau_k(x) y$
            If $k > 1$, call DTRMV to compute the k-th column of $T$ by
                $[T]_{1:(k-1),k} = [T]_{1:(k-1),1:(k-1)}[y]^T_{1:(k-1)}$
        End if
        Call DGER to perform 1 rank operation
            $[A^{(i)}]_{k_i:l_i,(k+1):m} := [A^{(i)}]_{k_i:l_i,(k+1):m} + [v]_{k_i:l_i}[y]_{(k+1):m}$
        If $i == 1$ then set $a_{kk} = \beta_k(x)$, $t_{kk} = \tau_k(x)$
    End for
```

6 Performance Results

Two different implementations of the QR factorization with our approach for tall-and-skinny matrices referred further as TSQR are used for performance comparison in this section. The first program is our own implementation of the recursive RGEQR3 algorithm from [9] which calls our implementation of Algorithm 5 instead of DGEQR2 and DLARFT. The internal column block size in the implementation of RGEQR3 was set to 64. A parallel matrix-vector multiplication and trailing matrix update are only used for the implementation of Algorithm 5. The triangular matrix-vector product and scaling of a vector are performed only by the master thread in a team since parallelization of these operations does not provide any significant benefit.

The sequential and parallel implementations of our approach have been recently incorporated into Intel MKL 11.0 to increase performance of xGEQRF routines. The implementation of QR factorization in Intel MKL is similar to PLASMA implementation. It also uses a DAG of tasks and these tasks are also executed asynchronously. Unlike the PLASMA, only block panels instead of blocks are used in Intel MKL xGEQRF implementation and each task can be performed by a small team of threads or sequentially. At the moment the sequential version is used inside of Intel MKL on Intel 64 platforms. For a large tall-and-skinny matrices considered in this section, the new sequential TSQR allowed increasing the performance of Intel MKL DGEQRF in 1.6-3 times.

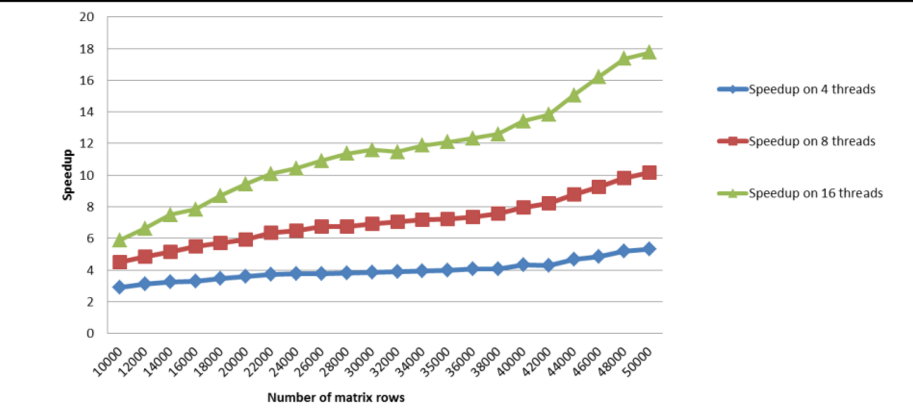
To get the best performance with PLASMA, we followed the recommendations described in [10] and so called "pruned search" from [10] was used by us for finding the *(NB, IB)* pairs that maximize the performance depending on the matrix size and on the number of cores. The following limited set of *(NB, IB)* pairs was selected: {(40, 10), (60, 20), (84, 28), (120, 40), (168, 56), (200, 40), (256, 64)}. Then we benchmark the PLASMA QR factorization only with this limited number of combinations and finally the best performance obtained is selected. The time for conversion from the Column Major Format to Block Data Layout in the case of PLASMA was not measured in order to make fair comparisons.

The first figure demonstrates good scalability of the parallel implementation of Algorithm 5 especially for large numbers of rows where the speedup of a parallel execution on q cores is computed by the following formula:

$$s_q = t_1/t_q$$

where q is the number of cores, t_1 the execution time of the sequential algorithm and t_q the execution time of the parallel algorithm when q cores are used. There was a serious performance drop of the matrix-vector multiplication performance in serial mode for large matrices when the memory needed for storing a panel exceeded the size of cache memory. Due to this circumstance the scalability factor for large sizes is larger than expected.

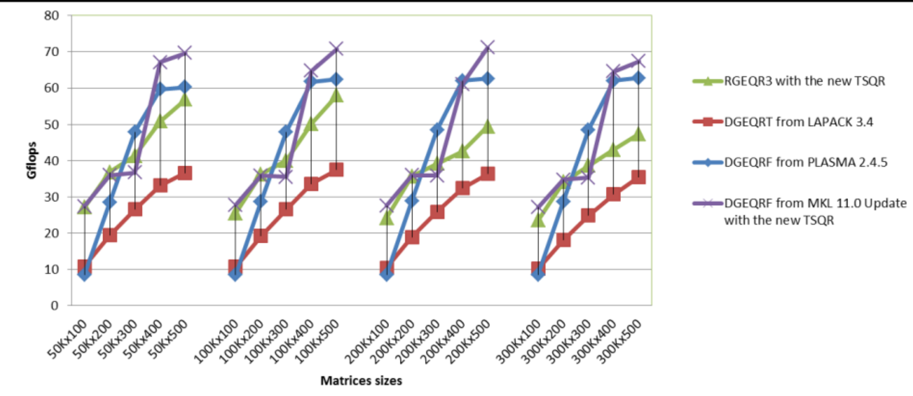
The next figure shows the performance comparison of our two implementations mentioned before with PLASMA_DGEQRF from PLASMA 2.4.5 and DGEQRT from LAPACK 3.4 for tall-and-skinny random matrices of different sizes. The LAPACK 3.4 DGEQRT subroutine is based on another algorithm from [9]. The figure 2 shows that the recursive RGEQR3 algorithm with our parallel TSQR is faster than DGEQRT. It also shows that DGEQRF form MKL 11.0 Update 1 is a bit faster than PLASMA_DGEQRF for this set of matrices except the matrix with 300 columns.



Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to www.intel.com/performance/resources/benchmark_limitations.htm.
* Other brands and names are the property of their respective owners

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Fig. 1. Scalability of the panel QR factorization on Intel Xeon processor E5-2680 (2.7 GHz, RAM 32 GB, Max Memory Bandwidth 51.3 GB/s), with 64 columns



Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to www.intel.com/performance/resources/benchmark_limitations.htm.
* Other brands and names are the property of their respective owners

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice

Fig. 2. Performance comparison of Intel MKL 11.0 Update 1 with the new TSQR inside, PLASMA_DGEQRF from PLASMA 2.4.5 and DGEQRT from LAPACK 3.4 on Intel Xeon processor E5-2680 (2.7 GHz, RAM 32 GB, Max Memory Bandwidth 51.3 GB/s). The number of threads is 16.

Acknowledgments. The author is grateful to Jeanette Feldhousen, Sergey Gololobov, Victor Kostin and Alexander Kobotov for helpful remarks and fruitful discussions of the paper. Thanks to the anonymous referee for suggestions that helped improve the paper.

References

1. Intel Math Kernel Library (Intel MKL), `http://www.intel.com/software/products/mkl/`
2. Anderson, E., Bai, Z., Bischof, C., Blackford, L., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. SIAM (1992)
3. Agullo, E., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Langou, J., Ltaief, H.: PLASMA UsersGuide. Technical report, ICL, UTK (2009)
4. Golub, G.H., Van Loan, C.F.: Matrix Computation. In: John Hopkins Studies in the Mathematical Sciences, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
5. Kuznetsov, S.: Orthogonal reduction of dense matrices to bidiagonal form on computers with distributed memory architectures. In: Parallel Computing, vol. 24/2, pp. 305–313. Elsevier Science B.V. (1998)
6. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: Parallel Tiled QR Factorization for Multicore Architectures. UT-CS-07-598. LAPACK Working Note 190 (July 2007)
7. Dongarra, J., Hammarling, S., Sorensen, D.: Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations, ANL, MCS-TM-99. LAPACK Working Note 2 (September 1987)
8. Schreiber, R.S., Van Loan, C.: A Storage Efficient WY Representation for Products of Householder Transformations. Technical Report. UMI TR87-864, Cornell University (1987)
9. Elmroth, E., Gustavson, F.G.: Applying recursion to serial and parallel QR factorization leads to better performance. IBM Journal of Research and Development 44(4), 605–624 (2000)
10. Agullo, E., Hadri, B., Ltaief, H., Dongarra, J.: Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In: SC (2009)

Approximate Incomplete Cyclic Reduction for Systems Which Are Tridiagonal and Strictly Diagonally Dominant by Rows

Carl Christian Kjelgaard Mikkelsen and Bo Kågström

Department of Computing Science and HPC2N
Umeå University, Sweden
{spock,bokg}@cs.umu.se

Abstract. Systems which are narrow banded and strictly diagonally dominant by rows can be solved in parallel using a variety of methods including incomplete block cyclic reduction. We show how to accelerate the algorithm by approximating the very first step. We derive tight estimates for the forward error and explain why our procedure is suitable for linear systems obtained by discretizing some common parabolic PDEs. An improved ScaLAPACK style algorithm is presented together with strong scalability results.

Keywords: Narrow banded, strictly and evenly diagonally dominant linear systems, approximate incomplete cyclic reduction.

1 Introduction

Once again we consider the problem of solving a nonsingular tridiagonal linear system

$$Ax \equiv \begin{bmatrix} d_1 & f_1 & & \\ e_2 & \ddots & \ddots & \\ & \ddots & \ddots & f_{n-1} \\ & & e_n & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{bmatrix} \equiv b \tag{1}$$

on a parallel machine with p processors. We assume that systems are large, i.e.

$$p \ll n, \tag{2}$$

and strictly diagonally dominant by rows, i.e.

$$\forall i \; : \; |e_i| + |f_i| < |d_i|, \tag{3}$$

where e_0 and f_n are undefined and should be treated as zeros. We shall make frequent use of the dominance factor ϵ defined by

$$\epsilon = \max_i \left\{ \frac{|e_i| + |f_i|}{|d_i|} \right\} < 1.$$

Finite difference methods, such as the implicit Euler method or the Crank-Nicolson method for the one dimensional heat equation

$$u_t = a(x,t)u_{xx} + c(x,t)u + f(x,t), \quad 0 < x < 1, \quad 0 < t \tag{4}$$

generate tridiagonal systems which are not only strictly diagonally dominant, but evenly so, in the sense that

$$\exists\, \epsilon \in (0,1)\, \forall i \,:\, \max\{|e_i|, |f_i|\} \leq \frac{1}{2}\epsilon |d_i|. \tag{5}$$

Our main result, Theorem 1, demonstrates the strength of this condition over condition (3).

Given a tolerance $\delta > 0$, our goal is to obtain an approximation y of the exact solution x such that

$$\|x - y\|_\infty \leq \delta \|x\|_\infty, \tag{6}$$

and to execute the computations with high parallel efficiency.

The linear system (1) can be partitioned as a block tridiagonal linear system as illustrated in Figure 1. Each matrix A_j is large and tridiagonal and each matrix C_j has just a single entry. Moreover, each of the four submatrices B_j, D_j, E_j, and F_j is a large vector containing at most one nonzero component. All blocks with the subscript j are stored in the local memory of processor P_j. This is exactly the partitioning scheme used in the ScaLAPACK implementation of cyclic reduction for certain banded matrices [1]. The blocks on the fringe, i.e. F_1, E_1, B_p, C_p, D_p, β_p, and ξ_p are all undefined and should be treated as zero.

$$\begin{bmatrix} A_1 & B_1 & & & & & & \\ D_1 & C_1 & \boxed{E_2} & & & & & \\ & \boxed{F_2} & \boxed{A_2} & \boxed{B_2} & & & & \\ & & \boxed{D_2} & \boxed{C_2} & E_3 & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & D_{p-1} & C_{p-1} & E_p \\ & & & & & F_p & A_p \end{bmatrix} \begin{bmatrix} x_1 \\ \xi_1 \\ \boxed{x_2} \\ \boxed{\xi_2} \\ \vdots \\ \xi_{p-1} \\ x_p \end{bmatrix} = \begin{bmatrix} b_1 \\ \beta_1 \\ \boxed{b_2} \\ \boxed{\beta_2} \\ \vdots \\ \beta_{p-1} \\ x_p \end{bmatrix}, \tag{7}$$

Fig. 1. The ScaLAPACK partitioning scheme used in PDDBTRF/PDDBTRS. The submatrices held by processor P_2 are emphasized.

If the central components $\{\xi_j\}_{j=1}^{p-1}$ have been computed and distributed, then we can recover the remaining components by solving the independent linear systems

$$A_j x_j = b_j - F_j \xi_{j-1} - B_j \xi_j, \quad j = 1, 2, \ldots, p. \tag{8}$$

The matrices A_j are all nonsingular, because A is strictly diagonally dominant by rows. Suppose that we have computed approximations $\{\nu_j\}_{j=1}^{p-1}$ such that

$$|\xi_j - \nu_j| \leq \delta \|x\|_\infty, \quad j = 1, 2, \ldots, p-1.$$

How should we proceed? In view of equation (8) it is only natural to define the vector y_j as the solution of

$$A_j y_j = b_j - F_j \nu_{j-1} - B_j \nu_j, \quad j = 1, 2, \dots, p, \tag{9}$$

and then inquire if the vector y given by

$$y = (y_1^T, \nu_1^T, y_2^T, \nu_2, \dots, \nu_{p-1}^T, y_p^T)^T$$

will satisfy the forward error bound (6)? The answer is yes! We have

$$x_j - y_j = -A_j^{-1} \begin{bmatrix} F_j \ B_j \end{bmatrix} \begin{bmatrix} \xi_{j-1} - \nu_{j-1} \\ \xi_j - \nu_j \end{bmatrix}. \tag{10}$$

By Corollary 3.2 [2] the strict diagonal dominance of A ensures

$$\left\| A_j^{-1} \begin{bmatrix} F_j \ B_j \end{bmatrix} \right\|_\infty \leq \epsilon < 1,$$

which together with (10) implies

$$\|x_j - y_j\|_\infty \leq \delta \|x\|_\infty,$$

and it is clear that y will satisfy the forward error bound (6).

Now, the real numbers $\{\xi_j\}_{j=1}^{p-1}$ can be found using, say, cyclic reduction [3] as in ScaLAPACK [1] or approximated using incomplete cyclic reduction [4], an option which we have explored in a previous paper [5]. By design the ScaLAPACK implementation accomplishes the vast majority of the calculations during the very first reduction step, unless condition (2) is violated and the system is small compared to the number of processors. If the system is large, then the benefits of skipping steps is minimal, unless the interconnect is very slow. In this paper, we are therefore concerned with accelerating the very first reduction step.

Our main results are derived in Section 2 and a parallel algorithm is formulated and presented together with some numerical experiments in Section 3. The new algorithm is an approximation of a single step of incomplete block cyclic reduction using the ScaLAPACK block partitioning [1].

2 The Main Result

High parallel efficiencies can be achieved by the rapid computation of accurate approximations of the real numbers $\{\xi_j\}_{j=1}^{p-1}$. Theorem 1 concerns the approximation of a single component of the solution to a large tridiagonal system.

Remark 1. In the statement of Theorem 1 the rows are numbered from $-\ell$ to m, rather than 1 through n as is customary. The goal is to focus the reader's attention on the central row with index 0.

Theorem 1. *Let x be the exact solution of the linear system*

$$Ax = \left[\begin{array}{cccc|c|cccc}
d_{-\ell} & f_{-\ell} & & & & & & & \\
e_{1-\ell} & \ddots & \ddots & & & & & & \\
 & \ddots & \ddots & \ddots & & & & & \\
 & & e_{-1} & d_{-1} & f_{-1} & & & & \\
\hline
 & & & e_0 & d_0 & f_0 & & & \\
\hline
 & & & & e_1 & d_1 & f_1 & & \\
 & & & & & \ddots & \ddots & \ddots & \\
 & & & & & & \ddots & \ddots & f_{m-1} \\
 & & & & & & & e_m & d_m
\end{array}\right]
\left[\begin{array}{c} x_{-\ell} \\ x_{1-\ell} \\ \vdots \\ x_{-1} \\ \hline x_0 \\ \hline x_1 \\ \vdots \\ x_{m-1} \\ x_m \end{array}\right]
= \left[\begin{array}{c} b_{-\ell} \\ b_{1-\ell} \\ \vdots \\ b_{-1} \\ \hline b_0 \\ \hline b_1 \\ \vdots \\ b_{m-1} \\ b_m \end{array}\right],$$

where A is strictly diagonally dominant by rows. Let q be an integer such that

$$q < \min\{\ell, m\},$$

and let $y = y(q)$ be the exact solution of the linear system

$$A_q y = \left[\begin{array}{cccc|c|cccc}
d_{-q} & f_{-q} & & & & & & & \\
e_{1-q} & \ddots & \ddots & & & & & & \\
 & \ddots & \ddots & \ddots & & & & & \\
 & & e_{-1} & d_{-1} & f_{-1} & & & & \\
\hline
 & & & e_0 & d_0 & f_0 & & & \\
\hline
 & & & & e_1 & d_1 & f_1 & & \\
 & & & & & \ddots & \ddots & \ddots & \\
 & & & & & & \ddots & \ddots & f_{q-1} \\
 & & & & & & & e_q & d_q
\end{array}\right]
\left[\begin{array}{c} y_{-q} \\ y_{1-q} \\ \vdots \\ y_{-1} \\ \hline y_0 \\ \hline y_1 \\ \vdots \\ y_{q-1} \\ y_q \end{array}\right]
= \left[\begin{array}{c} b_{-q} \\ b_{1-q} \\ \vdots \\ b_{-1} \\ \hline b_0 \\ \hline b_1 \\ \vdots \\ b_{q-1} \\ b_q \end{array}\right]. \tag{11}$$

Then

$$|x_0 - y_0| \leq \epsilon^{1+q} \|x\|_\infty. \tag{12}$$

If A is strictly and evenly diagonally dominant by rows, i.e. satisfies (5)*, then*

$$|x_0 - y_0| \leq \frac{2\rho^{1+q}}{1+\rho^{2+2q}} \|x\|_\infty, \tag{13}$$

where $\rho = \rho(\epsilon)$ is given by

$$\forall \epsilon \in (0,1) \; : \; \rho(\epsilon) = \frac{1 - \sqrt{1-\epsilon^2}}{\epsilon}. \tag{14}$$

Proof. Since A_q is nonsingular, we have a row equivalence relation of the form

$$\left[\begin{array}{c|cc|c|ccc|c|c}
e_{-q} & d_{-q} & f_{-q} & & & & & & b_{-q}\\
 & \ddots & \ddots & \ddots & & & & & \vdots\\ \hline
 & & e_0 & d_0 & f_0 & & & & b_0 \\ \hline
 & & & \ddots & \ddots & \ddots & & & \vdots\\
 & & & & & e_q & d_q & f_q & b_q
\end{array}\right] \overset{A_q}{\sim} \left[\begin{array}{c|cc|c|cc|c|c}
u_{-q} & 1 & & & & & v_{-q} & b'_{-q}\\
\vdots & & \ddots & & & & \vdots & \vdots\\ \hline
u_0 & & & 1 & & & v_0 & b'_0\\ \hline
\vdots & & & & \ddots & & \vdots & \vdots\\
u_q & & & & & 1 & v_q & b'_q
\end{array}\right],$$

and since A is strictly diagonally dominant by rows, Theorem 2 [5] gives a tight estimate for $\left\| \begin{bmatrix} u_j \ v_j \end{bmatrix} \right\|_\infty$. In particular, we have

$$\| \begin{bmatrix} u_0 \ v_0 \end{bmatrix} \|_\infty \leq \epsilon^{1+q}. \tag{15}$$

Moreover, we have

$$u_0 x_{-(1+q)} + x_0 + v_0 x_{1+q} = b'_0 = y_0,$$

which allows us to write

$$x_0 - y_0 = -u_0 x_{-(1+q)} - v_0 x_{1+q} = - \begin{bmatrix} u_0 \ v_0 \end{bmatrix} \begin{bmatrix} x_{-(1+q)} \\ x_{1+q} \end{bmatrix},$$

and deduce that

$$|x_0 - y_0| \leq \| \begin{bmatrix} u_0 \ v_0 \end{bmatrix} \|_\infty \left\| \begin{bmatrix} x_{-(1+q)} \\ x_{1+q} \end{bmatrix} \right\|_\infty \leq \epsilon^{1+q} \|x\|_\infty.$$

We will now prove the upper bound given by (13) subject to condition (5). To this end we define the matrix Ω_q as follows

$$\Omega_q = \begin{bmatrix} 0 & -d_{-q}^{-1} f_{-q} & & \\ -d_{1-q}^{-1} e_{1-q} & \ddots & \ddots & \\ & \ddots & \ddots & -d_{q-1}^{-1} f_{q-1} \\ & & -d_q^{-1} e_q & 0 \end{bmatrix}.$$

Then the strict diagonal dominance implies $\|\Omega_q\|_\infty < 1$ and

$$(I - \Omega_q)^{-1} = \sum_{j=0}^{\infty} \Omega_q^j \leq \sum_{j=0}^{\infty} |\Omega_q|^j = (I - |\Omega_q|)^{-1},$$

and if A satisfies (5) then

$$(I - |\Omega_q|)^{-1} \leq (I - H_q)^{-1},$$

where

$$H_q = \begin{bmatrix} 0 & \frac{1}{2}\epsilon & & \\ \frac{1}{2}\epsilon & \ddots & \ddots & \\ & \ddots & \ddots & \frac{1}{2}\epsilon \\ & & \frac{1}{2}\epsilon & 0 \end{bmatrix}.$$

Therefore

$$\left\| \begin{bmatrix} u_{-q} & v_{-q} \\ \vdots & \vdots \\ u_0 & v_0 \\ \vdots & \vdots \\ u_q & v_q \end{bmatrix} \right\| = \left| (I - \Omega_q)^{-1} \begin{bmatrix} d_{-q}^{-1} e_{-q} & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 0 & d_q^{-1} f_q \end{bmatrix} \right| \leq (I - H_q)^{-1} \begin{bmatrix} \frac{1}{2}\epsilon & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 0 & \frac{1}{2}\epsilon \end{bmatrix}.$$

Moreover, equality is achieved when $A = B(\epsilon)$, where

$$B(\epsilon) = I - H_q = \begin{bmatrix} 1 & -\frac{1}{2}\epsilon & & \\ -\frac{1}{2}\epsilon & \ddots & \ddots & \\ & \ddots & \ddots & -\frac{1}{2}\epsilon \\ & & -\frac{1}{2}\epsilon & 1 \end{bmatrix}. \tag{16}$$

It remains to analyze this specific matrix. It is not hard to see that the structure of $A = B(\epsilon)$ will ensure

$$u_0 = v_0,$$

and using Gaussian elimination without pivoting, see appendix, we discover that

$$v_0 = -\frac{\rho^{1+q}}{1 + \rho^{2+2q}}. \tag{17}$$

Therefore, if A is strictly and evenly diagonally dominant by rows, then

$$\left\| \left[u_0 \ v_0 \right] \right\|_\infty \leq \frac{2\rho^{1+q}}{1 + \rho^{2+2q}} \tag{18}$$

and equality is achieved for the matrix $A = B(\epsilon)$. Inequality (13) follows immediately. □

The upper bound given by (13) is completely determined by the central parameter ρ and the integer q. We have plotted ρ as a function of ϵ in Figure 2, and the following proposition summarizes its essential properties. The proof is straightforward and elementary.

Proposition 1. *Let $\rho = \rho(\epsilon)$ be defined as in Theorem 1. Then*

$$\forall \epsilon \in (0, 1) \ : \ \frac{\epsilon}{2} < \rho < \epsilon. \tag{19}$$

Moreover, ρ is strictly increasing with the limits

$$\rho \to 1, \quad \epsilon \to 1, \quad \epsilon \in (0, 1), \tag{20}$$

$$\rho \to 0, \quad \epsilon \to 0, \quad \epsilon \in (0, 1), \tag{21}$$

and the asymptotic behavior

$$\frac{\rho(\epsilon)}{\epsilon/2} \to 1, \quad \epsilon \to 0, \quad \epsilon \in (0, 1). \tag{22}$$

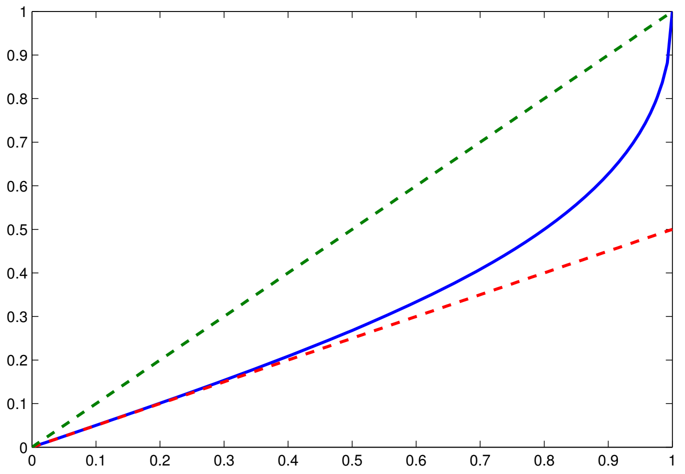


Fig. 2. The solid curve represents the central parameter ρ, i.e. $y = (1-\sqrt{1-\epsilon^2})/\epsilon$, while the two straight lines are given by $y = \epsilon$ and $y = \epsilon/2$

By Theorem 1 we can approximate the central components $\{\xi_j\}_{j=1}^{p-1}$ using Gaussian elimination and $8(2q+1)(p-1)$ arithmetic operations. In the following three examples we calculate the smallest possible value of q, which will allow us to compute $\{\xi_j\}_{j=1}^{p-1}$ to machine precision. The first example shows that inequality (15) is tight.

Example 1. Let $c > 0$ and consider the transport equation given by

$$u_t + cu_x = 0, \quad 0 < x < 1, \quad 0 < t,$$

with an initial condition $u(x,0) = u_0$ and a boundary condition $u(0,t) = g(t)$. Consider the following finite difference scheme

$$\frac{v_j^{(k+1)} - v_j^{(k)}}{\Delta t} + c\frac{v_j^{(k+1)} - v_{j-1}^{(k+1)}}{\Delta x} = 0, \quad j = 1,2,\dots,n, \quad \Delta x = 1/n, \tag{23}$$

together with the initial/boundary conditions

$$v_j^{(0)} = u_0(x_j), \quad j = 1,2,\dots,n, \quad \text{and} \quad v_0^{(k)} = g(t_k), \quad t_k = k\Delta t, \quad k \in \mathbb{N}.$$

The scheme can be written in matrix form as

$$Av^{(k+1)} = v^{(k)} + c\mu g(t_{k+1})e_1^{(n)}, \quad \text{where} \quad \mu = \frac{\Delta t}{\Delta x},$$

where $A \in \mathbb{R}^{n\times n}$ is the bidiagonal matrix given by

$$A = \begin{bmatrix} (1+c\mu) & & & & \\ -c\mu & (1+c\mu) & & & \\ & -c\mu & \ddots & & \\ & & \ddots & \ddots & \\ & & & -c\mu & (1+c\mu) \end{bmatrix},$$

and $e_1^{(n)}$ is the first column of the n by n identity matrix. We observe, that A is strictly diagonally dominant and the dominance factor ϵ is given by

$$\epsilon = \frac{c\mu}{1+c\mu} < 1.$$

Moreover, any value of $\epsilon \in (0,1)$ can be achieved with a suitable choice of $c\mu$. It is straightforward to verify that this matrix realizes the upper bound (15).

Which value of q will suffice to realize our forward error bound (6)? Clearly, we must have

$$q \geq q_0 = \left\lceil \frac{\log \delta}{\log \epsilon} \right\rceil - 1.$$

The choice of $\delta = 2^{-53}$ corresponds to the unit round off error in IEEE double precision floating point arithmetic, and for $\epsilon = 99/100$, we have $q_0 = 3655$.

The following example illustrates the strength of condition (5) over that of strict diagonal dominance (3).

Example 2. Consider the implicit Euler method for the parabolic equation (4) where $a = a(x,t) \geq 0$ and $c = c(x,t) \leq 0$. The finite difference equations are

$$\frac{v_j^{(k+1)} - v_j^{(k)}}{\Delta t} = a_j^{(k+1)} \frac{v_{j-1}^{(k+1)} - 2v_j^{(k+1)} + v_{j+1}^{(k+1)}}{(\Delta x)^2} + c_j^{k+1} v_j^{k+1} + g_j^{k+1},$$

or equivalently

$$A^{(k+1)} v^{(k+1)} = v^{(k)} + \Delta t g^{(k+1)},$$

where $A = A^{(k+1)}$ is the tridiagonal matrix given by

$$d_j = 1 + 2\lambda a_j^{(k+1)} - c_j^{(k+1)} \Delta t, \quad e_j = f_j = -\lambda a_j^{(k+1)} = -\frac{1}{2}\epsilon_j d_j,$$

and

$$0 \leq \epsilon_j = \frac{2\lambda a_j^{(k+1)}}{1 + 2\lambda a_j^{(k+1)} - c_j^{(k+1)} \Delta t} < 1,$$

so that

$$\max\{|e_j|, |f_j|\} \leq \frac{1}{2}\epsilon |d_j|, \quad \epsilon = \max_j \epsilon_j < 1,$$

and we can apply the improved bound (18). The worst case is realized when $a > 0$ and $c = 0$ are constants. Then

$$\epsilon = \frac{2a\lambda}{1 + 2a\lambda} < 1.$$

We observe that

$$q \geq q_0' = \left\lceil \frac{\log(\delta/2)}{\log \rho} \right\rceil - 1 \quad \Rightarrow \quad \frac{2\rho^{1+q}}{1 + \rho^{2+2q}} \leq \delta.$$

For $\epsilon = 99/100$ we have $\rho \approx 0.8676$ and with $\delta = 2^{-53}$ we have $q_0' = 263$ which is substantially smaller than the number found in Example 1, i.e. $q_0 = 3655$.

The final example demonstrates that it is critical that A is strictly diagonally dominant by rows.

Example 3. Consider the two point boundary value problem for the differential equation

$$u''(x) = g(x), \quad x \in (0,1).$$

If we use the standard space central discretization of the Laplace operator with uniform stepsize $h = 1/n$, then we arrive at a tridiagonal linear system $Ax = b$, where

$$A = n^2 \begin{bmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix}_{(n-1)\times(n-1)}$$

is diagonally dominant by rows, but not strictly diagonally dominant by rows. Therefore, Theorem 1 does not apply, but a direct calculation or a perturbation argument quickly establishes that

$$\left\| \left[u_0 \; v_0 \right] \right\|_\infty = 1$$

regardless of the value of q! It follows, that it is impossible to obtain an accurate approximation of y_0 using local information only.

3 A Parallel Algorithm and Some Experimental Results

We now derive a parallel algorithm by using our main result, Theorem 1 to approximate the very first step of block cyclic reduction implemented in ScaLAPACK, specifically subroutines `PDDBTRF/PDDBTRS`. The ScaLAPACK memory layout is used as discussed in Section 1.

The processors form a linear array and are numbered 1 through p. Subscripts continue to identify the processor storing the submatrix. It is worth restating that C_j has dimension 1. The superscripts are abbreviations. Specifically, $A_j^{(lr)}$

is the *lower right* q by q corner of A_j, while $A_j^{(ul)}$ is the *upper left* q by q corner of A_j. Moreover, $D_j^{(r)}$ refers to the q *rightmost* entries of the row vector D_j, while $E_j^{(l)}$ refers to the q *leftmost* entries of the row vector E_j. Finally, $b_j^{(b)}$ consists of the q components at the *bottom* of b_j, while $b_j^{(t)}$ consists of the q components at the *top* of b_j.

If $1 < p$, then processor P_j works together with P_{j-1} in order to solve the linear system

$$\left[\begin{array}{c|c|c} A_{j-1}^{(lr)} & B_{j-1}^{(b)} & \\ \hline D_{j-1}^{(r)} & C_{j-1} & E_j^{(l)} \\ \hline & F_j^{(t)} & A_j^{(ul)} \end{array}\right] \left[\begin{array}{c} z_{j-1}^{(b)} \\ \hline \nu_{j-1} \\ \hline z_{j1}^{(t)} \end{array}\right] = \left[\begin{array}{c} b_{j-1}^{(b)} \\ \hline \beta_{j-1} \\ \hline b_j^{(t)} \end{array}\right] \tag{24}$$

for ν_{j-1}, and if $j < p$ then processors P_j works together with P_{j+1} in order to solve the linear system

$$\left[\begin{array}{c|c|c} A_j^{(lr)} & B_j^{(b)} & \\ \hline D_j^{(r)} & C_j & E_{j+1}^{(l)} \\ \hline & F_{j+1}^{(t)} & A_{j+1}^{(ul)} \end{array}\right] \left[\begin{array}{c} z_j^{(b)} \\ \hline \nu_j \\ \hline z_{j+1}^{(t)} \end{array}\right] = \left[\begin{array}{c} b_j^{(b)} \\ \hline \beta_j \\ \hline b_{j+1}^{(t)} \end{array}\right] \tag{25}$$

for ν_j. Finally, processor P_j computes the vector x_j using equation (8). The details are given as Algorithm 1.

We implemented our algorithm in Fortran 90 using non-blocking MPI send and receive operations. As in ScaLAPACK [1] and the SPIKE/PSPIKE packages [6,7], the algorithm can be split in a factorization phase, which exclusively accesses the matrix to produce the necessary LU factorizations, and a solution phase, which accesses the right hand side and returns the actual solution.

Here we report on a test executed on the machine Akka, at HPC2N Umeå, which demonstrates that the algorithm is viable and exhibits good scalability. Using

$$p \in \{1, 2, \dots, 32\}$$

processors, we solved the linear systems

$$-x_{j-1} + \frac{2}{\epsilon} x_j - x_{j+1} = g_j, \quad j = 1, 2, \dots, n,$$

where $x_0 = x_{n+1} = 0$ and

$$n \in \{16, 32, 64, 128, 256, 512\} \times 10^3,$$

all with $\epsilon = 0.9$ and $q = 80$, for which $\rho \approx 0.6268$ and $\rho^{81} \approx 3.7 \times 10^{-17} < 2^{-53}$, and a single right hand side g corresponding to a known solution, specifically a vector of ones. The calculations were carried out using IEEE double precision floating point arithmetic and every system was solved with a normwise forward relative error equal to at most $5u$, $u = 2^{-53}$, a figure consistent with the infinity norm condition numbers which are bounded by 19. Plots of the strong scalability efficiency, i.e.

$$\eta = T_1/(pT_p),$$

for the factorization phase is given in Figure [3]. The plots for the solve efficiencies are similar and are omitted because of the page limitation. As expected, the efficiencies are high for small values of q/μ, where $\mu \approx n/p$ is the size of the largest tridiagonal block A_j. Developing a theory which explains the exact nature of the curves is still an open problem, as it is not merely a matter of counting floating point operations. The run-times were measured and averaged over all active processors. Cache effects were eliminated by flushing the entire cache hierarchy between each call to the factorization routine. The run-times include the time required to load the matrices from RAM memory. We repeated every experiment 10 times and found that the majority of the timings were tightly clustered except for one or two outliers which were frequently, but not consistently, among the first in every set of repetitions. We disregarded the anomalies and generated our plots from the rest of the data.

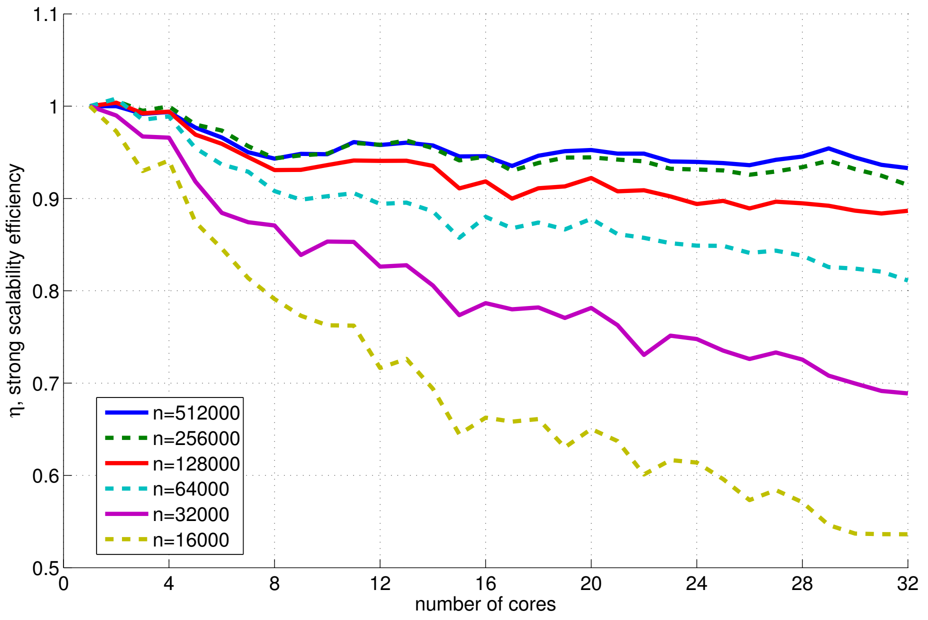


Fig. 3. Strong scalability test of the factorization phase of our algorithm. Please note that the efficiencies are all greater than 50%. All curves can be identified by their value at $p = 32$ where the ordering matches that of the legend.

4 Conclusions and Some Further Discussions

In exact arithmetic our algorithm will return the same approximation as the interface splitting algorithm introduced by Arpiruk Hokpunna [8]. There are minor differences in the memory layout and while we have the same number of

Algorithm 1. Approximate incomplete cyclic reduction

```
 1: if $1 < j$ then
 2:     Processor $P_j$ solves the linear system
            $$A_j^{(ul)} \left[ U_j^{(t)} \; u_j^{(t)} \right] = \left[ F_j^{(t)} \; b_j^{(t)} \right],$$
        computes the two scalars
            $$E_j^{(l)} U_j^{(t)} \quad \text{and} \quad E_j^{(l)} u_j^{(t)}$$
        and sends them to processor $j-1$.
 3: end if
 4: if $j < p$ then
 5:     Processor $P_j$ solves the linear system
            $$A_j^{(lr)} \left[ V_j^{(b)} \; v_j^{(b)} \right] = \left[ B_j^{(b)} \; b_j^{(b)} \right],$$
        computes the two scalars
            $$C_j - D_j^{(r)} V_j^{(b)} \quad \text{and} \quad \beta_j - D_j^{(r)} v_j^{(b)},$$
        and sends them to processor $P_{j+1}$.
 6: end if
 7: if $1 < j$ then
 8:     As the message from $P_{j-1}$ is received, processor $P_j$ completes the computation
            $$\nu_{j-1} = \left[ \left( C_{j-1} - D_{j-1}^{(r)} V_{j-1}^{(b)} \right) - E_j^{(l)} U_j^{(t)} \right]^{-1} \left[ \left( \beta_{j-1} - D_{j-1}^{(r)} v_{j-1}^{(b)} \right) - E_j^{(l)} u_j^{(t)} \right].$$
 9: end if
10: if $j < p$ then
11:     As the message from $P_{j+1}$ is received, processor $P_j$ completes the computation
            $$\nu_j = \left[ \left( C_j - D_j^{(r)} V_j^{(b)} \right) - E_{j+1}^{(l)} U_{j+1}^{(t)} \right]^{-1} \left[ \left( \beta_j - D_j^{(r)} v_j^{(b)} \right) - E_{j+1}^{(l)} u_{j+1}^{(t)} \right].$$
12: end if
13: if $1 < j$ then
14:     Processor $P_j$ updates its right hand side and solves for $y_j$
            $$A_j y_j = b_j - F_j \nu_{j-1} - B_j \nu_j, \quad j = 1, 2, \ldots, p.$$
15: else
16:     Processor $P_1$ updates its right hand side and solves for $y_1$
            $$A_1 y_1 = b_1 - B_1 \nu_1.$$
17: end if
```

messages, we have reduced the message size during the factorization phase from $O(q)$ to $O(1)$. This reduction is not significant, because both algorithms are only efficient for small values of q. Ahmed Sameh taught Mikkelsen how to reduce the need for communication by replicating calculations on different processors and how to use the corners of a matrix to approximate the action of the inverse. These techniques are utilized repeatedly in the SPIKE/PSPIKE packages [6,7]. Thus our primary contribution is to recognize the connection to cyclic reduction and to derive the tight error bounds given in Theorem 1. Moreover, Examples 1, 2, and 3 show that while truncation based methods can be very efficient for discretizations of certain parabolic operators, one must be cautious of hyperbolic operators and expect failure for elliptic operators. The numerical experiments verify that approximate incomplete cyclic reduction is accurate and scalable for systems which satisfy condition (5).

The concept of a tridiagonal and evenly diagonally dominant matrix has been defined differently in the past. Xian-He Sun, Hong Zhang and Lionel M. Ni required

$$|e_i| \leq \frac{1}{2}|d_i|, \quad |f_i| \leq \frac{1}{2}|d_i|, \quad e_{i+1}f_i > 0, \tag{26}$$

when they performed the initial analysis of the parallel diagonally dominant (PDD) algorithm [9]. In particular, they used this condition to show that the reduced system of the PDD algorithm is nonsingular. However, this system is identical to the reduced system of the truncated Spike algorithm [6], which in turn is nonsingular whenever A is strictly diagonally dominant by rows [2].

Another paper by Sun [10] considered the matrix which we have now proved exemplifies the worst possible behavior subject to condition (5). In particular, Sun derived asymptotic decay rates for this and related special cases all of which were Toeplitz matrices.

The standard space central discretization of the Laplace operator, see Example 3, produces a matrix which is evenly diagonally dominant in the sense of Sun, Zhang and Lionel, i.e. (26), but not in our sense (5). This is not a critical loss, because the decay rate of the elements of the inverse matrix is linear, rather than exponential, hence insufficient to be exploited even by the PDD algorithm.

In short, by changing the definition slightly we have arrived at a class of matrices which is large enough to contain physically relevant matrices and so small that the worst case behavior still exhibits exponential decay and for which a tight estimate can be given.

Acknowledgments. The first author thanks Arpiruk Hokpunna for answering some questions on the interface splitting algorithm and related issues in computational fluid dynamics. Birgitte Brydsø and Lars Karlsson both read earlier versions of the manuscript and made several valuable comments which contributed to the clarity of the presentation. The work is supported 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. In addition, support has been provided by the Swedish Research

Council grant A0581501. This research was conducted using the resources of High Performance Computing Center North (HPC2N).

Appendix

The purpose of the appendix is to derive a compact formula for the solution of the linear system

$$Ax = \begin{bmatrix} d_1 & f_1 & & \\ e_2 & \ddots & \ddots & \\ & \ddots & \ddots & f_{n-1} \\ & & e_n & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ f_n \end{bmatrix} = b,$$

where A is not only diagonally dominant by rows with dominance factor $\epsilon < 1$, but

$$|e_n| + |f_n| \leq \epsilon |d_n|, \quad \epsilon < 1,$$

so that f_n is not completely independent of the matrix A. We are primarily interested in the case of

$$d_i = 1, \quad e_i = f_i = -\frac{1}{2}\epsilon, \tag{27}$$

as well as the very specific choice of $n = 2q + 1$ and $j = q + 1$. However, it is convenient to proceed from the general case. A straight forward application of Gaussian elimination without pivoting reveals that

$$\left[\begin{array}{cccc|c} d_1 & f_1 & & & 0 \\ e_2 & \ddots & \ddots & & \vdots \\ & \ddots & \ddots & f_{n-1} & 0 \\ & & e_n & d_n & f_n \end{array}\right] \sim \left[\begin{array}{cccc|c} \gamma_1 & f_1 & & & 0 \\ & \ddots & \ddots & & \vdots \\ & & \ddots & f_{n-1} & 0 \\ & & & \gamma_n & f_n \end{array}\right],$$

where

$$\gamma_1 = d_1, \quad \gamma_{j+1} = d_{j+1} - e_j \gamma_j^{-1} f_j.$$

By back-substitution we have

$$x_j = (-1)^{n-j} \prod_{i=j}^{n} \left[\gamma_i^{-1} f_i\right] = (-1)^{n-j} \frac{\Gamma_{j-1}}{\Gamma_n} \prod_{i=j}^{n} f_i, \tag{28}$$

where we have defined

$$\Gamma_0 = 1, \quad \Gamma_j = \prod_{i=1}^{j} \gamma_j.$$

Equation (28) is useful, because the sequence $\{\Gamma_j\}$ satisfies a linear recurrence relation, specifically

$$\Gamma_{j+1} = \gamma_{j+1}\Gamma_j = (d_{j+1} - e_{j+1}\gamma_j^{-1} f_j)\Gamma_j = d_{j+1}\Gamma_j - e_{j+1} f_j \Gamma_{j-1}. \tag{29}$$

Now, in the case of (27) we have

$$\Gamma_j = c_1\lambda_-^j + c_2\lambda_+^j, \quad c_1 = \frac{\lambda_-}{\lambda_- - \lambda_+}, \quad c_2 = -\frac{\lambda_+}{\lambda_- - \lambda_+}, \quad \lambda_\pm = \frac{1 \pm \sqrt{1-\epsilon^2}}{2}.$$

It is straight forward to verify that

$$\frac{c_1}{c_2} = -\rho^2, \quad \frac{\lambda_-}{\lambda_+} = \rho^2, \quad \rho = \frac{1-\sqrt{1-\epsilon^2}}{\epsilon},$$

and so in the special case of $n = 2q+1$ and $j = 1+q$ we have

$$\begin{aligned} x_{1+q} &= -\left(\frac{\epsilon}{2\lambda_+}\right)^{1+q} \left\{ \frac{1+\frac{c_1}{c_2}\left(\frac{\lambda_-}{\lambda_+}\right)^q}{1+\frac{c_1}{c_2}\left(\frac{\lambda_-}{\lambda_+}\right)^{1+2q}} \right\} = -\rho^{1+q}\left\{\frac{1-(\rho^2)^{1+q}}{1-(\rho^2)^{2+2q}}\right\} \\ &= -\rho^{1+q}\left\{\frac{1-\rho^{2+2q}}{(1-\rho^{2+2q})(1+\rho^{2+2q})}\right\} = -\frac{\rho^{1+q}}{1+(\rho^2)^{1+q}}, \end{aligned} \tag{30}$$

which explains the origin of equation (17). We stress that (28) and (29) are valid only because multiplication of real numbers is commutative.

References

1. Arbenz, P., Cleary, A., Dongarra, J., Hegland, M.: A Comparison of Parallel Solvers for Diagonally Dominant and General Narrow-Banded Linear Systems II. In: Amestoy, P., Berger, P., Daydé, M., Duff, I., Frayssé, V., Giraud, L., Ruiz, D. (eds.) Euro-Par 1999. LNCS, vol. 1685, pp. 1078–1087. Springer, Heidelberg (1999)
2. Mikkelsen, C.C.K., Manguoglu, M.: Analysis of the truncated Spike algorithm. SIAM J. Matrix Analysis Applications 30, 1500–1519 (2008)
3. Hockney, R.W.: A fast direct solution of Poisson's equation using Fourier analysis. J. ACM 12, 95–113 (1965)
4. Heller, D.: Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. SIAM Journal on Numerical Analysis 13, 484–496 (1976)
5. Kjelgaard Mikkelsen, C.C., Kågström, B.: Incomplete Cyclic Reduction of Banded and Strictly Diagonally Dominant Linear Systems. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 80–91. Springer, Heidelberg (2012)
6. Polizzi, E., Sameh, A.H.: A parallel hybrid banded system solver: the Spike algorithm. Parallel Computing 32, 177–194 (2006)
7. Manguoglu, M., Sameh, A.H., Schenk, O.: PSPIKE: A Parallel Hybrid Sparse Linear System Solver. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 797–808. Springer, Heidelberg (2009)
8. Hokpunna, A.: Compact fourth order scheme for numerical simulations of Navier-Stokes equations. PhD thesis, Technische Universität München (2009)
9. Sun, X.-H., Sun, H.Z., Ni, L.M.: Parallel algorithms for solution of tridiagonal systems on multicomputers. In: Proceedings of the 3rd International Conference on Supercomputing, ICS 1989, pp. 303–312. ACM, New York (1989)
10. Sun, X.H.: Application and accuracy of the parallel diagonal dominant algorithm. Parallel Computing 21, 1241–1267 (1995)

Fast Poisson Solvers for Graphics Processing Units

Mirko Myllykoski[1], Tuomo Rossi[1], and Jari Toivanen[1,2]

[1] Department of Mathematical Information Technology,
P.O. Box 35 (Agora), FI-40014 University of Jyväskylä, Finland
[2] Department of Aeronautics and Astronautics, Stanford University,
Stanford, CA 94305, USA

Abstract. Two block cyclic reduction linear system solvers are considered and implemented using the OpenCL framework. The topics of interest include a simplified scalar cyclic reduction tridiagonal system solver and the impact of increasing the radix-number of the algorithm. Both implementations are tested for the Poisson problem in two and three dimensions, using a Nvidia GTX 580 series GPU and double precision floating-point arithmetic. The numerical results indicate up to 6-fold speed increase in the case of the two-dimensional problems and up to 3-fold speed increase in the case of the three-dimensional problems when compared to equivalent CPU implementations run on a Intel Core i7 quad-core CPU.

1 Introduction

The linear system solvers are a very popular research topic in the field of GPU (Graphics Processing Unit, Video Card) computing. Many of these transform the original problem into a set of sub-problems which can be solved more easily. In some cases, these sub-problems are in the form of tridiagonal linear systems and the tridiagonal system solver often constitutes a significant portion of the total execution time. Conventional linear system solvers such as the LU-decomposition, also known as the Thomas method [1] when applied to a tridiagonal system, do not perform very well on a GPU because of their sequential nature. For that reason, a different kind of method called cyclic reduction [2] has become one of the most widely used methods for this purpose [3–8].

The basic idea of the cyclic reduction method can be extended to block tridiagonal systems which arise, for example, from many PDE (Partial Differential Equation) discretisations. The idea of the block cyclic reduction (BCR) was first introduced in [2]. While the formulation is numerically unstable, it can be stabilized by combining it with the Fourier analysis method [2] as was shown in [9, 10]. The first stable BCR formulation, so called Buneman's variant [11], was introduced in 1969 and generalized in [12]. Later, the idea of the partial fraction expansions was applied to the matrix rational functions occurring in the formulas, thus leading to the discovery of a parallel variant [13]. The radix-q PSCR (Partial Solution variant of the Cyclic Reduction) method [14–17] represents a different kind of approach based on the partial solution technique [18, 19]. Excellent surveys on these kind of methods can be found in [20] and [21].

The cyclic reduction is a two-stage algorithm. The reduction stage generates a sequence of (block) tridiagonal systems by recursively eliminating (block) rows from the system and the back substitution stage solves all previously formed reduced systems in reverse order using the known rows of the solution from the previous back substitution step. Usually, the reduction is performed in such a way that all odd numbered (block) rows are eliminated, i.e., the radix-number is two. The method presented in [22] is such a method and in this paper it is called as the radix-2 BCR method. More generalized BCR methods, such as the radix-q PSCR, allow the use of higher radix-numbers.

Each radix-2 BCR reduction and back substitution step can be computed in parallel using the partial fraction expansions. However, the steps themselves must be performed sequentially. A method with a higher radix-number requires fewer steps to be taken and thus could be more suitable for parallel computation. A method analogous to the radix-2 BCR method can be easily obtained as a special case of the radix-4 PSCR method. This method reduces the systems size by a factor of four at each reduction step. Each radix-4 BCR reduction and back substitution step requires more computation than a radix-2 step, but the amount of sequential computation is reduced by a factor of two.

In this paper, the radix-2 and radix-4 BCR methods are applied to the following problem: Solve $u \in \mathbb{R}^{n_1 n_2}$ from

$$\begin{bmatrix} D & -I & & \\ -I & D & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & D \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n_1} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n_1} \end{bmatrix}, \tag{1}$$

where $D = \text{tridiag}\{-1, 4, -1\} \in \mathbb{R}^{n_2 \times n_2}$, when $f \in \mathbb{R}^{n_1 n_2}$ is given. It is assumed that $n_1 = 2^{k_1} - 1$ and $n_2 = 2^{k_2} - 1$ for some positive integers k_1 and k_2. This choice greatly simplifies the mathematical formulation and the implementation. The system (1) corresponds to a two-dimensional Poisson problem with Dirichlet boundary conditions posed on a rectangle. The implementations presented in this paper can be extended to cases where the diagonal block D is symmetric, tridiagonal and diagonally dominant.

The diagonal block can also be of the form $D = \text{tridiag}\{-I_{n_3}, \hat{D}, -I_{n_3}\} \in \mathbb{R}^{n_2 n_3 \times n_2 n_3}$, where $\hat{D} = \text{tridiag}\{-1, 6, -1\} \in \mathbb{R}^{n_3 \times n_3}$ and $n_3 = 2^{k_3} - 1$ for some positive integer k_3. In this case, the linear system (1) corresponds to a three-dimensional Poisson problem with Dirichlet boundary conditions posed in a rectangular cuboid.

The GPU implementations are compared with each other and to equivalent CPU implementations. The first objective is to find out how suitable the BCR methods are for GPU and how the radix-number effects the overall performance. The second objective is to introduce new ideas related to the tridiagonal system solvers. In particular, it is considered how to deal with the GPU's multilevel memory architecture and its limitations.

The rest of this paper is organized as follows: The second section briefly describes the two BCR methods considered in this paper and the third section covers the key aspects of the implementation. The fourth section presents the numerical results and discussion. Finally, the conclusions are given in the fifth section.

2 Methods

2.1 Radix-2 Block Cyclic Reduction

The radix-2 BCR method can be described using the following cyclic reduction formulation described in [23]. Let $T^{(0)} = I$, $D^{(0)} = D$ and $f^{(0)} = f$. Now the reduced systems are defined, for each reduction step $r = 1, 2, \ldots, k_1 - 1$, as

$$\begin{bmatrix} D^{(r)} & -T^{(r)} & & \\ -T^{(r)} & D^{(r)} & \ddots & \\ & \ddots & \ddots & -T^{(r)} \\ & & -T^{(r)} & D^{(r)} \end{bmatrix} \begin{bmatrix} u_1^{(r)} \\ u_2^{(r)} \\ \vdots \\ u_{2^{k_1-r}-1}^{(r)} \end{bmatrix} = \begin{bmatrix} f_1^{(r)} \\ f_2^{(r)} \\ \vdots \\ f_{2^{k_1-r}-1}^{(r)} \end{bmatrix}, \tag{2}$$

where

$$\begin{aligned} T^{(r)} &= \left(T^{(r-1)}\right)^2 \left(D^{(r-1)}\right)^{-1}, \\ D^{(r)} &= D^{(r-1)} - 2\left(T^{(r-1)}\right)^2 \left(D^{(r-1)}\right)^{-1}, \\ f_i^{(r)} &= f_{2i}^{(r-1)} + T^{(r-1)} \left(D^{(r-1)}\right)^{-1} \left(f_{2i-1}^{(r-1)} + f_{2i+1}^{(r-1)}\right). \end{aligned} \tag{3}$$

These reduced systems, $r = k_1 - 1, k_1 - 2, \ldots, 0$, can be solved recursively during the back substitution stage of the algorithm by using the formula

$$u_i^{(r)} = \begin{cases} \left(D^{(r)}\right)^{-1} \left(f_i^{(r)} + T^{(r)} \left(u_{(i-1)/2}^{(r+1)} + u_{(i-1)/2+1}^{(r+1)}\right)\right), & \text{when } i \notin 2\mathbb{N}, \\ u_{i/2}^{(r+1)}, & \text{when } i \in 2\mathbb{N}, \end{cases} \tag{4}$$

where $i = 1, 2, \ldots, 2^{k_1-r} - 1$ and $u_0^{(r+1)} = u_{2^{k_1-r-1}}^{(r+1)} = 0$. Finally, $u = u^{(0)}$.

As shown in [22], if the matrices $D^{(0)}$ and $T^{(0)}$ commute, then the matrices $T^{(r)}\left(D^{(r)}\right)^{-1}$ and $\left(D^{(r)}\right)^{-1}$ can be presented using matrix polynomials and rational functions. This observation greatly improves the computational complexity of the algorithm as it preserved the sparsity properties of the coefficient matrix. Otherwise the matrices $D^{(r)}$ and $T^{(r)}$ could fill up quickly. Assuming $T^{(0)} = I$ allows the use of the partial fraction expansion technique [13] and leads to

$$T^{(r)} \left(D^{(r)}\right)^{-1} = 2^{-r} \sum_{j=1}^{2^r} (-1)^{j-1} \sin\left(\frac{2j-1}{2^{r+1}}\pi\right) \left(D - \theta(j,r) I_{n_2}\right)^{-1}, \tag{5}$$

and

$$\left(D^{(r)}\right)^{-1} = 2^{-r} \sum_{j=1}^{2^r} \left(D - \theta(j,r) I_{n_2}\right)^{-1}, \tag{6}$$

where

$$\theta(j,r) = 2\cos\left(\frac{2j-1}{2^{r+1}}\pi\right). \tag{7}$$

These sum-formulations imply that each reduction and back substitution step can be carried out by first forming a large set of sub-problems, then solving these sub-problems (in parallel) and finally constructing the final result by computing collective sums over the solutions. This is the first point where some additional parallelism can be achieved and this level of parallelism is usually sufficient for contemporary multi-core CPUs.

The above described cyclic reduction formulas are well-defined (i.e. $\left(D^{(r)}\right)^{-1}$ exists for each $r = 1, 2, \dots, k_1 - 1$) if D^{-1} exists and the coefficient matrix is strictly diagonally dominant by rows [23]. In addition, the method has been shown to be numerically stable if the smallest eigenvalue of the matrix D is at least 2 [22]. All of these conditions are fulfilled in the case of the problem (1).

The arithmetical complexity of this method is $\mathcal{O}(n_1 n_2 \log n_1)$. If the diagonal block D is block tridiagonal as discussed in the introduction, then this method can be applied recursively. In this case, the arithmetical complexity is $\mathcal{O}(n_1 n_2 n_3 \log(n_1) \log(n_2))$.

Remark 1. The above formulated partial fraction method can be actually considered to be a special case of the radix-2 PSCR method in the sense that both methods generate exactly the same sub-problems [22].

2.2 Radix-4 Block Cyclic Reduction

The formulation of the radix-4 BCR method is slightly more complicated. One approach is to start from the radix-4 PSCR method and explicitly calculate all eigenvalues and eigenvector components associated with the partial solutions. The radix-4 PSCR method can be applied to a problem with a coefficient matrix of the form

$$A_1 \otimes M_2 + M_1 \otimes A_2 + c(M_1 \otimes M_2), \tag{8}$$

where $A_1, M_1 \in \mathbb{R}^{n_1 \times n_1}$ are tridiagonal, $A_2, M_2 \in \mathbb{R}^{n_2 \times n_2}, c \in \mathbb{R}$ and $\otimes$ denotes the matrix Kronecker (tensor) product. If $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{m \times m}$, then $A \otimes B = \{A_{i,j} B\}_{i,j=1}^{n} \in \mathbb{R}^{nm \times nm}$. The coefficient matrix in the system (1) can be expressed as

$$A \otimes I_{n_2} + I_{n_1} \otimes (D - 2I_{n_2}), \tag{9}$$

where $A = \text{tridiag}\{-1, 2, -1\} \in \mathbb{R}^{n_1 \times n_1}$.

The radix-4 PSCR method includes an initialization stage comprising generalized eigenvalue problems. Let $n_1 = 4^k - 1$ for some positive integer k. When the coefficient matrix is of the form (9), the generalized eigenvalue problems reduce to

$$\tilde{A}^{(r)} w_i^{(r)} = \lambda_i^{(r)} w_i^{(r)},\ i = 1, 2, \ldots, m_r, \tag{10}$$

where $r = 0, 1, \ldots, k-1$, $m_r = 4^{r+1}-1$ and $\tilde{A}^{(r)} = \text{tridiag}\{-1, 2, -1\} \in \mathbb{R}^{m_r \times m_r}$.

With the assumptions mentioned above, the radix-4 PSCR solution process goes as follows: Let $f^{(0)} = f$. First, for $r = 1, 2, \ldots, k-1$, a sequence of vectors is generated by using the formula

$$f_i^{(r)} = f_{4i}^{(r-1)} + \sum_{j=1}^{m_{r-1}} (w_j^{(r-1)})_{m_{r-1}} v_{i,j}^{(r)} + \sum_{j=1}^{m_{r-1}} (w_j^{(r-1)})_1 v_{i+1,j}^{(r)}, \tag{11}$$

where $i = 1, 2, \ldots, 4^{k-r} - 1$ and the vector $v_{i,j}^{(r)}$ can be solved from

$$\left(D + (\lambda_j^{(r-1)} - 2) I_{n_2}\right) v_{i,j}^{(r)} = \sum_{s=1}^{3} (w_j^{(r-1)})_{s4^{r-1}} f_{(i-1)4+s}^{(r-1)}. \tag{12}$$

Then, for $r = k-1, k-2, \ldots, 0$, a second sequence of vectors is generated by using the formula

$$\begin{aligned} u_{4d+i}^{(r)} &= \sum_{j=1}^{m_r} (w_j^{(r)})_{i4^r} y_{d,j}^{(r)},\ i = 1, 2, 3, \\ u_{4d+4}^{(r)} &= u_{d+1}^{(r+1)}, \end{aligned} \tag{13}$$

where $d = 0, 1, \ldots, 4^{k-r} - 1$ and the vector $y_{d,j}^{(r)}$ can be solved from

$$\begin{aligned} \left(D + (\lambda_j^{(r)} - 2) I_{n_2}\right) y_{d,j}^{(r)} = \sum_{s=1}^{3} (w_j^{(r)})_{s4^r} f_{4d+s}^{(r)} + \\ (w_j^{(r)})_1 u_d^{(r+1)} + (w_j^{(r)})_{m_r} u_{d+1}^{(r+1)}. \end{aligned} \tag{14}$$

In addition, $u_0^{(r+1)} = u_{k-r-1}^{(r+1)} = 0$. Finally, $u = u^{(0)}$.

It is well-known that the matrix $\tilde{A}^{(r)}$ has the following eigenvalues and eigenvectors

$$\lambda_i^{(r)} = 2 - 2\cos\left(\frac{i\pi}{4^{r+1}}\right) \text{ and } (w_i^{(r)})_j = \sqrt{\frac{2}{4^{r+1}}} \sin\left(\frac{ij\pi}{4^{r+1}}\right), \tag{15}$$

where $i, j = 1, 2, \ldots, m_r$. Now,

$$\begin{aligned} (w_i^{(r)})_1 &= \sqrt{2^{-2r-1}} \sin\left(i\pi/4^{r+1}\right) = (-1)^{i-1} (w_i^{(r)})_{m_r}, \\ (w_i^{(r)})_{1\cdot 4^r} &= \sqrt{2^{-2r-1}} \sin\left(i\pi/4\right) = (-1)^{i-1} (w_i^{(r)})_{3\cdot 4^r}, \\ (w_i^{(r)})_{2\cdot 4^r} &= \sqrt{2^{-2r-1}} \sin\left(i\pi/2\right). \end{aligned} \tag{16}$$

It is easy to see that $(w_j^{(r)})_{2\cdot 4^r} = 0$ when $j \in 2\mathbb{N}$ and $(w_j^{(r)})_{1\cdot 4^r} = (w_j^{(r)})_{3\cdot 4^r} = 0$ when $j \in 4\mathbb{N}$. For this reason, about one-quarter of the sub-problems required to compute the partial solutions are non-contributing and can be ignored.

Clearly each radix-4 BCR reduction and back substitution step is more computationally demanding than the corresponding radix-2 BCR step. However, the radix-2 BCR method generates a total of

$$N_{\text{count}}^2(n) = (n+1)(\log_2(n+1) - 1) + 1 \tag{17}$$

sub-problems and the radix-4 BCR method generates a total of

$$N_{\text{count}}^4(n) = (n+1)\left(\frac{3}{4}\log_2(n+1) - 1\right) + 1 \tag{18}$$

sub-problems. Thus the total number of sub-problems is reduced asymptotically by the factor

$$\lim_{n\to\infty} \frac{N_{\text{count}}^2(n)}{N_{\text{count}}^4(n)} = \frac{4}{3}. \tag{19}$$

In the case of three-dimensional problems, the ratio is even better

$$\lim_{n\to\infty} \left(\frac{N_{\text{count}}^2(n)}{N_{\text{count}}^4(n)}\right)^2 = \frac{16}{9}. \tag{20}$$

Remark 2. The above described method can be also derived by combining two radix-2 BCR reduction steps (3) into a single radix-4 BCR reduction step (11). Applying the partial fraction technique yields exactly the same sub-problems. The same procedure can be applied to the back substitution stage.

This simplified formulation can be only applied to problems with $n_1 = 4^k - 1$. However, this limitation can be easily relaxed in the following manner: Let $n_1 = 2^{\hat{k}} - 1$ for some integer $\hat{k} \geq 2$. The indexes in the reduction formula (11) are modified in such a way that $r = 1, 2, \ldots, \lceil \hat{k}/2 \rceil - 1$ and $i = 1, 2, \ldots, 2^{k-2r}$. Similarly, the indexes in the back substitution formula (13) are modified in such a way that $r = \lfloor \hat{k}/2 \rfloor - 1, \lfloor \hat{k}/2 \rfloor - 2, \ldots, 0$ and $d = 0, 1, \ldots, 2^{k-2r-2}$. If $\hat{k} \notin 2\mathbb{N}$, then it is necessary to perform one radix-2 BCR back substitution step at the radix-2 level $r = \hat{k} - 1$ in order to solve the block row $u_{2^{\hat{k}-1}}$.

The numerical experiments indicate that this method is numerically stable in the case of the Poisson problem (1). The arithmetical complexity of this method is $\mathcal{O}(n_1 n_2 \log n_1)$. If the diagonal block D is block tridiagonal as discussed in the introduction, then this method can be applied recursively. In this case, the arithmetical complexity is $\mathcal{O}(n_1 n_2 n_3 \log(n_1) \log(n_2))$.

2.3 Simplified Scalar Cyclic Reduction

In the case of the problem (1), all tridiagonal sub-problems generated by the methods described above are of the form

$$\begin{bmatrix} d & -1 & & \\ -1 & d & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & d \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}, \tag{21}$$

where $d \in]2, 10[$, $v_1, \ldots, v_n, g_1, \ldots, g_n \in \mathbb{R}$ and $n = 2^k - 1$ for some positive integer k. This system can be solved with the following cyclic reduction formulas analogous to (3) and (4): Let $t^{(0)} = 1$, $d^{(0)} = d$ and $g^{(0)} = g$. Now the reduced systems are defined, for each reduction step $r = 1, 2, \ldots, k-1$, as

$$\begin{aligned} t^{(r)} &= \left(t^{(r-1)}\right)^2 / \, d^{(r-1)}, \\ d^{(r)} &= d^{(r-1)} - 2\left(t^{(r-1)}\right)^2 / \, d^{(r-1)}, \\ g_i^{(r)} &= g_{2i}^{(r-1)} + \left(t^{(r-1)} / \, d^{(r-1)}\right)\left(g_{2i-1}^{(r-1)} + g_{2i+1}^{(r-1)}\right). \end{aligned} \tag{22}$$

The solution of each reduced system, $r = k-1, k-2, \ldots, 0$, is produced recursively during the back substitution stage of the algorithm by using the formula

$$v_i^{(r)} = \begin{cases} \left(g_i^{(r)} + t^{(r)}\left(v_{(i-1)/2}^{(r+1)} + v_{(i-1)/2+1}^{(r+1)}\right)\right) / \, d^{(r)}, & \text{when } i \notin 2\mathbb{N}, \\ v_{i/2}^{(r+1)}, & \text{when } i \in 2\mathbb{N}, \end{cases} \tag{23}$$

where $i = 1, 2, \ldots, 2^{k-r} - 1$ and $v_0^{(r+1)} = v_{2^{k-r-1}}^{(r+1)} = 0$. Finally, $v = v^{(0)}$. The arithmetical complexity of this method is $\mathcal{O}(n)$.

3 Implementation

3.1 GPU Hardware

The GPU implementations are written using the OpenCL [24] framework and the OpenCL terminology is used throughout the paper. The architecture of a GPU is very different compared to a CPU. The main difference is that while a contemporary high-end consumer-level CPU may contain up to 8 cores, a modern high-end GPU contains thousands of processing elements. This means that the GPU requires a very fine-grained parallelism.

Another important difference is the memory architecture. A computing oriented GPU may include a few gigabytes of global memory (Video RAM, VRAM) which can be used to store the bulk of data. In addition, the processing elements are divided into groups called the compute units and the processing elements belonging to the same compute unit share a fast memory area called the local memory. The effective use of this small memory area, together with a good understanding of the other underlying hardware limitations, is often the key to achieving good performance.

The GPU-side code execution begins when a special kind of subroutine called the kernel is launched. Every work-item (thread) starts from the same location in the code but each work-item is given a unique index number which makes branching possible. The work-items are divided into work groups which are then assigned to the compute units. The work-items which belong to the same work group can share a portion of the local memory.

3.2 Overall Implementation

The BCR implementations consist mostly of scalar-vector multiplications and vector-vector additions which can be implemented trivially, for example, by mapping each row-wise operation to one work-item. The large vector summations, especially during the last few reduction steps and first back substitution steps, require some additional attention. The kernels performing these summations divide the large summations into several sub-sums in order to better distribute the workload among the processing elements. The implementation employs three kernels per step approach: the first kernel generates the right-hand side vectors for the sub-problems, the second kernel solves the sub-problems and the third kernel computes the collective sums.

The implementation incorporates a simple parameter optimizer. The main application for this parametrization is to choose the optimal work group size for each kernel. Also, the kernels responsible for computing the vector sums are parametrized. The parametrization is used to choose the optimal size for each sub-sum. In addition, the parametrization is used to specify how much local memory can be used to solve a single tridiagonal sub-problem and how double precision numbers are stored into the local memory.

3.3 Previous Work on Tridiagonal System Solvers on a GPU

The GPU hardware presents many challenges to the tridiagonal system solver implementation. First, a work group can only contain a limited number work-items and the work groups cannot communicate with each other. These two limitations complicate the tasks of solving large systems. Secondly, the global memory is quite slow for scattered memory access and therefore work-items with successive index numbers should only access memory locations which are close to each other. In addition, the local memory is often divided into banks which may be subject to only one memory request at a time.

The idea of using the cyclic reduction for solving tridiagonal systems on a GPU first appeared in [3]. The cyclic reduction, the parallel cyclic reduction [25], the recursive doubling [26], and hybrid algorithms were compared with each other in [5]. All considered implementations utilize the local memory and hold the data in-place. The paper also suggested the possibility of reducing the system size by using the cyclic reduction and the global memory in order to fit the reduced system into the local memory. The cyclic reduction and the local memory were also used in [6]. The paper introduced a clever permutation pattern which reduces the number of bank conflicts.

The idea of hybrid algorithms was taken a step further in [27]. The implementation considered consist several phases. The system is first split into multiple sub-systems using the parallel cyclic reduction and the global memory. Then, the sub-systems are solved in the local memory using the parallel cyclic reduction and the Thomas method. The optimal switching points between different stages are chosen automatically with the help of auto-tuning algorithm.

The idea of using both the global and local memory in the context of the cyclic reduction and the recursive doubling was also studied in [8]. The cyclic reduction implementation stores the right-hand side vector into the global memory and divides the system into sections. Each section is then processed separately in the local memory and then the intermediate result are merged back into the global memory. Additional work was also done in [4, 7, 28, 29].

3.4 Tridiagonal System Solver Implementation

When the coefficient matrix is a symmetric Toeplitz matrix like in (21), using the simplified cyclic reduction method is probably the most suitable algorithm for solving the tridiagonal sub-problems. The tridiagonal system solver consists of three stages and the right-hand side vector is replaced by the solution vector. One tridiagonal system is mapped to one work group and the whole solution process is performed as a single kernel launch. The implementation can be easily extended to more generalized tridiagonal systems and to cases where one tridiagonal systems is mapped to multiple work groups.

First Stage. The first stage is performed only when when the system is too large to fit into the allocated local memory. It uses the global memory to store the right-hand side vector and the local memory to share odd numbered rows between work-items. The right-hand side vector is divided into sections which are the same size as the used work group. Then all sections are processed in pairs as follows: first every work-item computes one row, and then all odd numbered rows are stored into the first section, and computed rows are stored into the second section. At the next reduction step, the same procedure is repeated using the second section from each pair. This permutation pattern is reversed during the back substitution stage. Fig. 1 illustrates this process. This implementation differs from the one presented in [8].

The idea behind this segmentation and permutation pattern is to divide the right-hand side vector into independent parts which can be processed separately. In this case, these sections are processed sequentially and therefore the implementation is capable of solving systems that are too large to fit into the allocated local memory. In a more general implementation, these section can be processed in parallel using multiple work groups. The second benefit is that the rows which belong to the same reduced system are stored close to each other in the global memory, thus allowing a more coherent global memory access pattern.

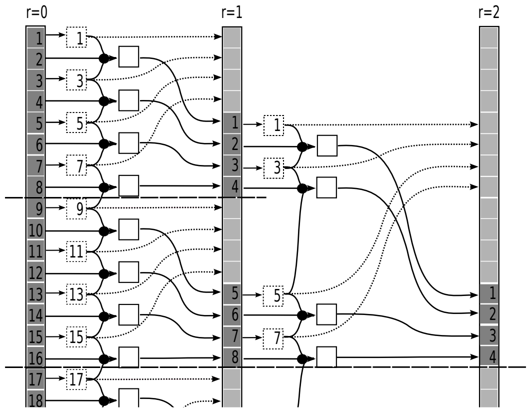


Fig. 1. The permutation pattern during the first stage of the tridiagonal system solver. The work group size is four. The numbers correspond to the row indexes. The row indexes highlighted with dotted rectangles are shared between the work-items using the local memory.

Second Stage. The second stage is only performed when the number of remaining even numbered rows is greater than the used work group size. It uses a similar segmentation and permutation approach as the first stage, but the rows are processed by four sections at a time and every work-item is responsible for computing two rows. The idea is that the rows belonging to these four parts are permuted before the beginning of the reduction process in such a way that all odd numbered rows are stored into the first and third section, and all even numbered rows are stored into the second and fourth section. This permutation pattern resembles the one presented in [6]. After the reduction step is performed, the rows are permuted in such a way that all rows, which are going to be odd numbered during the next reduction step, are stored into the second section and all rows, which are going to be even numbered during the next reduction step, are stored into the fourth section. This permutation pattern is reversed during the back substitution stage. Fig. 2 illustrates this process.

The biggest advantage of this approach is that the odd and even numbered rows are located in separate sections and stored in a condensed form, thus allowing a more effective local memory access pattern when the next reduction step begins. Of course, this access pattern can still lead to bank conflicts especially when double precision arithmetic is used, as was also noted in [6]. The most straightforward solution would be to split the words and store upper and lower bits separately but this approach was actually found to be slower. The second advantage is that the remaining right-hand side vector rows are once again divided into independent parts which can be processed separately and therefore the implementation is capable of solving systems with the number of even numbered rows higher than the used work group size.

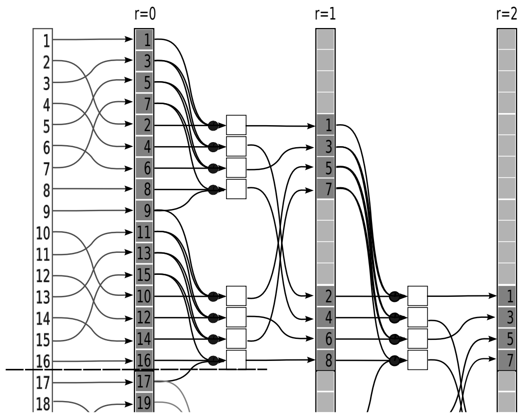


Fig. 2. The permutation pattern during the second stage of the tridiagonal system solver. The work group size is four. The numbers correspond to the row indexes.

Third Stage. The last stage uses a similar row permutations as the second stage. The system is preprocessed in such a way that all even numbered rows are stored to the beginning of the memory buffer, followed by all odd numbered rows. Every work item computes at most one row. After the reduction step is performed, the rows are permuted in such a way that all rows, which are going to be even numbered during the next reduction step, are stored into the beginning of the memory buffer, followed by all rows, which are going to be odd numbered during the next reduction step. This final stage seems to be identical with the algorithm used in [6].

4 Numerical Results

The GPU tests are carried out using Nvidia GeForce GTX580 GPU with 512 processing elements (cuda cores). The CPU tests are carried out using Intel Core i7-870 2.93 GHz processor with 4 cores (8 threads). The CPU implementations are written using standard C and OpenMP framework. The CPU implementations utilize the simplified cyclic reduction, which is in this case faster than the Thomas method. All test are performed using double precision floating point arithmetic.

Fig. 3 shows results for the two-dimensional Poisson problem. Expected-line shows the expected run time difference based on (17) and (18). However, it does not take into account the memory usage and other differences. The CPU results seem to show quite constant relative run time difference between the methods. The GPU results show a much more complicated pattern. The higher than expected run time difference in the case of the small problems can be explained by the fact that the radix-4 BCR method has more parallel and less serial computation. Thus the radix-4 BCR method is better capable of taking advantage of

GPU's parallel computing resources while the radix-2 BCR methods leave some of the processing elements partially unutilized.

While the radix-4 BCR method increased the amount of parallel computation, it also made it more difficult to achieve high memory throughput because the process of forming the right-hand side vectors for the sub-problems became more complicated. This is the most probable reason for the sudden drop in the performance when the problem size exceeds 1023^2. Fig. 4 shows the results for the three-dimensional Poisson problem. CPU and GPU results seem to correspond

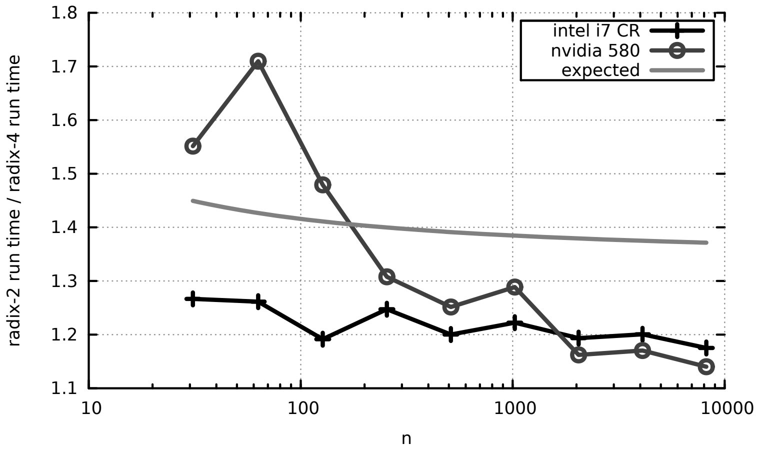


Fig. 3. Run time comparison between the radix-2 and radix-4 BCR methods, two-dimensional case, $n_1 = n_2 = n$

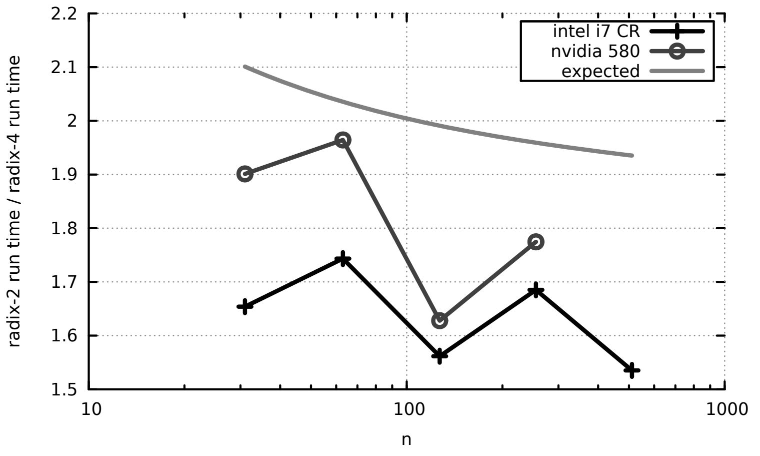


Fig. 4. Run time comparison between the radix-2 BCR and radix-4 BCR methods, three-dimensional case, $n_1 = n_2 = n_3 = n$

to the expectations. The sawtooth pattern is due to the modifications discussed in section 2.2.

Fig. 5 shows the relative run time differences between the radix-4 BCR CPU implementation and the radix-4 BCR GPU implementation. The GPU implementation is up to 6-fold faster when the transfer time between RAM and VRAM is ignored. The results for the three-dimensional GPU implementation are more modest but the GPU implementation is still up to 3-fold faster for the biggest problem.

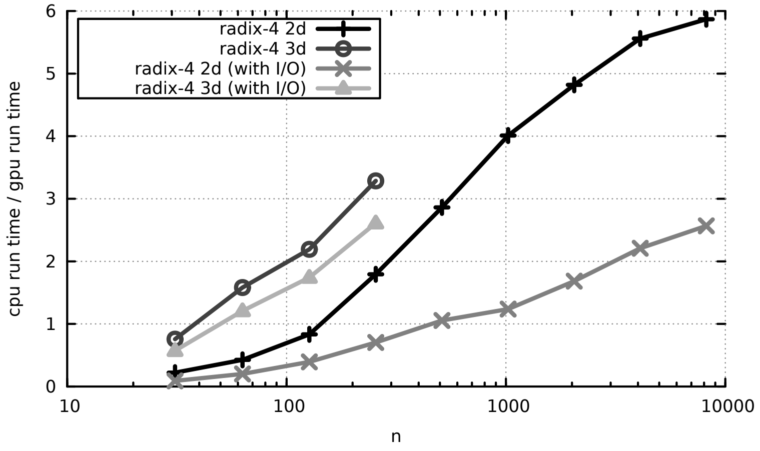


Fig. 5. Radix-4 BCR run time comparison between Intel Core i7 quad-core CPU and Nvidia GeForce GTX580 GPU, with and without initial RAM to VRAM transfer (I/O), $n_1 = n_2 = n_3 = n$

5 Conclusions

This paper covered the implementation of two block cyclic reduction methods for a GPU. Special attention was given to the tridiagonal system solver. A few new ideas were introduced to improve the efficiency of the tridiagonal solver on GPUs. According to the numerical results, the block cyclic reduction algorithm seems to offer a sufficient amount of fine-grained parallelism when combined with the cyclic reduction method. The observed speed differences between the radix-2 and radix-4 methods suggests that the radix-4 version is indeed better able to take advantage of GPU's parallel computing resources.

Acknowledgements. The authors thank the reviewers for offering valuable feedback. The presentation of the paper was significantly improved thanks to their comments and suggestions. The research of the first author was supported by the Academy of Finland, grant #252549.

References

1. Thomas, L.H.: Elliptic Problems in Linear Difference Equations Over a Network. Technical report, Watson Sc Comput. Lab. Rept, Columbia University, New York (1949)
2. Hockney, R.W.: A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. J. Assoc. Comput. Mach. 12, 95–113 (1965)
3. Kass, M., Lefohn, A., Owens, J.D.: Interactive Depth of Field Using Simulated Diffusion. Technical report, Pixar Animation Studios (2006)
4. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan Primitives for GPU Computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pp. 97–106. ACM, NY (2007)
5. Zhang, Y., Cohen, J., Owens, J.D.: Fast Tridiagonal Solvers on the GPU. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 127–136. ACM, New York (2010)
6. Gödeke, D., Strzodka, R.: Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid. IEEE Transactions on Parallel and Distributed Systems (TPDS), Special issue: High Performance Computing with Accelerators 22(1), 22–32 (2011)
7. Davidson, A., Owens, J.D.: Register Packing for Cyclic Reduction: a Case Study. In: Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units, pp. 4:1–4:6. ACM, NY (2011)
8. Lamas-Rodriguez, J., Arguello, F., Heras, D., Boo, M.: Memory Hierarchy Optimization for Large Tridiagonal System Solvers on GPU. In: IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, pp. 87–94. IEEE (2012)
9. Hockney, R.W.: The Potential Calculation and Some Applications. Methods Comput. Phys. 9, 135–211 (1970)
10. Swarztrauber, P.N.: The Method of Cyclic Reduction, Fourier Analysis and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle. SIAM Review 19, 490–501 (1977)
11. Buneman, O.: A Compact Non-Iterative Poisson Solver. Technical report 294, Stanford University Institute for Plasma Research, Stanford, CA (1969)
12. Sweet, R.A.: A Cyclic Reduction Algorithm for Solving Block Tridiagonal Systems of Arbitrary Dimension. SIAM J. Numer. Anal. 14, 706–719 (1977)
13. Sweet, R.A.: A Parallel and Vector Variant of the Cyclic Reduction Algorithm. SIAM J. Sci. Stat. Comput. 9, 761–765 (1988)
14. Vassilevski, P.: Fast Algorithm for Solving a Linear Algebraic Problem with Separable Variables. C. R. Acad. Bulgare Sci. 37, 305–308 (1984)
15. Kuznetsov, Y.A.: Numerical Methods in Subspaces. In: Marchuk, G.I. (ed.) Vychislitel'nye Processy i Sistemy II, vol. 37, pp. 265–350. Nauka, Moscow (1985)
16. Kuznetsov, Y.A., Rossi, T.: Fast Direct Method for Solving Algebraic Systems with Separable Symmetric Band Matrices. East-West J. Numer. Math. 4, 53–68 (1996)
17. Rossi, T., Toivanen, J.: A Parallel Fast Direct Solver for Block Tridiagonal Systems with Separable Matrices of Arbitrary Dimension. SIAM J. Sci. Comput. 20(5), 1778–1793 (1999)
18. Banegas, A.: Fast Poisson Solvers for Problems with Sparsity. Math. Comp. 32, 441–446 (1978)
19. Kuznetsov, Y.A., Matsokin, A.M.: On Partial Solution of Systems of Linear Algebraic Equations. Sov. J. Numer. Anal. Math. Modelling 4, 453–468 (1978)

20. Bini, D.A., Meini, B.: The Cyclic Reduction Algorithm: from Poisson Equation to Stochastic Processes and Beyond. Numer. Algor. 51(1), 23–60 (2008)
21. Bialecki, B., Fairweather, G., Karageorghis, A.: Matrix Decomposition Algorithms for Elliptic Boundary Value Problems: a Survey. Numer. Algor. 56, 253–295 (2011)
22. Rossi, T., Toivanen, J.: A Nonstandard Cyclic Reduction Method, Its Variants and Stability. SIAM J. Matrix Anal. Appl. 20(3), 628–645 (1999)
23. Heller, D.: Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems. SIAM J. Numer. Anal. 13(4), 484–496 (1976)
24. OpenCL — The Open Standard for Parallel Programming of Heterogeneous Systems, http://www.khronos.org/opencl/
25. Hockney, R.W., Jesshope, C.R.: Parallel Computers: Architecture, Programming and algorithms. Hilger (1981)
26. Stone, H.S.: An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations. Journal of the ACM 20(1), 27–38 (1973)
27. Davidson, A., Zhang, Y., Owens, J.D.: An Auto-Tuned Method for Solving Large Tridiagonal Systems on the GPU. In: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, pp. 956–965. IEEE (2011)
28. Alfaro, P., Igounet, P., Ezzatti, P.: Resolucion de Matrices Tri-diagonales Utilizando una Tarjeta Gráfica (GPU) de Escritorio. Mecanica Computacional 29(30), 2951–2967 (2010)
29. Kim, H., Wu, S., Chang, L., Hwu, W.W.: A Scalable Tridiagonal Solver for GPUs. In: Proceedings of the International Conference on Parallel Processing, pp. 444–453. IEEE (2011)

Parallel Feature Selection for Regularized Least-Squares

Sebastian Okser[1,2], Antti Airola[1,2], Tero Aittokallio[1,3,4], Tapio Salakoski[1,2], and Tapio Pahikkala[1,2]

[1] TUCS - Turku Centre for Computer Science, Finland
[2] Department of Information Technology, University of Turku, Finland
[3] Institute for Molecular Medicine Finland (FIMM), University of Helsinki, Finland
[4] Department of Mathematics, University of Turku, Finland
`forename.surname@utu.fi`

Abstract. This paper introduces a parallel version of the machine learning based feature selection algorithm known as greedy regularized least-squares (RLS). The aim of such machine learning methods is to develop accurate predictive models on complex datasets. Greedy RLS is an efficient implementation of the greedy forward feature selection procedure using regularized least-squares, capable of efficiently selecting the most predictive features from large datasets. It has previously been shown, through the use of matrix algebra shortcuts, to perform feature selection in only a fraction of the time required by traditional implementations. In this paper, the algorithm is adapted to allow for efficient parallel-based feature selection in order to scale the method to run on modern clusters. To demonstrate its effectiveness in practice, we implemented it on a sample genome-wide association study, as well as a number of other high-dimensional datasets, scaling the method to up to 128 cores.

1 Introduction

With the rapid growth in the amount of information stored in databases, it has become necessary to develop methods that allow for intelligent data analysis and mining of meaningful patterns from large-scale complex datasets. In data intensive areas of science and engineering, such as biology, medicine, image recognition, natural language processing and signal processing, methods from the field known as machine learning [1] are being increasingly used to automatically derive predictive models from data. Machine learning is a commonly used method in which explicit programming of the models is not necessary, rather the learner is able to make predictions on new data based on prior seen observations from a training set. Classification and regression are the most commonly considered tasks in supervised machine learning. In classification, the aim is to predict which class of possible outcomes an example belongs to, such as the case of whether or not an individual has a particular disease, while in regression the goal is to predict a real valued outcome for each subject, such as the blood pressure of an individual. These predictions are based on variables describing the characteristics of the data points, commonly referred to as features. Due to both the loss of predictive power that can be attributed to irrelevant features, and the interpretability of the models, one of the fundamental tasks that is addressed in machine learning is feature selection [2]. Its aim is to find a minimal subset of the features, capable of describing the data which allows the development of compact and accurate predictive models.

Previously, a feature selection method known as greedy RLS has been proposed by some of the present authors [3,4]. The method implements a greedy forward selection search, where at each search step the feature that most improves the cross-validation performance of an RLS trained classifier, when added to the already selected feature set, is selected. Through the use of matrix algebra based shortcuts first introduced in [3, 4], the greedy RLS training algorithm allows orders of magnitude reductions in computational costs compared to straightforward implementations of this same search procedure for other similar learning algorithms. In this work, we speed up greedy RLS further by parallelizing it through the use of the Message Passing Interface (MPI). The parallelization achieves a highly efficient speedup with respect to the size of the dataset, the number of selected features, and the number of cores used, allowing the use of the algorithm on much larger problem sizes than what has been previously feasible. As datasets continue to grow, such advanced algorithms will be a necessary component of future analyses.

To demonstrate parallel greedy RLS's ability to work both efficiently and accurately on high-dimensional datasets, we ran a series of tests on various large scale simulated datasets. These were generated to represent the dimensionality of those which are commonly encountered in genome profiling applications such as those originating from genome-wide association studies (GWAS) or whole-genome next generation sequencing (NGS) platforms. Further, to demonstrate its predictive power in real world studies, the algorithm was implemented on a GWAS aimed at identifying the single-nucleotide polymorphisms (SNPs) that are associated with the prediction of Type 1 Diabetes. As these studies often contain hundreds of thousands to millions of SNPs and several thousand examples, they are a prime example of the need to select a subset of the features that, when their aggregate effect is accounted for, allow making accurate predictions of an individual's disease status [5]. Using all of the in a model can have the adverse effect of both forcing the inclusion of non-relevant features along with limiting the interpretability of those features that are relevant with respect to the dependent variable being predicted.

2 Methods

2.1 Preliminaries

In this paper, we restrict our consideration to the binary classification problem. We assume that there is access to a training set of m examples, each having n real-valued features, as well as a class label denoting whether the example belongs to the positive or to the negative class.

We use the following notation. Let $\mathbb{R}^n$ and $\mathbb{R}^{m\times n}$ denote the sets of real-valued column vectors of length, m and $m\times n$-matrices, respectively. Bold capital letters are used to denote matrices, bold lower case letters denote vectors, and calligraphic capital letters denote sets. By $\mathbf{M}_{i,:}$, $\mathbf{M}_{:,j}$, and $\mathbf{M}_{i,j}$, we refer to the ith row, jth column, and (i,j)th entry of the matrix $\mathbf{M} \in \mathbb{R}^{m\times n}$, respectively. Similarly, for index sets $\mathcal{R} \subseteq \{1,\ldots,m\}$ and $\mathcal{L} \subseteq \{1,\ldots,n\}$, we denote the submatrices of $\mathbf{M}$ having their rows indexed by $\mathcal{R}$, the columns by $\mathcal{L}$, and the rows by $\mathcal{R}$ and columns by $\mathcal{L}$ as $\mathbf{M}_{\mathcal{R},:}$, $\mathbf{M}_{:,\mathcal{L}}$, and $\mathbf{M}_{\mathcal{R},\mathcal{L}}$, respectively. By v_i we refer to the ith entry of the vector $\mathbf{v}$. Let $\mathcal{N} = \{1,2,\ldots,n\}$ denote the index set of the features.

Running greedy RLS produces a set $\mathcal{S}$, consisting of the indices of selected features and a prediction function $f(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}_{\mathcal{S}}$, where $\mathbf{w}$ is the $|\mathcal{S}|$-dimensional vector representation of the function, $\mathbf{x}$ denotes the data point for which the prediction is to be made, and $\mathbf{x}_{\mathcal{S}}$ is its projection to a feature vector whose entries are indexed by $\mathcal{S}$, where $\mathcal{S} \in \mathcal{N}$. The learned parameters corresponding to the selected features are stored in the vector $\mathbf{w}$.

We assume that we are provided a set $\{(\mathbf{x}^i, y_i)\}_{i=1}^m$ of m training examples, where $\mathbf{x}^i$ are n-dimensional feature vectors and y_i are real valued labels, with $y_i \in \{-1,1\}$ being the standard encoding for binary classification problems. By $\mathbf{X} \in \mathbb{R}^{m\times n}$ we denote a data matrix containing the feature vectors as rows, and by $\mathbf{y}$ we denote the m-dimensional vector containing all the training set labels. The RLS prediction function, for a fixed set of feature indices $\mathcal{S}$, is obtained by solving the following minimization problem [6]:

$$\mathbf{w}^* = \operatorname*{argmin}_{\mathbf{w}\in\mathbb{R}^{|\mathcal{S}|}} \left\{ \sum_{j=1}^{m} \left(\mathbf{w}^{\mathrm{T}}\mathbf{x}^j_{\mathcal{S}} - y_j \right)^2 + \lambda \|\mathbf{w}\|^2 \right\}, \tag{1}$$

where the first term measures the sum of squared errors made by $\mathbf{w}$ on the training data and the norm of $\mathbf{w}$ acts as a measure of model complexity, and $\lambda > 0$ is a regularization parameter controlling the trade-off between the two terms.

Solving (1) with respect to $\mathbf{w}$, we get

$$\mathbf{w} = (\mathbf{X}_{:,\mathcal{S}})^{\mathrm{T}}\mathbf{a}, \tag{2}$$

where

$$\mathbf{a} = \mathbf{G}\mathbf{y}$$

is a vector consisting of the so-called dual variables of $\mathbf{w}$,

$$\mathbf{G} = (\mathbf{X}_{:,\mathcal{S}}(\mathbf{X}_{:,\mathcal{S}})^{\mathrm{T}} + \lambda\mathbf{I})^{-1}, \tag{3}$$

and $\mathbf{I}$ is the identity matrix of size $m\times m$. The symbols $\mathbf{a}$ and $\mathbf{G}$ are used in the description of the greedy RLS algorithm below.

Greedy RLS [3,4] performs a greedy forward selection for RLS, in which one feature is added at a time into the set of selected features $\mathcal{S}$ until the size of the set has reached the desired number of selected features k. During each iteration, every feature that has not yet been added into the set of selected features will be tested for addition. As a selection criterion, the method uses the leave-one-out cross-validation (LOOCV) [7,8] error for RLS, trained with the currently selected features, $\mathcal{S}$, and the new feature to be tested. For computing the LOOCV, we use the classical result (see e.g [8]) indicating that the leave-one-out prediction for the jth training example can be computed from:

$$y_j - \frac{a_j}{d_j}, \tag{4}$$

where y_j is the label of the jth training example, a_j is the jth entry of $\mathbf{a}$, and d_j is the jth diagonal element of $\mathbf{G}$. This is a constant time operation given that the dual variables $\mathbf{a}$ and the diagonal elements of $\mathbf{G}$ have already been computed.

Greedy RLS implements the feature updates and LOOCV computations using matrix algebra based computational shortcuts and caching of the previously computed results. Because of these, the time and space complexities of greedy RLS are only $O(kmn)$ and $O(mn)$, respectively, where k is the number of selected features, m is the number of training examples, and n is the total number of features. These complexities represent a significant improvement over the traditional approach, in which the prediction function requires training from scratch during each iteration of the selection process and the LOOCV, resulting in $O(\min\{k^3m^2n, k^2m^3n\})$ time complexity. Still, for massive data sets, such as those encountered in whole genome next-generation sequencing, even the linear complexities of greedy RLS can be too expensive, demanding new approaches: such as the need for efficient parallelization.

2.2 Algorithm Description

Next, we describe the parallel greedy RLS algorithm. In Algorithm 1 we present detailed pseudocode of the method. Further, due to the technically complex nature of the algorithm caused by the heavy use of matrix algebra based shortcuts, we describe the high-level structure of the computations in Table 1.

In the beginning, the feature indices are divided among the cores in approximately equal portions. The index set $\mathcal{N}_p$ denotes the feature indices assigned for the p^{th} core. Based on the supplied indices, each core loads from the disk the corresponding portion of $\mathbf{X}$, denoted as $\mathbf{X}_{:,\mathcal{N}_p}$. This set contains one vector of length m for each feature in $\mathcal{N}_p$, with the entries of the vector corresponding to the ith feature being the values for this feature in the m training examples. The cache matrix $\mathbf{C}_{:,\mathcal{N}_p}$ is initialized to $\lambda^{-1}\mathbf{X}_{:,\mathcal{N}_p}$, since $\mathbf{G} = \lambda^{-1}\mathbf{I}$ when no features have been selected yet. These caches are necessary for speeding up the forthcoming computations in the feature selection process. Further, two additional cache vectors $\mathbf{a} = \mathbf{G}\mathbf{y}$ and $\mathbf{d} = \text{diag}(\mathbf{G})$, where $\text{diag}(\mathbf{G})$ denotes a vector that consists of the diagonal entries of $\mathbf{G}$, are initialized for each processor. After the initialization, Algorithm 1 performs k iterations, selecting on each iteration greedily one additional feature. The selection is based on computing the LOOCV error with the short-cut (see Equation 4) for the temporarily updated set of features $\mathcal{S} \cup \{i\}$, which requires the vectors $\mathbf{a}$ and $\mathbf{d}$ to be updated to take account of the

Algorithm 1 Parallel greedy RLS

```
 1: Initialize $\mathcal{N}_p \subset \mathcal{N}$    ▷ Subset of global feature index set $\mathcal{N} = \{1, \ldots, n\}$
 2: Initialize $\mathbf{X}_{:,\mathcal{N}_p}$    ▷ Load data through MPI File I/O
 3: Initialize $\mathbf{C}_{:,\mathcal{N}_p} = \lambda^{-1}\mathbf{X}_{:,\mathcal{N}_p}$    ▷ Local cache for speeding up computations
 4: $\mathbf{a} \leftarrow \lambda^{-1}\mathbf{y}$    ▷ Local cache vector
 5: $\mathbf{d} \leftarrow \lambda^{-1}\mathbf{1}$    ▷ Local cache vector
 6: if rank=0 then
 7:     $\mathcal{S} \leftarrow \emptyset$    ▷ Global set of selected features.
 8: $\mathcal{S}_p \leftarrow \emptyset$    ▷ Set of locally selected features.
 9: while $|\mathcal{S}| < k$ do
10:     $e_p \leftarrow \infty$
11:     $b_p \leftarrow 0$
12:     for $i \in \mathcal{N}_p \setminus \mathcal{S}_p$ do
13:         $\mathbf{u} \leftarrow \mathbf{C}_{:,i}(1 + \mathbf{X}_{:,i}^{\mathrm{T}}\mathbf{C}_{:,i})^{-1}$
14:         $\widetilde{\mathbf{a}} \leftarrow \mathbf{a} - \mathbf{u}(\mathbf{X}_{:,i}^{\mathrm{T}}\mathbf{a})$
15:         $\widetilde{\mathbf{d}} \leftarrow \mathbf{d} - \mathbf{u} \odot \mathbf{C}_{:,i}$
16:         $\mathbf{p} \leftarrow \mathbf{y} - (\mathbf{1} \oslash \widetilde{\mathbf{d}}) \odot \widetilde{\mathbf{a}}$
17:         $e_i \leftarrow (\mathbf{p} - \mathbf{y})^{\mathrm{T}}(\mathbf{p} - \mathbf{y})$    ▷ Squared LOOCV error for the $i$th feature
18:         if $e_i < e$ then
19:             $e_p \leftarrow e_i$
20:             $b_p \leftarrow i$
21:     if rank=0 then
22:         Gather from all processes $e_p$, $b_p$
23:         $q \leftarrow \mathrm{argmin}_i\, e_i$
24:         Broadcast process index $q$ to all processes
25:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{b_q\}$
26:     if rank=$q$ then
27:         $\mathcal{S}_p \leftarrow \mathcal{S}_p \cup \{b_p\}$
28:         Broadcast $\mathbf{X}_{:,b_q}$ and $\mathbf{C}_{:,b_q}$ to all processes
29:     $\mathbf{u} \leftarrow \mathbf{C}_{:,b_q}(1 + \mathbf{X}_{:,b_q}^{\mathrm{T}}\mathbf{C}_{:,b_q})^{-1}$
30:     $\mathbf{a} \leftarrow \mathbf{a} - \mathbf{u}(\mathbf{X}_{:,b_q}^{\mathrm{T}}\mathbf{a})$
31:     $\mathbf{d} \leftarrow \mathbf{d} - \mathbf{u} \odot \mathbf{C}_{:,b_q}$
32:     for $i \in \mathcal{N}_p$ do
33:         $\mathbf{C}_{:,i} \leftarrow \mathbf{C}_{:,i} - \mathbf{u}(\mathbf{X}_{:,b_q}^{\mathrm{T}}\mathbf{C}_{:,i})$
34: for $i \in \mathcal{S}_p$ do
35:     $\mathbf{w}_i \leftarrow \mathbf{a}^{\mathrm{T}}\mathbf{X}_{:,i}$
36: if rank=0 then
37:     Gather $\mathbf{w}_{\mathcal{S}}$
38:     return $\mathcal{S}, \mathbf{w}_{\mathcal{S}}$
```

Table 1. Structure of Algorithm 1

Lines	Meaning of the operations
1-8	Loading data matrix and label vector from disk, initializations
9-33	The outer loop selects k features
12-20	The inner loop tests one remaining candidate feature at a time
13-16	LOOCV predictions are computed for a model that is based on the already selected features and the currently tested candidate. The predictions are stored in $\mathbf{p}$
17	Squared error computed between predictions $\mathbf{p}$ and true labels $\mathbf{y}$
18-20	The feature index corresponding to lowest error is stored
21-28	Communication in order to determine the selected feature, and to share information needed to update the cache matrices
29-33	Cache matrices and vectors are updated in order to prepare for next round of selection
34-35	Each core computes its portion of the model $\mathbf{w}$
37	The master gathers the coefficients of the model
38	Program returns selected features and the corresponding model

extra feature i. For this purpose, the algorithm computes the temporary vectors $\widetilde{\mathbf{a}}$ and $\widetilde{\mathbf{d}}$ that correspond to the updated feature set. Given that $\mathbf{a}$, $\mathbf{d}$, $\mathbf{X}_{:,i}$, $\mathbf{C}_{:,i}$ are stored in local memory, the temporary vectors can be computed in linear time due to the well-known inversion identity often referred to as the Woodbury formula (see [3,4] for technical details).

The evaluation of feature candidates is performed in a distributed manner in the individual cores as described in Algorithm 1. Each core stores the index of the feature with the lowest LOOCV error among the ones assigned to it. The indices and the errors corresponding to the best features identified by the individual cores are gathered by the master process that checks, on line 23 of Algorithm 1, which of the cores found the best feature to be added. The index of the core is then broadcast to every core and the core in question, in turn, broadcasts the information required for updating the cache memories. This information consists of two vectors of length m. After receiving the information about the selected feature, each core executes Algorithm 1, lines 29-33, that performs the cache updates which enable the fast computation of the selection criterion during the subsequent rounds. The cache update operations employ the same Woodbury identity as the temporary updates in the LOOCV computations. In contrast to updating only $\mathbf{a}$ and $\mathbf{d}$, here the the cache matrix $\mathbf{C}$ also has to be updated. However, since the matrix is updated in linear time, the overall time complexity is the same as that of the LOOCV separately for each feature index.

In the detailed descriptions, the operations $\odot$ and $\oslash$ denote the element-wise multiplication and division of two vectors, respectively. The correctness of the algorithm is a direct consequence of the results proven in our previous work [3,4] and we refer the readers to these papers for further details.

2.3 Discussion about Parallelization

The parallelization of greedy RLS is based on an MPI implementation, that may be considered the current de facto standard for parallelizing algorithms in distributed memory environments. The parallelization uses a master-slave paradigm in which the master determines how to navigate through the search space along with handling accounting. All cores including the master evaluate a static subset of the feature set. The features are distributed based on block-column distribution of $\mathbf{X}$. There is a sequential order amongst the feature indices considered among the cores, and the load is balanced so that all cores have approximately the same sized dataset (see Figure 1).

Efficient parallelization requires careful consideration of how to divide the data and computations between the cores. A high-level flow-chart description of the parallelization is presented in Figure 2. In order to evaluate new features, the individual cores need the information about the set of already selected features. In addition to speed up the selection process the cores also take advantage of the relevant portions of the cached temporary results computed by the other cores. This information needs to be communicated between the cores every time a new feature has been selected. The communication of the cached results, in turn, guarantees the linear speedup of the algorithm with respect to the number of cores. This makes parallelization of greedy RLS substantially more complicated compared to traditional filter type of feature selection methods based on univariate statistical tests, that can be parallelized simply by computing the statistic of interest independently at each core on different feature subsets. In this parallel implementation the master processor determines the global optimal feature at each iteration through communication with the cores. The master core is also allocated a subset of the data, and performs the same computations as the slave processes. This helps to improve efficiency, especially in runs where the number of cores being used is low. While the same parallelism could have been also achieved with domain decomposition by replacing lines 21–24 with an MPI_Allreduce command, it was found through initial empirical tests that this alternative approach did not lead to significant differences in runtime results.

While the parallel algorithm can be shown to be efficient for selecting large numbers of features from high dimensionality datasets, it has some obvious limitations. As each core is allocated a portion of the complete feature set, the algorithm will not be useful in situations where the number of features is smaller than the number of cores used as this would lead to many processing units being left with no calculations to be done. Moreover, at each iteration a bottleneck occurs in which processors can not progress to searching for the next feature, until all cores have completed their current selection and a global optimal feature has been selected. Further it has been previously established that feature selection methods can be prone to overfitting when $m \ll n$, so it is important to validate the models on independent data [4].

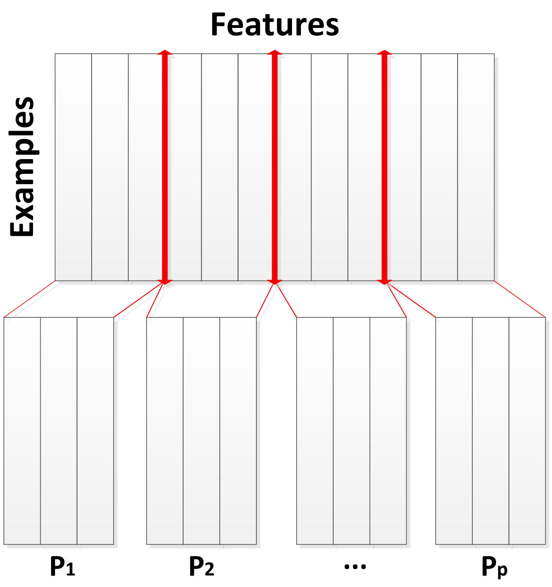


Fig. 1. The distribution of the dataset amongst the p processes. The data is split using block-column distribution.

3 Results

3.1 Scalability

The greedy RLS algorithm is expected to scale approximately linearly with respect to the number of features, examples and how many features are selected, an essential aspect to allowing the algorithm to be applied to large scale problems. To examine the scaling, efficiency and speedup of the method, a number of experiments were run to test the performance in various scenarios. These included varying the number of cores, adjusting the total number of features and the number of examples along with testing with selecting different numbers of features. While the algorithm can be expected to be highly scalable, there is a computational bottleneck to the parallelization, notably that after each core selects the optimal feature the data must be compared to the selected features from all other cores before they can move on to analyzing the next feature to select. Therefore, the parallel version is limited by the slowest core. We ran the experiments on Finland's IT Center for Computer Science's Vuori machine which is an HP CP4000 BL ProLiant supercluster with a theoretical peak of 34 Tflops/s and 304 compute nodes.

The tests run on the simulated data contained either 100,000, 500,000, 1,000,000 or 2,000,000 features, 1,000, 5,000 or 10,000 training examples and selected either 250, 500 or 1,000 variants. The sizes of the experiments were

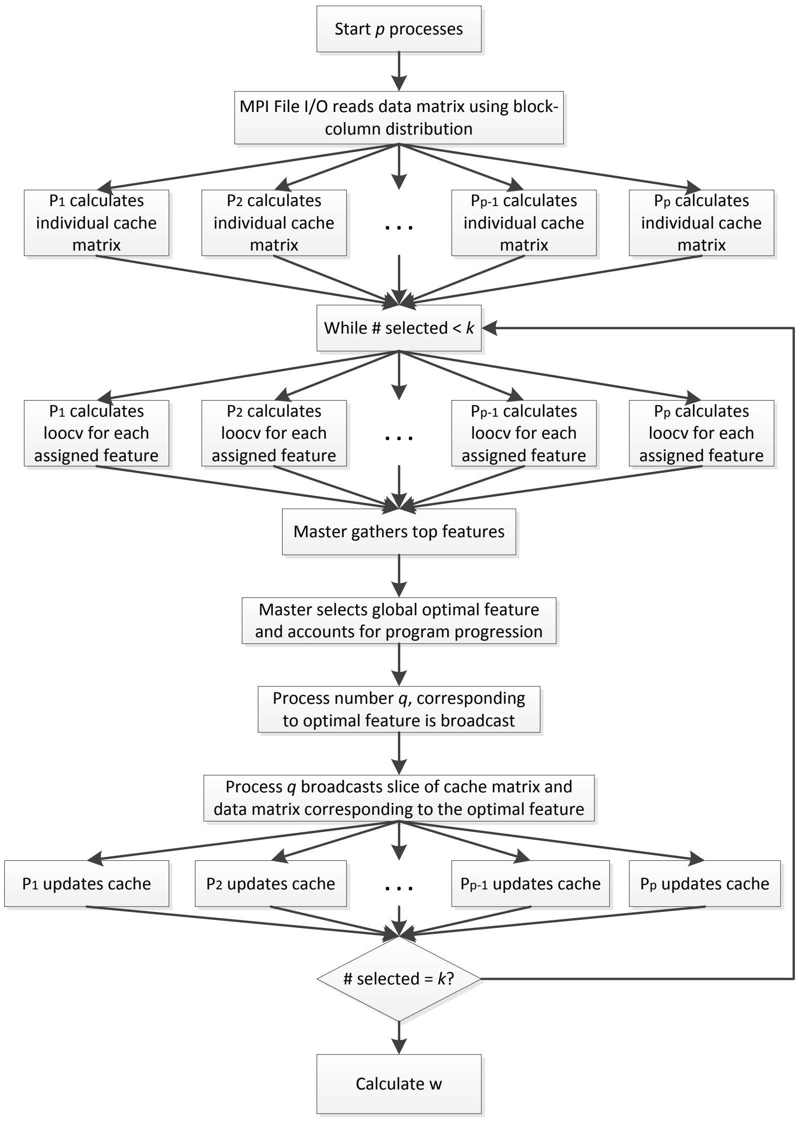

Start *p* processes
MPI File I/O reads data matrix using block-column distribution
P1 calculates individual cache matrix
P2 calculates individual cache matrix
. . .
Pp-1 calculates individual cache matrix
Pp calculates individual cache matrix
While # selected < *k*
P1 calculates loocv for each assigned feature
P2 calculates loocv for each assigned feature
. . .
Pp-1 calculates loocv for each assigned feature
Pp calculates loocv for each assigned feature
Master gathers top features
Master selects global optimal feature and accounts for program progression
Process number *q*, corresponding to optimal feature is broadcast
Process *q* broadcasts slice of cache matrix and data matrix corresponding to the optimal feature
P1 updates cache
P2 updates cache
. . .
Pp-1 updates cache
Pp updates cache
# selected = *k*?
Calculate w

Fig. 2. Flow-chart of the master-slave setup of parallel greedy RLS

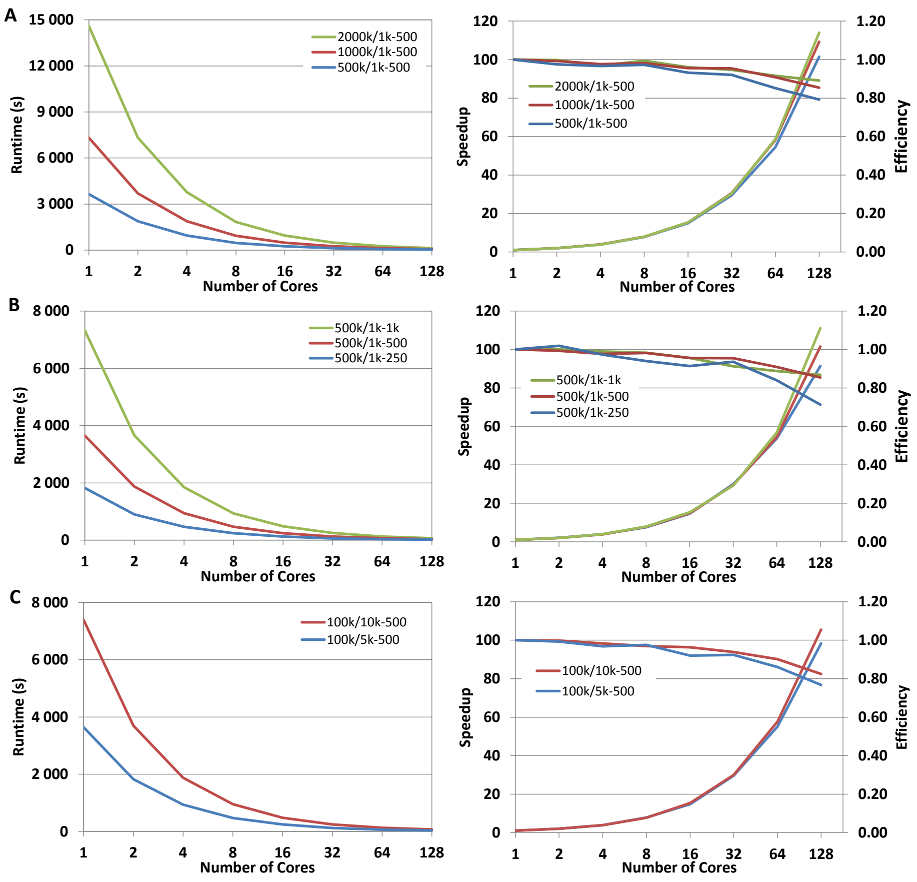


Fig. 3. Plots of the runtimes, speedup and efficiency measures for parallel greedy RLS. The different rows of the plot represent scaling the dimensionality of the dataset and selecting differing numbers of features. Rows ’A’, ’B’ and ’C’ represent scaling the feature set in the dimensions of the total number of features, the number of selected features and the number of examples, respectively. In the plot legend the specifications of each experiment are listed as # features/# examples-# selected. For example the line with the reference ’500k/1k-500’ means that the dataset contained 500,000 features, 1,000 examples and 500 features were selected.

selected to simulate dimensionalities that are a similar scale of those commonly used in GWAS. Each experiment was run with either 1, 2, 4, 8, 16, 32, 64 or 128 cores. Performing an analysis of these varying sized datasets, implemented on differing numbers of cores allows for verification of the algorithms scalability. The comparison for the speedup and efficiency were made against a sequential version of the algorithm. From the results seen in Figure 3, it can be observed that regardless of the number of data points, dimensionality of the dataset, or the number of selected features, doubling the number of cores roughly halves the required runtime. Thus, it is clear that the implementation scales well when increasing the number of cores.

A drop-off begins to become noticeable when using a higher number of cores. However, the performance decreases to a lesser scale on larger datasets implying that at this point the communication and startup costs are starting to use a significant portion of the running times. The largest drop-offs in both speedup and efficiency are noticed on the smaller datasets, in which the runtimes are minimal, being under one minute for the fastest runs. This indicates that using a large number of cores on small datasets, is inefficient and a waste of computing resources and these types of experiments should be done primarily on larger data studies.

Further tests of the algorithms capability of running on real data was carried out on the Wellcome Trust Case Control Consortium's Type 1 Diabetes combined with the UK National Blood Service's control GWAS cohort [9]. This allows us to examine the algorithms ability to make accurate predictions in a real world scenario where we can both quantitatively and qualitatively look at the results. An initial quality control filter was implemented to remove unsuitable features and examples [10]. The resulting dataset consisted of 405,508 features and 3,421 examples. To incorporate all possible data, a 3-fold external cross-validation was implemented, in which 2/3 of the dataset was used as a training set and the remaining 1/3 was used as an independent validation set. This is repeated until each 1/3 of the dataset has been used as the test fold once. Within each fold, the greedy RLS algorithm implemented an internal LOOCV when selecting the features in each external fold.

To gauge the ability for the algorithm to make accurate predictions, we used a scoring metric known as the area under the receiver operating characteristic curve (AUC). The AUC can be defined as the probability of ranking a randomly chosen positive instance higher than a randomly chosen negative one. Unlike many other performance measures, such as accuracy, the AUC is invariant to the relative class distributions. As defined in [11], given m^+ cases, m^- controls, $\hat{y}_j^+$ and $\hat{y}_k^-$ are predictions in which the individual is classified as a positive or negative class respectively and H is a transformation based on the Heaviside step function (5), the AUC can be defined by (6).

$$H(x) = \begin{cases} 0 & : x < 0 \\ 0.5 & : x = 0 \\ 1 & : x > 0 \end{cases} \tag{5}$$

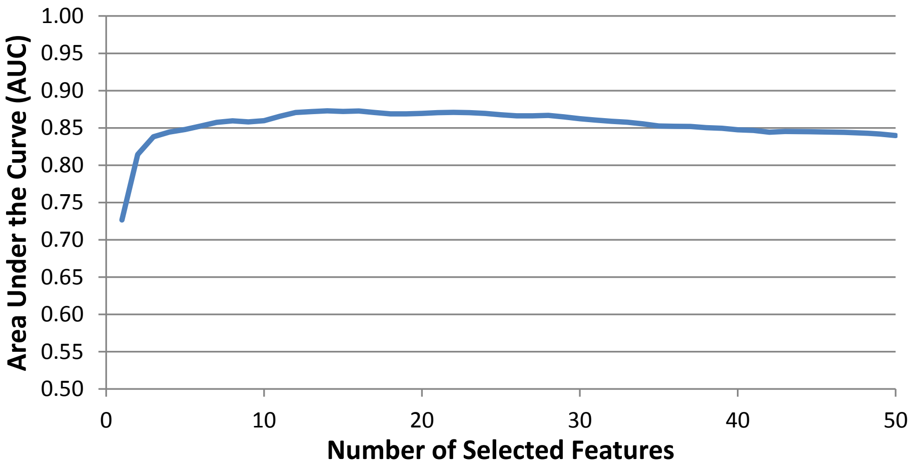


Fig. 4. AUC of the top fifty selected features when greedy RLS is averaged over the three test folds of the external cross-validation. The peak performance, 0.873, occurs after 14 features have been selected.

$$AUC = \frac{1}{m^+ m^-} \sum_{j=1}^{m^+} \sum_{k=1}^{m^-} H(\hat{y}_j^+ - \hat{y}_k^-) \tag{6}$$

The resulting scores over the external folds were then averaged to come up with an overall score for the model. As this score is an average, it should be interpreted as the potential of the algorithm to achieve a particular level of predictive performance rather than a final feature subset. The maximal AUC score of 0.873 was obtained after 14 features had been selected (see Figure 4).

3.2 Selected Features

The drawback of external-CV is that it does not readily allow the user to determine the set of selected features, since each round is likely to lead to a different set of selected features. Therefore, in order to select the final set of features, we ran the algorithm over the entire dataset. Then using the features selected over the complete cohort, it is possible to evaluate the model's suitability through qualitative analysis to see which features have been previously identified in similar studies. As well as these established features, it can be expected that a number of previously unidentified features would be selected. These features are significant in that they may be involved in epistasis interactions with the established variants.

Numerous studies have been conducted on Type 1 Diabetes GWAS datasets, allowing us to compare the results of our feature selection with other studies

Table 2. Final feature subset when selecting the top 14 features from the entire dataset. The number of features to select was determined by the point at which the average test AUC over the three folds of the external cross-validation peaked. The *Reported* column represents if the variant or its associated/nearest gene has been reported to have a possible association with T1D or is located in the *MHC* region. The base-pair locations are based off of the SNP positions in the GRCh37.p5 assembly. The reported SNPs are mapped to the closest gene within 20,000 base-pairs using dbSNP.

SNP	Gene	Reported	Chr.	Location
rs3957146	MTCO3P1	✓	6	32681530
rs377763	NOTCH4	✓	6	32199144
rs9270986	HLA-DRB1	✓	6	32574060
rs6679677	RSBN1	✓	1	114303808
rs492899	SKIV2L	✓	6	31933518
rs2894254	C6ORF10	✓	6	32345689
rs3130284	AGPAT1	✓	6	32140487
rs9272346	HLA-DQA1	✓	6	32604372
rs9275418	MTCO3P1	✓	6	32670244
rs17116117	HTR3B		11	113801591
rs17116145	HTR3B		11	113804326
rs2240063	CCHCR1	✓	6	31114745
rs4892855			X	2442667
rs16894932	OR10C1	✓	6	29415075

focusing on similar data sets. Even though the algorithm selected the top fifty features for each of the three nested-CV folds, we only evaluated the selected SNPs that were selected before the average optima in the AUC values were achieved. The SNPs were mapped to the closest located gene, with a threshold of 20,000 base-pairs using dbSNP [12]. Since the optimal AUC was reached after 14 features had been selected, we chose to only evaluate the features selected to this point. Of these features, many could be linked to the onset of Type 1 Diabetes in other studies giving a positive indication that our implementation is capable of identifying meaningful variants (see Table 2). Additionally, numerous selected SNPs are located in the MHC region, an area known to have a strong relation to the onset of Type 1 Diabetes [13]. This is an expected result as our method looks for sets of SNPs which are able to optimize the performance of a given model and in these sets we expect there to be both features that have been previously indicated to be causal SNPs along with those which remain to be identified as associated to the disease.

4 Conclusion

In this work, we have proposed a highly efficient parallelization of the greedy RLS feature selection algorithm. We show that the parallelization preserves the linear time complexity of the original serial algorithm and achieves a nearly inverse scaling with respect to the number of cores. We present a detailed description of the algorithm and a thorough experimental evaluation on real-world data. It was demonstrated that the algorithm was able to provide efficiently meaningful results in the WTCCC's Type 1 Diabetes GWAS. The results could be validated by qualitatively analyzing the selected variants, many of which had been noted in previous publications to be associated with the disease. The results demonstrate the potential of the proposed algorithm in solving large scale learning problems.

Acknowledgements. This project was funded by the Academy of Finland (grants 120569, 133227, 134020, 139884, 140880, 218310), Turun Yliopistosäätiö and the Finnish Cultural Foundation. This study makes use of data generated by the Wellcome Trust Case-Control Consortium. A full list of the investigators who contributed to the generation of the data is available from www.wtccc.org.uk. Funding for the project was provided by the Wellcome Trust under award 076113. Finally, we would like to thank CSC, the Finnish IT center for science, for providing us with extensive computational resources.

References

1. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
2. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. Journal of Machine Learning Research 3, 1157–1182 (2003)
3. Pahikkala, T., Airola, A., Salakoski, T.: Speeding up greedy forward selection for regularized least-squares. In: Draghici, S., Khoshgoftaar, T.M., Palade, V., Pedrycz, W., Wani, M.A., Zhu, X. (eds.) Proceedings of The Ninth International Conference on Machine Learning and Applications (ICMLA 2010). IEEE Computer Society (2010)
4. Pahikkala, T., Okser, S., Airola, A., Salakoski, T., Aittokallio, T.: Wrapper-based selection of genetic features in genome-wide association studies through fast matrix operations. Algorithms for Molecular Biology 7(1), 11 (2012)
5. He, Q., Lin, D.Y.: A variable selection method for genome-wide association studies. Bioinformatics 27(1), 1–8 (2011)
6. Hoerl, A.E., Kennard, R.W.: Ridge regression: Biased estimation for nonorthogonal problems. Technometrics 12, 55–67 (1970)
7. Lachenbruch, P.A.: An almost unbiased method of obtaining confidence intervals for the probability of misclassification in discriminant analysis. Biometrics 23(4), 639–645 (1967)
8. Elisseeff, A., Pontil, M.: Leave-one-out error and stability of learning algorithms with applications. In: Suykens, J., Horvath, G., Basu, S., Micchelli, C., Vandewalle, J. (eds.) Advances in Learning Theory: Methods, Models and Applications. NATO Science Series III: Computer and Systems Sciences, vol. 190, pp. 111–130. IOS Press, Amsterdam (2003)
9. Burton, P.R., Clayton, D.G., Cardon, L.R., Craddock, N., Deloukas, P., Duncanson, A., Kwiatkowski, D.P., McCarthy, M.I., Ouwehand, W.H., Samani, N.J., Todd, J.A., Donnelly, P., et al.: Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls. Nature 447, 661–678 (2007)

10. Evans, D.M., Visscher, P.M., Wray, N.R.: Harnessing the information contained within genome-wide association studies to improve individual prediction of complex disease risk. Human Molecular Genetics 18(18), 3525–3531 (2009)
11. Hanley, J.A., McNeil, B.J.: The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology 143(1), 29–36 (1982)
12. Sherry, S.T., Ward, M.H., Kholodov, M., Baker, J., Phan, L., Smigielski, E.M., Sirotkin, K.: dbsnp: the ncbi database of genetic variation. Nucleic Acids Research 29(1), 308–311 (2001)
13. Nejentsev, S., Howson, J.M.M., Walker, N.M., Szeszko, J., Field, S.F., Stevens, H.E., Reynolds, P., Hardy, M., King, E., Masters, J., Hulme, J., Maier, L.M., Smyth, D., Bailey, R., Cooper, J.D., Ribas, G., Campbell, D.R., Clayton, D.G., Todd, J.A.: Localization of type 1 diabetes susceptibility to the MHC class I genes HLA-B and HLA-A. Nature 450(7171), 887–892 (2007)

Part IV

Performance Analysis and Optimization

Topology Aware Process Mapping

Sebastian von Alfthan[1], Ilja Honkonen[1,2], and Minna Palmroth[1]

[1] Finnish Meteorological Institute, Finland
[2] Department of Physics, University of Helsinki, Finland
sebastian.von.alfthan@fmi.fi

Abstract. A parallel program based on the Message Passing Interface (MPI) commonly uses point-to-point communication for updating data between processes, and its scalability is ultimately limited by communication costs. To minimize these costs we have developed a library that reduces network congestion, and thus improves performance, by optimizing the placement of processes onto nodes allocated to the parallel job. Our approach is useful on production machines, as irregular communication patterns can at run-time be optimally placed on non-contiguous node allocations. It is also portable as it supports multiple architectures: Cray XT, IBM BlueGene/P and regular SMP clusters. We demonstrate on a Cray XT5m and an Infiniband cluster that good placement of processes doubles the total bandwidth compared to random placement and, furthermore, by up to a factor of 1.4 compared to to the original placement. It is not only important to place processes well on individual nodes, minimizing the number of link traversals on the Cray XT5m provides up to 20 % of additional performance. The scalability of a real-world application, Vlasiator, is also investigated and the scalability is shown to improve by up to 35 %. For communication limited applications the approach provides an avenue to improve performance, and is useful even with dynamic load balancing as the placement is optimized at run-time.

Introduction

In high performance computing good performance and scalability is of great importance to maximize scientific output. In order to achieve good parallel efficiency on a distributed memory computer it is important to maximize the fraction of total time a program spends on computing. In addition to computing time, parallel applications also spend time in transferring data and waiting in synchronization points, which does not directly help to advance the simulation. For a given implementation of a parallel algorithm based on domain decomposition, synchronization and communication costs can be minimized through load balancing. Load balancing with widely used libraries such as Zoltan [1] does no take into account the actual network topology of the machine, and thus processes handling neighboring subdomains can be far away from each other on the actual hardware. This can lead to higher communication and synchronization cost due to network contention as multiple messages have to cross the same network links, and due to higher latencies as messages have to hop over several network links. Optimizing the placement of subdomains to the compute units of a machine with a particular network topology has been studied extensively over the years, both based on physical optimization algorithms for mapping subdomains onto compute units, as well as in heuristic approaches. The problem itself is NP-hard [2], but to realize performance improvements only nearly optimal solutions are required. Bhanot et al. [3] introduced a off-line simulated annealing approach for optimizing placement on IBM Blue Gene/L supercomputers. More recently Bhatele et. al [4] developed a set of heuristics for mapping communication graphs onto Cray XT and IBM Blue Gene machines. The heuristic approach is not easily generalized; applications often exhibit irregular communication patterns and are allocated irregular computational sets of nodes. The MPIPP tool [5] is able to both discover the topology at run-time through ping-pong tests, as well as optimize the mapping problem using a K-way graph partitioning algorithm. In another recent set of publications Mercier and Jeannot [6,7] introduce the well performing TreeMatch algorithm for optimizing placement on multi-node NUMA architectures, and achieved good results when optimizing for data volume. The main limitation in this work is that the physical topology of the network is considered flat. Subramoni et. al designed [8] a scalable network topology detection service for InfiniBand networks, and a topology aware MPI library that takes advantage of this information.

Vlasiator [9] is a new simulation code where good scalability is a key requirement for simulating space plasma, both in local setups and on a global scale where space-weather, namely the interaction between the highly varying solar wind and Earth's magnetosphere, is simulated. This system is of great interest as near-Earth space provides a extremely rich environment for studying a wide range of plasma phenomena unreachable in laboratories. By combining simulation results with measurements from satellites and ground-based instruments, new fundamental insights may be reached. There are also practical interests, as events such as solar flares and coronal mass ejections may cause space storms disturbing the operation of satellites, GPS signals, and even electric grids and other long conductors on earth through geomagnetically induced currents.

To investigate the impact of process mapping and to improve the scalability of Vlasiator, we have developed a library that is able to optimize the placement of subdomains obtained from a partitioning algorithm onto compute nodes connected by a network. The hypothesis is that one can improve performance of parallel applications by improving the placement of processes in real-world supercomputers. To support this library has to be adaptable; both regular and irregular point-to-point communication patterns in applications should be described and mapped to irregular resource allocations. It also has to be portable; network topologies describing different machines should be automatically discovered. Finally, the optimization cost has to be reasonable so that it can be done repeatedly while the simulation is running and the load balancing changes. The other hypothesis is that one can gain additional speedup by also utilizing information on the actual network topology, but improvements are also seen by optimizing for compact placement on nodes.

Here we present the approach we have developed to tackle these challenges. We benchmark our library using a simple test case with nearest neighbor communication on two contemporary supercomputers. We also apply the approach to Vlasiator.

Optimizing the Topology Mapping

The approach taken here is geared towards minimizing communication time for an application where the majority of the communication is sparse, and each process only exchanges data with a few other processes. This situation occurs quite frequently, e.g., in applications where the solution of a numerical model at a certain location in a problem domain only requires local information to be solved. Such problems are typically parallelized using a domain decomposition approach, where each process solves the problem on a subdomain of the total problem domain. At each iteration the required data along the subdomain boundary is exchanged with the processes of the neighboring subdomains. If the processes of neighboring subdomains are distant on the actual machine network topology, the latency and bandwidth may degrade compared with the situation where they are on the same node, or on nodes close on the network topology. The domain is split into subdomains such that the computational load is equally divided using a load balancing scheme. These schemes do not take into account the network topology of the machine, and thus an imbalance in communication performance may still remain, even if the computational load is perfectly balanced. In problems with a static and regular domain decomposition it may be possible to place processes close on the network, but when the problem is irregular and dynamic, optimal placement requires a more involved scheme, such as the one described here.

The communication pattern of the application is described by a directed graph, where each vertex corresponds to a subdomain of the problem, and each directed edge corresponds to a send operation that transfers boundary data to a neighboring subdomain. Typically the edge weight $w_c(u_c, v_c)$ is the amount of data sent from the tail vertex u_c to the head vertex v_c. This communication graph is specified by the user in the application; each process adds its own relevant send operations and their weights.

The network topology of the machine is described by a complete graph. Each vertex corresponds to a compute unit, and each edge weight $w_n(u_n, v_n)$ describes the cost of sending data between the compute units u_n and v_n. This network topology is automatically discovered by the library; we have support for Cray XT and IBM BlueGene/P where the core location on the torus is known (torus-topology) and clusters where we only know which cores are on the same node (node-topology). The node coordinate on the Cray XT torus can be read using an undocumented system API [4]. For Cray XT and IBM BlueGene/P the weight of an edge between two processes on the same node is zero, and the weight between two processes on different nodes is the square root of the rectilinear distance of the two nodes. This is a heuristic choice that proved to give better results than a pure hop-byte metric [4] that is implied by having the weight being equal to the rectilinear distance. To model clusters we set the weight of inter-node edges to zero and intra-node edges to one, as we have no specific information on the network topology of the machine. This means that placement is optimized to minimize communication between nodes. Explicitly storing each weight would be unfeasible at scale as the storage requirement would scale as the square of the number of processes; instead we recompute the weights when needed. Earlier work has succeeded in measuring the network topology through ping-pong tests [5], or by directly discovering the InfiniBand topology [8]. In this work ping-pong tests were also briefly attempted, but on the fat-tree InfiniBand network and the Cray XT torus network used here, the measurement were dominated by noise and were not useful for constructing the network topology.

A mapping $v_n = M(v_c)$ of the communication graph onto the network topology graph describes on which core (network graph vertex), each subdomain (application communication vertex) is located. The cost of one communication edge is obtained by multiplying the weight of the communication edge with the network edge weight between the two processes to which the subdomains map. The total cost E of the mapping is the total cost of all communication edges averaged over all communication vertices,

$$E = \frac{1}{|V_c|} \sum_{\{u_c v_c\} \in E_c} w_c(u_c, v_c) w_n(M(u_c)M(v_c)), \tag{1}$$

where V_c is the set of communication vertices and E_c is the set of communication edges.

To optimize the mapping we use a standard simulated annealing algorithm. In this stochastic optimization scheme the mapping is iteratively modified by doing trial steps, which are accepted with a probability given by the Metropolis-Hastings acceptance probability function,

$$p(E_i, E_{i+1}, T_i) = \exp\left(-\frac{E_{i+1} - E_i}{T_i}\right), \tag{2}$$

where E_{i+1} is the cost of the mapping after the trial step, E_i is the cost of the current mapping, and T_i is a scaling factor corresponding to a temperature. If the trial step is rejected, the state is not modified. If it is accepted, the state of the trial step becomes the new state. The trial steps randomly pick two subdomains, and swap the processes onto which they are mapped. The cooling schedule defines how T_i depends on the time step i. In general, the optimization starts by randomizing the mapping, followed by an anneal starting from a high temperature where a large fraction of all steps are accepted, down to a low temperature with few accepted steps.

To get a good coverage of the phase space of the optimization problem we parallelize the algorithm so that each process independently optimizes the mapping using different cooling schedules. First the initial temperatures of each process is automatically tuned to achieve a desired acceptance rate, ranging from 0.1 on process 0, up to 0.5 on the last process. Then the cooling proceeds by decreasing the temperature every M steps by a process dependent factor selected randomly from the range of 0.975 to 0.995, until the temperature has been reduced 1000 times. We elected to scale M linearly with the number of processes N, so that the total number of states sampled in the optimization is proportional to N^2 indicating a good coverage of phase space. This also implies that the optimization time scales as $O(N)$. When testing various cooling schedules a value of $M = 4N$ was found to provide a good match between performance and quality for the cases studied here. At lower values the mapping had a higher cost, while slower anneals provided only marginal improvement.

Implementation

The library supports C++ applications, and comes in the form of a set of header files describing three kinds of classes: a communication graph describing the send operations of the application, a network graph describing the machine and a mapping class that can optimize the placement of communication vertices onto network vertices. The machine, and its queuing system does not need to provide any special support for our approach. The actual migration of the work has to be done by the application, based on the optimized mapping.

The communication graph is described by a graph class, and the vertices and edges and their weights are set by the application. First each process adds a vertex corresponding to its rank using the `addVertex` function in the graph, then each process adds its communication vertices (send operations) by setting receiving processes and the corresponding communication weights through the `addEdge` function. Finally the actual graph is constructed using the collective `commitChanges` call, after which all processes will have a complete copy of the communication graph. For example, if each process has two neighbors to which it sends data then one could initialize the communication graph for all processes in the `MPI_COMM_WORLD` communicator using the following piece of code.

```
Graph g(MPI_COMM_WORLD);
g.addVertex(rank);
g.addEdge(rank,neighbor1,weight1);
g.addEdge(rank,neighbor2,weight2);
g.commitChanges();
```

For each supported architecture there is a network graph class derived from the graph class. The constructor of the class will create a graph with a vertex for each compute unit allocated to the job, and will index these by the rank of the process located on the compute unit. As for the communication graph, each process will construct a complete copy of the whole network graph. For general clusters where only on-node placement is optimized the nodes are identified using `MPI_Get_processor_name`, and the graph describing the network is constructed as follows:

```
NodeNetwork t(MPI_COMM_WORLD);
```

On a Cray XT machine system libraries are used to discover the position of each process on the network, but the actual torus size has to be given by the user. For example, to initialize a torus with a size of $1 \times 12 \times 16$ the following piece of code is used:

```
CrayXtTorus t(1,12,16,MPI_COMM_WORLD);
```

The mapping class is used to optimize the placement, and its constructor takes as arguments the communication graph, the network graph and the communicator for which the graphs are defined. The mapping is then optimized using a call to the optimization function. After optimizing the mapping, one can read to which network vertex (i.e. rank) the subdomain on a rank should be transferred using the `getNetworkVertex` function. The actual exchange of subdomains has to be handled by the application. The following shows an example of optimizing the mapping:

```
Mapping<Graph,CrayXtTorus> m(g,t,MPI_COMM_WORLD);
m.simulatedAnnealingOptimizer();
destinationRank=m.getNetworkVertex(rank);
```

Results

We have tested the approach on two machines: Curie and Meteo. Curie is a Bull Bullx InfiniBand cluster based in France at Commissariat à l'énergie atomique et aux énergies alternatives (CEA). On the fat-node partition of Curie that was used for this work there are four eight-core processors per node, connected with a quad data rate InfiniBand network with a fat tree topology. Meteo is a Cray XT5m machine based in Finland at the Finnish Meteorological Institute. The SeaStar2 network is a 2D torus spanning 2 cabinets, and each node comprises two six-core processors.

On Curie we used the node-topology to model edge weights. This enables a optimized placement of simulation subdomains on nodes, minimizing network traffic. On a machine such as Curie, with a large number (32) of cores per node the network is potentially a significant bottleneck. As we have no in-depth knowledge of the network, we were not able to model the network itself in more detail. On Meteo we used both the same node-topology that minimize traffic sent from the node, as well as the torus-topology model that also reduce the number of hops that messages traverse over the network as subdomains exchanging data are placed closer to each other on the network.

Results from two tests are presented here: a benchmark showing the effective bandwidth and a real-world application Vlasiator for simulating space plasma.

Bandwidth Test

The first test is a benchmark corresponding to the communication needs of a stencil operation on a regular three-dimensional Cartesian grid with periodic boundary conditions. We construct a grid using the DCCRG [10] library[1], so that each process is allocated exactly one cell with a constant amount of data per cell. This removes potential performance artifacts by as it provides perfect load balancing, and enables us to send point-to-point data between two processes in one message from a contiguous block of memory. In the communication phase each process sends the cell data to its 26 nearest neighbors, and also receive their cell data in return. To measure the performance of the data exchange we measure the total bandwidth achieved in this operation. The performance is measured for three mappings: 1) The default one, where the subdomains (cells) are placed according to their index, so that columns of cells in the grid are placed on the same nodes. 2) Optimized placement where the total mapping cost has been minimized. 3) Randomized placement of subdomains.

In Figure 1 we have plotted the bandwidth as a function of the message (cell) size for a case with 1024 and 2048 cores on Curie. The benefit from proper placement is clear; the optimized placement provides significantly better throughput. The default placement has 20-50 % better bandwidth than the random placement. The optimized node placement further increases performance by 20 % for messages with a few kilobytes, and by 40 % for larger messages of several megabytes in size. Compared to random placement performance is close to 100 % better, both for smaller messages of a up to a kilobyte as well as for large messages with several megabytes of data. This suggests that the optimized placement affects both latency and bandwidth sensitive communication patterns.

The performance increase is readily understandable as a decrease in the total network traffic. For randomized mapping most cells that exchange data will be placed on different nodes, and almost all messages have to traverse the network. In our tests each

[1] http://gitorious.org/dccrg

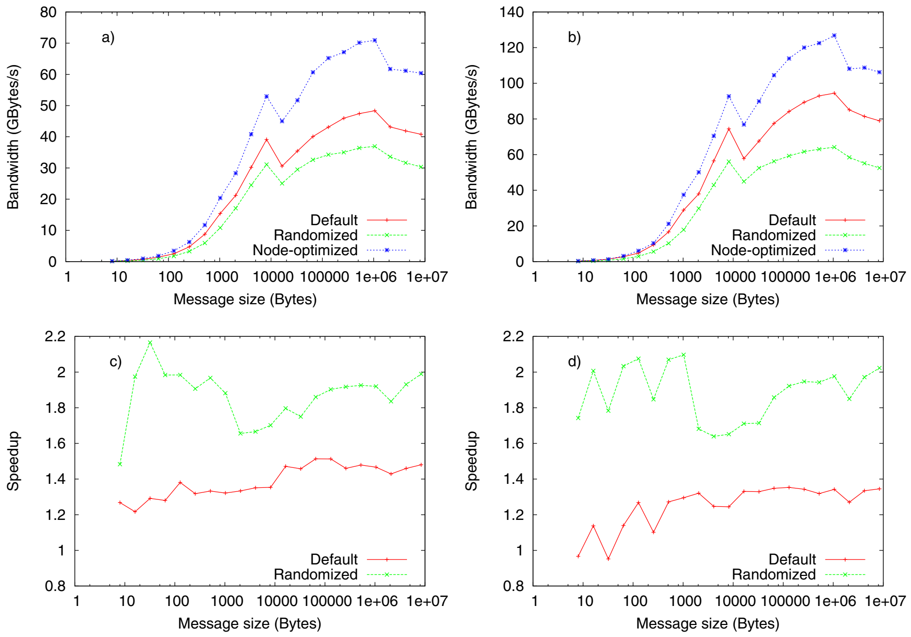


Fig. 1. Performance achieved using different mappings as a function of the size of each message on Curie, a quad data rate InfiniBand cluster with a fat tree topology. The total bandwidth is plotted for a) 1024 cores and b) 2048 cores. The speedup one can achieve by optimizing for the node-topology compared to random and default placement is also plotted for c) 1024 cores and d) 2048 cores.

process sent on average more than 25 out of a maximum of 26 messages over the network. The default placement is already better than the randomized one, as the column of cells assigned to a node is contiguous reducing the amount of messages to 21 in our tests. In the optimized placement the cells assigned to the processes on each node form a compact cluster, further decreasing the amount of data transferred over the network to on average 13.8 messages per process. The efficient on-node shared-memory MPI is thus able to handle nearly half of the messages, while the other half goes over the network. This corresponds closely to the doubling of bandwidth going from randomized placement to node-optimized placement.

Figure 2 shows the results on Meteo for 480 and 960 cores. The torus-topology optimized mapping shows up to 120 % better bandwidth than the random mapping and on the order of 30 % better bandwidth than the default mapping. If we only optimize for node placement the results are slightly worse, the added knowledge we have of the network gives us up to 20 % of additional performance.

Again the underlying reason for the performance increase is the reduced network traffic. In the default case on average 24 messages are sent over the network, showing that for the smaller nodes on Meteo the processes have neighbors on the same node only in one dimension of the grid. The randomized case again has close to the maximum

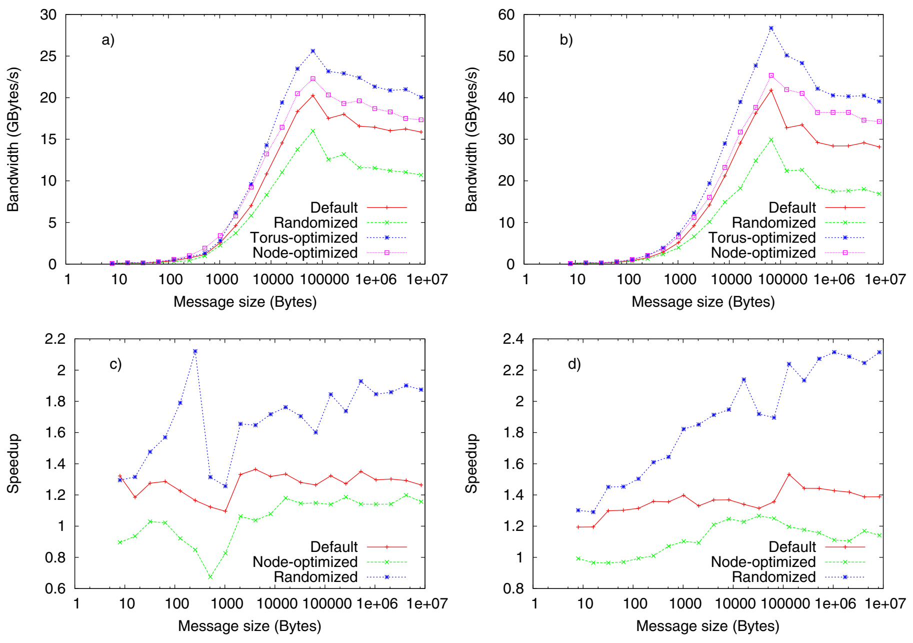


Fig. 2. Performance achieved using different mappings as a function of the size of each message on Meteo, a Cray XT5m machine with a SeaStar2 2D-torus network. The total bandwidth is plotted for a) 480 cores and b) 960 cores. The speedup achieved by optimizing for the torus-topology compared to random placement, default placement and placement optimized for the node-topology is plotted for c) 480 cores and d) 960 cores.

amount of messages sent over the network, with close to 25.7 messages being sent per process. As for the optimized case, only 18 cells on average need to be transmitted. This is higher than for Curie due to the fact that on Curie we can fit 32 cells (processes) per node, while on Meteo we can only fit 12 leading to a higher surface to volume ratio of the cluster of cells on each node.

Application Tests with Vlasiator

Vlasiator is based on the hybrid-Vlasov description, where ions are modeled as a 6-dimensional distribution function $f(\mathbf{r}, \mathbf{v}, t)$ in ordinary and velocity space, while electrons are modeled as a charge neutralizing massless fluid. The distribution function obeys the Vlasov equation,

$$\frac{\partial}{\partial t} f(\mathbf{r}, \mathbf{v}, t) + \mathbf{v} \cdot \nabla_r f(\mathbf{r}, \mathbf{v}, t) + \mathbf{a} \cdot \nabla_v f(\mathbf{r}, \mathbf{v}, t) = 0, \tag{3}$$

where $\mathbf{r}$ and $\mathbf{v}$ are the ordinary space and velocity space coordinates, acceleration $\mathbf{a}$ is given by the Lorentz force coupling the ion propagation to the electromagnetic fields, and $f(\mathbf{r}, \mathbf{v}, t)$ is the six-dimensional phase space density of ions with mass m and charge q. The distribution function is propagated forward in time with a finite volume method (FVM) method [11,12]. In the hybrid-Vlasov model electrons are not given a full Vlasov treatment, as that would be computationally too demanding for global simulations. Instead we couple the propagation of the ion distribution function to a field solver [13] for Maxwell's equation including Ohm's law, making self-consistent simulations possible. The ions couple to the field propagation through Ohm's law that includes the ion charge density and the bulk velocity from the zeroth and first moments of the velocity distribution.

We use DCCRG grid to construct a three-dimensional Cartesian mesh in the ordinary space, which is parallelized using a domain decomposition scheme with load balancing using the recursive coordinate bisection algorithm in the Zoltan [1] library. Each cell in ordinary space contains variables describing the electromagnetic field on its edges and faces, as well as a three-dimensional velocity mesh describing the full six-dimensional phase-space. In total the amount of data per cell is on the order of 1 MB, and a typical message size when sending data is 500 KB. Thus Vlasiator's main communication load is in the large-message range of the results from the bandwidth tests, where the benefit from the scheme was significant.

In Figure 3 the strong scalability of Vlasiator on Meteo is compared for two system sizes: The first has 27×27×1 cells in ordinary space, and the other has 54×54×1 cells. In the tests with Vlasiator we compared the default placement, with the one obtained by optimizing for torus-topology. The benefit from the placement optimization is most pronounced for larger core counts. For the large system we get at 1536 cores up to 35 % better performance through topology optimization. Contrary to the bandwidth benchmark, this test not only contains MPI communication, but also a significant portion of computation. At low process counts the compute time dominates, and improvements to MPI communication have little influence on the total performance. At higher process counts the MPI communication becomes dominant, as the amount of cells per process decreases. At low process count there is also a greater number of inner cells that can

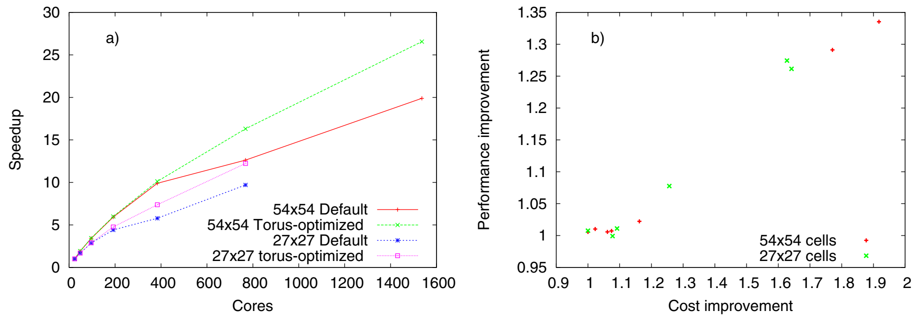


Fig. 3. The strong scalability of Vlasiator is shown in a) for two system sizes, using default placement as well as optimized placement for torus-topology. The speedup is computed relative to the performance on two nodes with 24 cores in total. In b) the correlation between the improvement in the cost of the optimized mapping and the improvement in the performance is plotted for the same two simulations.

be computed during communication, enabling one to better hide communication costs. Finally it is also evident from Figure 3 that the improvement in the cost closely correlates with the improvement in performance. Similar linear correlation also holds for the bandwidth tests.

Discussion

We have described the problem of minimizing MPI communication time of domain decomposed programs by optimizing the placement of subdomains on the physical hardware. This is done by mapping a graph representing the subdomains and their communication, to a graph representing cores and the network connecting them. The total cost of communicating with a certain mapping is minimized using simulated annealing, and thus irregular communication graphs can be mapped to irregular network topology graphs. This is important as that enables the approach to be used on production machines with multiple users, where the nodes allocated to users are not always close to each other on the network.

The optimization procedure does not guarantee that the global minimum is found; the final state will most likely be a good local minimum. The slower the cooling schedule is, the better minima are in general found as the algorithm is able to sample a greater portion of phase space. For optimizing the placement of subdomains on a network this is in general not a problem; as long as a local minimum with close to optimal cost is found the performance will increase. The time required to optimize the mapping was on 1000 cores on the order of 15 s, which is still reasonable if done infrequently. Slower cooling schedules only improved the cost marginally showing that close to optimal mappings are obtained. It should also be noted that the cooling schedule is most likely not optimal, and optimization time could be reduced by tuning it. As the time required to optimize the mapping scales linearly with the number of processes the mapping optimization would at petascale level (100 000 cores) already take almost half an hour for parallel applications utlizing only MPI, indicating that in the present form this technique is only useful for medium scale. However, the optimization problem should be scalable to higher core counts when optimizing a hybrid OpenMP-MPI program due to two reasons: the number of processes is smaller and each process could parallelize the optimization algorithm using threads. In modern supercomputers with tens of cores per node, this should enable the present approach to be viable at tens of thousands of cores.

On the two machines where we tested the approach we observed up to 40 % increase in total bandwidth compared to the default placement, and up to 100 % increase compared to randomized placement. This was true both for large and small messages, suggesting both improved bandwidth and latency. We also showed that the bandwidth was closely related to the amount of network traffic. Optimized placement of ranks on nodes through static placement rules is a common optimization technique, but it is not possible to do this if the program uses dynamic load balancing. This work shows how it is possible to achieve the same benefit even in this case. These results are very similar to the ones achieved by Subramoni et al. [8], with a similar 3D stencil benchmark they also observed up to 40 % increased performance. On the Cray XT5m machine we furthermore show that even a simple model for the cost of sending messages over multiple hops on the network boosts performance by up to 20 % over just minimizing the total amount of inter-node communication. For a hybrid OpenMP-MPI application with one process per node, the placement is already optimal for the node-topology. Optimizing for torus-topology can thus provide performance improvements even for these applications. For the Vlasiator application we achieved up 35 % speedup at higher core-counts enabling the code to scale further and thus achieve its scientific goals in the field of space-weather. This can be contrasted with the 15 % speedup achieved by Mercier et al.[7] for the ZEUS-MP astrophysics code, and the 10-15 % speedup observed for the MILC code by Subramoni et al [8].

Acknowledgments. We acknowledge that the results in this paper have been achieved using the PRACE Research Infrastructure resource Curie based in France at CEA.

The project has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Starting Grant agreement number 200141-QuESpace. The work of SVA, IH and MP is supported by the Academy of Finland.

References

1. Devine, K., Boman, E., Heapby, R., Hendrickson, B., Vaughan, C.: Zoltan data management service for parallel dynamic applications. Computing in Science and Engg. 4(2), 90–97 (2002)
2. Díaz, J., Petit, J., Serna, M.: A survey of graph layout problems. ACM Comput. Surv. 34(3), 313–356 (2002)
3. Bhanot, G., Gara, A., Heidelberger, P., Lawless, E., Sexton, J.C., Walkup, R.: Optimizing task layout on the blue gene/l supercomputer. IBM J. Res. Dev. 49(2), 489–500 (2005)
4. Bhatele, A.: Automating topology aware mapping for supercomputers. PhD thesis, Champaign, IL, USA, AAI3425400 (2010)
5. Chen, H., Chen, W., Huang, J., Robert, B., Kuhn, H.: Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In: Proceedings of the 20th Annual International Conference on Supercomputing (ICS 2006), pp. 353–360. ACM, New York (2006)
6. Jeannot, E., Mercier, G.: Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part II. LNCS, vol. 6272, pp. 199–210. Springer, Heidelberg (2010)
7. Mercier, G., Jeannot, E.: Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 39–49. Springer, Heidelberg (2011)
8. Subramoni, H., Potluri, S., Kandalla, K., Barth, B., Vienne, J., Keasler, J., Tomko, K., Schulz, K., Moody, A., Panda, D.K.: Design of a scalable infiniband topology service to enable network-topology-aware placement of processes. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2012), pp. 70:1–70:12. IEEE Computer Society Press, Los Alamitos (2012)
9. Palmroth, M., Honkonen, I., Sandroos, A., Kempf, Y., von Alfthan, S., Pokhotelov, D.: Preliminary testing of global hybrid-vlasov simulation: Magnetosheath and cusps under northward interplanetary magnetic field. J. Atm. Solar Terr. Phys. (in press), http://dx.doi.org/10.1016/j.jastp.2012.09.013

10. Honkonen, I., von Alfthan, S., Sandroos, A., Janhunen, P., Palmroth, M.: Parallel grid library for rapid and flexible simulation development. Comp. Phys. Comm. (in press), http://dx.doi.org/10.1016/j.cpc.2012.12.017
11. LeVeque, R.J.: Wave propagation algorithms for multidimensional hyperbolic systems. J. Comput. Phys. 131(2), 327–353 (1997)
12. Langseth, J.O., LeVeque, R.J.: A wave propagation method for three-dimensional hyperbolic conservation laws. J. Comput. Phys. 165(1), 126–166 (2000)
13. Londrillo, P., Zanna, L.D.: On the divergence-free condition in godunov-type schemes for ideal magnetohydrodynamics: the upwind constrained transport method. J. Comput. Phys. 195(1), 17–48 (2004)

Numprof: A Performance Analysis Framework for Numerical Libraries

Olli-Pekka Lehto

CSC - IT Center for Science Ltd.,
PO Box 405, 02101 Espoo, Finland

Abstract. This paper introduces Numprof, a profiling framework for performance analysis of numerical libraries. The framework consists of a profiler and replayer for the BLAS and FFTW3 libraries. The profiler records library call events with a user configurable amount of detail. The replayer can be used to execute library calls based on the profiling trace files generated by the profiler. We explore real-world use cases for the framework and demonstrate that due to its low overhead it is feasible to be used for continuous statistical analysis of numerical library calls.

Keywords: numerical libraries, linear algebra, fourier transforms, profiling.

1 Introduction

Today's High Performance Computing (HPC) resources are being leveraged for an increasing array of different applications in a variety of disciplines. However, there are common numerical methods which which are employed by many of these applications. They often comprise of the most computationally intensive sections of programs and thus are central to their overall performance. Spectral methods involving Fourier Transforms and dense linear algebra are prevalent. [SA]

Different implementations of these common numerical methods can be found in various numerical libraries. These are developed by the Open Source Community such as Netlib BLAS [JD] and FFTW [MF], available from third parties such as NAG IMSL and system vendors such as Intel MKL.

The performance of these numerical libraries typically varies depending on the exact routine called, its parameters and underlying architecture. Thus in cases where multiple implementations of a library are available, such as in the case of BLAS, the selection of the implementation also plays a major role. [DE] [AS]

The ubiquitous nature of the numerical libraries means that understanding how they are used and how they perform is fundamental in analyzing the performance of a variety of HPC applications. Due to the variety of parameters affecting performance, fairly detailed information about each library call should be recorded. Ideally this type of analysis should have low impact on the overall runtime of the application and be as simple as possible to implement, ideally with no changes to the source code and no need for recompilation. This has served as the motivation to develop the Numprof framework.

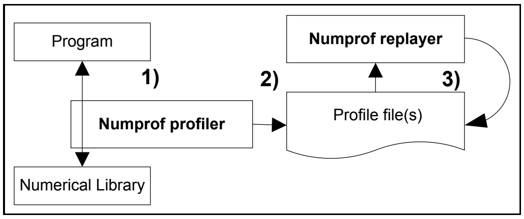


Fig. 1. Numprof dataflow

2 Numprof

Numprof is a stand-alone C-based performance analysis framework for numerical libraries that is simple and intuitive to install and use by both end users and HPC site administrators. The framework consists of two main components, the *profiler* and *replayer*. The basic dataflow is illustrated in Figure 1: The profiler intercepts calls from the program to the numerical library and gathers data from the calls (1). At the end of execution, the data is saved to profile output file(s) (2). The replayer takes a profile file as input and re-executes the sequence of calls listed in it (3). The replayer can be executed on the same or a different system and linked with a different implementation of the numerical library. The output from the replayer is in the same profile file format as the profiler output files.

At the time of writing, Numprof supports the BLAS API and has limited support for the FFTW3 numerical library API. The framework currently consists of separate profiling libraries and replayer tools for both libraries. The framework is designed to be modular and extendable to support new libraries.

2.1 Profiler

The profiler is compatible with any application that is dynamically linked to a supported numerical library and there is no need to recompile the application. The processing and memory overhead of the profiler is designed to be as small as possible and level of profiling detail adjustable by the user according to their requirements.

The profiling mechanism is illustrated in Figure 2. The profiler library is loaded using the `LD_PRELOAD` environment variable which forces the library to be loaded before any other library. The profiler contains a set of "interposer" symbols which intercept the calls to the numerical library and redirect them to the profiler library (1). The library collects data from the call parameters to an internal data structure (2) and relays the call to the actual numerical library (3). When the numerical library call returns, the execution time and (optionally) detailed information about the return data is also recorded (4). Once the program reaches its end, the collected data is written into a profile file (5).

A similar interposer-based profiling methodology has previously been successfully implemented in domain-specific profiling libraries focusing on other performance-critical areas such as MPI communication [JV] and disk I/O [PR].

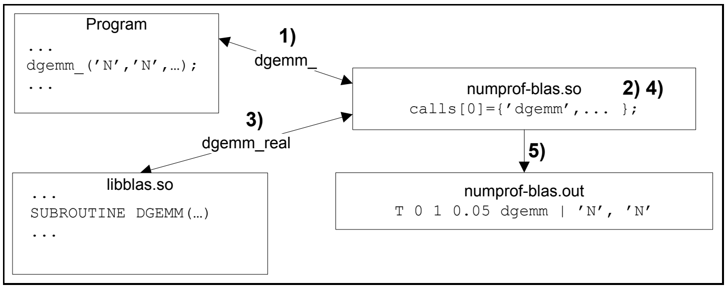


Fig. 2. Numprof profiler structure

Currently the library has been tested on x86-64 -based architectures using the Linux OS. This approach can also be utilized in other architectures such as Windows and OS X, although the exact implementation mechanisms differ. [MD]

On Linux, the interposer symbols are created by using the GNU linker's `--wrap` command-line parameter. The wrapper for each routine is in a separate, self-contained source file containing all the functions and data structure associated with the call. This self-containment makes it possible to easily customize the library to include a subset of the routines available simply by linking just their respective files to the profiler library.

Below is a sample of the command to launch an application with the profiles:

```
NUMPROF_LEVEL=1 NUMPROF_OUTFILE=myrun-blas.prof \
LD_PRELOAD=./numprof-blas.so ./myprogram
```

The level of detail can be adjusted using the `NUMPROF_LEVEL` environment variable: The profiler currently has three levels of detail for the profiling records:

1. The amount of calls and accumulated time for each routine. Also basic overall resource usage statistics are produced such as memory overhead of the profiler and time spent in the numerical library calls.
2. Basic data is gathered for each individual call. This includes the timestamp, call duration, call name and basic call parameters, such as matrix dimensions.
3. Deep analysis of input and output data arrays. Currently the minimum and maximum values and the fill rate of each array is recorded.

In addition to the summary data, there is also the capability to dump input and output data of individual calls using the `NUMPROF_DUMPMAT` environment variable. The variable can be used to specify a comma-separated list of call sequence numbers and/or a library routine names to be dumped. For example, to dump calls 12 and 41 as well as all DGEMM calls:

```
NUMPROF_DUMPMAT=12,41,dgemm
```

The data files are stored in the Matrix Market (MM) [RB] format which makes them straightforward to postprocess with third party tools.

The detail levels offer a progressively increasing amount of information, but each level also has higher memory and processing overhead.

Below is an example of a basic level 1 output summary from a GPAW run:

```
Callname: zaxpy    callcount: 18438     time_sec: 11.267141
Callname: dgemm    callcount: 3         time_sec: 0.003322
Callname: zgemm    callcount: 15087     time_sec: 160.734917
Callname: zher2k   callcount: 21        time_sec: 47.707324
Callname: zherk    callcount: 5778      time_sec: 33.176687

Total_time: 580.639131    BLAS_time_sec: 252.889391
BLAS_time_pct: 43.55
BLAS_cputime_sec: 252.890000   BLAS_cputime_pct: 43.55
callcount: 39327    Memory_overhead_bytes: 7977840
```

With detail level ≥ 2, individual calls are traced: The following is an example of a `dgemm` call:

```
T 2 86790 79.367988 1.274598 | dgemm | transa=N transb=N m=4772
n=4772 k=676 alpha=-1.000000 *a=[d_array] lda=5448 *b=[d_array]
ldb=5448 beta=1.000000 *c=[d_array] ldc=5448
```

The output is split into different fields by using the pipe characters and displayed on a single line for easy parseability. The first field contains the MPI task number, an identifier number for the call, time offset for the call (seconds from start of program execution) and duration of call. The second field contains the function name. The third field contains the function parameters for the routine. In case of pointers to arrays, the parameter is replaced by `[*_array]` notation where * indicates the datatype (In the example case, `d`stands for double precision floating point).

With detail level 3, the contents of each array are analyzed. The data is appended as a fourth field into the aforementioned trace file (after function parameters). Here is an example of an output:

```
a_max: 18.000000 a_min: 2.000000 a_fill: 0.900000
b_max: 36.000000 b_min: 4.000000 b_fill: 0.900000
c_max: 3810.000000 c_min: 570.000000 c_fill: 1.000000
```

The minimum and maximum values for each array (a,b,c) are displayed as well as the fill ratio (ratio of non-zero entries to total number of entries).

2.2 Replayer

The replayer tool reads in a Numprof profile file with detail level ≥ 2 and executes the routines in the same sequence as described in the profile file. Currently the

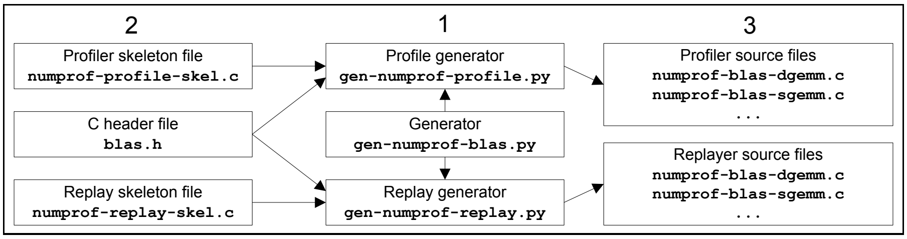


Fig. 3. Numprof generator structure

tool autogenerates dummy arrays for input. However, support for using data from dumped arrays as input is planned.

2.3 Generators

The Numprof generators are a set of of Python scripts for creating new libraries based on function prototypes. The generators simplify the process of extending Numprof to support additional numerical libraries.

The structure of the BLAS generator is illustrated in Figure 3. The script `gen-numprof-blas.py` is the main script for starting the generation. It calls routines in the module files
`gen-numprof-profile.py` and `gen-numprof-replay.py`.

These scripts read the function prototypes from the C header file `blas.h` and the function template files
`numprof-profile-skel.c` and `numprof-replay-skel.c`.

The library generation is fully automatic from a C header file if all the functions parameters consist of basic datatypes. If the parameters or return values include pointers to arrays, the basic data collection (detail level 1) will still be generated but more advanced functionality (profile detail level ≥ 2 and replay) require defining the relationship between the array dimensions and parameters.

3 Use Cases

In this section we explore some of the potential applications of the Numprof framework.

3.1 Accelerator Capacity Planning

In the last few years computation using graphics processing units (GPUs) has become an integral part of the HPC ecosystem and Intel's upcoming Xeon Phi coprocessor is also gathering considerable interest. Both technologies feature considerably higher theoretical computational performance than traditional CPUs. However, both the actual achievable optimal performance as well as the porting and optimization effort to reach this performance varies depending on the application.

Implementations of FFT and BLAS APIs which use accelerators, such as CUFFT [CF] and CUBLAS [CB] have been developed. These libraries are considered as the easiest way to leverage the power of accelerators, provided the applications use numerical library calls extensively and have sufficiently large data arrays that the data transfer cost between the host and accelerator is amortized.

The Numprof profiler can be used to identify which applications satisfy this requirement and to what extent. Based on this information, an estimate can be made of what kind of minimum baseline application performance increase one could expect from simply offloading the numerical library calls to the accelerators. Having such "performance guarantee" data could be particularly useful for HPC site administrators investigating the feasibility of adopting accelerators.

3.2 Benchmarking

Using the Numprof profiler, HPC site administrators can collect information on the numerical library calls on a systemwide basis. This data can be directly used by the Numprof replayer to create a numerical library benchmark which represents the actual usage patterns of the libraries. These benchmarks could potentially be very practical for system procurements.

3.3 Debugging and Optimization

The Numprof profiler can be used for extracting intermediate results from the program using the matrix dump functionality. This is much faster and dynamic than the traditional way of instrumenting the code manually with print commands, for example.

Using single precision numerical library calls offer superior performance to double precision calls. However, using single precision can be detrimental to the accuracy of results. The profile data enables the user to analyze the individual library calls in order to determine the error introduced by using single precision operations.

The fill rate information makes it possible to quickly identify operations involving very sparse data sets (with many zeroes). For these, replacing the routines optimized for dense data with sparse algorithms and data storage types tends to offer superior performance and a smaller memory footprint. The matrix dump capability can be used to extract the matrices to study the sparsity pattern, which affects the choice of optimal algorithm and storage format.

The FFTW library optimizes execution for various sizes of FFTs on a given architecture using plans. Before executing a specific type of FFT with specific dimensions, a planning phase must be executed where the plan is constructed. The level of optimization can be controlled using environment variables. The planning step using the higher levels of optimization can be extremely time consuming. To save time, the precomputed plans can be saved in a "wisdom" file and loaded from this file whenever a FFT routine using the same parameters is executed. With Numprof it is possible to gather data from FFTW call parameters. This enables the site administrator to discover if there are any recurring

FFT routines and parameters. This information could then be used to produce a wisdom file with highly optimized plans for the most commonly used FFT routines.

4 Existing Approaches

There are several existing strategies for analyzing numerical libraries, which each have some limitations:

1. Analyzing the code visually. This is feasible only if the application is very simple and the source code is available.
2. Instrumenting the code manually. This also requires changes to the source code and can be time-consuming if there are multiple different callsites to the library.
3. Using a instrumenting profiling tool such as `gprof` [SG]. The application may need to at least be relinked with the profiler flag. The `tprof` profiler is a notable exception as no relinking is needed. However it is only available on the AIX OS.
4. Using the `ltrace` command. While no code changes are necessary, there is a slight performance penalty.

As can be observed, each of these approaches have limitations which make them infeasibile for low-overhead, non-intrusive operation required by the aforementioned use cases. Furthermore, the automated methods (`gprof` and `ltrace`) are only able to collect the basic information on each callsite but lack the capability to provide qualitative information such as the fill rate of each matrix. Furthermore, none of the approaches incorporate a similar replay capability for the profile data that we have implemented.

5 Measurements

5.1 Overhead

Ideally the instrumentation should provide as much information as possible with little interference with the application being profiled. The following tests were performed on a dual-socket Intel Xeon E5 (Sandy Bridge) 2.7GHz with CentOS Linux 6.2.

In Table 1 we compare the overhead caused by the basic profiling overhead between the `ltrace` command, Intel Compiler 12.1 profiler and Numprof detail level 2. The overhead was measured as the average per-call overhead from 1000 successive runs of 100x100 DGEMM. It can be observed that the overhead of Numprof is roughly half of the overhead of ltrace. The overhead of Intel's profiler is negligible, but this required the code to be recompiled with the -p flag. To investigate the impact of the overhead we tested the benchmark using two different levels of verbosity. The results are illustrated in Figure 4. As can be observed, the overhead with detail level 2 is negligible. With level 3 the overhead

Table 1. Per-call temporal overhead of instrumentation

Numprof	1.14E-4 s
ltrace	3.1E-4 s
Intel	<1E-7 s

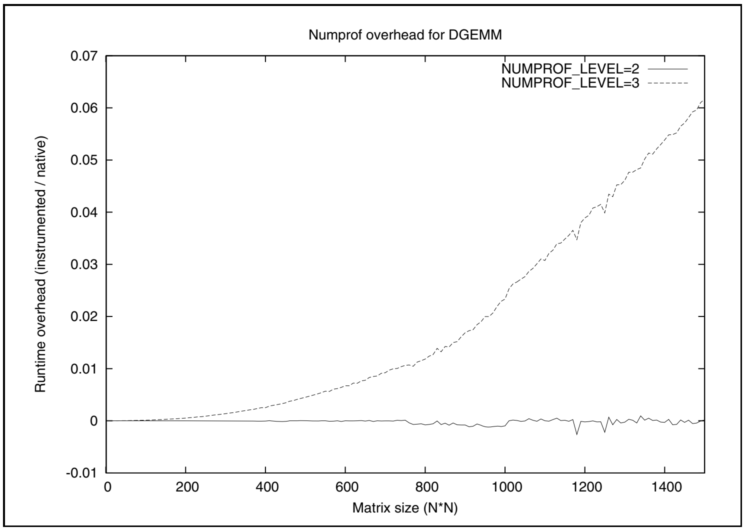


Fig. 4. Numprof profiler structure

increases as the input and output arrays are more thoroughly analyzed (fill rate, min/max). Still, the relative performance penalty is reasonable for doing explicit profiling runs.

6 Conclusions

In this paper we have demonstrated that Numprof is a versatile framework for both profiling numerical libraries and generating benchmarks based on the profiles without any need to augment existing code. It can also be readily extended to support other libraries as well. We also established that the profiler has a considerably lower per-call overhead than an alternative, non-intrusive library profiling mechanism: ltrace. Furthermore, Numprof is also set apart by providing deep analysis of the numerical library calls and facilitates extracting data from the calls to assist with debugging and optimization.

7 Future Work

Future work includes improving the coverage of Numprof to fully support all the BLAS and FFTW calls and supporting additional libraries such as MAGMA

[RN] and LAPACK [EA]. The robustness of the library should be improved and thoroughly assessed in order to reliably run continuous analysis during production workloads. Most importantly memory conservation and thread safety should be investigated. In order to handle large multidimensional data sets the matrix dump should be extended to support the HDF5 file format. [KQ] For deeper performance analysis of individual calls, leveraging the performance counters using PAPI [SB] or a similar framework could also be a potential future development.

References

[JD] Dongarra, J.J., Croz, J.D., Hammarling, S., Hanson, R.J.: An extended set of Fortran basic linear algebra subprograms. ACM Transactions on Mathematical Software 14, 117 (1986)

[MF] Frigo, M., Johnson, S.G.: The design and implementation of fftw3. In: Proceedings of the IEEE, pp. 216–231 (2005)

[SG] Graham, S.L., Kessler, P.B., McKusick, M.K.: gprof: a call graph execution profiler (1982)

[MD] Myers, D.S., Bazinet, A.L.: Intercepting arbitrary functions on Windows, UNIX, and Macintosh OS X platforms. Institute for Advanced Computer Studies. University of Maryland, CS-TR-4585, UMIACS-TR-2004-28 (2004)

[PR] Roth, P.C.: Characterizing the i/o behavior of scientific applications on the cray xt. In: Proceedings of the 2nd International Workshop on Petascale Data Storage: held in Conjunction with Supercomputing 2007 (PDSW 2007), pp. 50–55. ACM, New York (2007)

[AS] Sunderland, A., Pickles, S., Nikolic, M., Jovic, A., Jakic, J., Slavnic, V., Girotto, I., Nash, P., Lysaght, M.: An Analysis of FFT Performance in PRACE Application Codes, PRACE whitepaper (2012)

[DE] Benchmarking Single- and Multi-Core BLAS Implementations and GPUs for use with R, http://cran.r-project.org/web/packages/gcbd/vignettes/gcbd.pdf

[RB] Boisvert, R.F., Boisvert, R.F., Pozo, R., Pozo, R., Remington, K.A., Remington, K.A.: The matrix market exchange formats: Initial design. NISTIR, 5935

[JV] Vetter, J.S., Mueller, F.: Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In: International Parallel and Distributed Processing Symposium (2002)

[RN] Nath, R., Tomov, S., Dongarra, J.: Accelerating GPU Kernels for Dense Linear Algebra. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 83–92. Springer, Heidelberg (2011)

[CF] NVidia CUDA FFT Library, http://developer.nvidia.com/cuda/cufft

[CB] NVidia CUDA BLAS Library, http://developer.nvidia.com/cublas

[EA] Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammerling, S., Demmel, J., Bischof, C., Sorensen, D.: Lapack: a portable linear algebra library for high-performance computers. In: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, Supercomputing 1990, pp. 2–11. IEEE Computer Society Press, Los Alamitos (1990)

[SA] Simpson, A.D., Bull, M., Hill, J.: Identification and Categorisation of Applications and Initial Benchmarks Suite. PRACE Technical Report (2008)

[SB] Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications 14(3), 189–204 (2000) (Fall)

[KQ] Koziol, Q., Matzke, R.: HDF5 - A New Generation of HDF: Reference Manual and User's Guide. NCSA (1998)

Multi-core Scalability Measurements: Issues and Solutions

Mattias Linde

Department of Computing Science, Umeå University, Sweden

Abstract. We discuss how power management development in multi-core processors to achieve higher performance using automatic frequency scaling can cause artifacts when doing performance comparisons and give pessimistic efficiency estimates for algorithms. Overclocking also causes underestimates of the theoretical peak performance of the CPU as can be seen in some cases on the TOP500 list. We show that overclocking capabilities, when available, must be taken into account in thread scheduling for better overall performance.

1 Introduction

Performance and scalability measurements on new algorithms or standard tests is typically performed by running a programs with an increasing number of threads. Scalability and efficiency is then measured using timing routines and compared with theoretical values.

This method is correct provided the computational capacity is independent of power consumption, temperature and computational load. This assumption does not hold when power saving features are enabled, but also with these disabled, the CPU speed can change.

Current processors are subject to a Thermal Design Power (TDP) specification, which specifies its maximum thermal power dissipation. If a processor has 8 cores and only 1 is used, the power consumption and heat production will be small which allows parts of the processor to automatically increase the frequency without exceeding the TDP specification. This feature is available in processors from Intel (Intel Turbo Boost[8]) as well as AMD (AMD Turbo CORE). Similar features are also available in GPUs such as NVIDIA's Kepler[9] and some GPUs from AMD in the Radeon 7000-series with PowerTune technology[2].

Using more cores consumes more power and produces more heat and consequently, there is more room for overclocking when only a few cores are active. Hence, when performing tests with different number of threads, the CPUs in the different tests run at different speeds due to automatic overclocking. Therefore this will give a pessimistic view of parallel efficiency since a gradually slower machine will be used when increasing the number of threads, unless one explicitly accounts for the hardware execution rate.

2 Related Work

Intel introduced its Nehalem architecture in 2008, which includes Turbo Boost features[8]. AMD presented similar features in 2010[1]. Performance evaluations have been performed on different architectures, e.g. [3] and [4] investigate the Nehalem architecture. Charles et al. performed extensive performance evaluation of Turbo Boost features focusing on their potential to increase performance. Results included an average decrease of execution time of 6% with boost features active and also increased power consumption up to 16%.

Much attention has also been spent on asymmetric multi-core computing and power management features such as Dynamic Voltage and Frequency Scaling (DVFS). Such studies are motivated by the fact that energy efficiency is becoming more important and that features to lower the CPU frequency are common, not only in laptops but also desktop and server CPUs.

This differs from our perspective since we focus on how the increase in performance from automatic overclocking can affect further analysis and give an incorrect view of computational efficiency, specially since advancements in boost technology to date allows for frequency increases larger than 50% in some processors.

3 Frequency Scaling

The CPU frequency is determined by a multiplier and a base clock frequency. Lowering the multiplier will reduce the CPU frequency and the power consumption will go down. There is also a dependency between frequency and voltage so when the CPU frequency is lowered, the voltage can also be lowered, further decreasing power consumption.

Unused parts of the CPU still dissipate energy, but using a technique called Power Gating, these parts can be turned off, leaving more energy available for other parts. This is the principle behind Intel's and AMD's automatic overclocking capabilities.

The presence of boost technology can be tested via the `cpuid` instruction and if available, the boost multipliers can be read from Model Specific Registers (MSR). The MSRs lists the highest allowed boost multiplier for different number of active cores.

Whether frequency boost should be allowed can often be controlled via BIOS and for some CPUs the boost multipliers are unlocked and can be changed by the user. When the boost feature is enabled, Intel's implementation is controlled by a Power Control Unit (PCU) within the processor. The PCU is a micro controller that decides the current multiplier from on-chip sensor readings measuring e.g. temperatures and voltages.

4 Theoretical Scenario

Assume an algorithm A that solves a trivially parallelizable problem in an optimal way. It should achieve linear speedup when more processing elements are added and 100% efficiency if we also assume that the problem is compute-bound and no memory effects occur. Under these assumptions, the best serial and the best parallel implementation are the same. The amount of work, W, to solve the problem is set arbitrarily to 30360 units in order to produce integer numbers in the following tables.

The algorithm will be evaluated on two 6-core CPUs with a fixed clock speed at 2.66GHz. Each thread can perform one work unit per time unit. This gives the theoretical values presented in Table 1, assuming linear speedup.

Next we will estimate what happens when executing algorithm A on two Intel Xeon X5650 CPUs with clock speed 2.66GHz. This CPU supports two threads per core with hyperthreading, but we assume hyperthreading is not used in this analysis. The CPU supports self-overclocking and the maximum boost multipliers from the MSRs are listed in Table 2.

Assuming adequate cooling, the CPU temperature should not prevent the CPU from using boosted frequencies and further assuming that the highest boost multiplier for each active core is reached, each workload will give a different result in Table 1, scenario 1, compared to theoretical values.

If the overclocking is not assumed to be active all the time, different scenarios can occur and two more are presented in Table 1, scenario 2, where no boost was available when running with one thread and then available for later runs. While this scenario is unlikely, it will show an efficiency larger than 100% when many threads are used. The opposite, having full boost when using one thread and no boost otherwise is presented as scenario 3 and will show the most pessimistic view on efficiency. In the three scenarios, threads are distributed evenly on the CPUs.

Table 1. Theoretical results for three different scenarios for a trivially parallelizable problem executed on Intel Xeon X5650 CPUs. Scenario 1, full boost all the time. Scenario 2, no boost for 1 thread, full boost otherwise. Scenario 3, full boost on one thread and no boost otherwise. Boost being used is indicated with numbers in red. Efficiency is calculated per scenario.

	Theoretical	Scenario 1		Scenario 2		Scenario 3	
#cores	Time	Time	Eff. (%)	Time	Eff. (%)	Time	Eff. (%)
1	30360	26400	100	30360	100	26400	100
2	15180	13200	100	13200	115	15180	87
4	7590	6600	100	6600	115	7590	87
6	5060	4600	96	4600	110	5060	87
8	3795	3450	96	3450	110	3795	87
10	3036	2760	96	2760	110	3036	87
12	2530	2300	96	2300	110	2530	87

Table 2. Turbo boost multipliers for a hexa-core Intel Xeon X5650 CPU listed as 2.66 GHz (20×133MHz) and a quad-core Intel Core i7 950 listed as 3.06GHz (23×133MHz)

	Intel Xeon X5650			Intel Core i7 950		
# Active cores	Mult.	Speedup	Freq.	Mult.	Speedup	Freq.
1	23	15%	3.06 GHz	25	8.7%	3.33 GHz
2	23	15%	3.06 GHz	24	4.3%	3.19 GHz
3	22	10%	2.92 GHz	24	4.3%	3.19 GHz
4	22	10%	2.92 GHz	24	4.3%	3.19 GHz
5	22	10%	2.92 GHz	-	-	-
6	22	10%	2.92 GHz	-	-	-

5 Experiments

Theory indicates that artifacts related to non-linear speedup can be easily observed and thus standard efficiency calculations can produce non-intuitive results. Two artificial and two linear algebra test cases will be performed on two different Intel CPU models to see which artifacts occur in practice. Time is the common measurement when comparing different results, but to be able to do valid comparisons, the CPU operating frequency is also needed in order to determine if overclocking was used and to what extent.

Using hardware performance counters, as described in [8], the actual frequency can be calculated and the multiplier that was in use determined. There are also many overclocking tools that provide information about the CPU frequency, available instruction sets and active operating frequency. On Linux, the `perf` framework is recommended and has been used for some of the measurements.

The artificial tests are executed on all cores. Both tests are designed to put as high load on the CPU as possible. The first one is called `cpuburn` and runs an endless loop of floating point and integer instructions in an attempt to maximize heat production. The second one is the more recent `stresscpu2` which comes from the Gromacs molecular dynamics package[7] and contains hand coded assembly kernels from the inner molecular dynamics simulation loop.

The linear algebra test cases are matrix-matrix multiply using OpenBLAS which is based upon the highly optimized GotoBLAS2[5,6]. OpenBLAS supports threading, but for fine-grained control on thread placement on a dual socket node with SMP support, we decided not to use the built-in threading and thread placement, but instead launch several single threaded processes, each locked to a different physical core. With this method, the workload is increased with number of running processes, but the runtime for each process should remain the same.

The second linear algebra test case is The High-Performance Linpack (HPL) benchmark[10], which is the software used to create TOP500 lists. This benchmark solves a dense linear system via Gaussian elimination with partial pivoting.

6 Results

The artificial test cases are used to find out what happens under full load, if the CPU temperature reaches a steady-state and what frequency that is used. Figure 1 and 2 show that `stresscpu2` puts a higher load on the system with higher temperature compared to `cpuburn` and in Figure 1 the frequency is also lower. Important to note is that overclocking is active on both CPUs.

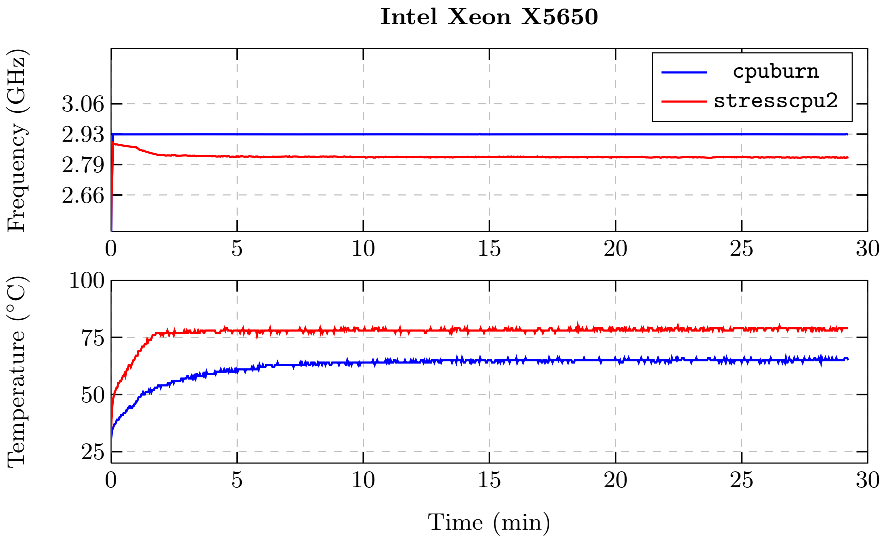


Fig. 1. Temperature and frequency measurements for two different test suites. The grid-lines in the frequency plot corresponds to the base multiplier and possible boost multipliers.

Table 3 shows the results from the matrix-matrix multiplication test. The workload is increased with number of cores and the runtime is expected to be the same regardless of the number of active cores. Due to the scaled workload, the efficiency is calculated as $E = \frac{pT_1}{T_p}/p = T_1/T_p$.

Finally, the results from the HPL benchmark are illustrated in Table 4. The benchmark is designed to run in an MPI environment and we have been executing it on a single node, which should make it easier to achieve good performance. The performance from the test is parameter dependent and many tests where executed with different block and matrix sizes and the best results are listed. The result from the test is not the actual performance, but rather that overclocking was active when the benchmark was performed. The frequencies measured in practice exceeded the theoretical values used for standard computation of peak performance and hence using the base frequency will not give a value that represents the maximum theoretical peak performance.

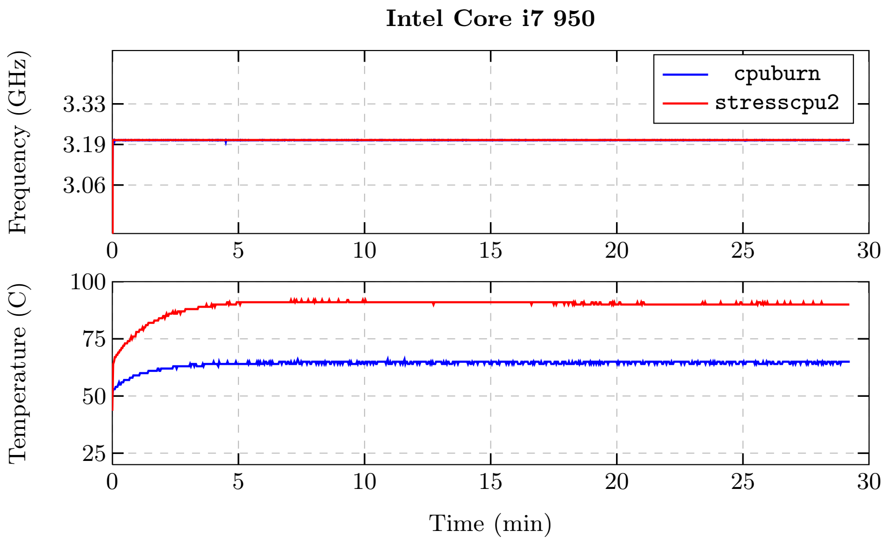


Fig. 2. Temperature and frequency measurements for two different test suites

Table 3. Execution time and average frequency for the first CPU core when all active cores compute the same matrix-matrix multiply workload with OpenBLAS

	Intel Xeon X5650 (2.66 GHz)					
# Active cores	1	2	3	4	5	6
Runtime (s)	212.23	213.17	222.12	221.84	222.61	229.51
Frequency (GHz)	3.057	3.049	2.928	2.927	2.926	2.855
Active multiplier	22.98	22.92	22.02	22.01	22.01	21.47
Efficiency (%)	100	99.55	95.55	95.66	95.33	92.47
Rate adj. efficiency	100	99.82	99.76	99.91	99.60	99.01

	Intel Core i7 950 (3.06 GHz)			
# Active cores	1	2	3	4
Runtime (s)	202.11	202.44	202.75	203.81
Frequency (GHz)	3.208	3.208	3.208	3.208
Active multiplier	24.12	24.12	24.12	24.12
Efficiency	100	99.84	99.68	99.17

Table 4. HPL benchmark results. The numbers for theoretical peak performance does not take boost into account.

CPU	HPL result	Theoretical peak	Frequency	Multiplier
Intel Xeon X5650	54.89 Gflop/s	63.98 Gflop/s	2.798 GHz	21.04
Intel Core i7 950	43.55 Gflop/s	48.96 Gflop/s	3.206 GHz	24.11

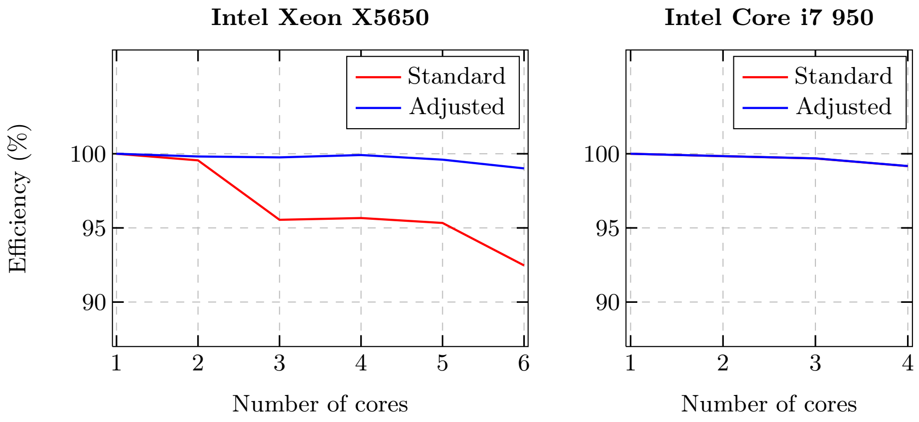


Fig. 3. Efficiency of matrix-matrix computation on different CPUs. The adjusted measurements have been scaled with R_s/R_p to account for the frequency. For the Core i7 950 CPU the measured frequency was the same during the computations.

7 Conclusions

One big cost for HPC hardware is power and low energy CPUs are therefore a viable and good alternative. Using on-demand frequency scaling to save energy is another way to reduce the energy footprint.

Recent multi-core processors that support underclocking to save energy often also support automatic overclocking and we have shown that this can affect performance measurements.

The artifacts that can be observed are CPU dependent. The Intel Core i7 950 ran at the same overclocked speed during all the tests and if the frequency listed on the chip is used for calculating the theoretical peak performance, the value is underestimated as the chip in practice performed faster.

The Intel Xeon X5650 behaved differently and when 1-5 cores where active, the measured frequency was close to the maximum overclocked frequency for that number of active cores. When all cores were active the lowest overclocked frequency gave a frequency multiplier of 21 which is a frequency increase with 5% out of a maximum of 10%.

Without turning off the boost feature while performing measurements, linear speedup will be hard or impossible to achieve on dynamically clocked hardware and hence efficiency decreases or is underestimated. The classical measurement of efficiency, $E = \frac{T_s}{p \cdot T_p}$, can be adjusted to account for the hardware, such that $E = \frac{T_s}{p \cdot T_p} \frac{R_s}{R_p}$, where R_p and R_s are the frequencies for the parallel- and serial runs, respectively. If the hardware does not support overclocking, this factor will be 1 and the standard measurement is used.

There are processors with higher boosted clock frequencies than the ones used in these tests, ranging from both desktop processors to more server oriented versions. One example from each category from Intel is the Intel Core i7-870s, which is listed at 2.66GHz, but can reach a 35% higher maximum frequency of 3.6GHz with one active core. The Intel Xeon Processor E3-1260L, which is listed at 2.4GHz, can reach 3.3GHz, a 37.5% increase.

Achieving higher clock speeds helps performance and this is important to take into consideration when scheduling threads on multi-socket systems or systems where resources are shared when not all available resources are being used. One such architecture is AMD's Bulldozer where the processor is built from modules, where each Bulldozer-module is seen as two cores by the operating system. A module has duplicated integer pipelines but the 256bit wide floating point pipeline is shared. For this architecture, the boost possibilities are affected by the number of active modules. Therefore there are situations where it is better to use two threads in one module instead of two modules with one thread each. If the shared resource becomes the bottleneck, which is likely to occur if both threads use AVX instructions, then it's better to schedule the threads in different modules for increased floating point performance. For a multi socket system, scheduling the threads unevenly between the CPUs can allow for one of the CPUs to reach a higher boosted frequency while the others remain at the same speed as long as the memory bus for the most utilized CPU does not become the bottleneck.

We looked for super-computers among the first 100 computers on the TOP500 list from November 2011, that use Intel Xeon X5650 and do not use GPUs as accelerators. We then checked how the listed theoretical peak performance, R_{peak}, was calculated. Machine number 87 and 88 on the list matched what we searched for and they both used 2.66GHz $\times$ 4 $\times$ N_{cores} as their value for theoretical peak performance. In theory the CPU can reach 2.92GHz when all cores are active. If using 2.92GHz instead of 2.66GHz for calculating the peak performance, those two super-computers reach 72-73% instead of $\sim$80% of their peak performance. We have shown that automatic overclocking do occur, also when running the HPL benchmark, which show that dynamically clocked hardware can not use the base frequency when calculating maximum theoretical values and should instead use the theoretical maximum boosted frequency when all cores are active.

Our future work includes software tools regarding performance measurements and analysis and as an example, reading boost related information from MSRs is both vendor specific and requires the use of privileged instructions. Instead these values can be measured in a series of test and become accessible with normal user privileges.

Acknowledgements. This research was supported by EU Mål 2 Structural Funds (UMIT) and the High Performance Computing Center North (HPC2N). We thank our reviewers for their feedback and valuable comments which have improved the manuscript significantly.

References

1. AMD. AMD Phenom^{TM} II Key Architectural Features, http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-key-architectural-features.aspx (accessed February 16, 2012)
2. AMD. AMD PowerTune Technology. White Paper (2010)
3. Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S., Sancho, J.C.: A performance evaluation of the Nehalem quad-core processor for scientific computing. Parallel Processing Letters 18(4), 453–469 (2008)
4. Charles, J., Jassi, P., Ananth, N.S., Sadat, A., Fedorova, A.: Evaluation of the Intel® Core^{TM} i7 Turbo Boost feature. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 188–197. IEEE (October 2009)
5. Goto, K., Van De Geijn, R.A.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Soft. 34(3), 12 (2008)
6. Goto, K., Van De Geijn, R.: High-performance implementation of the level-3 BLAS. ACM Transactions on Mathematical Software (TOMS) 35(1), 4 (2008)
7. Hess, B., Kutzner, C., van der Spoel, D., Lindahl, E.: Gromacs 4: Algorithms for highly efficient, load-balanced and scalable molecular simulation. Journal of Chemical Theory and Computation 4(3), 435–447 (2008)
8. Intel. Intel® Turbo Boost Technology in Intel® Core^{TM} Microarchitecture (Nehalem) Based Processors. White Paper (2008)
9. NVIDIA. Technology Overview - NVIDIA GeForce GTX 680. White Paper (2012)
10. Petitet, A., Whaley, C., Dongarra, J., Cleary, A.: HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers (2008)

AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications

Renato Miceli[1], Gilles Civario[1], Anna Sikora[2], Eduardo César[2], Michael Gerndt[3], Houssam Haitof[3], Carmen Navarrete[4], Siegfried Benkner[5], Martin Sandrieser[5], Laurent Morin[6], and François Bodin[6]

[1] Irish Centre for High-End Computing, Trinity Tech & Ent Campus, Dublin, Ireland
{renato.miceli,gilles.civario}@ichec.ie

[2] Universitat Autònoma de Barcelona, CAOS Department, Barcelona, Spain
ania@caos.uab.cat, eduardo.cesar@uab.es

[3] Technische Universität München, Institut für Informatik, Garching, Germany
{gerndt,haitof}@in.tum.de

[4] Leibniz-Rechenzentrum, The Bavarian Academy of Sciences, Garching, Germany
carmen.navarrete@lrz.de

[5] University of Vienna, Faculty of Computer Science, Wien, Austria
{sigi,ms}@par.univie.ac.at

[6] CAPS Entreprise, Immeuble CAP Nord, 4 Allée Marie Berhaut, Rennes, France
{laurent.morin,francois.bodin}@caps-entreprise.com

Abstract. Performance analysis and tuning is an important step in programming multicore- and manycore-based parallel architectures. While there are several tools to help developers analyze application performance, no tool provides recommendations about how to tune the code. The AutoTune project is extending Periscope, an automatic distributed performance analysis tool developed by Technische Universität München, with plugins for performance and energy efficiency tuning. The resulting Periscope Tuning Framework will be able to tune serial and parallel codes for multicore and manycore architectures and return tuning recommendations that can be integrated into the production version of the code. The whole tuning process – both performance analysis and tuning – will be performed automatically during a single run of the application.

1 Introduction

The pervasiveness of multi- and many-core processors nowadays makes any computer a parallel system. Most embedded devices, desktop machines, servers and HPC systems now include multicore processors coupled to accelerators (GPGPUs, FPGAs). This recent shift to multi- and many-core architectures hinders the development of hardware-optimized applications. Programming parallel architectures requires careful co-optimization of the following interrelated aspects:

– Energy consumption: Energy reduction has become a major issue on HPC architectures, given their power costs almost reach the purchase price over a lifetime. Careful application-specific tuning can help reduce energy consumption without sacrificing an application's performance.

– Inter-process communication: The overall scalability of parallel applications is significantly influenced by the amount and the speed of communication required. Reducing the communication volume and exploiting the physical network topology can lead to great performance boosts.
– Load balancing: Implicit/explicit process synchronization and uneven distribution of work may leave a process idle waiting for others to finish. The computing power within all parallel processes must be exploited to the fullest, otherwise program scalability may be limited.
– Data locality: Frequent accesses to shared or distant data creates a considerable overhead. Reducing the contention for synchronization resources and ensuring data locality can yield significant performance improvements.
– Memory access: Even the best arithmetically-optimized codes can stall a processor core due to latency in memory access. Careful optimization of memory access patterns can make the most of CPU caches and memory bandwidth on GPGPUs.
– Single core performance: To achieve good overall performance each core's compute capabilities need to be optimally exploited. By providing access to the implementation details of a targeted platform, application optimizations can be specialized accordingly.

Application tuning addressing these areas is an important step in program development. Developers analyze an application's performance to identify code regions that can be improved. They then perform different code transformations and experiment with setting parameters of the execution environment in order to find better solutions. This search is guided by experience and by the output of performance analyses. After the best set of optimizations is found for the given – and supposedly representative – input data set, a verification step with other input data sets and process configurations is executed.

The research community and vendors of parallel architectures developed a number of performance analysis tools to support and partially automate the first step of tuning process, i.e. application analysis. However, none of the current tools supports the developer in the subsequent step of tuning, i.e. application optimization. The most sophisticated tools can automatically identify the root cause of a performance bottleneck but do not provide developers with hints about how to tune the code.

Therefore, the AutoTune project's goal is to close the gap in the tuning process and simplify the development of efficient parallel programs on a wide range of architectures. To achieve this objective, we aim to develop the Periscope Tuning Framework (PTF), the first framework to combine and automate both analysis and optimization into a single tool. AutoTune's PTF will:

– Identify tuning variants based on codified expert knowledge;
– Evaluate the variants online (i.e. within the execution of the same application), reducing the overall search time for a tuned version; and
– Produce a report on how to improve the code, which can be manually or automatically applied.

This project focuses on automatic tuning for multicore- and manycore-based parallel systems, ranging from parallel desktop systems with accelerators, to petascale and future exascale HPC architectures. The next sections show how AutoTune addresses the automatic analysis and tuning of parallel applications.

2 Related Work

The complexity of today's parallel architectures has a significant impact on application performance. In order to avoid wasting energy and money due to low utilization of processors, developers have been investing significant time in tuning their codes. However, tuning implies searching for the best combination of code transformations and parameter settings of the execution environment, which can be fairly complicated. Thus, much research has been dedicated to the areas of performance analysis and auto-tuning.

The explored techniques can be grouped into the following categories:

– Self-tuning libraries for linear algebra and signal processing like ATLAS, FFTW, OSKI and SPIRAL [3,4,5,6];
– Tools that automatically analyze alternative compiler optimizations and search for their optimal combination [7,8,9,10,11];
– Auto-tuners that search a space of application-level parameters that are believed to impact the performance of an application [12,13]; and
– Frameworks that try to combine ideas from all the other groups [14,15].

Performance analysis and tuning are currently supported via separate tools. AutoTune aims to bridge this gap and integrate support for both steps in a single tuning framework.

3 Approach: The Periscope Tuning Framework (PTF)

AutoTune is developing the Periscope Tuning Framework (PTF) as an extension to Periscope [2]. It follows Periscope's main principles, i.e. the use of formalized expert knowledge and strategies, automatic execution, online search based on program phases, and distributed processing. Periscope is being extended by a number of tuning plugins, each of which performs the tuning according to a certain code aspect [1].

PTF's tuning process begins by pre-processing the source files. It takes codes written in C/C++ or Fortran using MPI and/or OpenMP and performs instrumentation and static analysis to generate a SIR file (Standard Intermediate Representation). Extensions to support HMPP and OpenCL codes and parallel patterns are under development. The tuning is then started via the frontend either interactively or in a batch job.

Periscope's analysis strategy becomes part of a higher-level tuning strategy, which controls PTF's sequence of analysis and optimization steps. The analysis guides the selection of a tuning plugin and the actions it performs. After the plugin execution ends, the tuning strategy may restart the same or another analysis strategy to continue with further tuning. All plugins – each with its own tuning strategy, which may employ expert knowledge or machine learning – combine with Periscope's analysis strategies to perform a multi-aspect application tuning.

A tuning plugin is controlled by a plugin strategy meant to guide the search for a tuned version. This strategy often performs several iterations of selection and transformation of tuning parameters and experimental evaluation. Plugins try to tune the code by changing the values of certain tuning parameters, which are the features that influence the performance of a code region.

Code regions are often influenced not by one but by several tuning parameters – the tuning space. The plugin experimentally assesses a code region for a code variant, which is the set of values assigned to region's tuning space. Before experimenting, however, the plugin restricts the search space based on the output both of the previous analyses and of each plugin executed. The reduced search space, called variant space, defines the remaining code variants to be evaluated experimentally.

In the tuning process, the plugin's search strategy explores the variant space to optimize certain tuning objectives. A tuning objective is a function that takes a code region and a variant and outputs a real value, generally a performance measurement (e.g. runtime, energy consumed). The plugin executes a tuning scenario, evaluating one or more tuning objectives for a specific code region and variant.

Once the tuning process is finished, PTF generates a tuning report to document the recommended tuning actions. These tuning actions, i.e. the changes performed to tune the code, can be integrated either manually or automatically into the application for production runs.

The concrete output of the AutoTune project is the Periscope Tuning Framework and its tuning plugins for:

- High-level parallel patterns for GPGPUs;
- Hybrid manycore HMPP codelets;
- Energy consumption via CPU frequency;
- MPI runtime; and
- Compiler flag selection.

The framework follows an extensible architecture so that additional plugins can expand its functionalities. The effectiveness, efficiency and software quality of PTF will be demonstrated with real-world multi- and many-core applications.

3.1 General Design of a Tuning Plugin

PTF's high-level flow is controlled by the frontend; for auto-tuning, a predefined sequence of operations is enforced. However, the frontend allows loadable plugins to specify what is done in certain steps of the execution and to determine how many iterations are required for certain loops.

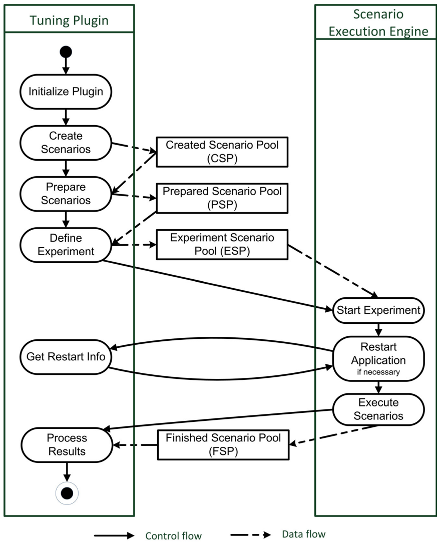

Tuning Plugin
Scenario Execution Engine
Initialize Plugin
Create Scenarios
Prepare Scenarios
Define Experiment
Created Scenario Pool (CSP)
Prepared Scenario Pool (PSP)
Experiment Scenario Pool (ESP)
Start Experiment
Get Restart Info
Restart Application if necessary
Execute Scenarios
Process Results
Finished Scenario Pool (FSP)
Control flow
Data flow

Fig. 1. Simplified work flow of a tuning plugin

The predefined sequence has to cover all possible scenarios given the programming models and parallel patterns supported for tuning, besides any preparation steps required by the system (software and hardware) where the tool is running. As a consequence, the full state machine is relatively complex. For illustration purposes, a simplified version of PTF's flow is presented in Figure 1.

All steps are involved in the creation and processing of the scenarios to be experimented. Scenarios are stored in pools accessible by all plugins as well as the frontend. These pools are:

- Created Scenario Pool (CSP): Scenarios created by a search algorithm;
- Prepared Scenario Pool (PSP): Scenarios already prepared for execution;
- Experiment Scenario Pool (ESP): Scenarios selected for the next experiment;
- Finished Scenario Pool (FSP): Scenarios executed.

All steps in a plugin's workflow relate to the Tuning Plugin Interface (TPI). All methods in this interface must be implemented by all plugins; PTF checks their conformance at loading time. The TPI's major methods are the following:

Initialize: This method is called when the frontend instantiates the plugin. The plugin's internal data structures, tuning space and search algorithms are initialized and the tuning parameters are established.

Create Scenarios: From the defined variant space, the plugin generates the scenarios using a search algorithm and inserts them into the CSP, so the frontend can access them. The plugin combines the region, a variant, and the objectives (e.g. execution time and energy consumption) to generate the scenarios, using either a generic search algorithm (like exhaustive search) or its own search algorithm. The search algorithm may go through multiple rounds of scenario generation. Selecting new scenarios to be generated may depend on the objective values for the previous scenarios. Before the frontend calls the final method to process the results, it checks if the search algorithm needs to generate additional scenarios. If so, the frontend triggers an additional iteration of creation, preparation and execution of scenarios.

Prepare Scenarios: In this method, scenarios are selected from the CSP, prepared and moved into the PSP. Before experiments can be executed, some scenarios require preparation, which typically covers tuning actions that cannot execute at runtime – e.g. recompiling with a certain set of compiler flags or generating special source code for the scenario's variant. Only the plugin can decide which scenarios can be simultaneously prepared. For example, scenarios requesting conflicting compiler flags for a same file cannot be prepared together. If no preparation is needed, this method simply moves all created scenarios from the CSP to the PSP.

After the execution of an experiment, the frontend checks if the CSP is empty. If it is not, the frontend calls the Prepare Scenarios method again.

Define Experiments: Once generated and prepared, the scenarios need to be assembled into an experiment. An experiment goes through at least one execution of the application's phase region. Multiple scenarios can be run in a single experiment assigned either to a single process (if they affect different regions) or to different processes; only the plugin can decide whether this is possible or not. For example, two scenarios for the same program region of an MPI application can only be executed in a single experiment if assigned to different processes. The plugin decides upon the assignment of scenarios to processes/threads in this method. A subset of the prepared scenarios is selected for the next experiment and moved from the PSP to the ESP. Once the experiment is defined, the frontend transfers the control to the Scenario Execution Engine (SEE), which triggers the experiment.

Get Restart Info: During the experiment execution, the SEE first checks with the plugin whether a restart of the application is necessary to implement the tuning actions. For example, the scenarios generated by the MPI tuning plugin explore certain parameters of the MPI runtime environment that can only be set before launching the application. This method returns *true* if an application restart is needed to execute the experiment. It also permits to return parameters to the application launch command, e.g. MPI configuration parameters to be set upon application launch as required by the scenario.

After the potential restart of the application, the SEE runs the experiment by releasing the application to execute a phase region. The SEE takes care of the execution in case multiple phases are required to gather all the measurements for the objectives, and even restarts the application if it terminates before gathering all measurements. At the end, the SEE moves the executed scenarios from the ESP to the FSP and returns the objectives to the plugin.

Process Results: The frontend calls this method if the CSP is empty and the search algorithm is finished. Here the plugin analyzes the objectives and acquired properties – implemented as standard Periscope properties – and either commands that there are extra steps necessary for tuning; or indicates that the tool is finished, selects the best scenario and generates the tuning advices for the user.

The individual steps and methods may be repeated if scenarios still remain in the scenario pools.

As introduced here, the tuning control flow is dictated by the frontend. While the general flow is predetermined, the specific combination of valid execution paths is determined by the plugin based on its internal logic and the search algorithms used.

The specific implementation of our tuning plugins based on the TPI follows in separate sections.

3.2 Tuning of High-Level Patterns for GPGPU

Tuning Objective. We investigate tuning of high-level patterns for single-node heterogeneous manycore architectures comprising CPUs and GPUs. Our focus is on pipeline patterns, which are expressed in C/C++ programs with pragma-annotated while-loops, where pipeline stages correspond to functions for which different implementation variants may exist. The associated programming framework has been developed in the EU project PEPPHER [16] and comprises a component framework, a transformation system, a coordination layer and a heterogeneous runtime system.

The following example gives an impression of a high-level image processing pipeline for face detection, where the computational stages are based on OpenCV library routines [17]. For the functions of the middle stage, different implementation variants – one for CPUs and one for GPUs – are provided. Annotations enable the user to specify a stage replication factor, which results in the generation of multiple stage invocations processing different data packets in parallel. This may be advantageous if a stage executes for much longer than its predecessor stage. Moreover, the size of the buffers that are automatically generated by the framework to pass data between stages may be specified. Merging stages can be achieved by enclosing two or more function calls within a *stage* pragma.

```
#pragma pph pipeline with buffer (PRIORITY, N*2)
while (inputstream >> file) {
    readImage(file, image);
    #pragma pph stage replicate(N)
    {
        resizeAndColorConvert(image);
        detectFace(image, outimage);
    }
    writeFaceDetectedImage(file, outimage);
}
```

Such a high-level code is transformed into a representation that utilizes a coordination layer and the StarPU [18] heterogeneous runtime system to exploit pipeline parallelism across stages. The coordination layer and runtime system decide when to schedule a stage for execution and which implementation variant to execute on which execution unit of the targeted heterogeneous architecture.

The goal of tuning pipeline patterns is to maximize the throughput by efficiently exploiting all CPU cores and GPUs of a specific targeted architecture.

Tuning Parameters and Tuning Actions. The main tuning parameters for pipeline patterns comprise stage replication factors and buffer sizes. Moreover, the tuning of the underlying coordination and runtime layers, including implementation variant selection, runtime scheduling policy and resource allocation strategy will be investigated.

To support these tuning parameters, the PTF monitor and the coordination layer provide extensions to measure the time spent executing the stages, processing the buffers and executing the overall pipeline. Tuning actions vary these tuning parameters aiming to maximize the pipeline throughput.

To restrict the variant space, a range is specified per tuning parameter. Moreover, the tuning plugin may take into account the historical execution data and static information about the concrete runtime environment (e.g. number of CPU cores, number of GPUs).

3.3 Tuning of HMPP Codelets

Tuning Objective. The objective is to tune the performance of a codelet computation in an application using the CAPS Workbench. The plugin targets many-core accelerators like GPGPUs. The performance is evaluated by analyzing the execution time of the codelet.

A codelet is a computational unit of a HMPP program [20]. It is typically implemented as a C function or a Fortran subroutine annotated with OpenHMPP directives [19]. The CAPS compiler automatically translates codelets into a hardware-specific language such as CUDA or OpenCL. Therefore, the execution of a codelet from the CPU perspective is considered atomic (no identified intermediate state or data).

As a portable programming standard, HMPP provides a way to access a varied set of hybrid accelerators; the CAPS compiler can currently generate code for NVIDIA and AMD GPUs and Intel Xeon Phi coprocessors. In spite of this, we restrict the plugin to Linux host systems; the plugin will come with a prototype of the CAPS Workbench different from the mainstream release version.

Tuning Parameters and Tuning Actions. This plugin has to manage a wide set of parameters that depend on the targeted architecture. Thus, this plugin allows the user to define the HMPP tuning experience via the plugin API. This API is currently accessed via OpenHMPP directives to the HMPP compiler, although more specific and autonomous auto-tuning methods – e.g. a user-friendly graphical interface for a specific domain or architecture, or automatic tools for a particular machine or application domain – may come in the future.

The tuning object is the HMPP codelet. Relative to a codelet, there are two kinds of tuning parameters to consider:

– Static codelet tuning parameters: operations, transformations or algorithms used to implement a codelet. Examples are the unrolling factor, the HMPP grid size and loop permutations.
– Dynamic codelet tuning parameters: variable or callback available at run-time, which are generally target-specific.

The variant space will be initially searched using either exhaustive or random search. Further search algorithms may be added in the future.

In this first implementation, the plugin looks for the tuning regions, the tuning parameters and the search space in the code, as they should be statically defined by the user. The information about the tuning parameters is extracted at compilation time into an extended SIR file. PTF starts after the HMPP compilation and no more compilation steps are necessary.

The scenarios executed by this plugin also specify the hardware requirements for a valid execution – the type and number of accelerators needed and if the scenario requires hardware exclusivity – since the CAPS Workbench can cope with various models of accelerators but the tuning is hardware-specific.

This plugin also extends the monitoring infrastructure provided by Periscope to deal with hardware accelerators. The HMPP profiling interface (PHMPP [21]) is tailored to extract the exact execution time of a codelet; for NVIDIA GPUs, its implementation is partially based on the NVIDIA CUPTI [22].

3.4 Tuning of Energy Consumption via CPU Frequency

Tuning Objective. The main tuning objective is to minimize the energy consumption of an application. The code may belong to any application field; however, emphasis is given to scientific fields where codes are usually arithmetic-operation-intensive. On the hardware side we can change frequency policies and read the RAPL counters (Running Average Power Limit [23]). We specifically target the IBM System x iDataPlex thin-node islands on LRZ's SuperMUC machine, whose nodes contain two processors in a shared-memory fashion.

The power consumption can only be measured per package according to the RAPL counter: PP0_ENERGY:PACKAGE0, PP0_ENERGY:PACKAGE1, PACKAGE_ENERGY:PACKAGE0, PACKAGE_ENERGY:PACKAGE1. Each package also includes un-core elements such as the last level cache, Integrated IO, QPI and Memory controllers.

Tuning Parameters and Tuning Actions. We define two different tuning parameters: the available governors/policies and the frequencies to be used. Once an application is tuned for performance, its energy- and time-related costs can be optimized.

Regarding the first tuning parameter, there are five governors: Performance, Powersave, Ondemand, Conservative and Userspace. We discarded the Performance and the Powersave governors from our search space since they are special cases of the Userspace governor. Thus, we define the tuning parameter as an indexed vector of the three governors Ondemand, Conservative and Userspace.

Regarding the second tuning parameter, the Sandy Bridge-EP Intel Xeon E5-2680 supports, among others, the following frequencies: 2.7, 2.4, 2.2, 2, 1.8, 1.6, 1.4 and 1.2 GHz. We leave the turbo mode frequency[1] outside the range of explored frequencies.

Ideally, measurements are captured for all combinations of the available tuning parameters to select the best performing set per governor. However, trying all three governors each with all possible frequencies per region may be too time consuming (24 frequencies per region). Our experience shows that the range of frequencies describes a soft parabolic-like shape to the energy consumed. Therefore, we perform a ternary search per governor, akin to the bisection method. First, we run three experiments: the highest (f2), the lowest (f0) and the median frequency (f1). The two neighboring frequencies that resulted in the lowest energy consumption define the upcoming search interval, and the search continues until the best performing frequency is found.

3.5 Tuning of MPI Runtime

Tuning Objective. Using Periscope's standard MPI analysis, this plugin implements a combined tuning strategy to determine the values of multiple tuning parameters: a set of MPI environment variables and variants of MPI communication functions.

There are many environment variables associated with specific implementations of the MPI library. In particular, the IBM MPI library on SuperMUC offers more than 50 configurable parameters. Changes to some of these parameters could significantly affect the time an application spends in communication. The plugin assumes that the inputs are SPMD applications, so the same optimization is applied to all processes.

The MPI variables that relate to the communication buffer/protocol and to the application mapping have potential for tuning. Tuning the former variables could be effective on applications with clustered communications, while tuning the latter variables could be particularly effective on applications that exchange messages of a uniform size. The initial tuning objective is to find a proper combination of values for the pair of variables *(MP_BUFFER_MEM, MP_EAGER_LIMIT)*. In addition, the tuning strategy consists of systematically launching the application using different combinations of values for both variables. Finally, the tuning recommendation consists of the pair of values that led to the application's lowest execution time.

Tuning Parameters and Tuning Actions. For this plugin we have two kinds of tuning parameters: the MPI environment parameters and the code variants for the MPI communications.

[1] In theory, the Sandy Bridge-EP Intel Xeon E5-2680 can reach a turbo frequency of 3.5GHz; in practice, however, this frequency varies, depending on several factors such as the CPU load, the quality of thermal solution and the ambient temperature.

The MPI environment parameters may be set before the application executes. We propose two tentative pairs of parameters:

– MPI application mapping: adapting tasks per node/core, adapting the affinity of the processes.
  - MP_TASK_PER_NODE: Specifies the number of tasks run on each physical node; and
  - MP_TASK_AFFINITY: Attaches a parallel task to one of the system's cpusets.
– MPI communication buffer/protocol: adapting the sending/receiving buffer, analyzing the message size patterns, adapting the communication protocol (eager/rendezvous).
  - MP_BUFFER_MEM: Controls the amount of memory for buffering data from early arrival messages[2]; and
  - MP_EAGER_LIMIT: Changes the message size threshold that defines the message passing protocol used (eager or rendezvous).

These parameters can be adjusted using environment variables or *mpirun* options, forcing all processes in the SPMD application to use the same options. The tuning action is to set the values of the MPI parameters and run a new experiment.

Regarding the code variants for the MPI communications, the user must provide different versions of the functions within the application (in function *main*) and annotate the code with user regions and their attributes:

– Pragma *mchoice* (multiple choice) indicates that there are many versions of the same functions. This pragma has an attribute *v* that indicates how many versions exist. This attribute is treated as a tuning parameter and exported to the SIR file.
– Pragma *dependency_mchoice* indicates a condition and a range of variants to explore when the condition is satisfied. For example, *dependency_mchoice v==5, bsize=128-128KB* means that for function version 5, the range for *bsize* is 128B to 128KB.

The tuning action is to execute the application changing the implementation of the functions and using different attribute values (if applicable).

3.6 Tuning of Compiler Flag Selection

Tuning Objective. The tuning objective is to reduce the execution time of the application's phase region. Besides the choice of algorithm and the way the high-level source code is written, the most influential factor to the runtime is the quality of the generated machine code. Compilers apply a large number of

[2] Message data sent without knowing if the corresponding receive is posted is said to be sent eagerly. A message data arriving before its corresponding receive is posted is called an early arrival and must be buffered at the receiving side.

code transformations to generate the best code for a given architecture, e.g. loop interchange, data prefetching, vectorization and software pipelining. Although the compiler ensures the correctness of the transformations, it is very difficult to predict the performance impact and select the right sequence of transformations. Therefore, compilers offer a long list of flags and directives to allow the programmer to guide the compilation in the optimization phase.

Due to the required background knowledge about compiler transformations, compiler interactions with the application and hardware, and the large number of flags, programmers find it difficult to select the best flags and guide the compiler by inserting directives. Thus, typically only the standard flags -*O2* and -*O3* are used to change the approach of the compiler optimization.

Tuning Parameters and Tuning Actions. This plugin's tuning parameters are the individual compiler flags. Each parameter can be switched either ON or OFF, hence the parameters have only two values. The tuning action is to switch a flag in the program recompilation. All tuning actions for each parameter are combined in the preparation step.

4 Evaluation

4.1 The Application Repository

The Application Repository is a central workspace composed of representative and reliable inputs for test cases. It is the main toolbox to be used during the design, development and integration of the tuning techniques and plugins to PTF. Each tool is an application – comprised of source code, configuration files and input data sets – that composes a use case of HPC software and hardware, specifically selected to match the requirements of our tuning plugins.

The repository applications currently have a dual objective:

- To guide the development of the tuning techniques and plugins; and
- To assess the quality of automatic tuning via the plugins.

The applications will serve as input to the tuning plugins and hence will provide a global view of the AutoTune behavior on scientific applications. Because of this, the applications must display the characteristics required by each plugin, be they MPI, HMPP, OpenCL, OpenACC, pipelining, data distribution or the master-worker pattern.

We also used the application repository to perform a preliminary validation of our tuning techniques by manually applying them over the applications. It aimed to provide first insights into the approach and practical implementation of the tuning plugins, besides representing proof why investing in plugin development is a worthy task.

With our manual tuning, our tuning techniques could find code variants that performed at least 25% better than the worst performing variant. Half of our techniques excelled and could find variants performing 75% better. All tuning techniques managed to find optimized code variants, on average 60.43% more efficient than the least performing variant.

4.2 The Proof of Concept

We developed a demo plugin to demonstrate and validate the auto-tuning extensions to Periscope. In this section we describe the plugin itself as well as a simple Fortran application used for evaluation.

Sample Application and Tuning Objective. The tuning objective of this plugin is to minimize the execution time of a code region for which several variants are defined in the program. The region is marked in the Fortran source file with an AutoTune pragma, which defines a single tuning parameter manipulated through a variable tuning action. Each value of the variable defines a unique variant.

Tuning Parameters and Tuning Actions. This tuning plugin can process a single tuning parameter as defined in the program via a directive, as it is shown below in the sample Fortran application:

```
do k=1,20
    var=k
    !$MON USERREGION TP name(Test) variable(var) variants(10)
    tstart=MPI_Wtime()
    !<user compute code depending on the value of variable ‘var’>
    tend=MPI_Wtime()
    !$MON END USERREGION
enddo
```

The code region to be tuned must be surrounded by the directives “USERREGION” and “END USERREGION”. The tuning parameter is specified in the directives using the keyword “TP” followed by (i) the name of the tuning parameter, (ii) the variable identifier and (iii) the number of variants.

In this code, the value of variable *var* is initially set to *k*; however, since *var* is a tuning parameter, the monitor overwrites its value at runtime upon entry in the tuning region. On the other side, setting *var* to *k* allows the code to be run without PTF.

When instrumenting the application, the Periscope instrumenter parses the Fortran source file to extract information from the AutoTune pragmas. The tuning parameters (variable *var* in our example) are then added to the SIR file, which the frontend takes as input. The instrumentation also maps between the name of a tuning parameter and the variable. Whenever the region is entered, the monitor assigns a value to the variable to trigger the execution of a certain variant. In our example, PTF sweeps over all variants from 1 to 10.

5 Conclusions and Future Works

This paper presents the Periscope Tuning Framework, currently under development in the FP7 project AutoTune. PTF is a framework that allows for the easy development of tuning plugins. The tuning plugins explore tuning parameters based on performance information resulting from PTF's analysis strategies; different code variants are explored by running experiments that return measurements for each variant.

We implemented a first demonstration prototype of PTF to validate the overall design. In the next project cycles, we will develop initial versions of the outlined tuning plugins and test them on the application repository. Although the initial versions will be based on exhaustive or random search, we plan to add intelligence in the form of expert knowledge or model-driven search.

We also intend to design and develop a plugin for MPI programs that display the Master-Worker pattern. Furthermore, we will conduct a feasibility study and design a plugin to address I/O bottlenecks, in order to enable its implementation as a future PTF plugin. Other plugins, such as for OpenMP thread affinity, are under consideration for development and integration into PTF.

Finally, we plan to investigate combined plugin tuning strategies for inclusion into PTF. Combined strategies use multiple plugins to perform code tuning, taking different application aspects into consideration at the same time. We will especially target the design and analysis of tuning strategies for justified trade-off between energy tuning and runtime tuning.

Acknowledgements. The authors thank the European Union for supporting this project under the Seventh Framework Programme, grant no. 288038.

References

1. Miceli, R., Civario, G., Bodin, F.: AutoTune: Automatic Online Code Tuning. In: NVIDIA GPU Technology Conference 2012 (GTC 2012), San Jose, USA (2012)
2. Benedict, S., Petkov, V., Gerndt, M.: PERISCOPE: An Online-Based Distributed Performance Analysis Tool Tools for High Performance Computing 2009, pp. 1–16. Springer, Heidelberg (2010)
3. Whaley, C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the atlas project. Parallel Computing 27, 2001 (2000)
4. Frigo, M., Johnson, S.G.: Fftw: An adaptive software architecture for the fft, pp. 1381–1384. IEEE (1998)
5. Vuduc, R., Demmel, J.W., Yelick, K.A.: Oski: A library of automatically tuned sparse matrix kernels. Institute of Physics Publishing (2005)
6. Püschel, M., Moura, J.M.F., Singer, B., Xiong, J., Johnson, J., Padua, D., Veloso, M., Johnson, R.W.: Spiral: A generator for platform-adapted libraries of signal processing algorithms. Journal of High Performance Computing and Applications 18, 21–45 (2004)
7. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.I.: Compiler optimization-space exploration. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 204–215. IEEE Computer Society (2003)
8. Haneda, M., Knijnenburg, P., Wijshoff, H.: Automatic selection of compiler options using non-parametric inferential statistics. In: 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005), pp. 123–132 (September 2005)

9. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 319–332 (2006)
10. Leather, H., Bonilla, E.: Automatic feature generation for machine learning based optimizing compilation. In: Code Generation and Optimization (CGO), pp. 81–91 (2009)
11. Fursin, G., Kashnikov, Y., Wahid, A., Chamski, M.Z., Temam, O., Namolaru, M., Yom-tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C.K.I.: Milepost gcc: machine learning enabled self-tuning compiler (2009)
12. Chung, I.H., Hollingsworth, J.: Using information from prior runs to improve automated tuning systems. In: Supercomputing. Proceedings of the ACM/IEEE SC2004 Conference, vol. 30 (November 2004)
13. Nelson, Y., Bansal, B., Hall, M., Nakano, A., Lerman, K.: Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008), pp. 1–8 (April 2008)
14. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.: A scalable autotuning framework for compiler optimization. In: IEEE International Symposium on Parallel Distributed Processing (IPDPS 2009), pp. 1–12 (May 2009)
15. Ribler, R., Vetter, J., Simitci, H., Reed, D.: Autopilot: adaptive control of distributed applications. In: Proceedings of the Seventh International Symposium on High Performance Distributed Computing, pp. 172–179 (July 1998)
16. Benkner, S., Pllana, S., Traff, J., Tsigas, P., Dolinsky, U., Augonnet, C., Bachmayer, B., Kessler, C., Moloney, D., Osipov, V.: Peppher: Efficient and productive usage of hybrid computing systems. IEEE Micro. 31(5), 28–41 (2011)
17. Gary, B.: Learning opencv: Computer vision with the opencv library (2008)
18. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A., Inria, L., Sud-ouest, B.: Author manuscript, published in "euro-par (2009)" starpu: A unified platform for task scheduling on heterogeneous multicore architectures (2009)
19. CAPS Entreprise: HMPP Directives Reference Manual, version 3.2.0 (2012)
20. CAPS Entreprise: The HMPP Workbench, http://www.caps-entreprise.com/products/hmpp/ (accessed on October 16, 2012)
21. CAPS Entreprise: H4H - HMPP Profiling Event specification, version 2.3.3 (2012)
22. The CUDA Profiling Tools Interface, http://docs.nvidia.com/cupti/ (accessed on October 16, 2012)
23. David, H., Gorbato, E., Hanebutte, U., Khanna, R., Le, C.: RAPL: memory power estimation and capping. In: Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design (2010)

Vectorized Higher Order Finite Difference Kernels

Gerhard Zumbusch

Friedrich-Schiller-Universität Jena,
Institut für Angewandte Mathematik,
07743 Jena, Germany
gerhard.zumbusch@uni-jena.de
http://cse.mathe.uni-jena.de

Abstract. Several highly optimized implementations of Finite Difference schemes are discussed. The combination of vectorization and an interleaved data layout, spatial and temporal loop tiling algorithms, loop unrolling, and parameter tuning lead to efficient computational kernels in one to three spatial dimensions, truncation errors of order two to twelve, and isotropic and compact anisotropic stencils. The kernels are implemented on and tuned for several processor architectures like recent Intel Sandy Bridge, Ivy Bridge and AMD Bulldozer CPU cores, all with AVX vector instructions as well as Nvidia Kepler and Fermi and AMD Southern and Northern Islands GPU architectures, as well as some older architectures for comparison. The kernels are either based on a cache aware spatial loop or on time-slicing to compute several time steps at once. Furthermore, vector components can either be independent, grouped in short vectors of SSE, AVX or GPU warp size or in larger virtual vectors with explicit synchronization. The optimal choice of the algorithm and its parameters depend both on the Finite Difference stencil and on the processor architecture.

1 Introduction

Finite Differences are a classical numerical scheme for the solution of differential equations. However, the stencils on structured grids are also computationally efficient on current computers, which explains their widespread use in science.

By the introduction of a new vector instruction set for x86 architecture CPUs, vector length increases from 128 bit SSE to 256 bit AVX vectors, i.e. from 4 to 8 single precision numbers (float), with a road map to even larger vectors. Other CPUs provide long vectors already (Intel Phi 512 bit, 16 floats). In GPU computing, vector lengths of 16 to 64 floats are common, which can be combined to virtual vectors of length 256 to 1024 by hardware multi-threading. Automatic vectorization of loop and array expressions in Fortran style codes has been developed successfully for classic style vector computers. However, current architecture's memory, caches or GPU local processor memories do not provide enough bandwidth anymore. Algorithmic modifications are needed to reduce memory traffic, streamline memory access and to feed the vector units by data placed in registers and memory closer to the processor. Further issues are to provide enough instruction parallelism for long pipelines and multi-threading.

We make the following contributions: We propose interleaved data layouts of the Finite Difference grid points especially suited for vector instructions based on memory aligned vector load and store operations. We develop highly tuned implementations of Finite Difference stencil computations for single CPU cores with SSE and AVX vector instructions on older and most recent CPU architectures by Intel and AMD. Furthermore, OpenCL and Nvidia Cuda implementations for AMD and Nvidia GPUs are presented, again organizing Finite Difference stencil computations as vector operations and adapting the data layout accordingly. By an analysis of simple Finite Difference stencils, we obtain efficient implementation techniques and upper performance limits also applicable to more complex numerical expressions.

2 Model Problem Finite Difference Stencils

We consider Finite Difference stencil computations on structured grids. The stencils represent discretized versions of constant coefficient second order differential operators. Both application of the operator within a linear equation solver or within a time stepping scheme are included. Furthermore, an iterative solver of Jacobi type can be implemented this way. We consider arbitrary approximation orders, spatial dimensions of the structured grid, and isotropic and anisotropic self adjoint operators. Isotropic stencils of order p in one to three dimensions with constant coefficients c_l can be written as

$$u_i^{\text{new}} = c_0 u_i + \sum_{l=1}^{p/2} c_l (u_{i-l} + u_{i+l})$$

$$u_{i,j}^{\text{new}} = c_0 u_{i,j} + \sum_{l=1}^{p/2} c_l (u_{i-l,j} + u_{i+l,j} + u_{i,j-l} + u_{i,j+l})$$

$$u_{i,j,k}^{\text{new}} = c_0 u_{i,j,k} + \sum_{l=1}^{p/2} c_l (u_{i-l,j,k} + u_{i+l,j,k} + u_{i,j-l,k} + u_{i,j+l,k} + u_{i,j,k-l} + u_{i,j,k+l})$$

and may approximate the isotropic Δ Laplace operator applied to the grid function u. Anisotropic operators represent linearly distorted versions of the operator, like the two dimensional version:

$$\begin{aligned} u_{i,j}^{\text{new}} &= c_{0,0} u_{i,j} + \textstyle\sum_{l=1}^{p/2} \left(c_{l,0}(u_{i-l,j} + u_{i+l,j}) + c_{0,l}(u_{i,j-l} + u_{i,j+l})\right) \\ &\quad + \textstyle\sum_{l=1}^{p/2} c_{l,l}(u_{i-l,j-l} + u_{i+l,j+l} - u_{i+l,j-l} - u_{i-l,j+l}) \\ &\quad + \textstyle\sum_{l}^{p/2} \sum_{m=l+1}^{p/2} c_{l,m}(u_{i-l,j-m} + u_{i-m,j-l} + u_{i+l,j+m} + u_{i+m,j+l} \\ &\qquad\qquad - u_{i+l,j-m} - u_{i+m,j-l} - u_{i-l,j+m} - u_{i-m,j+l}) \end{aligned}$$

The three dimensional anisotropic stencil of the Laplace operator approximation is a collection of two dimensional stencils along each x_i, x_j coordinate area. The

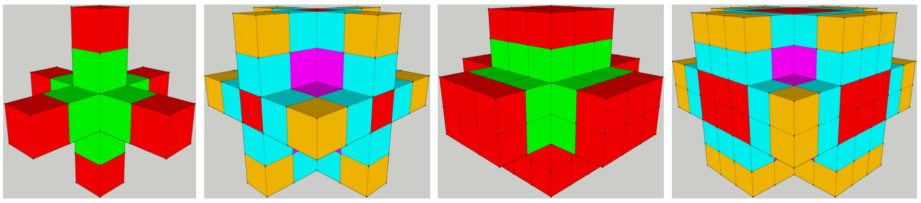


Fig. 1. Schematic 3D 4th order FD stencil, isotropic and anisotropic. Single stencils (left) and $3 \times 3 \times 2$ block (right).

shape of the three dimensional isotropic and anisotropic stencils are depicted in Fig. 1.

The number of grid points and arithmetic operations, separated into adds and subs, multiplications (mul) and alternatively fused multiply-adds (fma) are summarized in Tab. 1. The number of fma operations equals the number of grid points, which increases with order and dimension. The number of multiplications equals the number of coefficients and is at most the number of adds.

Table 1. Finite Difference stencils of order p, number of floating point operations per grid point

name	**operator**	**load points**	**total flops**	**add**	**mul**	**fma**
1D	D_{xx}	$1+p$	$1+\frac{3}{2}p$	p	$1+\frac{1}{2}p$	$1+p$
2D	$D_{xx}+D_{yy}$	$1+2p$	$1+\frac{5}{2}p$	$2p$	$1+\frac{1}{2}p$	$1+2p$
3D	$D_{xx}+D_{yy}+D_{zz}$	$1+3p$	$1+\frac{7}{2}p$	$3p$	$1+\frac{1}{2}p$	$1+3p$
anisotropic						
2D	$a_{11}D_{xx}+2a_{12}D_{xy}+a_{22}D_{yy}$	$(1+p)^2$	$1+\frac{13}{4}p+\frac{9}{8}p^2$	$2p+p^2$	$1+\frac{5}{4}p+\frac{1}{8}p^2$	$(1+p)^2$
3D	$a_{11}D_{xx}+2a_{12}D_{xy}+2a_{13}D_{xz}$ $+a_{22}D_{yy}+2a_{23}D_{yz}+a_{33}D_{zz}$	$1+3p+3p^2$	$1+\frac{21}{4}p+\frac{27}{8}p^2$	$3p+3p^2$	$1+\frac{9}{4}p+\frac{3}{8}p^2$	$1+3p+3p^2$

3 Target Processor Architectures

We will implement several Finite Difference algorithms for CPU and GPU (graphics processing unit) architectures. We consider single processor cores of x86 CPUs by Intel and AMD and GPUs by Nvidia and AMD, see Tab. 2 and 3. We consider the smallest independent processor unit (core), called 'streaming multiprocessor' by Nvidia, 'shader cluster' on AMD GPUs or 'module' for AMD Bulldozer. Most recent CPUs feature AVX arithmetic vector instructions, that is 256 bit vectors with 8 single precision (float) or 4-double precision values. Add and mul operations take two vectors to compute a result vector, fma takes three input vectors. Previous CPUs offered SSE vectors of half the size. The CPUs have independent floating point add and mul pipelines, with the exception of AMD Bulldozer. Hence the number of adds are an upper performance limit for the Finite Difference implementations, given that data flow between registers, caches and memory is fast enough.

GPU architectures are based on vectors of length 32 or 64 floats within a processor. Hardware multi-threading enables the program to combine the vectors

Table 2. CPU and GPU processors, micro architectures and their single precision (32 bit float) performance. All numbers of a single processor core.

processor		performance				cache		
		clock	vector	vector	flop/s	L1	L2	shared
architecture	name		instructions	op/cycle				LL
		[GHz]			[GF]	[kB]	[kB]	[MB]
Intel Ivy Bridge	i5-3450	3.1/3.5	AVX	add + mul	56	32	256	6
Intel Sandy Bridge	i7-2600	3.4/3.8	AVX	add + mul	60.8	32	256	6
Intel Core	Xeon E5405	2.0	SSE4.1	add + mul	16	32	–	6
AMD Bulldozer	FX-8150	3.4/3.9/4.2	AVX, FMA4	fma	67.2	16	2048	8
AMD K10	Opteron 6168	1.9	SSE4	add + mul	15.2	64	512	6
AMD K8	Opteron 865	1.8	SSE3	1/2(add + mul)	7.2	64	–	1
Nvidia Kepler	GK110, Tesla K20c	0.705/0.758	32	6 fma	291.1	16 – 48	–	1/8
Nvidia Kepler	GK104, GTX 680	1.006/1.059	32	6 fma	406.5	16 – 48	–	1/8
Nvidia Fermi	GF108, GT 540M	1.344	32	3/2 fma	129	16 or 48	–	1/8
Nvidia Fermi	GF110, GTX 590	1.215	32	1 fma	77.8	16 or 48	–	1/8
Nvidia Fermi	GF100, Tesla C2050	1.15	32	1 fma	73.6	16 or 48	–	1/8
Nvidia GT200	GTX 260	1.296	32	1/4 fma	20.7	–	–	–
AMD South. Isl.	GCN, HD 7970	0.925	64	1 fma	118.4	16	–	1/8
AMD North. Isl.	VLIW4, HD 6990	0.83	64	1 fma	106.3	8		

transparently to larger virtual vectors of sizes 256 to 1024. The GPUs have fma floating point pipelines. Hence the number of fmas serve as an upper performance limit, like in the case of Bulldozer CPUs. We have listed the properties of a single CPU core and a single GPU processor, although usually chips and systems with different numbers of cores are available. Note that the double precision performance of the CPUs is half of the single precision in Tab. 2 and the GPU double precision performance is between 1/24 and one half of single precision.

Table 3. Double precision peak performance of a single GPU core

processor		performance				
architecture	name	clock	cores	vector	vector	flop/s
		[GHz]		length	fma/cycle	[GF]
Nvidia Kepler	GK110, Tesla K20c	0.705/0.758	13	32	2	97.0
Nvidia Kepler	GK104, GTX 680	1.006/1.059	8	32	1/4	16.9
Nvidia Fermi	GF108, GTX 540M	1.344	2	32	1/8	10.8
Nvidia Fermi	GF110, GTX 590	1.215	16	32	1/8	9.7
Nvidia Fermi	GF100, Tesla C2050	1.15	14	32	1/2	36.8
AMD South. Isl.	GCN, HD 7970	0.925	32	64	1/4	29.6
AMD North. Isl.	VLIW4, HD 6990	0.83	24	64	1/4	26.6

4 Scalar Cache Aware Algorithms and Data Layout

First of all we will discuss scalar versions of the Finite Difference algorithms before we turn them into the vectorized algorithms necessary to fully exploit the target processors. For reasons of simplicity, we ignore boundary conditions close to the border of the spatial grid and start-up procedures for algorithms with multiple time steps.

Naive implementations of a Finite Difference stencil will simply take a loop over all grid points and apply the stencil. Such an algorithm has to load the same number of values from memory as number of fma operations are performed. This is a memory bandwidth bound algorithm in a range of about 1 GF. It is up to a good cache mechanism to make this efficient. However, even the bandwidth of the L1 cache does not match the demand of the floating point pipelines.

4.1 Sliding Window Algorithm

The memory hierarchy of main memory RAM, last level (LL) to first level (L1) caches and processor registers offers different memory capacity at vast differences of access bandwidth and latency. Only the bandwidth of the register file is able to fully match the processor floating point pipelines. An improvement of the naive Finite Difference implementation goes like this: Data re-use is explicitly organized in register space. A cache aware 1D space loop for $p+1$-point wide Finite Difference stencil (order p in Tab. 1) implementing a single time step may look like this:

```
r[0..p-1] = grid[0..p-1];                             // load memory
for (int x=0; x<stepx*(p+1); x=x+p+1) {
  for (int x0=0; x0<p+1; x0++) {                      // unroll loop
    r[(x0+p)%(p+1)] = grid[x+x0+p];                   // load memory
    grid[x+x0] = calc (r[(x0)%(p+1)..(x0+p)%(p+1)]); // store memory
  }
}
```

Values `r` and `c` are to be placed in registers, routine `calc` represents the inlined stencil and `grid` values u. For reasons of simplicity a single array u is used both input and output, over writing old values of u with new ones. The `x0` loop needs to be explicitly unrolled, at least for most of the compilers, such that index expressions for `r` can be removed. The code for example for a 3-point stencil (order $p=2$) can be expanded to

```
r0 = grid[0]; r1 = grid[1];      // load memory
for (int x=0; x<stepx*3; x=x+3) {
  r2 = grid[x+2];                // load memory
  grid[x  ] = calc (r0, r1, r2); // store memory
  r0 = grid[x+3];                // load memory
  grid[x+1] = calc (r1, r2, r0); // store memory
  r1 = grid[x+4];                // load memory
  grid[x+2] = calc (r2, r0, r1); // store memory
}
```

by a source-to-source preprocessor. This way copying of registers can be avoided. Data re-use takes place via registers only. There is one memory load and one store per grid point, compared to e.g. $1+p$ fma operations.

The sliding window algorithm is discussed by [1,2,3,4,5,6,7,8] (often in the GPU context) and can be generalized to higher dimensions based on the memory hierarchy: The inner most loop data re-use through the register file can be complemented by L1 and L2 caches for the surrounding loops. In this case $1+d\cdot p$ fma operations have to be compared to one main memory load and one store per grid point and additional cache loads.

4.2 Time Slicing Algorithm

In case the ratio of arithmetic operations to memory operations is still too small, several time steps can be aggregated into the time slicing (or time skewing, temporal tiling) algorithm, see [9,10,11,12] and for more recent work [1,5,13,14,15].

Table 4. Scalar sliding window algorithm for order p Finite Difference stencils. All memory loads expect for one can be cached.

name	**register storage**	**load**	**store**	**fma**
1D	$1+p$	1	1	$1+p$
2D isotropic	$1+2p$	$1+p$	1	$1+2p$
3D isotropic	$1+3p$	$1+2p$	1	$1+3p$
2D anisotropic	$(1+p)^2$	$1+p$	1	$(1+p)^2$
3D anisotropic	$1+\frac{21}{4}p+\frac{27}{8}p^2$	$(1+p)^2$	1	$1+\frac{21}{4}p+\frac{27}{8}p^2$

Initially developed to perform most of the computations in cache rather than main memory, time slicing with wide slices does most of the computations in register with memory operations mainly to cache. A one dimensional version of time slicing with $p+1$ point stencils looks like this:

```
for (int x=(stepx-1)*s; x>=0; x=x-s) {
  r[0..s-1] = grid[x..x+s-1];                    // load memory
  for (int t=0; t<stept*p; t=t+p) {
    r[s..s+p-1] = grid[x+t+s..x+t+s+p-1];        // load cache
    r[0..s-1] = calc (r[0..s+p-1]);              // unroll
    grid[x+t+p..x+t+2*p-1] = r[0..p-1];          // store cache
  }
  grid[x+(stept+1)*p..x+stept*p+s-1] = r[p..s-1];// store memory
}
```

A spatial tile of size `s` is used to compute `stept` time steps at once. Additional input data for the inlined Finite Difference stencils `calc` is loaded from a section of `grid`, which is presumably in cache. This algorithm uses `grid` as input and output for u and as cached intermediate storage of the stencil halo zones. To make sure that `r` is mapped to registers and the number of s difference stencils are in fact unrolled, some compilers require explicit source code unrolling. Note that this implementation requires the tile size to be large enough $s \geq p$. Furthermore, an initialization of the time slices next to the border is needed, which is ignored here, see [8] for a 1D version with separate auxiliary storage.

In two dimensions the loop and storage structure is the similar. The grid pattern to compute a rectangle of $s_x \times s_y$ points is a rectangle of $(s_x+p)\times(s_y+p)$ points for the anisotropic stencils. The points to be loaded and stored in each time step are an old rectangle minus a new one, forming L-shaped domains. If we assume that data re-use of the innermost x-loop fits into L1 cache and the y-loop into L2 cache, one leg of the L-domain is mapped to L2 cache and the remaining rectangle to L1 cache. Some snapshots of a two dimensional scheme are depicted in Fig. 2.

```
for (int y=(stepy-1)*sy; y>=0; y=y-sy) {
  for (int x=(stepx-1)*sx; x>=0; x=x-sx) {
    load rectangle [0..sy-1]*[0..sx-1] at grid[y][x] from memory
    for (int t=0; t<stept*p; t=t+p) {
      load L-domain [0..sy+p-1]*[0..sx+p-1] minus [0..sy-1]*[0..sx-1]
           at grid[y+t][x+t] from caches
      calc rectangle [0..sy-1]*[0..sx-1]
      store L-domain [0..sy-1]*[0..sx-1] minus [p..sy-1]*[p..sx-1]
            at grid[y+t+p][x+t+p] to caches
    }
    store rectangle [0..sy-p-1]*[p..sx-p-1]
          at grid[y+stept*p][x+stept*p] to memory
  }
}
```

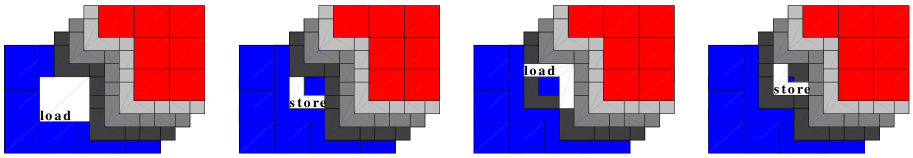


Fig. 2. Snapshots of the 2D time slicing algorithm, left to right, initial rectangle and later L-shaped pattern loads and the consecutive stores. The grid points are organized in the y-x plane. The time level is color-coded, values are shifted in x- and y-direction proportional to t. Initial values on the left (blue), results on the right (red).

The storage patterns of the isotropic stencils are rectangles minus rectangles $p/2 \times p/2$ at the four corners. In three dimensions cubes and differences of cubes are mapped to three levels of cache. The memory access pattern change accordingly.

The computation to memory operation ratio is comparable to the sliding window algorithm, with the advantage to substitute main memory access by cache access, at least for large numbers of time steps.

Table 5. Scalar time slice algorithm for order p Finite Difference stencils with tile size $s \geq p$. All memory loads and stores can be cached.

name	register storage	load / store
1D	$s+p$	p / p
2D isotropic	$(s_x+p)(s_y+p)-p^2$	$(s_x+p)(s_y+p)-s_xs_y-\frac{3}{4}p^2$
		$s_xs_y-(s_x-p)(s_y-p)-\frac{1}{4}p^2$
3D isotropic	$(s_x+p)(s_y+p)(s_z+p)-p^3$	$(s_x+p)(s_y+p)(s_z+p)-s_xs_ys_z-\frac{7}{8}p^3$
		$s_xs_ys_z-(s_x-p)(s_y-p)(s_z-p)-\frac{1}{8}p^3$
2D anisotropic	$(s_x+p)(s_y+p)$	$(s_x+p)(s_y+p)-s_xs_y$
		$s_xs_y-(s_x-p)(s_y-p)$
3D anisotropic	$\leq (s_x+p)(s_y+p)(s_z+p)$	$(s_x+p)(s_y+p)(s_z+p)-s_xs_ys_z$
		$s_xs_ys_z-(s_x-p)(s_y-p)(s_z-p)$

5 Vectorization and Data Layout

All processor architectures under consideration are (parallel) vector processors. For reasons of performance, vector operations have to be used rather than scalar operations. A naive approach would be to block the innermost loop in the size of the vector length, exploit instruction level parallelism and use vector load operations not aligned to the length of a vector. However, unaligned memory access can imply more memory access if it crosses cache lines. Further, data re-use in registers is inhibited by this procedure.

An alternative approach would be to emulate unaligned memory access by loading consecutive aligned vectors and additional vector shift operations. This is again expensive, partly because arbitrary vector shift instructions are not available. We will come back to this topic in the next subsections.

5.1 Vectors of Independent Tasks

One strategy to fully exploit the potential of vector instructions to consider them as SIMD parallel, independent tasks. Hence each vector component represents one grid, e.g. one octant of the full domain for vector length 8. Since a single step of the Finite Differences is fully parallel, a coupling just appears through the boundary conditions, which is cheap to implement.

The vector load and store operations have to be aligned for performance reasons. Hence the independent sub-grids have to be interleaved in memory accordingly. The scalar algorithm is simply vectorized by substitution of the scalar data type by the vector type and additional boundary procedures.

5.2 Shifted Vectors on CPUs

No changes in memory layout and a more efficient use of vector registers would be the introduction of vector shift instructions like

```
vec shift (vec a, vec b, int i) { // vector length n, 0<=i<=n
  return [a[i..n-1] b[0..i-1]];
}
```

A CPU of Ivy Bridge type for example is able to issue one floating point vector add, one mul and one permute operation per cycle in addition to integer and memory instructions. Unfortunately, the permute instructions have limited capabilities only:

The SSE instruction set offers the two vector argument instruction `_mm_shuffle`, which is sufficient for double precision, but in single precision can only be used for groups of two float2 values. The AVX instructions have a similar `_mm256_shuffle` instruction operating on each half-vector and `_mm256_permute2f128` to permute both half-vectors. This is again sufficient for four double values, such that one instruction can be used for shift one and the other for shift two, and the combination gives shift three. However, we still can only handle groups of float2 values. Hence, two or four interleaved parts of the domain are stored in memory in addition to some partial vector shift by one or two permute instructions.

Note that Intel Phi 512 bit vector instructions lead to flexible permute operations of groups of float4 `_mm512_mask_permute4f128`. The IBM Power AltiVec instructions set is more flexible in vector rotate, but leads to a multi instruction vector shift implementation.

5.3 Shifted Vectors on GPUs

In the GPU case vector shift can be implemented through Cuda `__shared__` memory respectively OpenCL `__local` memory. In warp synchronous parallel computing, i.e. short vectors of warp size, length 32 or 64 without explicit synchronization instructions, shared memory has to be marked as `volatile`.

Nvidia Kepler (capability ≥ 3.0) offers an additional vector shift for one vector without shared memory by `__shfl_up` and `__shfl_down` instructions. However, an implementation of a two argument vector shift still requires several instructions.

The hardware multi threading of GPUs allows for larger vector sizes. Vector shift can be implemented again through shared memory. Explicit synchronization is needed via Cuda `__syncthreads` or OpenCL `barrier`. Now the sliding window algorithms can be used again with shifted vectors [2].

6 Experiments

For reasons of comparison we perform experiments on a number of different generations of CPUs and GPUs. Since there are large absolute performance differences, in many cases we consider relative performance with respect to the speed of the add or the fma pipeline, i.e. peak performance of the Finite Difference stencil. The absolute numbers for single processor cores can be recovered by the number of operations per second times vector length in Tab. 2 and the number of operations per stencil in Fig. 1.

6.1 Compiler

The experiments depend on the quality of the code. We have used the compilers listed in Tab. 6. In most cases the gcc compilers gave superior x86 execution code. The larger the code and the number of unrolled instructions, the larger was gap between gcc version 4.7 and other compilers. Inspection of x86 assembly code generated by the different compilers gave no obvious explanation and we speculate that the different clock-per-instruction rates are due to the optimization level of instruction scheduling.

Table 6. List of C/C++ compilers in use. Sample compile time and performance comparison for a 3D 4th order example on an Intel Ivy Bridge CPU, compiler optimization options `-O3 -march=native`, code with AVX intrinsics.

name		**version**	**source**	**compile**	**performance**
C	**C++**			**time [s]**	**[GF]**
gcc	g++	4.7.0	FSF	40.907	6.09
gcc	g++	4.6.3	FSF	14.284	6.14
clang	clang++	3.2	llvm	55.899	3.59
icc	icpc	13.0.0	Intel	80.858	2.66

Source code loop unrolling and placement of vector elements in variables to be mapped to registers was done by a custom source-to-source preprocessor. The codes were written in C++ using SSE, AVX and FMA x86 vector intrinsics (if applicable), Cuda, and OpenCL respectively. Experiments were run on Linux Ubuntu 12.04 64-bit for x86 CPUs and Nvidia Cuda 5.0 for Nvidia GPUs. AMD GPUs were run with AMD APP OpenCL 2.7 on Linux Ubuntu 11.04.

6.2 Single Precision on a Single Core

The one dimensional experiments are summarized in the top rows of Figs. 3 and 5. The time slicing algorithm seems to be superior in most of the cases compared to the sliding window. The amount of required registers grows with the stencil order, such that the tile size is limited for time slicing, while the amount of data re-use grows for sliding windows. Hence there will be a turn over point at high order when sliding window is more efficient. However, this point is not reached in the 1D experiments. The 1D results show extremely high relative performance both for CPU and GPU. Some architectures have difficulties for higher order time slicing due to a shortage of registers compared to their floating point pipeline length. The detailed analysis in Fig. 4 shows that vector shift instructions can help reduce the tile sizes and the register pressure and improve higher order results.

In the 2D case, sliding window becomes superior for higher order stencils on CPUs and Nvidia GPUs, while time slicing is still better on AMD GPUs and generally for lower order stencils. Vector shift operations improve efficiency above 2nd order stencils and the optimal tile sizes are rather small, see Fig. 6. In the 3D case sliding window is superior to time slicing for all higher order stencils on CPUs and generally on GPUs. In the 3D case and in the higher order 2D case, large virtual vectors start to become superior to small warp synchronous vectors on GPUs.

Anisotropic stencils in Fig. 7 introduce substantially more operations with roughly the same memory traffic. However, the number of intermediate registers required grows, such that time slicing for higher order or 3D runs out of registers and almost constant performance sliding window starts to outperform time slicing. The problem of a small register file is much more pronounced in 3D, where performance always drops with increasing order.

6.3 Double Precision Arithmetic

A summary of the absolute performance in Tab. 7 and Tab. 8 show the performance drop with increasing spatial dimension, which is mainly due to cache access to fetch halo values in the outer loop directions. Furthermore, the time slicing seems to be superior for the 1D case, while sliding window is improving for higher dimensions.

Table 7. Absolute performance numbers of a single core CPU. Numbers of the time slicing algorithm in single and double precision arithmetic. Colored numbers indicate the shifted vector version. * marks SSE on AVX enabled CPUs.

processor		**1D**		**2D**		**3D**	
architecture	**name**	**single [GF]**	**double [GF]**	**single [GF]**	**double [GF]**	**single [GF]**	**double [GF]**
Intel Ivy Bridge	i5-3450	55.2	27.6	36.9	18.7	20.1	9.4
Intel Sandy Bridge	i7-2600	59.6	29.8	40.8	20.4	21.2	10.4
Intel Core	Xeon E5405	15.8	7.9	10.8	5.4	7.2	3.2
AMD Bulldozer	FX-8150	44.2	22.1	22.7*	11.3*	13.0	7.2*
AMD K10	Opteron 6168	15.1	7.5	9.1	4.5	5.9	2.3
AMD K8	Opteron 865	5.8	2.5	4.3	2.1	2.6	0.9

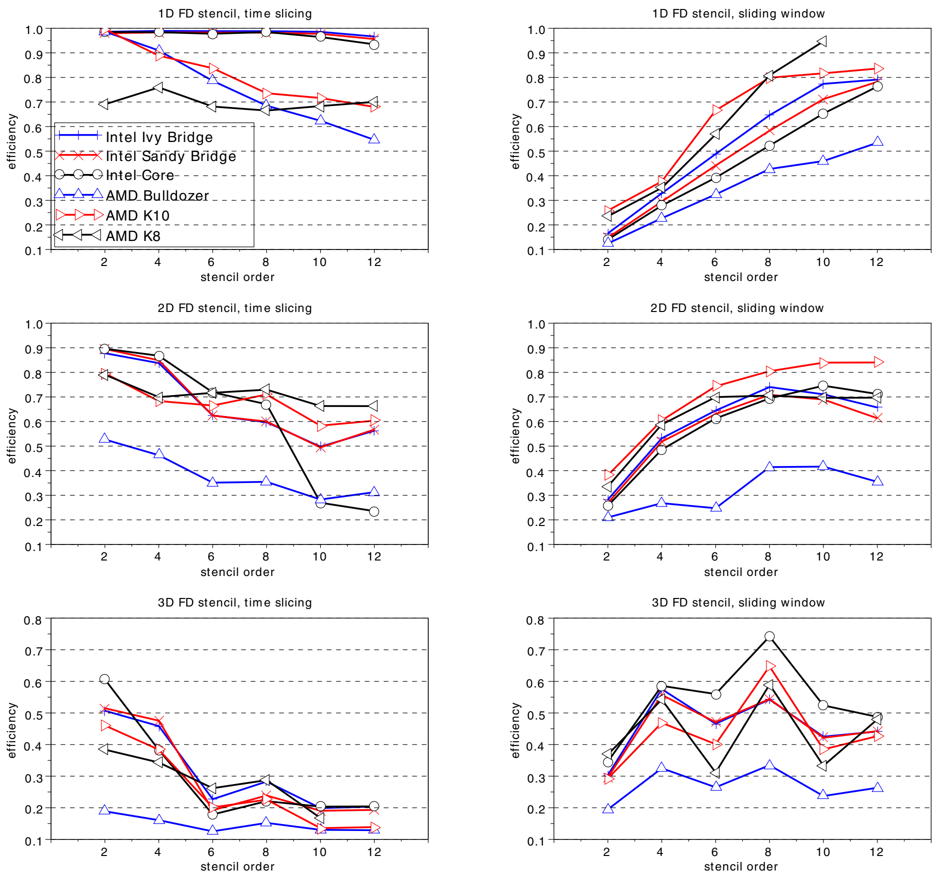


Fig. 3. Relative performance vs. stencil order of the sliding window and time slicing algorithms on CPUs

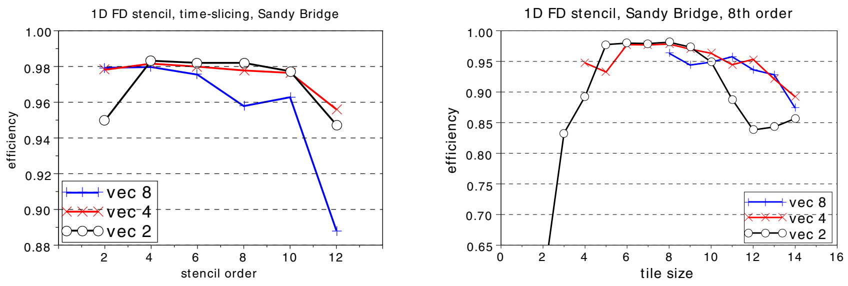


Fig. 4. 1D FD stencil. Time slicing on Sandy Bridge CPU. Independent vector components (vec 8), permute of float4 (vec 4) and permute+shuffle of float2 (vec 2). Relative performance vs. order p and tile size s.

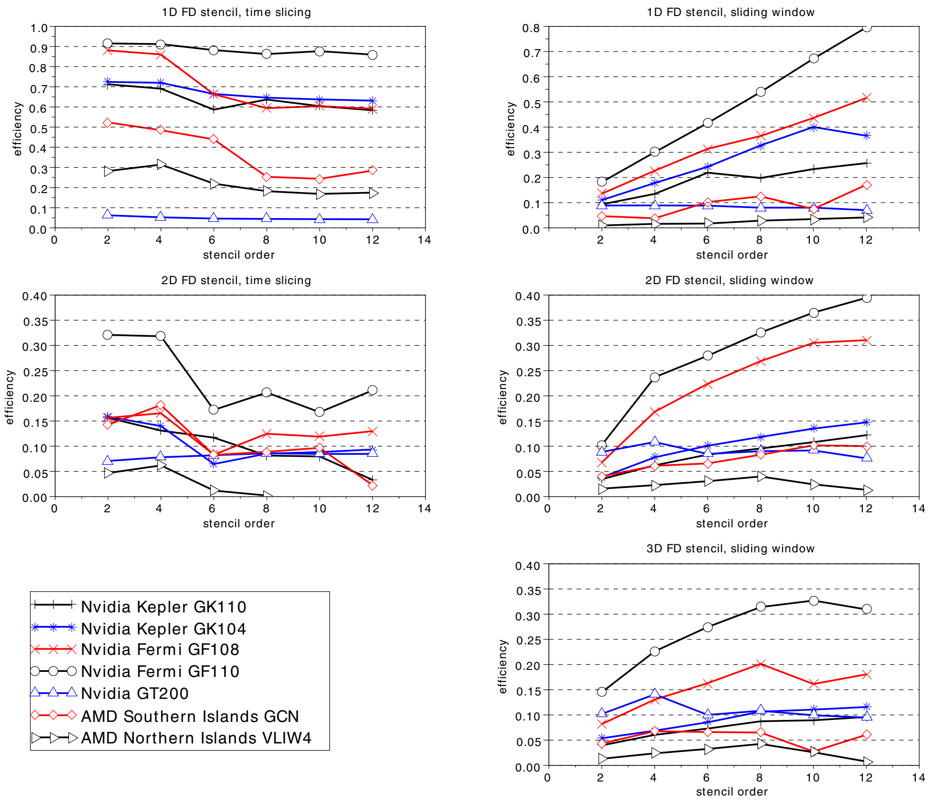


Fig. 5. Relative performance vs. stencil order of the sliding window and time slicing algorithms on GPUs

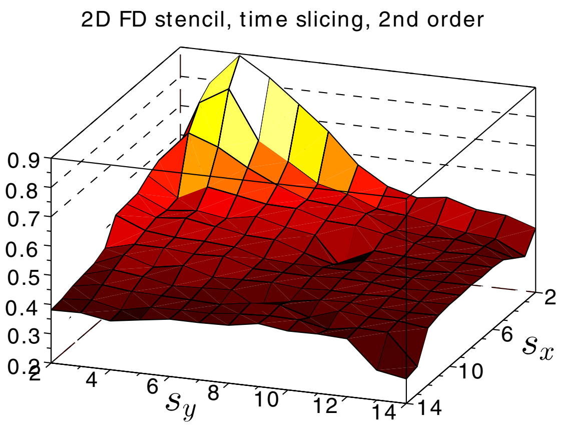


Fig. 6. 2D FD stencil. Time slicing on Sandy Bridge CPU. Relative performance vs.tile size $s_x \times s_y$.

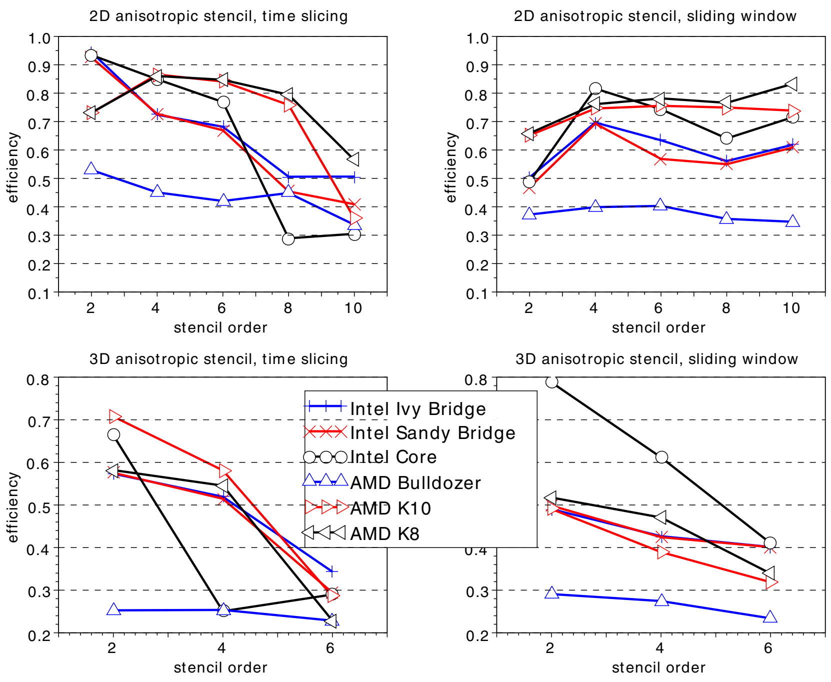


Fig. 7. 2D and 3D compact anisotropic FD stencil. Relative performance vs. stencil order of sliding window and time slicing on CPUs.

The CPU double precision numbers are consistently one half of the single precision. A 64 bit double number requires double the space in vector registers, caches and memory. The throughput and latency of the floating point pipelines remains the same in terms of vectors per time.

Double precision on GPUs is different: Cache and memory capacity and the number of available registers is halved, the throughput changes by a factor of 1/24 to 1/2, see Tab. 2. Slower floating point pipelines relative to the memory bandwidth results in an increased relative performance for memory bandwidth bound algorithms. The limited register file however results in smaller tile sizes. The Nvidia Tesla numbers demonstrate roughly half the double precision than single precision, consistent to the floating point performance. Slower double precision pipelines show higher relative performance. The ratio of single to double precision with increasing dimensions tends approach the ratio of memory throughput. Note that the AMD GPUs show even better double precision performance.

6.4 Outlook

The answer to the general question of CPU versus GPU performance mainly depends on metric of comparison, whether it be performance per core, per chip, per price or per electric power. A single CPU or GPU architecture is available as chips

Table 8. Absolute performance numbers of a multi-processor GPU chip. Numbers in single and double precision arithmetic of a time slicing (black) or a sliding window shifted-vector (grey) algorithm. * marks a shifted vector version of time slicing.

processor		1D		2D		3D	
architecture	**name**	**single [GF]**	**double [GF]**	**single [GF]**	**double [GF]**	**single [GF]**	**double [GF]**
Nvidia Kepler	GK110, Tesla K20c	1829	685	355	179	211	123
Nvidia Kepler	GK104, GTX 680	1638	98.3	309	75.0	219	60.4
Nvidia Fermi	GF100, Tesla C2050	655	302	241	116	196	98.6
Nvidia Fermi	GF110, GTX 590	794	114	304	78.5	236	69.4
AMD South. Isl.	GCN, HD 7970	1322	699	421	473	145	137
AMD North. Isl.	VLIW4, HD 6990	560	403	96.4	79.1*	62.7	52.5

of different numbers of cores. These can be further aggregated into shared and distributed memory systems of several chips. A comparison will have to balance the number of cores to compare, based on some criterion. So far we compared single cores, which includes instruction level parallelism and vector instructions.

Memory bandwidth bound algorithms on shared memory systems, especially on multi-core processors, will not show substantial parallel speed-ups. However, algorithms on private caches do scale. This is the case at least for one time slicing algorithm in 1D [8] and we expect it for 2D (L1 and L2 cache) and for the sliding window algorithm up to 3D. Beyond this, we expect a slow down due to shared LL cache and main memory. However, also shared LL cache and main memory usually scale for large numbers of cores. Note that the GPU experiments already take this into account, as all GPU cores execute the algorithm. Hardware multi-threading on Intel CPUs and two cores per module on AMD Bulldozer may accelerate a multi-threaded code, as long as a single thread does not fully load the floating point pipeline.

So far we neglected the treatment of the boundary nodes. However, in a parallelization based on domain decomposition, expensive inter-processor communication takes place by exchange of boundary data. Note that the algorithms differ in the communication pattern, but not in the total amount of data to transfer. The communication would again be more pronounced in higher dimensions and for higher order stencils, where the ratio of boundary nodes to inner nodes increases. We refer to [6,14] for multi-core and to [4,7] for distributed memory.

7 Conclusions

We were able to develop efficient vectorized sliding window and time slice implementations of Finite Differences in one, two and three dimensions, orders two to twelve and isotropic and anisotropic symmetric operators, for CPUs with x86 AVX vector instrinsics and GPUs in Cuda and OpenCL. The optimization techniques include various vectorization strategies, a change of data layout and loop unrolling. The result showed whether one of the algorithms was able to sustain high processor performance and to tolerate main memory latency: Time slicing tends to be superior for smaller Finite Difference stencils and large cache hierarchies found on current CPUs, while sliding window was better for larger stencils and larger register files like on GPUs.

Interesting generalizations of the model problem include variable coefficient difference stencils and systems of equations and hierarchies of grids with mesh refinement and multigrid algorithms [6,11,12,13]. The data access patterns are more complex and the amount of data per grid point increases.

Acknowledgments. The authors would like to thank DFG for partial support under grant SFB/TR7 “gravitational wave astronomy” and the anonymous referees for their helpful comments.

References

1. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Rev. 51(1), 129–159 (2009)
2. Micikevicius, P.: 3D finite difference computation on GPUs using Cuda. In: Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, pp. 79–84. ACM (2009)
3. Weyhausen, A.: Numerical algorithms of general relativity for heterogeneous computing environments. Diplomarbeit, Universität Jena, Physics Dept. (2010)
4. Maruyama, N., Nomura, T., Sato, K., Matsuoka, S.: Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In: Supercomputing. IEEE (2011)
5. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance code generation for stencil computations on GPU architectures. In: Proceedings of the 26th ACM International Conference on Supercomputing (2012)
6. Williams, S., Kalamkar, D., Singh, A., Deshpande, A., Straalen, B.V., Smelyanskiy, M., Almgren, A., Dubey, P., Shalf, J., Oliker, L.: Optimization of geometric multigrid for emerging multi- and manycore processors. In: Supercomputing. IEEE (2012)
7. Zhang, Y., Mueller, F.: Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In: Proc. 10th Int. Symp. Code Gen. Optim., San Jose, CA (2012)
8. Zumbusch, G.: Tuning a finite difference computation for parallel vector processors. In: 11th Int. Symp. Parallel and Distrib. Comput. CPS, pp. 63–70. IEEE (2012)
9. Song, Y., Li, Z.: New tiling techniques to improve cache temporal locality. In: Proc. ACM SIGPLAN Conf. Prog. Lang. Design Impl., Atlanta, pp. 215–228 (1999)
10. McCalpin, J., Wonnacott, D.: Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Rutgers Univ. (1999)
11. Rivera, G., Tseng, C.: Tiling optimizations for 3D scientific computations. In: Supercomputing (2000)
12. Weiß, C.: Data Locality Optimizations for Multigrid Methods on Structured Grids. PhD thesis, TU München (2001)
13. Stürmer, M., Treibig, J., Rüde, U.: Optimising a 3D multigrid algorithm for the IA-64 architecture. Int. J. Computational Science and Engineering 4, 29–35 (2008)
14. Wellein, G., Hager, G., Zeiser, T., Wittmann, M., Fehske, H.: Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In: Int. Comput. Soft. and Applications Conf. (COMPSAC), pp. 579–586 (2009)
15. Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: Supercomputing. IEEE (2010)

Part V

Applications of Parallel Computing in Industry and Engineering

Preconditioning for Large Scale Micro Finite Element Analyses of 3D Poroelasticity

Peter Arbenz* and Erhan Turan

Department of Computer Science, ETH Zürich, Zürich, Switzerland
arbenz@inf.ethz.ch,
turane@inf.ethz.ch, turane@gmail.com

Abstract. Osteoporosis is considered as a major health problem in the world. An understanding of the behavior of human bone under cyclic load requires numerical simulation of the physics. For that purpose, a large scale poroleastic solver is developed based on the mixed finite element method. This approach is free of numerical instabilities yet the discretization leads to an indefinite system that needs special attention. In this work, a comparison is made on several preconditioners that work efficiently in parallel environments.

Keywords: poroelasticity, finite elements, flexible GMRES, optimal preconditioning.

1 Introduction

Poroelasticity is the science of deformation of porous media filled with fluids. The foundations of the classical theory of poroelasticity is stated by Biot [5] at which a porous Hookean solid is coupled with Darcy's law in conjunction with continuity to model the fluid passing through the pores of the solid matrix. Although there exist other methodologies like the mixture theory [6] and also recently introduced microscopic theory and multiscale approaches [8,9], Biot's consolidation has a wide range of application in groundwater flow, biomechanics, coastal engineering and also in geothermal problems [7]. This work, on the other hand, is motivated by the disease called osteoporosis.

Osteoporosis is considered a major health problem that affects nearly 200 million people worldwide [17]. 37.5% of them are in Europe, the USA and Japan which cover only around 11% of the world population. This is mainly because the life expectancy is higher in developed countries and the disease mostly affects elderly people and especially women. In fact, according to the WHO, the risk for an osteoporotic fracture in women above 50 years is about 50%, for men the risk is about 20%. The European Union recognized this issue and supported a project called VPHOP to create a framework that aims at detecting possible patients precociously that potentially might suffer from osteoporosis. In this connection, a solver called ParFE [3] has been developed to model linear elastic response of

* Corresponding author.

realistic bone structures to exterior forces. The code is highly adapted to voxel-based models that have been generated by CT-scans. It is used by researchers that focus on bone remodeling which investigates the changes on the bone structure exposed to cyclic load. This adaption, the so called remodeling of the bone to environmental loading conditions is known as Wolff's law. Recently, Adachi *et al.* [1] analyzed this issue using a formulation that takes wall shear stresses at the lacuno-canalicular level of trabecular bone into account. Calculations of these values, on the other hand, require the analysis of bone poroelasticity [10] hence our aim was to extend ParFE to include poroelastic effects.

There are a couple of approaches to discretize the governing equations of Biot's Consolidation. In $\boldsymbol{u}$-p formulation, specific velocity, $\boldsymbol{f}$ is eliminated from the equations. Unfortunately, when piecewise linear (so-called Q_1-Q_1) basis functions are used with the standard Galerkin approach, oscillatory pressure fields are observed. To date, various strategies are suggested to stabilize $\boldsymbol{u}$-p formulation [2], [25], [27]. One of the proposed remedies is to use higher-order approximations for $\boldsymbol{u}$ than for p (Q_2-Q_1). Even if this pair is stable, second or higher order basis functions are prohibitive in terms of memory consumption. Further, in the analysis of bone structures the number of finite elements cannot be reduced since the geometry is approximated quite inaccurately anyway. Therefore, higher order elements do not improve the accuracy. Another approach would be to use a stability mechanism based on additional terms in the governing equations. In this scenario, the original form of the equations is not used anymore, and, additionally, there is a need for tuning of some stability parameter which is not known beforehand and requires computational experiments.

Instead of pursuing a displacement-pressure formulation, a $\boldsymbol{u}$-$\boldsymbol{f}$-p approach is employed in this study. This approach is free of instabilities and treats $\boldsymbol{f}$ as a primary variable. It also ensures continuity across element faces [12]. Furthermore, it acts as a stepping stone to more advanced analysis that might include Stoke's Flow for instance. On the other hand, this formulation requires storing more unknowns, i.e. all flux variables, and also a mixed finite element formulation should be used. In this formulation, the continuous problem turns into a saddle point problem. There are special iterative methods to solve symmetric indefinite systems, most notably SYMMLQ and MINRES [20]. These methods can suffer from loss of orthogonality among the Krylov vectors whence often GMRES [22] is used instead. A Krylov method is only efficient with an appropriate preconditioner. There is a variety of preconditioners available for saddle point problems, see [4] for a survey. Some preconditioners are specially adapted to problems from geomechanics [14]. In this study, we investigate block-diagonal and block-triangular preconditioners with a similar structure as the original system. We use AMG V-cycles to approximatively solve with the symmetric positive definite diagonal blocks.

Motivation behind this study is the simulation of bone remodeling which is a time-dependent phenomena where the changes in the geometry should be taken into consideration. Here, we focus on the fast and efficient solution of linear systems arising from the discretization of the problems bearing in mind the transient behavior of the governing equations. We envisage this solver to be a corner stone towards an efficient simulation of bone remodeling.

This article is organized as follows: In the next section, the governing equations and physical parameters of poroelasticity will be introduced. In Section 3 we discuss the numerical methods both in terms of the finite element discretization and the solution of the resulting indefinite linear problem. In section 5, results of rectangular domains and real bone geometries are discussed with particular interest on parallel processing. Finally, we summarize our study and make some suggestions for future work.

2 Mathematical Modeling

In order to model a poroelastic material, a set of three equations are needed. These are **(i)** the equilibrium equation which is to model elastic deformation, **(ii)** Darcy's law to model fluid flow, and **(iii)** mass conservation [26]. For a linear isotropic material, the equilibrium equation can be stated as

$$\nabla\cdot(2\mu\varepsilon(\boldsymbol{u}(\boldsymbol{x},t))) + \nabla(\lambda\nabla\cdot\boldsymbol{u}(\boldsymbol{x},t)) - \alpha\nabla p(\boldsymbol{x},t) + \boldsymbol{F}(\boldsymbol{x},t) = 0, \tag{1}$$

where $\boldsymbol{u}$ is the displacement, ε is the linearized strain tensor,

$$\varepsilon(\boldsymbol{u}) = \frac{1}{2}\left(\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^T\right),$$

p is the pressure, $\boldsymbol{F}$ is the external force, λ and μ are the Lamé parameters, and α is the Biot–Willis coefficient. The Lamé parameters are related to the Young's modulus E and the Poisson ratio ν by

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} = \frac{2G\nu}{(1-\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}.$$

The second equation of poroelasticity is Darcy's law,

$$\boldsymbol{f} = -\frac{k}{\eta}\nabla p, \tag{2}$$

that states that the fluid velocity $\boldsymbol{f}$ depends on the gradient of the pressure which is defined in excess of the hydrostatic pressure. Here, k is the permeability and η is the dynamic viscosity. Mass conservation can be written as

$$\alpha\frac{\partial}{\partial t}\nabla\cdot\boldsymbol{u} + S_\varepsilon\frac{\partial p}{\partial t} + \nabla\cdot\boldsymbol{f} = S_f, \tag{3}$$

where the specific storage at constant strain S_ε is a measure of released fluid volume per unit pressure in the control volume. S_f is an external source or sink.

The boundary conditions are given by

$$\begin{aligned} \boldsymbol{u}(\boldsymbol{x},t) &= \boldsymbol{u}_D, && \text{on } \partial\Omega_D,\\ \boldsymbol{\sigma}(\boldsymbol{x},t)\,\boldsymbol{n}(\boldsymbol{x}) &= \boldsymbol{t}(\boldsymbol{x},t), && \text{on } \partial\Omega_N,\\ \boldsymbol{f}(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x}) &= 0, && \text{on } \partial\Omega. \end{aligned} \tag{4}$$

We approximate the temporal derivatives in (3) by the implicit Euler method with time-step τ to obtain

$$\alpha\nabla\cdot\boldsymbol{u}(\boldsymbol{x},t) + S_\varepsilon p(\boldsymbol{x},t) + \tau\nabla\cdot\boldsymbol{f}(\boldsymbol{x},t) = \tau S_f + \alpha\nabla\cdot\boldsymbol{u}(\boldsymbol{x},t-\tau) + S_\varepsilon p(\boldsymbol{x},t-\tau).$$

We impose the initial conditions

$$\boldsymbol{u}(\boldsymbol{x},0) = u_0, \quad p(\boldsymbol{x},0) = p_0, \qquad \boldsymbol{x} \in \Omega. \tag{5}$$

In numerical tests, initial conditions might be assumed to be zero which satisfy the original set of equations. Or $\boldsymbol{u}_0$ and p_0 could be calculated from the loading like detailed in Section 5.1.

Although the system of equations is closed with (1), (2) and (3) including (4) and (5), some additional formulae are needed to compute the material properties. The constrained storage coefficient S_ε seems to be one of the parameters the calculation of which requires special attention. We can use,

$$S_\varepsilon = \frac{1}{K_s'}\left(1 - \frac{K}{K_s'}\right) + \phi\left(\frac{1}{K_f} - \frac{1}{K_\phi}\right) \tag{6}$$

where the porosity, ϕ, and the bulk moduli of the phases are used. In (6), K_s' is the unjacketed bulk modulus, K_f is the fluid bulk modulus, and K_ϕ is the unjacketed pore bulk modulus [26]. Material properties tabulated by Wang [26] are computed with the assumption[1] that solid-grain modulus K_s is equal to both K_s' and K_ϕ.

3 Numerical Methods

The governing equations are discretized with the $\boldsymbol{u}$-$\boldsymbol{f}$-p formulation using mixed finite elements. Each of the main variables is treated as a primary variable of the discrete problem. In this approach, the full set of equations (1), (2) and (3) is used. To employ the finite element method we need weak formulations of these equations. The weak formulation of Biot's model of poroelasticity uses the displacements $\boldsymbol{u} \in (H_1(\Omega))^3$, the fluid flux $\boldsymbol{f} \in H(\text{div};\Omega)$, and the pressure $p \in L_2(\Omega)$ [18] with the appropriate boundary conditions (4). The equations

$$\int_\Omega [2\mu\varepsilon(\boldsymbol{u}):\varepsilon(\boldsymbol{v}) + \lambda\,\nabla\cdot\boldsymbol{u}\,\nabla\cdot\boldsymbol{v}]\,d\Omega - \alpha\int_\Omega p\,\nabla\cdot\boldsymbol{v}\,d\Omega = 0, \tag{7}$$

$$\int_\Omega K^{-1}\boldsymbol{f}\cdot\boldsymbol{g}\,d\Omega - \int_\Omega p\,\nabla\cdot\boldsymbol{g}\,d\Omega = 0, \tag{8}$$

$$-\alpha\int_\Omega q\,\nabla\cdot\boldsymbol{u}\,d\Omega - \int_\Omega q\,\nabla\cdot\boldsymbol{f}\,d\Omega - S_\varepsilon\int_\Omega p\,q d\Omega = -\int_\Omega S_f q\,d\Omega, \tag{9}$$

have to hold for all $\boldsymbol{v} \in (H_1(\Omega))^3$, $\boldsymbol{g} \in H(\text{div};\Omega)$, and $q \in L_2(\Omega)$ satisfying homogeneous boundary conditions (4).

[1] Conditions that hold for this assumption are given by Detournay and Cheng [11].

For the finite element discretization we assume that the connected domain Ω is obtained by a 3-dimensional CT scan and is composed of *voxels*. In the above $\boldsymbol{u}$-$\boldsymbol{f}$-p formulation, we approximate each of the three displacement components by piecewise trilinear polynomials (Q_1), and the pressure by piecewise constant polynomials (P_0). The flux is approximated by Raviart–Thomas (RT_0) elements that have a continuous normal component across element interfaces [15]. The tangential components can jump. The displacements are determined by their values at the element corners, the pressure can be represented by its value at the element center. The RT_0 fluxes have a constant normal component on each element face. Hence, they can be represented by the normal components at the center of element faces. In a voxel, there are 24 degrees of freedom for the displacements, 6 degrees of freedom for the fluxes, and one degree of freedom for the pressure.

The advantage of this formulation over some other approaches like $\boldsymbol{u}$-p, is that the resulting discrete problem is stable and does not need additional terms to provide a stable numerical solution [12]. Because of the same reason the original set of governing equations are kept intact. Also, since there is no stability term, there is no need to perform computational experiments to find a optimized stability parameter. Even if the total number of unknowns are higher than with the $\boldsymbol{u}$-p formulation, fewer nonzeros are generated in the global matrix. This is more decisive when subblocks of the global stiffness matrix are to be stored. Additionally, flux values are continuous across element boundaries as stated by Ferronato *et al.* [12] which secures mass conservation. Having available the full set of primary variables, the improvement of the model to Stoke's flow can be realized with less difficulty since the fluxes are kept as primary variables. This approach avoids the calculation of the flux values in post-processing, as well.

The finite element discretization of the $\boldsymbol{u}$-$\boldsymbol{f}$-p formulation leads to the 3×3 block matrix

$$\mathscr{A}\boldsymbol{x} \equiv \begin{bmatrix} \boldsymbol{A}_{uu} & O & \boldsymbol{A}_{pu}^T \\ O & \boldsymbol{A}_{ff} & \boldsymbol{A}_{pf}^T \\ \boldsymbol{A}_{pu} & \boldsymbol{A}_{pf} & -\boldsymbol{A}_{pp} \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{f} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{b} \end{bmatrix}. \tag{10}$$

With appropriate boundary conditions, the diagonal blocks $\boldsymbol{A}_{uu}$, $\boldsymbol{A}_{ff}$, and $\boldsymbol{A}_{pp}$ are symmetric positive definite, such that the whole 3×3 block matrix is symmetric indefinite. Note that $\boldsymbol{A}_{pp}$ is diagonal. Also $\boldsymbol{A}_{ff}$ has a simple structure. Since the RT_0 vector basis functions on cubical finite elements have only one nonzero component, $\boldsymbol{A}_{ff}$ consists of essentially 1-dimensional, i.e. tridiagonal, pieces.

There are three widely used Krylov space methods for solving the symmetric indefinite system in (10). While MINRES and SYMMLQ are designed for solving precisely this type of equationss [20] the generalized minimal residual method, GMRES, is a solver for general linear system [22]. GMRES can be adapted to symmetric system such that it generates a symmetric tridiagonal projected matrix but still performs complete reorthogonalizations [23].

A crucial ingredient for the success of any Krylov space method is the preconditioner. We consider first symmetric positive-definite block-diagonal preconditioners similar to the one suggested by Lipnikov [18] for mimetic finite difference methods. Preconditioning $\mathscr{A}$ in (10) by the symmetric positive-definite block-diagonal matrix

$$\mathscr{M}_0 := \begin{bmatrix} \boldsymbol{A}_{uu} & & \\ & \boldsymbol{A}_{ff} & \\ & & \boldsymbol{S}_{pp} \end{bmatrix}, \tag{11}$$

with the *negative* Schur complement

$$\boldsymbol{S}_{pp} = \boldsymbol{A}_{pp} + \boldsymbol{A}_{pf}\boldsymbol{A}_{ff}^{-1}\boldsymbol{A}_{pf}^T + \boldsymbol{A}_{pu}\boldsymbol{A}_{uu}^{-1}\boldsymbol{A}_{pu}^T, \tag{12}$$

has just three (multiple) eigenvalues *provided* that $\boldsymbol{A}_{pp}$ vanishes [19, Remark 1].

Here, $\boldsymbol{A}_{pp} \neq \boldsymbol{0}$. But, since $\|\boldsymbol{A}_{pp}\|$ is small ($\mathscr{O}(h^3 S_\varepsilon)$), we argue that $\mathscr{M}_0^{-1}\mathscr{A}$ has three clusters of eigenvalues that make Krylov methods still converge in a few iteration steps.

Lipnikov [18] suggests two simplifications that make $\mathscr{M}_0$ cheaper to implement without compromising its quality: **(i)** the matrix $\boldsymbol{A}_{ff}$ is replaced by its diagonal $\boldsymbol{D}_{ff}$, and **(ii)** the exact Schur complement $\boldsymbol{S}_{pp}$ is approximated by

$$\tilde{\boldsymbol{S}}_{pp} = \boldsymbol{A}_{pp} + \boldsymbol{A}_{pf}\boldsymbol{D}_{ff}^{-1}\boldsymbol{A}_{pf}^T, \tag{13}$$

i.e., the last term in (12) is neglected. This term is $\mathscr{O}(1)$ while $\boldsymbol{A}_{pf}\boldsymbol{D}_{ff}^{-1}\boldsymbol{A}_{pf}^T \approx \boldsymbol{A}_{pf}\boldsymbol{A}_{ff}^{-1}\boldsymbol{A}_{pf}^T$, that approximates the Laplacian $-\nabla\cdot\nabla$, behaves like $\mathscr{O}(h^{-2})$. The approximate Schur complement $\tilde{\boldsymbol{S}}_{pp}$ is spectrally equivalent to $\boldsymbol{S}_{pp}$ [4].

While $\boldsymbol{S}_{pp}$ is dense, $\tilde{\boldsymbol{S}}_{pp}$ is a finite difference approximation of $-\alpha_1 \Delta p + \alpha_2 p$ with $\alpha_1 \sim \alpha\Delta t h^3 \frac{k}{\eta}$ and $\alpha_2 \sim h^3 S_\varepsilon$. Therefore, we suggest to use the preconditioner

$$\mathscr{M} := \begin{bmatrix} \boldsymbol{M}_{uu} & O & O \\ O & \boldsymbol{D}_{ff} & O \\ O & O & \boldsymbol{M}_{pp} \end{bmatrix}, \tag{14}$$

where $\boldsymbol{M}_{uu}$ and $\boldsymbol{M}_{pp}$ are multilevel preconditioners (a fixed number of V-cycles) calculated directly from $\boldsymbol{A}_{uu}$ and $\tilde{\boldsymbol{S}}_{pp}$, respectively.

Instead of a fixed number of V-cycles, we could solve these systems by the conjugate gradient method, preconditioned by $\boldsymbol{M}_{uu}$ and $\boldsymbol{M}_{pp}$.

Remark 1. We could replace the diagonal block $\boldsymbol{D}_{ff}$ in $\mathscr{M}$ by $\boldsymbol{A}_{ff}$. The factorization of $\boldsymbol{A}_{ff}$ does not produce fill-in.

We also consider preconditioners with block-triangular structure,

$$\mathscr{M}_1 := \begin{bmatrix} \boldsymbol{A}_{uu} & O & \boldsymbol{A}_{pu}^T \\ O & \boldsymbol{A}_{ff} & \boldsymbol{A}_{pf}^T \\ O & O & \boldsymbol{S}_{pp} \end{bmatrix}. \tag{15}$$

A Krylov space method for solving (10) preconditioned by $\mathscr{M}_1^{-1}$ converges in two iteration steps [4,19]. As with $\mathscr{M}_0$ the diagonal blocks of $\mathscr{M}_1$ are approximated by a fixed number of V-cycles or solved by PCG to high accuracy. The latter potentially leads to a very small number of iteration steps.

4 Parallel Implementation

Our code is written in C++ using software from the Trilinos project [16,24]. The foundation of Trilinos is the Epetra package which allows to define and build parallel matrices and vectors. Epetra hides the MPI communication behind sophisticated C++ constructs. GMRES is implemented in the package Belos which is a templated library of basic Krylov solvers. Belos also provides Krylov subspace recycling that will be helpful in the time dependent problems. The ML package provides smoothed aggregation-based algebraic multilevel preconditioners. For repartitioning of the variables a segregated approach is followed meaning that the primitive variables are distributed independently. We employed the procedure inherited from ParFE [3] to distribute the nodes (displacements) by means of ParMETIS [21]. The faces (flow) and elements (pressure) were distributed individually by calls to the Isorropia package of Trilinos. This may cause some unnecessary communication when we apply off-diagonal blocks in $\mathscr{A}$ or $\mathscr{M}$.

In addition to the connectivity list (element-to-node table) used in the elasticity problem, two additional tables are needed to form the system matrix: an element-to-face table and an element-neighbor list. Both tables have six columns. They can be easily created from the initial CT image.

5 Results and Discussion

In this section, results of two sets of problems will be discussed. First, a well-known benchmark problem is presented which is used to verify the solver. Rectangular full domains in 3D are analyzed where the parallel performance of the solver is investigated. Later, a real life problem will be tested to monitor the strength of the solver on bone-like structures.

5.1 Full Domains

There are a number of test problems used in geomechanics to check the validity of solvers. Here, we use Terzaghi's consolidation problem for which there exist a transient analytical solution [26]. Mathematically, the problem is 1D but with appropriate boundary conditions, the loading can be modeled in 3D, as we did in our analysis. The geometry of the model is depicted in Fig. 1. The solutions u and p, respectively, are given by

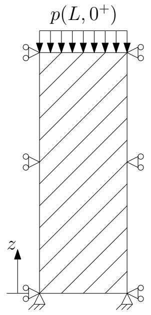


Fig. 1. Loading on the body

$$u(z,t) = c_M p_0 \left[(L-z) - \frac{8Lp_0}{\pi^2} \sum_{n=0}^{\infty} \frac{1}{(2n+1)^2} \exp\left(-N^2 ct\right) \cos(Nz) \right] + u_0,$$

$$p(z,t) = \frac{4p_0}{\pi} \sum_{n=0}^{\infty} \frac{1}{2n+1} \exp\left(-N^2 ct\right) \sin(Nz),$$

in which L is the length of the column, c is the diffusivity coefficient [11] and $N \equiv \frac{(2n+1)\pi}{2L}$. K_u is the undrained bulk modulus and M is the Biot modulus. The initial conditions for u and p are given by

$$w_0(z) = -\frac{z}{K_u + \frac{4\mu}{3}} p_L, \qquad p_0(z) = \frac{\alpha M}{K_u + \frac{4\mu}{3}} p_L, \tag{16}$$

In this test problem, the medium is assumed to be incompressible, hence α is set to unity. Material properties listed in Table 1 are taken similar to those used in [12] and the references therein.

Table 1. Material properties for benchmark problems

parameter	value	parameter	value
λ	40.0 MPa	k	1.02×10^{-6} mm^2
μ	40.0 MPa	η	1.0×10^{-9} MPa s
α	1	S_ε	1.65×10^{-4} MPa^{-1}

The sizes of the model problems are given in Table 2. As observed from the table, the number of unknowns scales approximately with factor eight. The mesh of model b_4 is shown in Fig. 2.

The tests are performed with right-preconditioned FGMRES(100) with an outer tolerance of 10^{-5}. The inner tolerances are kept equal for inner solves except that of $\boldsymbol{A}_{uu}$ block where a lower convergence criteria, 10^{-8}, is pursued for a high accuracy solution. Clearly, the solver scales well for large problems as seen from Fig. 3. However, for each of the three cases, b_32, b_64, and b_128, the block-diagonal preconditioner is faster. This is counterintuitive despite the fact that the number

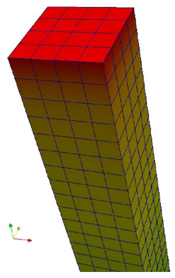


Fig. 2. Mesh b_4

Table 2. Test meshes for the first benchmark problem

mesh_id	elements	nodes	faces	total dof
b_1	30	124	151	553
b_2	240	529	964	2 851
b_4	1 920	3 025	6 736	17 731
b_8	15 360	19 521	49 984	123 907
b_16	122 880	139 009	384 256	924 163
b_32	983 040	1 046 529	3 011 584	7 134 211
b_64	7 864 320	8 116 225	23 842 816	56 055 811
b_128	62 914 560	63 918 081	189 743 104	444 411 904

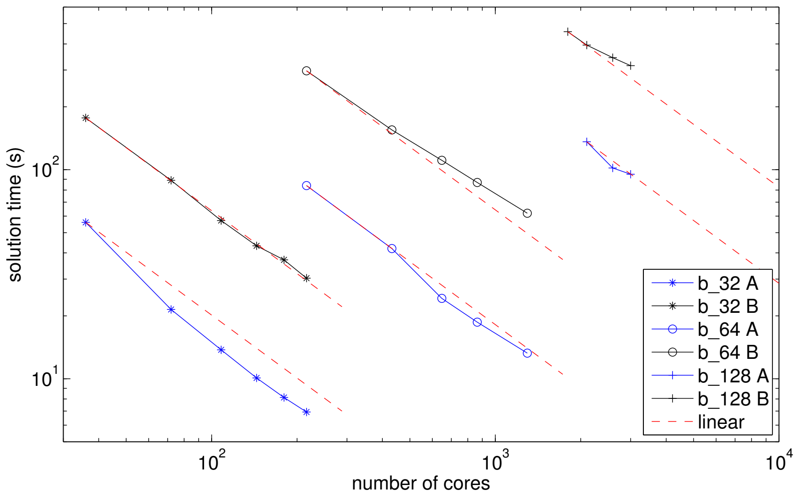


Fig. 3. Strong scaling on the test problem (A: block-diagonal, B: block-triangular)

of iterations are increasing with the mesh size and requires more iterations compared to block-triangular preconditioner as given in Table 3. This is a sign that a simpler implementation of the second preconditioner would suffice. One idea might be to relax the necessity to get a high accuracy solution for inner block solves. This is discussed next.

Table 3. Number of outer iterations

preconditioner	b_1	b_2	b_4	b_8	b_16	b_32	b_64	b_128
diagonal	58	82	84	86	84	90	92	94
triangular	20	31	33	36	38	39	39	41

Effect of Inner Tolerances. As explained before, the block-triangular preconditioner performs inner PCG solves. For each three block solutions, different stopping criteria are applied. In our numerical tests, we observed that the success of this second preconditioner depends solely on the tolerance of the $\mathbf{A}_{uu}$ problem while the effect of the accuracy on $\mathbf{A}_{ff}$ and $\mathbf{S}_{pp}$ blocks are merely significant. The performance of the block-triangular variant tested on the b_64 model is depicted in Fig. 4. Although the number of outer iterations are reduced to a constant value for lower inner tolerances on $\mathbf{A}_{uu}$, the run-time is shortest for a stopping criterion of 10^{-5}. The same trend is observed both for 216 and 1296 cores. Since the outer convergence criteria was also set to 10^{-5}, this raises the question whether the tolerance of the best inner stopping criterion is equal to the tolerance of the outer iteration. For that purpose, we performed additional

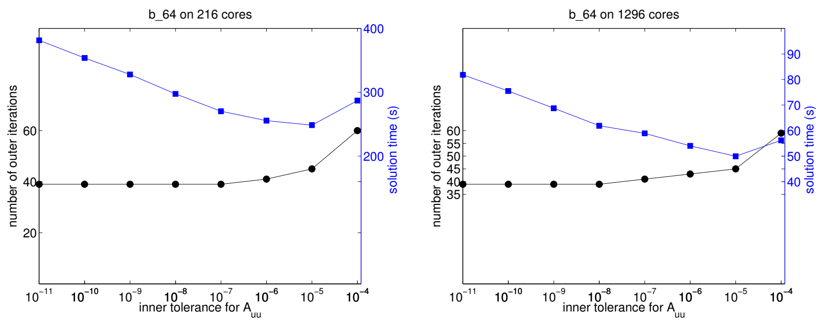


Fig. 4. Comparison of inner tolerances on the elasticity block, $\mathbf{A}_{uu}$

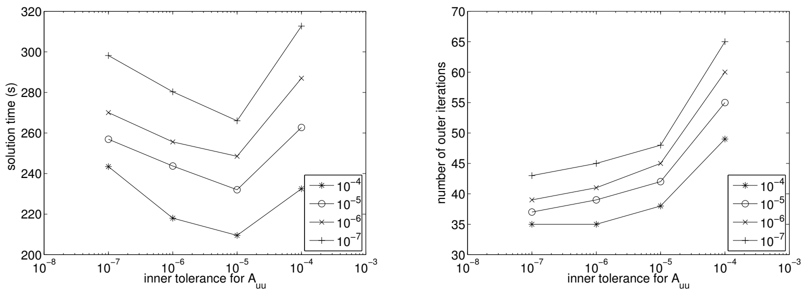


Fig. 5. Comparison of outer tolerances

simulation with varying outer tolerances. The results are given in Fig. 5 for the `b_64` model on 216 cores.

Clearly, 10^{-5} seems to be a good choice for the inner tolerance of the elasticity block when solution time is considered. The number of outer iterations tend to stay constant for smaller inner tolerances even for different stopping criteria for FMGRES. Yet, an inner tolerance of 10^{-5} is not enough to beat the block-diagonal preconditioner in run time. This suggests modifications on the implementation of the inner solves, which is considered next.

Variations on the Preconditioners. Another approach might be to avoid PCG and to use the multilevel preconditioner only. Similarly, we can add PCG acceleration on the first preconditioner with a probable sacrifice on run time but with an iteration count that is less sensitive to problem size. As a result, we test two additional preconditioner variants. The results for `b_32` and `b_64` are listed in Tables 4 and 5, respectively. Diagonal-I and Triangular-I are the original preconditioners introduced before. Diagonal-II is accelerated by PCG (tol=10^{-5}). The same tolerance is also used in Triangular-II instead of 10^{-5} as suggested in previous section. Triangular-III uses also the multilevel preconditioner only and that actually pays off when compared against the block-diagonal preconditioner, Diagonal-I. In addition to the absence of inner solves, $\boldsymbol{A}_{ff}$ block is replaced by its diagonal, $\boldsymbol{D}_{ff}$. The number of iterations for diagonal-II are fixed, 75 and 73 iterations are required for b_32 and b_64, respectively meaning that the iteration numbers are staying constant both in weak and strong scaling. Yet the run-time is prohibitive. On the other hand, the number of outer iterations for triangular-III shows a slight increase. Instead of 39 iterations as carried out by the original triangular preconditioner, now 68 and 79 iterations are needed for those two model problems. So, for fast solutions we must sacrifice on the outer iterations. Depending on the geometry and material properties, a different set of solution parameters can also be used to reduce the computational time. For the given geometries, however, we found the current setting leading to optimal solution times.

Table 4. Comparison of the preconditioners for b_32 – run time (in seconds) and iteration count

# cores	diagonal-I	diagonal-II	triangular-I	triangular-II	triangular-III
36	56.0 (79)	301 (75)	177 (39)	158 (48)	57.2 (65)
72	21.4 (79)	147 (75)	89.0 (39)	77.9 (48)	24.9 (67)
144	10.1 (81)	95.2 (75)	43.3 (39)	38.3 (48)	12.3 (68)
216	6.95 (83)	49.7 (75)	30.3 (39)	27.6 (48)	8.43 (69)

Table 5. Comparison of the preconditioners for b_64 – run time (in seconds) and iteration count

# cores	diagonal-I	diagonal-II	triangular-I	triangular-II	triangular-III
216	84.0 (91)	497 (73)	298 (39)	249 (45)	93.7 (74)
648	24.2 (91)	183 (73)	111 (39)	92.0 (46)	30.6 (81)
1296	13.3 (95)	103 (73)	61.9 (39)	50.8 (45)	18.0 (84)

5.2 Bone Structures

After testing full domains of various sizes, we performed several computations on bone samples. A bone sample, c_1, with a voxel size of 0.05 mm is taken as the basic mesh from which larger meshes, c_2, c_4, and c_8, are created by mirroring in 3D [3]. The problem sizes are listed in Table 6 and the two smallest sample models are shown in Fig. 6. The bodies are compressed along the z-direction with a fixed displacement value of 0.032 mm, 0.064 mm, 0.128 mm, 0.256 mm. Pressure and displacement values are kept as zero, initially. Computations are held on 2, 16, 128, and 1024 cores for the model geometries, respectively.

When we analyze the results on all four problems, we observe that the block-diagonal preconditioner is still the fastest in solution time. The block-triangular preconditioner is optimal in terms of the number of outer iterations, 5 and 7 using inner tolerances of 10^{-8} and 10^{-5} in the $\boldsymbol{A}_{uu}$ solves. However, when considering the time spent on computations to perform a weak-scaling analysis, we see that the run-time does not stay constant.

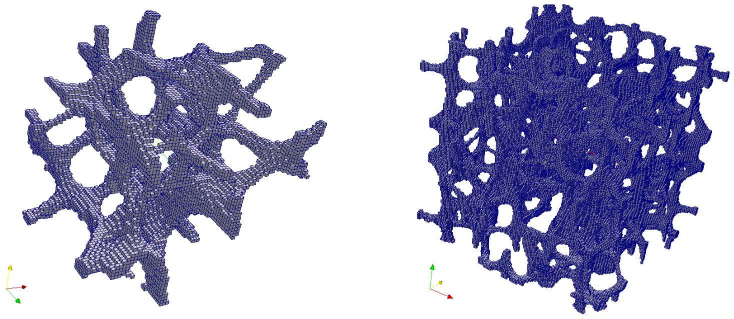


Fig. 6. Meshes `c_1` and `c_2`

Table 6. Test meshes on a bone sample

mesh_id	elements	nodes	faces	total dof
`c_1`	17 919	32 413	66 857	182 015
`c_2`	143 352	255 884	532 656	1 353 660
`c_4`	1 146 816	2 033 868	4 252 720	11 501 140
`c_8`	9 174 528	16 218 248	33 987 648	91 816 920

Table 7. Bone samples: Belos time (in seconds) and number of outer iterations

preconditioner	`c_1`	`c_2`	`c_4`	`c_8`
diagonal-I	3.42 (33)	14.8 (68)	21.6 (85)	34.9 (98)
triangular-I	8.67 (5)	22.2 (5)	34.0 (5)	56.1 (5)
triangular-II	6.94 (6)	20.6 (7)	30.3 (7)	50.0 (7)
triangular-III	5.94 (35)	22.1 (71)	33.5 (92)	51.9 (111)

6 Conclusions

In this study, the micro finite element analysis of bone poroelasticity is investigated. The mixed finite element method is used to discretize the problem using the $\boldsymbol{u}$-$\boldsymbol{f}$-p formulation. Displacement components are modeled with trilinear voxel elements whereas flux values are discretized with lowest order Raviart–Thomas elements. The pressure is represented by piecewise constants. The resulting saddle point problem is solved with flexible GMRES. Two different types of preconditioners are introduced, a block-diagonal and a block-triangular. An approximate Schur complement is calculated on the pressure block which improves the performance of the solver. Multilevel preconditioners are applied separately to each diagonal block. Additionally, PCG is employed in the block-triangular preconditioner to improve the accuracy of the inner solves. The solver is first tested against benchmark problems and validated in time. Strong scaling tests are performed and it is observed the solver scales well. In comparison of both preconditioners; it is noticed that the triangular variant is optimal in the sense that the iteration count is independent on the problem size. On the contrary, the block-diagonal preconditioner, although not optimal, is faster in solution time. In top of those preconditioners, two additional modifications are performed to test preconditioners that lay in between blocks diagonal and triangular preconditioners. It is seen that the inner solutions should not be too accurate and even they can be avoided and replaced my multilevel preconditioners, only. The computations are repeated for bone samples. At the end, it is observed that the resulting solver is successful to simulate bone poroelasticity.

Future work on large scale bone poroelasticity includes better distribution of the unknowns by graph repartitioning. Primary variables should be considered coupled instead of independent of each other. Future studies will also include the implementation of bone remodelling where the adaptation of the bone structure to external loading is considered. In remodelling, the mesh geometry changes. This affects **(i)** the problem size and **(ii)** the connectivity of both nodes and faces. It will be a big challenge to update the affected data structures efficiently. Furthermore, in the context of the solver, **(iii)** the initial data and **(iv)** the preconditioner have to be modified to take into account the geometric changes. The envisaged computational overhead might be alleviated by remodelling only after some number of time steps. To simplify the geometric representation of the domain a larger than necessary domain of voxels may be generated. Individual voxel are turned on or off by varying material properties ranging from 0 to real values [13].

Finally, the poroelastic solver can be extended to cover nonlinear effects to examine large deformations and fracture and further, non-isothermal features can be implemented which is useful to analyze geothermal systems.

Acknowledgments. This project is supported by the Swiss National Science Foundation grant #205320_125114 and by grant #223865 - VPHOP in the 7^{th} RTD framework programme of the European Commission.

References

1. Adachi, T.J., Kameo, Y., Hojo, M.: Trabecular bone remodelling simulation considering osteocytic response to fluid-induced shear stress. Phil. Trans. R. Soc. A 368, 2669–2682 (2010)
2. Aguilar, G., Gaspar, G., Lisbona, F., Rodrigo, C.: Numerical stabilization of Biot's consolidation model by a perturbation on the flow equation. Int. J. Numer. Methods Eng. 75, 1282–1300 (2008)
3. Arbenz, P., van Lenthe, G.H., Mennel, U., Müller, R., Sala, M.: A scalable multilevel preconditioner for matrix-free μ-finite element analysis of human bone structures. Int. J. Numer. Methods Eng. 73(7), 927–947 (2008)
4. Benzi, M., Golub, G.H., Liesen, J.: Numerical solution of saddle point problems. Acta Numer. 14, 1–90 (2005)
5. Biot, M.: General theory of three-dimensional consolidation. J. Appl. Phys. 12, 155–164 (1941)

6. Bowen, R.M.: Theory of mixtures. In: Eringen, A.C. (ed.) Continuum Physics, vol. 3, pp. 1–127. Academic Press, New York (1976)
7. Bundschuh, J., Arriaga, M.C.S.: Introduction to the Numerical Modeling of Groundwater and Geothermal Systems. Taylor and Francis, London (2010)
8. Coussy, O.: Poromechanics, 2nd edn. Wiley, Chichester (2004)
9. Coussy, O.: Mechanics and Physics of Porous Solids. Wiley, Chichester (2010)
10. Cowin, S.C.: Bone poroelasticity. J. Biomech. 32, 217–238 (1999)
11. Detournay, E., Cheng, D.A.H.: Fundamentals of poroelasticity. In: Fairhurst, C. (ed.) Comprehensive Rock Engineering: Principles, Practice & Projects, vol. 2, pp. 113–171. Pergamon, Oxford (1993)
12. Ferronato, M., Castelletto, N., Gambolati, G.: A fully coupled 3-D mixed finite element model of Biot consolidation. J. Comput. Phys. 229, 4813–4830 (2010)
13. Flaig, C., Arbenz, P.: A scalable memory efficient multigrid solver for micro-finite element analyses based on CT images. Parallel Comput. 37(12), 846–854 (2011)
14. Gambolati, G., Ferronato, M., Janna, C.: Preconditioners in computational geomechanics: A survey. Int. J. Numer. Anal. Meth. Geomech. 35, 980–996 (2011)
15. Girault, V., Raviart, P.A.: Finite Element Approximation of the Navier–Stokes Equations. Springer, Berlin (1979)
16. 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)
17. International Osteoporosis Organization (2012), http://www.iofbonehealth.org
18. Lipnikov, K.: Numerical Methods for the Biot Model in Poroelasticity. Ph.D. thesis, University of Houston (2002)
19. Murphy, M.F., Golub, G.H., Wathen, A.J.: A note on preconditioning for indefinite linear systems. SIAM J. Matrix Anal. Appl. 21(6), 1969–1972 (2000)
20. Paige, C.C., Saunders, M.A.: Solution of sparse indefinite systems of linear equations. SIAM J. Numer. Anal. 12, 617–629 (1975)
21. METIS-Family of Multilevel Partitioning Algorithms, http://glaros.dtc.umn.edu/gkhome/views/metis
22. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM, Philadelphia (2003)
23. Sleijpen, G.L.G., van der Vorst, H.A., Modersitzki, J.: Differences in the effects of rounding errors in Krylov solvers for symmetric indefinite linear systems. SIAM J. Matrix Anal. Appl. 22(3), 726–751 (2001)
24. The Trilinos Project Home Page, http://trilinos.sandia.gov/
25. Truty, A.: A Galerkin/least-squares finite element formulation for consolidation. Int. J. Numer. Methods Eng. 52, 763–786 (2001)
26. Wang, H.F.: Theory of Linear Poroelasticity with Applications to Geomechanics and Hydrogeology. Princeton University Press, Princeton (2000)
27. White, J.A., Borja, R.I.: Stabilized low-order finite elements for coupled solid-deformation/fluid-diffusion and their application to fault zone transients. Comput. Methods Appl. Mech. Engrg. 197, 4353–4366 (2008)

Parallel Solvers for Numerical Upscaling

R. Blaheta[1], O. Jakl[1], J. Starý[1], and Erhan Turan[2]

[1] Institute of Geonics AS CR, IT4Innovations Department, Ostrava, Czech Republic
[2] ETH Zürich, Department of Computer Science, Zürich, Switzerland

Abstract. This contribution deals with numerical upscaling of the elastic material behaviour, namely of geocomposites, from microscale to macroscale through finite element analysis. This computationally demanding task raises many algorithmic and implementation issues related to efficient parallel processing. On the solution of the arising boundary value problem, considered with either Dirichlet or Neumann boundary conditions, we discuss various parallelization strategies, and compare their implementations in the specialized in-house finite element package GEM and through the general numerical solution framework Trilinos.

Keywords: upscaling in elasticity, large scale linear systems, singular systems, iterative solvers, aggregation based preconditioners, parallel computation, Trilinos.

1 Introduction

Processes in materials that allow us to distinguish at least two separate scales (macroscale and microscale) are investigated in many applications including biomechanics (see e.g. [1]) and geomechanics (see [9]). The solved problems have dimensions typical for the macroscale whereas the microscale heterogeneity is not visible in their discretization by the finite element (FE) method. However the microscale influences the (material) properties which are used at the macroscale analysis.

This paper considers the upscaling of elastic behaviour of the material by means of numerical simulation of laboratory of in-situ testing of the materials, i.e. by loading material samples possessing the microstructure and observing their global deformation response.

We particularly investigate the mechanical behaviour of geocomposites arising from grouting a rock matrix by a polyurethane resin. The properties of specific coal geocomposites, which are characterized by a very complicated inner structure, can be achieved by numerical upscaling, which implements stress and strain driven tests based on microscale FE analysis.

The upscaling uses FE computations on very dense finite element meshes corresponding to 3D computer tomography images. The computations then require solution of large scale systems, which can be in addition ill-conditioned due to material heterogeneity and the use of pure Neumann boundary conditions in the case of stress driven tests. This motivates application of efficient iterative solvers running in parallel computing environments.

2 Numerical Upscaling

The upscaling in elasticity aims at obtaining macroscale elasticity tensor $\mathcal{C}_M$, which appears as representation of the macroscale (averaged) stress-strain relation,

$$\langle\sigma\rangle = \mathcal{C}_M \langle\varepsilon\rangle , \tag{1}$$

where ε and σ are strains and stresses in the microstructure, $\langle \cdot \rangle$ denotes componentwise averaging operator. The strains and stresses in the microstructure are computed by solving the following boundary value problem

$$\left.\begin{aligned} -\operatorname{div}\sigma &= 0 \\ \sigma &= \mathcal{C}_m\varepsilon \\ \varepsilon &= \tfrac{1}{2}\left(\nabla u + (\nabla u)^T\right) \end{aligned}\right\} \text{ in } \Omega \text{ + boundary conditions on } \partial\Omega. \tag{2}$$

Here $\mathcal{C}_m = \mathcal{C}_m(x)$ denotes local elasticity tensor in the microstructure, which can change rapidly (oscillatorily) with $x \in \Omega$, where Ω denotes the body of a sample (a representative volume). The volume forces are zero and the loading is given by boundary conditions, e.g.

$$u(x) = \varepsilon_0\, x \text{ on } \partial\Omega, \tag{3}$$

$$\sigma(x) \cdot \nu(x) = \sigma_0 \cdot \nu(x) \text{ on } \partial\Omega, \tag{4}$$

Here ν denotes the unit outward normal to the boundary $\partial\Omega$, (3) are pure Dirichlet boundary conditions with given strain tensor ε_0 and (4) are pure Neumann boundary conditions with given stress tensor σ_0. The pure Neumann (traction) boundary value problem is solvable, if

$$\int_{\partial\Omega} (\sigma_0 \cdot \nu(x)) v(x)\, dx = 0 \quad \forall v \in V_0, \tag{5}$$

where V_0 is the space of all rigid body motions compatible with the boundary conditions, in our case $V_0 = \{v : \ v(x) = a + b \times x, a, b \in R^3\}$. Note that the compatibility conditions (5) follows directly from the variational formulation of the elasticity problem and guarantee the existence of its solution, see e.g. [12]. The verification of (5) is straightforward. For the two described types of loading, we have no specific requirements on Ω. The use of six independent choices of ε_0 or σ_0 allows us to determine generally anisotropic elasticity tensors $\mathcal{C}_\varepsilon$ or $\mathcal{C}_\sigma$, respectively.

The loading with mixed boundary conditions like

$$u(x) \cdot \nu(x) = 0 \text{ on } \Gamma_{01}, \quad u(x) \cdot \nu(x) = u_0 \text{ on } \Gamma_{02}, \quad \sigma_t(x) = 0 \text{ on } \Gamma_{01} \cup \Gamma_{02}, \tag{6}$$

$$\sigma(x) \cdot \nu(x) = 0 \text{ on } \Gamma_1 = \partial\Omega \setminus (\Gamma_{01} \cup \Gamma_{02}), \tag{7}$$

where σ_t denotes tangential stress, allows us to simulate laboratory tests, as a uniaxial compression/tension. For (6), we assume cubic or cylindrical domain Ω with bottom and top faces Γ_{01} and Γ_{02}, respectively. Again the consistency conditions can be verified.

The $\mathcal{C}_\varepsilon$ and $\mathcal{C}_\sigma$ provide bounds (in the energetic sense) for macroscopic elasticity tensors C_M computed with the use of different boundary loading. Moreover, $\mathcal{C}_\varepsilon - \mathcal{C}_\sigma$ is positive semidefinite and its norm can be used for assessing the sufficiency of the size of Ω as a representative volume for the heterogeneous material, see e.g. [15] for more details. These facts motivate solving of the elasticity problems with both pure Dirichlet and pure Neumann boundary conditions.

Note that one loading with ε_0 or σ_0 is enough for establishing isotropic macroscale elasticity tensor $C_M\varepsilon = 3K\varepsilon_V + 2G\varepsilon_D$, where ε_V and ε_D denote the volumetric and deviatoric part of ε, respectively. Obviously, $3K = \|\sigma_V\|/\|\varepsilon_V\|$ and $2G = \|\sigma_D\|/\|\varepsilon_D\|$.

To compute the macroscale elasticity tensor $\mathcal{C}_M$, we solve the boundary value problem (2) with selected boundary conditions by the FE method. Particularly, we use FE software GEM (see [6]).

The FE mesh is prepared from digital images produced by the industrial X-ray computer tomography (CT) [9], which allows us noninvasive and nondestructive data acquisition on the inner microstructure of the domain of interest. The discretization divides regular rectangular voxel based elements into six tetrahedra in a way known as the Kuhn's decomposition and further uses piecewise linear FE approximation.

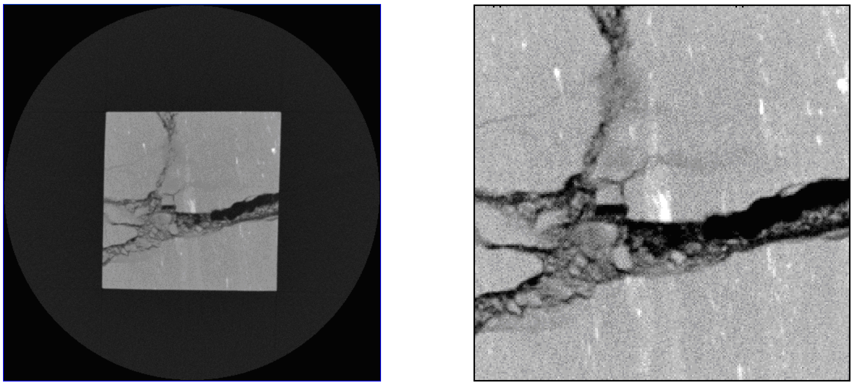


Fig. 1. An image taken by X-ray computer tomography in full size (512×512 pixels) and the investigated geocomposite area only (231×231 pixels)

In our experiments we employed a sample of a coal geocomposite of a cubic shape and 75 mm edge. Its microstructure was taken by the X-ray computer tomograph Toshiba, used with the courtesy of the Kumamoto University, which produced a set of images, see Figure 1, corresponding to parallel cuts through the sample. The reconstructed voxel grid has 231×231×37 elements 0.3×0.3×2.0 mm in size, and each voxel is supposed to be filled by a homogeneous material (coal, polyurethane resin, empty volume). The resulting linear system has then 6 135 936 degrees of freedom.

3 Parallel Iterative Solvers

The challenge in the modelling sequence is the solution of the arising large FE systems. The complicated microstructure geometry in 3D leads to systems of millions of degrees of freedom. The paper [1] reports even the solution of biomedical problems with over a billion degrees of freedom. Therefore we consider iterative solution methods like the conjugate gradient or another Krylov type iterations, see [13].

Moreover, the systems are ill-conditioned due to fine discretization and heterogeneity with possible large jumps in the material coefficients. Aiming at efficient and parallelizable preconditioners, we exploit two-level or multilevel preconditioners with coarse spaces created by aggregation, see e.g. [2], [14], [11], [7].

Finally the systems can be singular as a consequence of the used boundary conditions like (4) or (6) or presence of finite elements weekly hanged in the void space of the CT images. The latter phenomena can be easily removed by filling the voids with very week elastic materials. The former singularity due to non-trivial but known null space of the FE matrices can be solved by incorporation of an orthogonal projection to the matrix range into the iterative algorithms, see e.g. [10].

3.1 GEM Software

For practical computations we enhanced the in-house finite element modelling package called GEM, which is oriented on problems of geotechnics [6]. It is based on structured grids, linear tetrahedral finite elements and features several parallel iterative solvers based on the preconditioned conjugate gradient method, differing in the implemented problem decomposition technique (displacement decomposition, domain decomposition).

With one-dimensional domain decomposition (DD), GEM partitions the domain in the $\mathcal{Z}$ dimension into m subdomains (slices) with a two-layer minimal overlap, performs corresponding decomposition of vectors and matrices and assigns them to m concurrent processes, which pursue the conjugate gradient (CG) algorithm on the corresponding blocks of data. During the matrix-by-vector multiplication just local communication with neighbour slices is needed and the amount of communicated data is fairly small and proportional to the overlapped region.

Moreover, efficient Schwarz type preconditioners can be constructed. Our version of these preconditioners can be characterized by inexact solution of subproblems with the aid of incomplete factorization (one-level preconditioner), and by algebraic creation of an auxiliary global coarse grid by automatic aggregation of degrees of freedom (two-level preconditioner).

From the technical point of view, the developed parallel realizations of the preconditioned CG algorithm in GEM follow the message passing paradigm, and can be implemented by means of any message passing system. We have made use of the MPI message passing library, called from Fortran codes. The design gives rise to one master process and a number of slave processes, one per domain partition. As a computationally inexpensive task with just controlling functions, the master can share its processor with one of the workers.

3.2 Trilinos Framework

An effort to state and compare the efficiency and performance of GEM with another software oriented to the solution of similar problems encouraged us to try parallel solvers from Trilinos libraries. We chose Trilinos [16] because this project aims to develop and implement robust algorithms and enabling technologies within an object-oriented software design for the solution of large-scale complex multi-physics engineering and scientific problems.

We implemented new parallel code in C++ using only AztecOO, Belos, IFPACK, ML, Epetra and Teuchos from more than 50 software packages included in Trilinos. The first program was prepared in two weeks, when the implementation of data interface from Fortran (GEM) to C++ (Trilinos) showed to be the most time demanding part of work. Comparing with GEM, the code development was much faster (weeks versus months).

Within the code, we used Epetra primarily for construction and manipulation of distributed data structures like matrices and vectors and Teuchos mainly for smart pointers and parameter lists. The code can switch between the parallel CG solvers from the older AztecOO and newer Belos packages, always in combination with ILU or ML preconditioners. ILU denotes the one-level additive Schwarz preconditioner with subproblems solved by incomplete factorization (equivalent to the DD solver from GEM). ML denotes multi-level preconditioner with smoothed algebraic aggregations, where the Chebyshev smoother for the fine level and symmetric Gauss-Seidel smoother for the coarse level are used in our study.

The Trilinos code works with the same one-dimensional domain decomposition and appropriate distribution of data among processors as the parallel solvers of GEM.

3.3 Parallel Incomplete Factorization Preconditioners

Both GEM and Trilinos through its IFPACK library use the same parallel incomplete factorization strategy, which can be also viewed as a one-level additive Schwarz technique with subproblems replaced by incomplete factorization. If B is the preconditioner, then

$$B^{-1} = \sum_{k=0}^{NumProcs-1} R_k^T \widetilde{A}_k^{-1} R_k,$$

where R_k is the restriction operator from the global vector to the overlapping subdomain Ω_k, R_k^T is the corresponding prolongation operator, $\widetilde{A}_k$ is generally an approximation to the subdomain matrix $A_k = R_k A R_k^T$. The overlap can be defined by user and we shall apply a minimal nonzero overlap for both the GEM and Trilinos codes.

It is assumed that each subdomain is assigned to a different processor and the matrix storage is a special fixed stencil modification of Compressed Row Storage format (CRS) in GEM, which moreover stores only one symmetric part of the matrix. The matrix storage in Trilinos uses CRS. The matrix decomposition provided by both software packages corresponds to splitting of the one-dimensional vector of indices.

The incomplete factorization $\widetilde{A}_k = L_k L_k^T$ is defined as follows. GEM first approximates the local elasticity matrix A_k by A_k^0, which arises by putting to zero all elements representing coupling between nodal displacement components in different space directions. Then $\widetilde{A}_k = L_k L_k^T$ is a modified incomplete factorization of A_k^0 characterized by zero-fill with respect to A_k^0. The detailed description of this displacement decomposition-incomplete factorization can be found in [3].

IFPACK offers many choices, see [17], which are a bit difficult to compare. Using a recommendation provided by the default setting, we construct the ILU preconditioner as

```
Teuchos::RCP<Ifpack_Preconditioner> ILU;
Teuchos::ParameterList ILU_List;
Ifpack Factory;
int OverlapLevel = 1;
ILU = Teuchos::rcp( Factory.Create( "ILU", &*A, OverlapLevel ) );
ILU_List.set( "fact: drop tolerance", 1e-9 );
ILU_List.set( "fact: level-of-fill", 1 );
ILU_List.set( "schwarz: combine mode", "Add" );
IFPACK_CHK_ERR( ILU->SetParameters( ILU_List ) );
IFPACK_CHK_ERR( ILU->Initialize() );
IFPACK_CHK_ERR( ILU->Compute() );
```

It means that the fill can result only from combining entries on the original matrix positions and moreover small entries under the drop tolerance are ignored. Some experiments with parameter setting were done, but no alternative outperformed the default setting. Concerning particularly the class of elasticity stiffness matrices, no recommendation was found in IFPACK documentation.

3.4 Two-Level Schwarz Type Preconditioners

The one-level additive Schwarz preconditioner from the previous section can be enhanced by a coarse grid correction. GEM basically uses an additive variant of the Schwarz method, i.e. a preconditioner B such that

$$B^{-1} = R_c^T \widetilde{A}_c^{-1} R_c + \sum_{k=0}^{NumProcs-2} R_k^T \widetilde{A}_k^{-1} R_k .$$

Here R_c denotes a restriction on a coarse space, which is constructed in GEM by direct aggregation of 3×3 blocks corresponding to the nodes of the fine FE mesh.

As GEM works with structured meshes, we use a structured (regular) aggregations and a rather aggressive aggregation strategy. The matrix $\widetilde{A}_c = R_c A R_c^T$

is a Galerkin matrix on the coarse space and it is cheap to construct, because it preserves the regular grid pattern and is computed only by summing the matrix components. Thus the preprocessing phase is cheap. The coarse grid problem is solved by inner CG iterations, which use a lower accuracy (in our case $\varepsilon = 0.01$) and the same displacement decomposition-incomplete factorization preconditioning technique as described above.

The use of inner iterations results in the fact that B is a variable preconditioner. It frequently does not mind, but we stabilize the outer iterations by explicit orthogonalization, see [4]. More details concerning the GEM two-level additive Schwarz preconditioner can be found in [5].

Trilinos through its ML package [18] also provides possibility to construct a multi-level Schwarz preconditioner of a hybrid type, the smoother involves one level Schwarz method, the coarse space correction is added multiplicatively. These two steps provide nonsymmetric preconditioner, which can be used directly, see [4], or symmetrized by adding a third post-smoothing step. The coarse space is in ML created by smoothed aggregations, see e.g. [14]. The parameters used in our experiments, further denoted as ML-DD, are as follows

```
ML_Epetra::MultiLevelPreconditioner *MLDD = 0;
Teuchos::ParameterList DD_List;
ML_Epetra::SetDefaults( "DD", DD_List );
DD_List.set( "max levels", 10 );
DD_List.set( "aggregation: type", "MIS" );   // default: METIS
DD_List.set( "aggregation: nodes per aggregate", 128 );
if( NullSpaceActivated ) {
  DD_List.set( "null space: type", "pre-computed" );
  DD_List.set( "null space: dimension", v->NumVectors() );
  DD_List.set( "null space: vectors", v->Values() ); }
MLDD = new ML_Epetra::MultiLevelPreconditioner( *A, DD_List, true );
```

3.5 Multilevel Preconditioners

Trilinos through its ML package [18] allows us also to construct multilevel preconditioners based on smoothed aggregations and different smoothers. We make use of some default choice characterized by the following parameters for comparison of the multi-level techniques.

```
ML_Epetra::MultiLevelPreconditioner *ML = 0;
Teuchos::ParameterList ML_List;
ML_Epetra::SetDefaults( "SA", ML_List );
ML_List.set("smoother:type(level0)","Chebyshev" );
ML_List.set("smoother:type(level1)","symmetricGauss-Seidel" );
ML_List.set("coarse: max size", 8192 );
if( NullSpaceActivated ) {
  ML_List.set( "null space: type", "pre-computed" );
  ML_List.set( "null space: dimension", v->NumVectors() );
  ML_List.set( "null space: vectors", v->Values() ); }
ML = new ML_Epetra::MultiLevelPreconditioner( *A, ML_List, true );
```

3.6 Iterative Solution of Singular Systems

For getting some both-side estimates of the homogenized elasticity tensors, we are interested in solving singular systems corresponding to FE discretization of elasticity problems with pure Neumann boundary conditions. Theoretically, there is no problem to solve the singular and consistent systems by CG and similar methods. But practically, because of round-off errors, the solved systems can be ill-conditioned, due to presence of close to zero eigenvalues, or become slightly inconsistent. In this case, we can observe bad convergence or even divergence of iterations, see Figure 2.

A remedy is in use of a projection to the theoretical range $R(A)$, which can be constructed if we know a basis of the theoretical null space $N(A)$. For elasticity problems, such basis can be constructed from six rigid body modes $\{v_1, \dots, v_6\}$ and the projection $P\colon R^n \to R(A)$ can be evaluated from the formula

$$Pv = \sum_{i=1}^{6} \frac{(v,\, v_i)}{(v_i,\, v_i)} v_i.$$

With the aid of P we solve the projected system $PAPu = Pb$, see [10] for details.

In our applications, we can also meet another kind of singularity, corresponding to some weakly hanging elements coming to the FEM discretization from CT scan of complicated microstructure. This kind of singularity can be effectively removed by filling the voids with an artificial, very soft material.

4 Numerical Experiments

Our practical numerical experiments were carried out on a Linux workstation called Hubert, which is based on the TYAN VX50B4985-E barebone and powered by eight quad-core AMD Opteron 8830/2.5 GHz processors (32 cores in total), 128 GB of DDR2 RAM and RAID10 disk subsystem, made up from eight 15 000 RPM SAS drives. On this platform, Trilinos 10.8 and Intel Cluster Studio XE 2011 (comprising Fortran and C++ compilers and an MPI implementation) were employed in the experiments.

4.1 A Comparison of the GEM and Trilinos Solvers

The test results of different solvers for the Dirichlet problem are summarized in Table 1, where one can compare basic solution characteristics (in terms of number of iterations (#It) and wall-clock computation time (T)). In the case of GEM, solutions based on one-level additive Schwarz preconditioner combined with an incomplete factorization (DD) and two-level additive Schwarz preconditioner with a coarse grid created by aggregates of 6×6×3 original nodes (DD+CG) are presented. The Trilinos solution is represented by the AztecOO and Belos solvers. The behaviour of ILU, ML-DD (which are counterparts of DD and DD+CG) and ML preconditioners was practically the same in those solvers, therefore the results are shown for one choice only.

Table 1. Solution of the Dirichlet problem. For various numbers of subdomains (#Sd; corresponds to the number of processing elements employed), the number of iterations (#It) and wall-clock computation time (T) are provided. The symbol × means that the tests crashed due to insufficient operational memory.

	GEM				Trilinos					
	DD		*DD+CG*		*ILU(Aztec)*		*ML-DD(Belos)*		*ML(Belos)*	
# Sd	# It	T [s]	# It	T [s]	# It	T [s]	# It	T [s]	# It	T [s]
1	*108*	*407.8*			140	737.5	×		27	482,1
2	111	193.3	69	127.6	187	607.4	20	202.7	26	228,2
4	115	122.9	65	66.1	183	304.1	21	115.8	24	134.3
8	123	71.6	61	39.1	176	195.6	22	66.0	22	79,1
16	144	60.7	61	34.0	152	162.1	25	55.8	20	91,8

In the table, one can observe in general good scalability up to 8 processors, which is lost with greater number of processors due to the shared memory bottleneck. This happens for all GEM and Trilinos solvers. GEM solution makes profit from the coarse grid computations involved in the preconditioner, which shortens the computation (iterations, time) almost to a half. In comparison of incomplete factorization preconditioners, the number of iterations shows that the displacement decomposition involved in GEM outperforms the direct application of ILU. However there is the issue of tuning the Trilinos parameters. Similarly, it seems that ML-DD and ML applications are not optimally tuned for the elasticity problems. In any case, the assembly times, which are not included in the table, are much higher for ML and the highest for ML-DD.

4.2 Neumann Boundary Conditions

In Figure 2, we can observe the impact of the projections in the modified method. As one can see on the dependence of the relative accuracy on the number of iterations, projection is crucial for both one-level and two-level domain decomposition solution.

Now, let us compare the Neumann (with projections) and Dirichlet solutions, realized in GEM. According to Table 2, the Neumann formulation results in approximately twice higher iteration counts and computation times.

Table 2. Comparison of the Dirichlet and projected Neumann solution

	Dirichlet				Neumann			
	DD		*DD+CG*		*DD (P)*		*DD+CG (P)*	
# Sd	# It	T [s]	# It	T [s]	# It	T [s]	# It	T [s]
1	*108*	*407.8*						
2	111	193.3	69	127.6	293	541.4	137	256.4
4	115	122.9	65	66.1	302	302.2	124	125.9
8	123	71.6	61	39.1	300	175.3	115	75.7
16	144	60.7	61	34.0	350	148.5	116	73.6

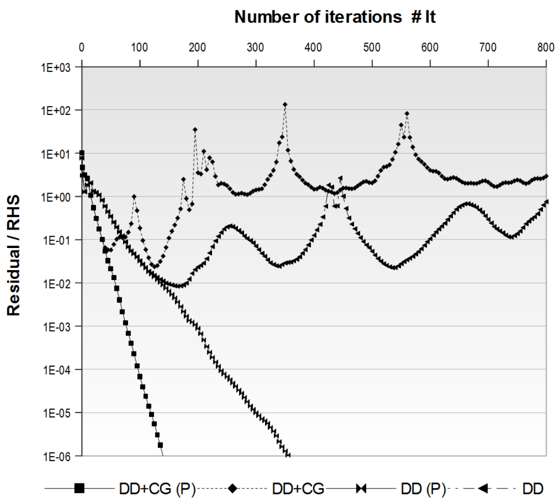


Fig. 2. Practical impact of the projection (P) on the solution of the Neumann problem case through one-level and two-level domain decomposition

4.3 Influence of Heterogeneity

Figure 3 illustrates the influence of heterogeneity on the behaviour of our most efficient solution method, namely on GEM's domain decomposition solution with Dirichlet boundary conditions.

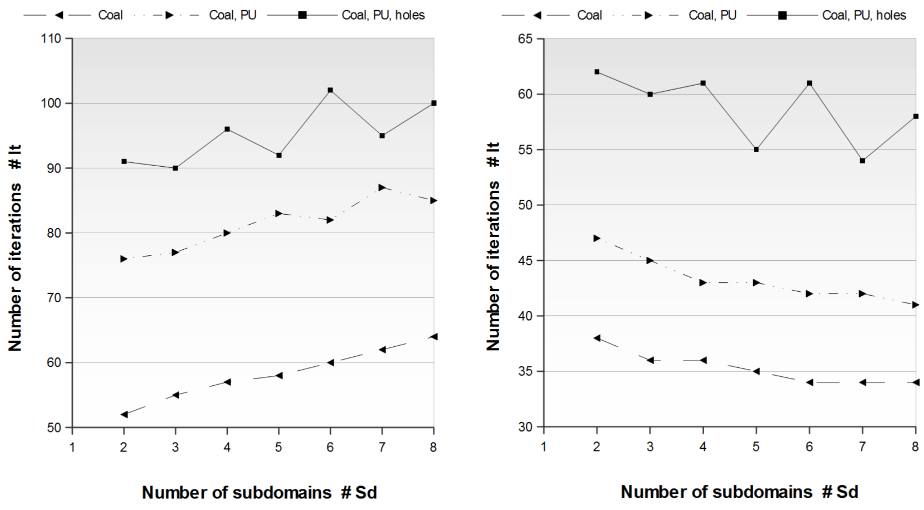


Fig. 3. Influence of heterogeneity illustrated on three compositions of the geocomposite sample: coal only (a fully homogeneous case, dashed line), coal and polyurethan resin (dash-dotted line), coal, polyurethan resin and holes / empty space (the most heterogeneous case, solid line). The numbers of iterations for GEM solvers: DD (left) and DD+CG (right).

The solution with a coarse grid (right) boasts better behaviour than the solution without a coarse grid (left) not only for the lower number of iterations, but also because its iterations can even decrease with increasing number of subdomains, whereas without a coarse grid their number increases. For both solver alternatives one observes a negative impact of more heterogeneous material compositions in terms of increasing number of iterations. Moreover the curves loose their smoothness.

5 Conclusions

In this paper, we dealt with the micro FEM analysis and upscaling techniques related to computer tomography and its application in geotechnics. We investigated solvers for numerical upscaling based on a formulation of boundary value problems either with pure Dirichlet or with pure Neumann conditions, with special attention to efficient parallel processing. Moreover, we utilized the Dirichlet case for a comparison study of two parallel FEM solvers: one in-house, written from scratch in Fortran and MPI, the other built up from components provided by the Trilinos framework. Our case study so far recognized a better performance of the former approach, on the other hand the advantage of the latter was in the fractional development time. Better performance of Trilinos solvers might be probably obtained by optimized combination of procedure and parameter selection, but this optimization is very difficult to accomplish only on the basis of generally available Trilinos documentation.

Acknowledgement. This work was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

References

1. Arbenz, P., van Lenthe, G.H., Mennel, U., Müller, R., Sala, M.: A scalable multi-level preconditioner for matrix-free μ-finite element analysis of human bone structures. Int. J. Numer. Methods Eng. 73(7), 927–947 (2008)
2. Blaheta, R.: A multilevel method with overcorrection by aggregation for solving discrete elliptic problems. J. Comput. Appl. Math. 24, 227–239 (1988)
3. Blaheta, R.: Displacement Decomposition - Incomplete Factorization Preconditioning Techniques for Linear Elasticity Problems. Numerical Linear Algebra with Applications 1, 107–128 (1994)
4. Blaheta, R.: GPCG - generalized preconditioned CG method and its use with non-linear and non-symmetric displacement decomposition preconditioners. Numerical Linear Algebra with Applications 9, 527–550 (2002)
5. Blaheta, R.: Space Decomposition Preconditioners and Parallel Solvers. In: Feistauer, M., Dolejíší, V., Knobloch, P., Najzar, K. (eds.) ENUMATH 2003, pp. 20–38. Springer, Berlin (2004)

6. Blaheta, R., Jakl, O., Kohut, R., Starý, J.: GEM – A Platform for Advanced Mathematical Geosimulations. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 266–275. Springer, Heidelberg (2010)
7. Blaheta, R., Sokol, V.: Multilevel Solvers with Aggregations for Voxel Based Analysis of Geomaterials. In: Lirkov, I., Margenov, S., Wasniewski, J. (eds.) LSSC 2011. LNCS, vol. 7116, pp. 489–497. Springer, Heidelberg (2012)
8. Blaheta, R., Hrtus, R., Kohut, R., Axelsson, O., Jakl, O.: Material Parameter Identification with Parallel Processing and Geo-applications. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 366–375. Springer, Heidelberg (2012)
9. Blaheta, R., Kohut, R., Kolcun, A., Souček, K., Staš, L., Vavro, L.: Digital image based numerical micromechanics of geocomposites with application to chemical grouting (submitted)
10. Bochev, P., Lehoucq, R.B.: On the Finite Element Solution of the Pure Neumann Problem. SIAM Review 47, 50–66 (2005)
11. Gee, M.W., Seifert, C.M., Hu, J.J., Tuminaro, R.S., Sala, M.G.: ML 5.0 Smoothed Aggregation User's Guide. Sandia Report SAND2006-2649 (2006)
12. Nečas, J., Hlaváček, I.: Mathematical theory of elastic and elasto-plastic bodies: an introduction. Elsevier, Amsterdam (1981)
13. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
14. Vaněk, P., Mandel, J., Brezina, M.: Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems. Computing 56, 179–196 (1996)
15. Zohdi, T.J., Wriggers, P.: An Introduction to Computational Micromechanics. Springer, Berlin (2005, 2008)
16. The Trilinos Project Home Page. Sandia National Laboratories (2012), http://trilinos.sandia.gov/
17. IFPACK Home. IFPACK Object-Oriented Algebraic Preconditioner Package, Sandia National Laboratories (2012), http://trilinos.sandia.gov/packages/ifpack/
18. ML Home. ML: Multi Level Preconditioning Package, Sandia National Laboratories (2012), http://trilinos.sandia.gov/packages/ml/

Parallel Numerical Algorithms for Simulation of Multidimensional Ill-Posed Nonlinear Diffusion-Advection-Reaction Models

Raimondas Čiegis and Andrej Bugajev

Vilnius Gediminas Technical University
Saulėtekis av. 11, LT10223 Vilnius, Lithuania
{rc,andrej.bugajev}@vgtu.lt

Abstract. This paper presents finite difference approximations of two dimensional in space mathematical model of a bacterial self-organization. Due to the chemotaxis process some instability of the solution can be developed in the system, in this paper we show that such instability can be connected to the ill-posed problem defined by the backward in time diffusion process. The ADI and splitting type methods are used to construct robust parallel numerical approximations. Domain decomposition method is applied to distribute subtasks among processors. The scalability analysis of the parallel algorithm is done and results of computational experiments are presented.

Keywords: diffusion-advection-reaction models, splitting schemes, parallel algorithms, ill-posed.

1 Introduction

Many mathematical problems of biological systems are described by non-stationary and non-linear diffusion–advection-reaction equations. The dynamics of their solutions can be very complicated, the interaction of different physical processes can lead to development of spatial and temporal patterns and instabilities [8]. We consider a two-dimensional mathematical model for simulation of important self-organization processes in biochemistry applications (see, e.g. [1,2,9,11]):

$$\frac{\partial u}{\partial t} = \sum_{j=1}^{2} \left[D \frac{\partial^2 u}{\partial x_j^2} - \frac{\partial}{\partial x_j} \left(\frac{\chi u}{(1+\alpha v)^2} \frac{\partial v}{\partial x_j} \right) \right] + \gamma r u(1-u), \tag{1}$$

$$\frac{\partial v}{\partial t} = \sum_{j=1}^{2} \frac{\partial^2 v}{\partial x_j^2} + \gamma \Big(\frac{u^p}{1+\beta u^p} - v \Big), \quad X \in (0,1)\times(0,1), \ \ t > 0,$$

where u is the dimensionless cell density, v is the dimensionless chemoattractant concentration, D defines the constant cell diffusion, χ is the chemotactic coefficient, r denotes the growth rate, α defines the receptor sensitivity, β stand for saturating of the signal production, γ defines the ratio of the spatial and temporal scales, and $p \geq 1$. Boundary conditions can vary according to the biological system, zero-flux and periodical conditions are popular and natural choices. Here, we use the periodicity conditions.

In the mentioned above papers and books, on the basis of theoretical analysis and computational simulations it is demonstrated that solutions of chemotaxis mathematical model can develop complicated spatial-time patterns, which are observed also in real bioluminescence images. For example, Painter and Hillen have shown in [9] that the long-time dynamics of the solutions fall into four main classes: a) homogeneous steady state solutions, b) stationary spatial solutions, c) stationary-temporal periodic solutions, d) spatio-temporal irregular solutions.

The dynamics of nonlinear systems can be investigated by various mathematical techniques, an extensive review with many examples and applications can be found e.g. in [8,9].

It is well-known that the theory of exponential attractors explains important properties of dynamical systems in infinite-dimensional spaces. Exponential attractors have a very strong stability in approximation. This gives a possibility to show a global reliability of numerical computations. A review on application of such results for problem (1) is given in [11].

For any mathematical model it is important to investigate the sensitivity of the solution with respect to initial data, i.e. to analyze the well-posedness of the mathematical model. It is well-known that many technological and physical processes can lead to development of spatial and temporal instabilities in solutions.

It is shown in [3] that chemotaxis-driven instability can be correlated to the ill-posed problem defined by the backward in time diffusion process. In general, the well-posedness of the model is connected to the important property of chemotaxis process, that the velocity of advection of u depends on the gradient of the chemo-attractor. Thus, if the cell density depends monotonically on the chemo-attractor, then such a dependence leads to anti-diffusion changes of the cell density. For simplicity of analysis, let us assume that parameter γ is sufficiently large, but $\gamma r \sim \mathcal{O}(1)$, then due to the fast relaxation we get that

$$\frac{\partial u}{\partial t} = \sum_{j=0}^{2} D \left[\frac{\partial^2 u}{\partial x_j^2} - \frac{\partial}{\partial x_j} \left(\frac{\chi p u^p}{(1+\alpha v)^2 (1+\beta u^p)^2} \frac{\partial u}{\partial x_j} \right) \right] + \gamma r u (1-u). \tag{2}$$

For specific values of parameters, equation (2) describes a 2D backward in time parabolic problem and therefore general mathematical model (1) can be ill-posed.

In order to investigate the long-time dynamics of solutions of mathematical models similar to (1), the solution should be computed until $t = 10^4$ or even $t = 10^5$ in realistic problems. The number of space mesh points also can be close 10^6 for 2D problems and 10^8 points for 3D problems (see, [6,9] and references contained therein). Thus the usage of parallel numerical algorithms is an essential step in simulation of real-world problems.

We consider the main steps in construction, analysis and implementation of efficient parallel numerical algorithms for solution of multidimensional nonlinear diffusion-advection-reaction type problems. The two-stage approximation strategy is a standard approach for development of sequential numerical schemes (see, [3,5,6] and references contained therein). First, a Method of Lines (MOL) approach is used to discretize operators in space on a uniform mesh. Second, the obtained system of ODEs is discretized by using the combination of ADI splitting and IMEX (a mixture of implicit and explicit) methods. The discrete advection operator is approximated by the explicit scheme and diffusion and reaction operators are solved by using the full-approximation ADI splitting scheme. The full-approximation is required if we are interested in using this algorithm to march towards a steady state. The 2D diffusion operators are split in space coordinates, thus the linear algebra part of the finite difference scheme is reduced to solving tridiagonal systems of equations. A more simple Locally One Dimensional (LOD) splitting of diffusion operators requires formulation of artificial boundary conditions, since the intermediate vectors at fractional time steps are not consistent approximations of the exact solution [6]. The second drawback of LOD approximations is that the full-approximation property is not valid for such schemes.

We also mention the Rosenbrock AMF methods which are popular tools for approximation of (1) type problems [5,6]. But the computational templates of parallel subproblems obtained after the factorization of Jacobian matrix are the same as for the ADI splitting algorithms investigated in this paper.

Parallelization of the ADI splitting algorithm is done by using the domain decomposition method. Here we are interested to investigate the maximal efficiency of the parallel splitting algorithm, therefore the reaction step is also split from the diffusion operator by using a simple LOD splitting technique. A modification of the Wang's factorization algorithm is constructed to solve systems with tridiagonal matrix obtained after approximation the diffusion operator with periodical boundary conditions. The scalability analysis of the parallel algorithm is based on [7,10].

The rest of the paper is organized as follows. In Section 2 we give details of the numerical techniques which are used for the construction of finite-difference schemes. Parallel ADI type algorithm is presented in Section 3. The modification of Wang's algorithm is presented for solution of linear systems of equations with tridiagonal matrices and periodic boundary conditions. The scalability of the parallel algorithm is investigated and the optimal number of processors is defined by using the complexity estimates of the parallel algorithm. Results of numerical experiments are presented in Section 4.

2 Numerical Technique

In this section we present numerical techniques which are used to approximate solutions of system (1). In general we follow papers [5,6].

2.1 The Method of Lines: Discretization in Space

At the first step we approximate the spatial derivatives in the PDEs (1) by applying robust and accurate approximations targeted for special physical processes described by differential equations. We cover the computational domain $\omega := (0,1)^2$ by an equi-spaced grid ω_h with J computational cells in each direction, the width of the cell is denoted by $h := 1/J$. On the semidiscrete domain $\omega_h \times [0,T]$ we define functions $U_{ij}(t) = U(x_{1i}, x_{2j}, t)$, $V_{ij}(t) = V(x_{1i}, x_{2j}, t)$, $ij = 0, \dots, N-1$, here U_{ij}, V_{ij} approximate exact solutions $u(x_1, x_2, t)$, $v(x_1, x_2, t)$ on the discrete grid ω_h at time moment t.

Using the finite volume approach, we approximate the diffusion and reaction terms by the following finite difference equations:

$$\begin{aligned}
&A_{D_k}(U) = D\partial_{\bar{x}_k}\partial_{x_k}U_{ij}, \quad k=1,2, \quad A_R U = \gamma r U_{ij}(1-U_{ij}), \quad (x_{1i}, x_{2j}) \in \omega_h,\\
&\widetilde{A}_{D_1}(V) = \partial_{\bar{x}_1}\partial_{x_1}V_{ij} - \gamma V_{ij}, \quad \widetilde{A}_{D_2}(V) = \partial_{\bar{x}_2}\partial_{x_2}V_{ij}\\
&\widetilde{A}_R U = \gamma \frac{U_{ij}^p}{1+\beta U_{ij}^p}, \quad (x_{1i}, x_{2j}) \in \omega_h.
\end{aligned}$$

For chemotaxis term we consider the upwind-based discrete 1D fluxes, e.g., for x_2 coordinate [3,6]:

$$\begin{aligned}
&F_T(U,a,2) = a_{i,j+\frac{1}{2}}\big[U_{ij} + \psi(\theta_{ij})\big(U_{i,j+1} - U_{ij}\big)\big], \quad a_{i,j+\frac{1}{2}} \geq 0,\\
&F_T(U,a,2,j+\frac{1}{2}) = a_{i,j+\frac{1}{2}}\big[U_{i,j+1} + \psi(1/\theta_{i,j+1})\big(U_{ij} - U_{i,j+1}\big)\big], \; a_{i,j+\frac{1}{2}} < 0,
\end{aligned}$$

with the Koren limiter function. Here the discrete spatial approximation of the velocity is computed by

$$a_{i,j+\frac{1}{2}}(t) = -\frac{\chi}{\big(1+\alpha(V_{ij}+V_{i,j+1})/2\big)^2}\,\partial_{x_2}V_{ij}.$$

We denote the discrete advection operator as

$$\begin{aligned}
A_T(U,V) = &\frac{1}{h}\big(F_T(U,a,1,i+1/2) - F_T(U,a,1,i-1/2)\big)\\
&+ \frac{1}{h}\big(F_T(U,a,2,j+1/2) - F_T(U,a,2,j-1/2)\big).
\end{aligned}$$

Then we get a nonlinear ODE system for the evolution of semi-discrete solutions

$$\begin{aligned}
&\frac{dU}{dt} = A_T(U,V) + A_{D_1}(U) + A_{D_2}(U) + A_R U, \qquad (3)\\
&\frac{dV}{dt} = \widetilde{A}_{D_1}(V) + \widetilde{A}_{D_2}(V) + \widetilde{A}_R(U),
\end{aligned}$$

here discrete periodical boundary conditions are included into the definition of discrete diffusion and advection operators.

2.2 Time Stepping

In order to develop efficient solvers in time for the obtained large ODE systems we take into account the different nature of the discrete operators defining the advection and the diffusion-reaction processes.

Let ω_τ be a uniform temporal grid

$$\omega_\tau = \{t^n : t^n = n\tau,\, n = 0, \dots, M,\, M\tau = T_f\},$$

here τ is the time step.

Given approximations U_j^n, V_j^n at time t^n, we compute solutions at $t^{n+1} = t^n + \tau$ by the following IMEX ADI type scheme: A_T is a non-stiff term suitable for explicit time integration

$$\widetilde{U}_{ij}^n = U_{ij}^n + \tau A_T(U^n, V^n), \quad (x_{1i}, x_{2j}) \in \omega_h, \tag{4}$$

A_{D_k}, $\widetilde{A}_{D_k}$, A_R, $\widetilde{A}_R$ are stiff terms requiring an implicit ADI treatment

$$\begin{aligned} U_{ij}^{n+\frac{1}{2}} &= \widetilde{U}_{ij}^n + \tau\big(A_{D_1}(U^{n+\frac{1}{2}}) + A_{D_2}(U^n) + A_R(U^{n+\frac{1}{2}})\big), \quad (5)\\ U_{ij}^{n+1} &= U_{ij}^{n+\frac{1}{2}} + \tau\big(A_{D_2}(U^{n+1}) - A_{D_2}(U^n)\big),\\ V_{ij}^{n+\frac{1}{2}} &= V_{ij}^n + \tau\big(\widetilde{A}_{D_1}(V^{n+\frac{1}{2}}) + \widetilde{A}_{D_2}(V^n) + \widetilde{A}_R(U^{n+1})\big), \quad (6)\\ V_{ij}^{n+1} &= V_{ij}^{n+\frac{1}{2}} + \tau\big(\widetilde{A}_{D_2}(V^{n+1}) - \widetilde{A}_{D_2}(V^n)\big). \end{aligned}$$

Such a mixture of implicit and explicit methods gives efficient solvers for each sub-step of the algorithm and due to the full approximation the stationary solution exactly satisfies the finite difference scheme.

The accuracy of the approximation in time can be increased by applying the symmetrical version of this scheme, for details see [6].

3 Parallel Algorithm

The main aim of this paper is to investigate the efficiency and scalability of parallel numerical algorithms for approximation of problem (1). Thus in order to find a bound on the parallel efficiency of split type approximations and to measure the influence of different parts of the full mathematical model, we have increased the amount of computations which can be done in parallel by modifying the basic full-approximation scheme (4)–(6). Using the operator splitting method, we further split operator $A_{D_1} + A_R$ by separating diffusion and nonlinear reaction terms. Note that for the obtained new discrete scheme the requirement of full approximation of the stationary solution is not satisfied.

$${U_{ij}}^{n+\frac{1}{4}} = U_{ij}^n + \tau A_T(U^n, V^n), \quad (x_{1i}, x_{2j}) \in \omega_h, \tag{7}$$

$${U_{ij}}^{n+\frac{m+s}{4m}} = {U_{ij}}^{n+\frac{m+s-1}{4m}} + \frac{\tau}{m}\gamma r U_{ij}^{n+\frac{m+s-1}{4m}}\big(1 - U_{ij}^{n+\frac{m+s}{4m}}\big), \quad s = 1, \dots, m, \tag{8}$$

$$\begin{aligned}
U_{ij}^{n+\frac{3}{4}} &= U_{ij}^{n+\frac{1}{2}} + \tau\big(A_{D_1}(U^{n+\frac{3}{4}}) + A_{D_2}(U^{n+\frac{1}{2}})\big), \qquad (9)\\
U_{ij}^{n+1} &= U_{ij}^{n+\frac{3}{4}} + \tau\big(A_{D_2}(U^{n+1}) - A_{D_2}(U^{n+\frac{1}{2}})\big),\\
V_{ij}^{n+\frac{1}{2}} &= V_{ij}^{n} + \tau\big(\widetilde{A}_{D_1}(V^{n+\frac{1}{2}}) + \widetilde{A}_{D_2}(V^{n}) + \widetilde{A}_{R}(U^{n+1})\big), \qquad (10)\\
V_{ij}^{n+1} &= V_{ij}^{n+\frac{1}{2}} + \tau\big(\widetilde{A}_{D_2}(V^{n+1}) - \widetilde{A}_{D_2}(V^{n})\big).
\end{aligned}$$

The reaction problem (8) is split into $m \geq 1$ sub-steps in order to resolve a stiff reaction problem accurately. We note that such an additional split step does not change the asymptotical efficiency of the parallel algorithm, as it is shown by results of the scalability analysis.

A parallel version of the proposed algorithm is developed by using the domain decomposition method. The two-dimensional decomposition of the space mesh ω_h into $P = P_1 \times P_2$ subdomains is applied. The advection and reaction steps are resolved by the same sequential algorithms, only data exchange among processors is implemented when the stencil of the finite difference scheme leads to overlapping of local sub-meshes of different processors.

The standard factorization algorithm for solving linear systems of equations with tridiagonal matrices is fully sequential and should be changed to some parallel solver. Here we use the well-known Wang's algorithm, but the modification is done in order to adapt this algorithm for periodic boundary conditions.

3.1 Parallel Factorization Algorithm

Let denote by U_j the 1D vector of unknowns, corresponding to 2D vector $\{U_{ij}\}$, where one coordinate, e.g. $1 \leq j \leq J$, is fixed:

$$U_j = \big(U_{1j}, U_{2j}, \dots, U_{M_1,j}\big),$$

where $M_1 = J/P_1$ is the number of unknowns in x_1 direction belonging to each processor. Thus we solve M_1 one dimensional systems with tridiagonal matrix, where periodic boundary conditions are taken into account in the first and the last equations:

$$\begin{cases} -a_1 U_J + c_1 U_1 - b_1 U_2 = f_1, \\ -a_j U_{j-1} + c_j U_j - b_j U_{j+1} = f_j, \quad j = 2, \dots, J-1, \\ -a_J U_{J-1} + c_J U_J - b_J U_1 = f_J. \end{cases} \qquad (11)$$

Let assume that P_2 processes are used to solve this system. We partition the system into P_2 blocks and denote the number of unknowns in each block by $M_2 = J/P_2$. The p-th process updates a block of equations for $j = s_p, \dots, f_p$, where $s_p = (p-1)M_2 + 1$, $f_p = pM_2$.

1. First, using the modified forward factorization algorithm, vector U_j is expressed in the following form

$$U_j = \alpha_j U_{j+1} + \gamma_j U_{s_p-1} + \beta_j, \;\; j = s_p, \dots, f_p, \; s_1 - 1 := J, \; f_P + 1 := 1, \qquad (12)$$

where

$$\alpha_{s_p} = \frac{b_{s_p}}{c_{s_p}}, \quad \gamma_{s_p} = \frac{a_{s_p}}{c_{s_p}}, \quad \beta_{s_p} = \frac{F_{s_p}}{c_{s_p}},$$

$$\alpha_j = \frac{b_j}{c_j - a_j\alpha_{j-1}}, \ \gamma_j = \frac{a_j\gamma_{j-1}}{c_j - a_j\alpha_{j-1}}, \ \beta_j = \frac{F_j + a_j\beta_{j-1}}{c_j - a_j\alpha_{j-1}}.$$

All processes implement the first step in parallel and the complexity of it is $8M_1M_2$ flops. No data communication among processors is required.

2. Second, the block matrix is diagonalized, except the first and last columns of the block, i.e. vector U_j is expressed in the form

$$U_j = \alpha_j U_{f_p} + \gamma_j U_{s_p-1} + \beta_j, \quad j = s_p, \dots, f_p, \tag{13}$$

where factorization coefficients are recomputed as:

$$\alpha_j = \alpha_{j+1}\alpha_j, \ \gamma_j = \gamma_j + \gamma_{j+1}\alpha_j, \ \beta_j = \beta_j + \beta_{j+1}\alpha_j, \ \ j = f_p - 1, \dots, s_p.$$

The computations are implemented from right to left. All processes run the second step in parallel and the complexity of it is $5M_1M_2$ flops. No data communication among processors is required.

3. In the third step, all processes $p > 1$, excluding the first one, send their first row of the matrix to its neighbour $(p-1)$, then all processes $p < P_2$, excluding the last one, modify the last row:

$$U_{f_p} = \alpha_{f_p} U_{f_{p+1}} + \gamma_{j_p} U_{f_{p-1}} + \beta_{f_p}, \quad p = 1, \dots, P_2 - 1. \tag{14}$$

where

$$\alpha_{f_p} = \frac{\alpha_{f_{p+1}}\alpha_{f_p}}{1 - \gamma_{f_{p+1}}\alpha_{f_p}}, \quad \gamma_{f_p} = \frac{\gamma_{f_p}}{1 - \gamma_{f_{p+1}}\alpha_{f_p}}, \quad \beta_{f_p} = \frac{\beta_{f_p} + \beta_{f_{p+1}}\alpha_{f_p}}{1 - \gamma_{f_{p+1}}\alpha_{f_p}}.$$

All processes implement the third step in parallel and the complexity of it is $8M_1$ flops. They exchange between neighbours $3M_1$ elements. The complexity of data communication is $2\big(\alpha + 3\beta M_1\big)$, where α is the message startup time, β is the time required to send one element of data.

Now, taking the last equation of each local sub-system and adding to the new system the first equation of the first process we get a $(P_2+1) \times (P_2+1)$ tridiagonal system of linear vector equations

$$\begin{cases} U_1 = \alpha_1 U_{f_1} + \gamma_1 U_J + \beta_1, \\ U_{f_p} = \alpha_{f_p} U_{f_{p+1}} + \gamma_{f_p} U_{f_{p-1}} + \beta_{f_p}, \quad p = 1, \dots, P_2 - 1. \\ U_J = \alpha_J U_1 + \gamma_J U_{f_{p-1}} + \beta_J. \end{cases} \tag{15}$$

The obtained system of linear equations again depends on periodic boundary conditions. It can be solved sequentially by one process and results distributed to the remaining processes. We propose to use a two-side factorization algorithm: all processes are implementing the factorization algorithm, but local subtasks are computed sequentially, the required data is exchanged among neighbour processes only. Thus we avoid any global data reduction and distribution operations. A small scale parallelization of the algorithm is still preserved since the computations are divided among two groups of processes.

4. In the fourth step, processes are divided into two groups and sequentially sweep from left to the right transformations of the last row of local system. E.g., process $p > 1$ of the left group gets $(p-1)$-th neighbour's last equation, then modifies its own last equation and sends new coefficients to $(p+1)$-th neighbour. The last equation of the local sub-system is transformed to the form:

$$\begin{aligned} U_{f_p} &= \alpha_{f_p} U_{f_{p+1}} + \gamma_{f_p} U_J + \beta_{f_p}, \quad p = 2, \dots, P_2/2, \\ U_{f_p} &= \gamma_{f_p} U_{f_{p-1}} + \alpha_{f_p} U_J + \beta_{f_p}, \quad p = P_2/2+1, \dots, P_2-1, \end{aligned} \tag{16}$$

where, e.g. for $p \le P_2/2$:

$$\alpha_{f_p} = \frac{\alpha_{f_p}}{1 - \gamma_{f_p}\alpha_{f_{p-1}}}, \quad \gamma_{f_p} = \frac{\gamma_{f_p}\gamma_{f_{p-1}}}{1 - \gamma_{f_p}\alpha_{f_{p-1}}}, \quad \beta_{f_p} = \frac{\beta_{f_p} + \beta_{f_{p-1}}\gamma_{f_p}}{1 - \gamma_{f_p}\alpha_{f_{p-1}}}.$$

The computational complexity of the fourth step is $4P_2M_1$ flops and the complexity of data communication is $\big(\alpha + 3\beta M_1\big)P_2$.

5. In the fifth step, both groups exchange information, and then sequentially transform the last raw equations into the form:

$$\begin{aligned} U_{f_p} &= \gamma_{f_p} U_J + \beta_{f_p}, \quad p = P_2/2, \dots, 1, \quad U_1 = \gamma_1 U_J + \beta_1, \\ U_{f_p} &= \alpha_{f_p} U_J + \beta_{f_p}, \quad p = P_2/2+1, \dots, P_2-1, \\ U_J &= \alpha_J U_1 + \beta_J, \end{aligned} \tag{17}$$

where, e.g. for $p \le P_2/2$:

$$\gamma_{f_p} = \gamma_{f_p} + \alpha_{f_p}\gamma_{f_{p+1}}, \quad \beta_{f_p} = \beta_{f_p} + \beta_{f_{p+1}}\alpha_{f_p}.$$

The computational complexity of the fifth step is $2P_2M_1$ flops and the complexity of data communication is $\big(\alpha + 2\beta M_1\big)P_2$.

6. In the sixth step, the first and last processes exchange information on the first and last equations of the system, compute vectors U_1 and U_J and then all processes of the left and right groups compute sequentially the remaining vectors U_{f_p}. The computational complexity of the sixth step is P_2M_1 flops and the complexity of data communication is $\big(\alpha + \beta M_1\big)P_2$.

7. In the last step, all processes in parallel compute vectors U_j, $j = 1, \dots, J$. The computational complexity of this step is $4M_1M_2$ flops.

We apply the same algorithm for solution of linear systems of the ADI scheme resolving 1D diffusion process A_{D_1}. Thus the total complexity of the parallel factorization algorithm to solve a 2D diffusion problem with periodical boundary conditions is equal to

$$T_{D,P} = 34\frac{J^2}{P} + 7\Big(\frac{P_1}{P_2} + \frac{P_2}{P_1}\Big)J + 8\Big(\frac{1}{P_1} + \frac{1}{P_2}\Big)J \\ + \alpha\big(2 + 3(P_1 + P_2)\big) + 3\beta\Big(1 + 2\Big(\frac{P_1}{P_2} + \frac{P_2}{P_1}\Big)\Big)J. \tag{18}$$

Note that the complexity of the sequential factorization algorithm is $28J^2$ floating point operations, thus the additional costs of the parallel algorithm mainly depend on data communication costs.

In the case of the Dirichlet or Neumann boundary conditions the complexity of the sequential factorization algorithm is $16J^2$ operations, and the complexity of the parallel algorithm implemented on one processor is $T_{D,1} = 34J^2$ operations, thus computational cost is increased more than twice.

3.2 Scalability Analysis

In this section we give the main scalability analysis results.

Nonlinear Reaction Step. Nonlinear reaction discrete equations (8) are solved by fully data parallel algorithm and no communication costs are required. The complexity of this part of operator splitting algorithm is

$$T_{R,P} = C_R\frac{J^2}{P}, \tag{19}$$

constant C_R depends on the number of splitting steps m, required to solve the stiff reaction equations in an accurate way.

Transport Step. The transport step (7) parallel algorithm is data parallel and coincides with the sequential explicit algorithm. Additional communication costs depend on the stencil of the finite difference scheme. For the approximation of the chemotaxis advection process, values of function U on the cross stencil with one point radius are used, and values of function V on the cross stencil with two point radius are required. Thus the total complexity of the advection step is given by

$$T_{A,P} = C_T\frac{J^2}{P} + 4\alpha + 12\beta\Big(\frac{J}{P_1} + \frac{J}{P_2}\Big). \tag{20}$$

Adding all estimates (18)–(20), we get that the complexity of the parallel ADI algorithm (4)–(10) is given by

$$T_P = (C_R + C_T + 68)\frac{J^2}{P} + 14\Big(\frac{P_2}{P_1} + \frac{P_1}{P_2}\Big)J + 16\Big(\frac{J}{P_1} + \frac{J}{P_2}\Big) \\ + 2\alpha\big(4 + 3(P_1 + P_2)\big) + 6\beta\Big[1 + 2\Big(\frac{P_1}{P_2} + \frac{P_2}{P_1} + \frac{1}{P_1} + \frac{1}{P_2}\Big)\Big]J.$$

Complexity of the sequential ADI algorithm (4)–(10) is given by

$$T_0 = (C_R + C_T + 56)J^2.$$

It follows from these estimates that the efficiency of the parallel algorithm is limited by the solution of reduced tridiagonal systems (15). Let us assume that $P_1 = P_2 = \sqrt{P}$, then the asymptotical optimal number of processors is determined from the equation, which defines the equidistribution of computational and data communication costs (the worst term of communication cost is of order cJ for $P_1 = P_2 = \sqrt{P}$):

$$\frac{J^2}{P} = cJ \quad \Longrightarrow \quad P = \mathcal{O}(J).$$

In future work, we plan to solve diffusion sub-problems by using parallel iterative solvers based on AMG preconditioners, see e.g. [4].

4 Computational Experiments

Computations were performed on Vilkas cluster of computers at Vilnius Gediminas Technical University, consisting of nodes with Intel®Core^{TM} processor i7-860 @ 2.80 GHz and 4 GB DDR3-1600 RAM. Each of the four cores can complete up to four full instructions simultaneously. Results of computational experiments are given.

Table 1. Scalability analysis of the parallel algorithm. The speed-up S_p and efficiency E_p coefficients for two problems of dimension 600×600 and 1200×1200.

	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
$S_p(600)$	1.39	3.22	5.96	10.1	12.5
$E_p(600)$	0.70	0.81	0.75	0.63	0.39
$S_p(1200)$	1.32	2.60	5.58	12.0	20.9
$E_p(1200)$	0.66	0.65	0.70	0.75	0.65

It follows from the presented results, that parallel ADI type algorithms are efficient in solving real-world biochemistry applications. The modified Wang's algorithms efficiently solves systems of linear equations with tridiagonal matrices and periodic boundary conditions. The conclusions of the scalability analysis about a linear scalability of the parallel algorithm are confirmed in computational experiments. In the future work we plan to consider an alternative 1D domain decomposition, when the data is transposed after the solution of the tridiagonal systems of one direction, such that tridiagonal systems are always solved sequentially, as is done in 2D FFT codes. We also plan to compare the direct ADI solvers with parallel solvers based on Krylov type iterative algorithms.

References

1. Baronas, R., Šimkus, R.: Modelling the bacterial self-organization in circular container along the contact line as detected by bioluminescence imaging. Nonl. Anal. Model. Contr. 16(2), 270–282 (2011)
2. Brenner, M.P., Levitov, L.S., Budrene, E.O.: Physical mechanisms for chemotactic pattern formaton by bacteria. Biophys. J. 74, 1677–1693 (1998)
3. Čiegis, R., Bugajev, A.: Numerical approximation of one model of the bacterial self-organization. Nonl. Anal. Model. Contr. 17(3), 253–270 (2012)
4. Evans, M., Žurič, Z., Basara, B., Frolov, S.: A novel SIMPLE-based pressure-enthalpy coupling scheme for engine flow problems. Math. Model. Anal. 17(1), 1–20 (2012)
5. Gerisch, A., Chaplan, M.A.: Robust numerical methods for taxis-diffusion-reaction systems: applications to biomedical problems. Math. Comput. Model. 43, 49–75 (2006)
6. Hundsdorfer, W., Verwer, J.G.: Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations. Springer Series in Computational Mathematics, vol. 33. Springer, Heidelberg (2003)
7. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Algorithms. Benjamin/Cummings, Redwood City (1994)
8. Murray, J.D.: Mathematical Biology I: An Introduction. Springer, Berlin (2002)
9. Painter, K.J., Hillen, T.: Spatio-temporal chaos in a chemotaxis model. Physica D 240, 363–375 (2011)
10. Starikovičius, V., Čiegis, R., Iliev, O.: A parallel solver for optimization of oil filters. Math. Model. Anal. 16(2), 326–342 (2011)
11. Yagi, A.: Abstract Parabolic Evolution Equations and their Applications. Springer Monographs in Mathematics. Springer, Heidelberg (2010)

Efficient Setup of Aggregation AMG for CFD on GPUs

Maximilian Emans[1] and Manfred Liebmann[2]

[1] Johann Radon Institute for Computational and Applied Mathematics and IMCC GmbH, both Linz, Austria
maximilian.emans@ricam.oeaw.ac.at

[2] Institute for Mathematics and Scientific Computing, University of Graz, Graz, Austria
manfred.liebmann@uni-graz.at

Abstract. We explore a GPU implementation of a Krylov-accelerated algebraic multigrid (AMG) algorithm with flexible preconditioning. We demonstrate by means of two benchmarks from an industrial computational fluid dynamics (CFD) application that the acceleration with multiple graphics processing units (GPUs) speeds up the solution phase by a factor of up to 13. In order to achieve good performance for the whole AMG algorithm, we propose for the setup a substitution of the double-pairwise aggregation by a simpler aggregation scheme skipping the calculation of temporary grids and operators. The version with the revised setup reduces the total computing time on multiple GPUs by further 30% compared to the GPU implementation with the double-pairwise aggregation. We observe that the GPU implementation of the entire Krylov-accelerated AMG runs up to four times faster than the fastest central processing unit (CPU) implementation.

1 Introduction

When numerical simulations of physical phenomena are considered, the notion of most engineers and scientists outside the high-performance computing community is that computers similar to the well-known desktop personal computers are used. Eventually, some of these computers might be linked by some kind of network interconnect to local networks or to clusters. In fact, the hardware architecture the vast majority of simulations in science and engineering runs on, is the same as that of a modern workstation, i.e. one or several many-core CPUs that are classified as multiple instruction – multiple data (MIMD) systems according to Flynn's taxonomy [1].

Since a couple of years, however, also particular single instruction – multiple data (SIMD) architectures in the form of graphics processing units (GPUs) have started to attract the attention of both, users and developers of numerical simulation software. Recent development in both, hardware design and software tools made it possible to exploit the large computational power of the GPUs for numerical calculations. However, so far not for every single algorithm a dedicated efficient GPU implementation is available. It is not unusual that an algorithm is modified in order to suit the hardware it is implemented on, e.g. the Gauß-Seidel smoother is in practice only used for serial calculations. The "Gauß-Seidel-like" smoother that is typically used for parallel calculations based on a domain decomposition is a hybrid Gauß-Seidel/ω-Jacobi smoother, see Emans [2]. Since for GPU-implementations, the parallelism of the hybrid Gauß-Seidel/ω-Jacobi smoother is still not enough, it is usually substituted by a ω-Jacobi smoother, see e.g. Haase et al. [3], although this algorithm has less favourable smoothing properties. But due to such substitutions or due to algorithmic adjustments in other cases it is possible today that compute-intensive parts of many calculations in science and engineering are executed on GPUs. Nevertheless, it appears that such GPU-accelerated simulations have not yet gained enough momentum to be competitive with the conventional CPU-based ones in relevant applications in science and engineering. Since the reduction of computing times is the main motivation for the use of GPUs, the improvement of the performance of GPU-accelerated simulations is a major research topic. We will report on a fast implementation of an AMG solver for linear systems that has been shown to be efficient for problems in fluid dynamics, but that can also be used in other applications.

It is known that Krylov-accelerated (k-cycle) AMG, described by Notay [4], has a particularly simple and computationally inexpensive setup since it uses double-pairwise aggregation. It is therefore well suited as linear solver within the iterative algorithms used in CFD. Compared to other common methods like Smoothed Aggregation, see Vaněk et al. [5], the inexpensive setup makes this algorithm also attractive for GPU calculations, since it is particularly difficult to implement the setup on the GPU efficiently. The absolute run-time of the setup of the double-pairwise aggregation is small compared to the run-time of the setup of other AMG algorithms. But it is still large in comparison to the time spent in the GPU-accelerated solution phase.

In this contribution we show that the attractive run-times of k-cycle algorithms on GPU-accelerated hardware can be even more reduced if the double-pairwise aggregation is replaced by a simple greedy aggregation algorithm that we refer to as plain aggregation. The latter method has the advantage that it does not require the computation of an intermediate and finally discarded grid level, like the algorithm originally chosen by Notay [4]. With GPU-acceleration, the additional cost due to the slightly worse convergence of the simpler aggregation scheme is outweighed by the dramatically reduced run-time of the setup.

2 Aggregation AMG Algorithms

The AMG algorithm is here applied as a preconditioner to the pressure-correction equation in a finite volume based CFD-code, see Emans [6]. We denote this system as

$$A\boldsymbol{x} = \boldsymbol{b} \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is symmetric and positive definite or semi-definite; $\boldsymbol{b} \in \mathbb{R}^n$ is some right-hand side vector, $\boldsymbol{x} \in \mathbb{R}^n$ is the solution vector, and n is the number of unknowns. We assume that the matrix is given in Compressed Row Storage (CRS) format as e.g. described by Falgout et al. [7]. For parallel computations, a domain decomposition is used to assign a certain set of nodes to each of the parallel processes. The influences of the values associated with nodes assigned to neighbouring processes are handled through a buffer layer, i.e. those values are calculated by the process they are associated with and then exchanged each time they are needed by a neighbouring process.

Any AMG scheme requires the definition of a grid hierarchy with L levels. The matrices representing the problem on grid level l are $A_l \in \mathbb{R}^{n_l \times n_l}$ $(l = 1, \ldots, L)$ with system size n_l where $n_{l+1} < n_l$ holds for $l = 1, \ldots, L-1$ and $A_1 = A$ as well as $n_1 = n$. As it is common practice in algebraic multigrid, the coarse-grid operators are defined recursively (starting with the finest grid) by

$$A_{l+1} = P_l^T A_l P_l \qquad (l = 1, \ldots, L-1). \tag{2}$$

where the prolongation operator P_l has to be determined for each level l while the restriction operator is defined as P_l^T. It is the choice of the coarse-grid selection scheme that determines the elements of P_l and consequently the entire grid hierarchy. The definition of the elements of P_l for all levels and the computation of the operators A_l $(l = 2, \ldots, L)$ are referred to as the setup phase of AMG.

The prolongation operator P_l maps a vector on the coarse grid $\boldsymbol{x}_{l+1}$ to a vector on the fine grid $\boldsymbol{x}_l$:

$$\boldsymbol{x}_l = P_l \boldsymbol{x}_{l+1} \tag{3}$$

The aggregation methods split the number of nodes on the fine grid into a lower number of disjoint sets of nodes, the so-called aggregates. The mapping from the coarse grid to the fine grid is then achieved by simply assigning the coarse-grid value of the aggregate to all the fine-grid nodes belonging to this aggregate. This corresponds to a constant interpolation. The prolongation operator of this scheme has only one non-zero entry in each row with the value one such that the evaluation of eqn. (2) is greatly simplified to an addition of rows of the fine-grid operator. In the following we restrict ourselves to this type of methods.

Double-Pairwise Aggregation. The first step of the double-pairwise aggregation that has been used by Notay [4] is the aggregation of the set of nodes into aggregates of pairs of nodes. For this we use Algorithm 1.

For the pairwise aggregation the output of this algorithm, i.e. the set of aggregates G_i $(i = 1, \ldots, n_{l+1})$, is used to define the prolongation operator P_l. The calculation of the elements of the coarse-grid matrix $A_l = P_l^T A_l P_l$ with eqn. (2) is implemented as the addition of two rows in two steps: First, the rows are extracted from the matrix storage structure in a way that the corresponding elements of the data array are put in a single array and the row pointers in another array of the same size. Second, the column pointers are replaced by the indices of the corresponding coarse-grid aggregates; then both arrays are sorted with respect to the new column pointers where matrix elements with the same column

Algorithm 1 Pairwise aggregation (by Notay [4], simplified version)

```
Input:          Matrix $A = (a_{ij})$ with $n$ rows and columns.
Output:         Number of coarse variables $n_c$ and aggregates $G_i$, $i = 1, \dots, n_c$
                (such that $G_i \cap G_j = \emptyset$ for $i \neq j$).
Initialisation: $U = [1, n]$
                for all $i$: $S_i = \left\{ j \in U \setminus \{i\} \mid a_{ij} < -0.25 \max_{k \in U} |a_{ik}| \right\}$,
                for all $i$: $m_i = | \{j | i \in S_j\} |$,
                $n_c = 0$.
Algorithm:      While $U \neq \emptyset$ do:
                1. select $i \in U$ with minimal $m_i$; $n_c = n_c + 1$.
                2. select $j \in U$ such that $a_{ij} = \min_{k \in U} a_{ik}$
                3. if $j \in S_i$: $G_{n_c} = \{i, j\}$, otherwise $G_{n_c} = \{i\}$
                4. $U = U \setminus G_{n_c}$
                5. for all $k \in G_{n_c}$: $m_l = m_l - 1$ for $l \in S_k$
```

pointers are added. The parallel version of the method restricts the aggregates to nodes belonging to the same parallel domain.

The double-pairwise aggregation, see Notay [4], consists of two passes of the pairwise aggregation Algorithm 1. For the first pass, the input of Algorithm 1 is A_l. Let us denote the prolongation operator resulting from the first pass as P_{l1}. With this, an intermediate coarse-grid operator $A_{l+1/2} = P_{l1}^T A_l P_{l1}$ is obtained in the described manner. For the second pass, this intermediate coarse-grid operator is the input of Algorithm 1. The resulting prolongation operator P_{l2} is used to obtain the coarse-grid operator with $A_{l+1} = P_{l2}^T A_{l+1/2} P_{l2}$. The final prolongation operator that is $P_l = P_{l2} P_{l1}$. Since it contains only the information to which coarse-grid element or aggregate a fine-grid node is assigned, it is sufficient to store it as an array of size n_l carrying the index of the coarse-grid node. The operators $A_{l+1/2}$, P_{l1}, and P_{l2} are discarded after A_{l+1} has been calculated. We refer to the method as within a k-cycle AMG as K-P4.

Plain Aggregation Algorithm. Our plain aggregation algorithm comprises the following steps:

1. **Determine strong connectivity:** Edges of the graph of the matrix A_l for which the relation

$$|a_{ij}| > \beta \cdot \max_{1 \leq j \leq n_l} |a_{ij}| \tag{4}$$

holds, are marked as strong connections. The criterion β depends on the level of the grid hierarchy l and it is defined according to Vaněk et al. [5] as

$$\beta := 0.08 \left(\frac{1}{2} \right)^{l-1} \tag{5}$$

2. **Start-up aggregation:** All nodes are visited in the arbitrary order of their numeration. Once a certain node i is visited in this process, a new aggregate

is built if this node is not yet assigned to another aggregate. Each of the neighbours of node i that is strongly connected to this node and that is not yet assigned to another aggregate is grouped into this aggregate as long as the number of nodes is lower than the maximum allowed aggregate size.
3. **Enlarging the decomposition sets:** Remaining unassigned nodes are joined to aggregates containing any node they are strongly connected to as long as the number of nodes in this aggregate is lower than twice the maximum allowed aggregate size. If there is more than one strongly connected node in different aggregates, the one with the strongest connection determines the aggregate this node is joined with.
4. **Handling the remnants:** Unassigned nodes are grouped into aggregates of a strongly connected neighbourhood. Twice the maximum allowed aggregate size is allowed.

This algorithm follows closely the one proposed by Vaněk et al. [5] with the essential difference that we restrict the number of nodes per aggregate which gives rise to a parameter of this algorithm. In step (3) we allow twice the maximum number of nodes in order to avoid a large number of single-point aggregates. Usually only a few such enlarged aggregates are formed. In parallel, only nodes assigned to the same process are grouped into aggregates.

The aggregates generated in this way are used in a scheme with constant interpolation, i.e. in a way that all fine-grid nodes assigned to a certain aggregate receive the value of the coarse-grid node this aggregate forms on the coarse-grid. The corresponding prolongation operator will have a similarly simple structure as the one of the described pairwise aggregation method. The coarse-grid operator is again obtained by adding the rows of the fine-grid matrix that are associated with the nodes assigned to an aggregate. This is done exactly in the same way as for the pairwise aggregation. The coarsening scheme that is defined by this procedure will be only useful, if the maximum number of nodes per aggregate is kept relatively small. In preliminary experiments we found that if we choose the maximum number of nodes per aggregate to be 6 we obtain an efficient and robust algorithm with good convergence properties. We denote this aggregation scheme in a k-cycle scheme as K-R6.

Smoothed Aggregation. The Smoothed Aggregation algorithm of Vaněk et al. [5] is derived from this algorithm: It refines the aggregation scheme by applying a ω-Jacobi smoothing step (along the paths of the graph of the fine-grid matrix) to the prolongation operator to obtain the final prolongation operator. In this way the quality of the interpolation is improved, but the structure of the operator is now similarly complex as the structure of the operators of classical AMG with the consequence, that the calculation of the coarse-grid operator with eqn. (2) can be no longer simplified in the described way. For the use as Smoothed Aggregation scheme, the number of nodes per aggregate is not limited. The Smoothed Aggregation is then used in a v-cycle scheme and the algorithm is denoted as V-SA.

3 Implementation on GPUs

The setup phase is implemented conventionally on the CPU as it has been described in Emans [2]. The standard CRS format, see e.g. Falgout et al. [7], is used for the matrices. After any matrix has been defined or calculated, it is translated into the Interleaved Compressed Row Storage (ICRS) format on the CPU and then transferred to GPU memory. The definition of this format is found e.g. in Haase et al. [3]. The corresponding algorithm devised in the same publication is used to carry out matrix-vector operations. This applies to the system matrices on all levels. The particularly simple structure of the restriction and prolongation operators of the aggregation algorithms K-P4 and K-R6 gives rise to a simplified version of the ICRS format: Since the value of all non-zero matrix elements is the same, one, it does not make sense to store these values explicitly. Therefore, only the number of elements per row, the displacement and the column index for each element is stored. The fill-in elements (due to the different number of elements per row) are identified by a negative column index and ignored in the matrix-vector multiplication kernel.

For an efficient parallel implementation, the concept of overlapping the data exchange with the internal operations, implemented by means of the asynchronous point-to-point exchange mechanism of MPI, see e.g. Emans [8], needed to be modified: While the internal work is done on the graphics board by the GPU, the CPU manages the data exchange by means of the same asynchronous point-to-point exchange mechanism of MPI, and computes $\boldsymbol{t}_d^{(b)} := E_d \boldsymbol{x}^{(e)}$, see Figure 1.

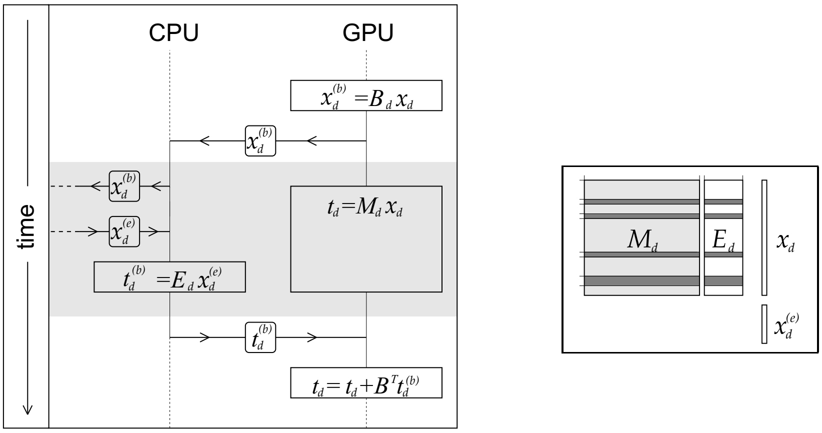


Fig. 1. Flow chart of parallel matrix-vector product on machines with multiple GPUs, the parallel execution on CPU and GPU is marked grey (left), notation (right)

The solver algorithms described above are integrated into the program FIRE 2011, developed and distributed by AVL GmbH, Graz. The solver part of this program is coded in FORTRAN 90 and compiled by the HP FORTRAN compiler, version 11.1. The used MPI library is Platform MPI, version 7.1. The GPU related code is written in CUDA and compiled by the Nvidia compiler version 4.0. The linking between the MPI library and the CUDA part is provided by C-binding.

4 Benchmarks

Problem 1 is a simulation of the unsteady flow of cold air into the cylinder of a gasoline engine of a car. The cylinder has a diameter of 0.08m. During the observed time the piston head is moving from its top position downwards and air at a temperature of 293K flows at a rate of around 1 kg/s into the cylinder. The mass flow at the boundary is prescribed as a function of time according to experimental data for this engine. The volume of the computational domain is initially 0.18l. The mesh consists of around $1.4 \cdot 10^6$ finite volumes 80% of which are hexagonal.

Problem 2 is a simulation of the steady, internal flow through a water-cooling jacket of an engine block. Cooling water, i.e. a 50% water/glycol mixture, flows at a rate of 2.21 kg/s into the geometry through an inlet area of $0.61 \cdot 10^{-3}\text{m}^2$ and leaves it though an outlet area of $0.66 \cdot 10^{-3}\text{m}^2$. The maximum fluid velocity is 4.3m/s. The volume of the cooling jacket is 1.14l. The turbulence is modelled by a k-ζ-f model according to Hanjalic et al. [9]. The computational domain is discretised by an unstructured mesh of about $5 \cdot 10^6$ cells of which around 88% are hexagonal.

For both problems, the Navier-Stokes equations are solved by the finite-volume based SIMPLE scheme with collocated variable arrangement, see Patankar and Spalding [10]. We apply our AMG algorithms to the pressure-correction equation only. This system is symmetric and positive definite and its solution is usually the most time consuming part of the whole simulation. In the case of problem 1, the SIMPLE iteration is terminated after 50 iterations, i.e. for this problem we consider the solution of 50 systems with different matrices and different right-hand sides. For problem 2 three time steps with together 89 SIMPLE iterations are calculated, i.e. here 89 different systems are solved.

The benchmarks were run on two different computers. Computer 1 is a cluster with four nodes where each node is equipped with two Intel X5650 and four Nvidia graphics boards of the type Tesla C2070. The network interconnect between the nodes of computer 1 is a 40 GB/s QDR Infiniband. Computer 2 is a single machine with two Intel X5650 per node, but it is equipped with four Nvidia GeForce GTX480 graphics boards instead of the Tesla graphics boards. The most important specifications of the hardware are compiled in Table 1. In this section we describe the parallel performance of several calculations. For the parallelisation, the entire calculation is split into a certain number of tasks that are, on a conventional CPU, carried out by the same number of processes.

Table 1. Hardware specification

	computer 1 & 2
CPUs	Intel X5650
per node	2
cores	2×6
main memory	96 GB
L3-cache	4×6 MB, shared
clock rate	2.67 GHz
memory bus	QPI, 26.7 GB/s

	computer 1	computer 2
GPUs (Nvidia)	Tesla C2070	GeForce GTX480
per node	4	4
multiprocessors	4×14	4×15
L2-cache	768 KB	768 KB
global memory	5375 MB	1535 MB
chip clock rate	585 MHz	700 MHz
memory clock rate	1.49 GHz	1.85 GHz
memory bus	136.8 GB/s	169.2 GB/s

Each task or process corresponds to one sub-domain that results from the domain decomposition. For our GPU-accelerated calculations, each of the available GPUs is associated with one process running on the CPU. In the case of GPU-accelerated calculations, the term process therefore refers to the number of sub-domains also.

Problem 1: Different Types of GPUs The average number of iterations in the left diagram of Figure 5 shows that the number of iterations is increased by about 10 % if the Gauß-Seidel smoother is replaced by the Jacobi smoother, which is common practice in AMG on GPU. The substitution of the double-pairwise aggregation (K-P4-J) by the plain aggregation with six nodes per aggregate (K-R6-J) increases the accumulated number of iterations by about a similar amount.

The comparison of the average computing time per iteration shows that computer 2 with the GeForce GTX480 graphics boards is around 40% faster than computer 1 with the Tesla C2070. This is due to the faster memory bus and the

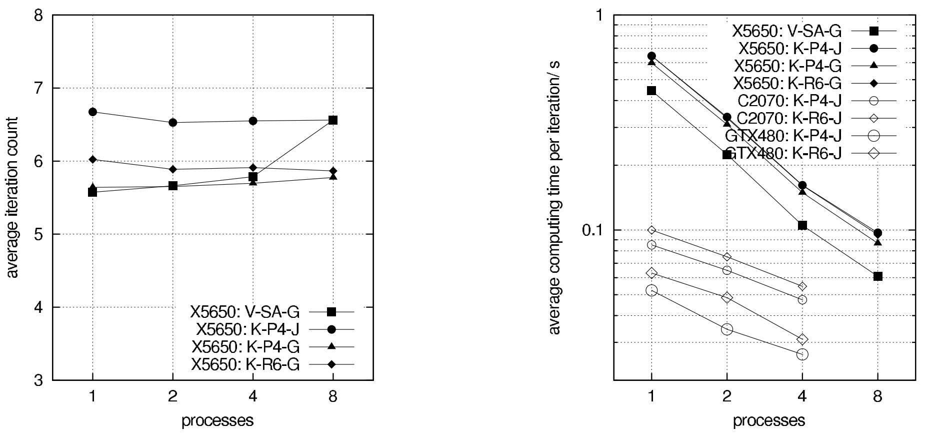


Fig. 2. Problem 1: average number of iterations (left) and average time per iteration on CPU and different GPUs (right)

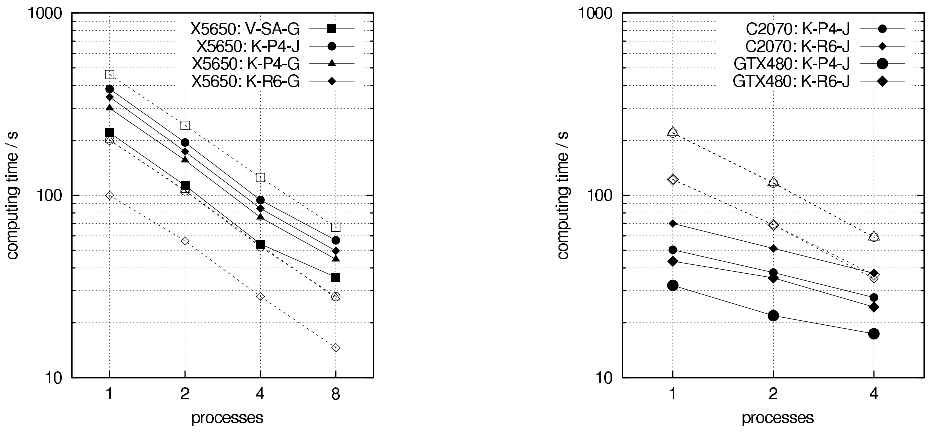


Fig. 3. Computing times for problem 1: CPU calculations (left), GPU calculations (right), filled symbols: AMG solution, empty symbols: AMG setup

higher chip clock rate of the GeForce GTX480. The execution of the same solution algorithm on the faster GPU is up to 13 times faster than on one core of the CPU. The comparison of the computing times for setup phase and solution phase in Figure 2 shows that for the Smoothed Aggregation AMG (V-SA-G) the setup is the dominant part. Since on the GPU we speed-up only the solution phase, this algorithm appears not to be favourable, although the time per iteration is short due to the v-cycle. The setup of the algorithm with the plain aggregation scheme (K-R6-J) is significantly faster than that of the double-pairwise aggregation (K-P4-J). Since the setup becomes dominant in the calculations involving GPUs, this leads to a significant reduction of the total computing time, too, see Figure 3. Thus, while on the CPU the fastest algorithm employs the double-pairwise aggregation, the fastest algorithm on the GPU employs the plain aggregation scheme. The total computing times with computer 2 using the GeForce GTX480 graphics boards is up to four times faster than the fastest calculation on the CPU, see again Figure 3. Finally, the parallel efficiency

$$E_p := \frac{t_1}{p \cdot t_p} \tag{6}$$

where t_p is the run-time with p parallel processes, is presented in the right diagram of Figure 4. Although the parallel efficiency of the calculations with GPU acceleration is inferior to that of the CPU calculations, it is within an acceptable range for practical applications.

Problem 2: Multiple GPUs on Different Nodes. With regard to the increase of the number of iterations due to the use of algorithms better adapted to the requirements of the GPU, i.e. the replacement of the Gauß-Seidel smoother by the Jacobi smoother and the replacement of the double-pairwise aggregation

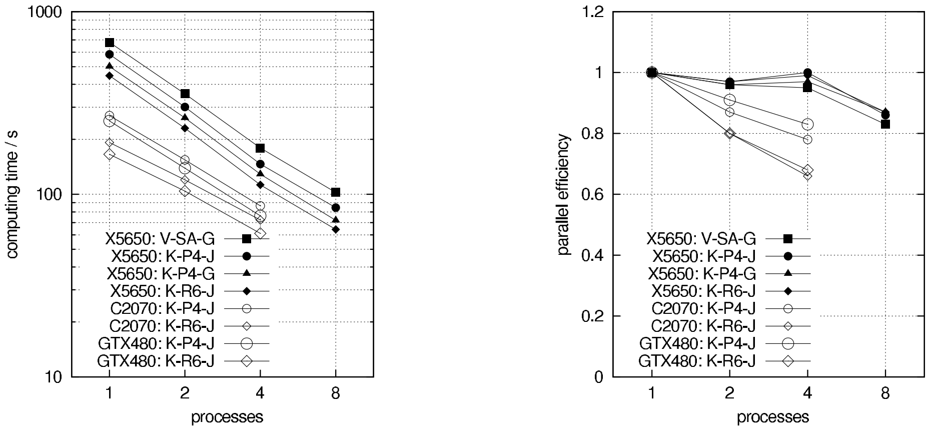


Fig. 4. Total computing times for problem 1 (left), Parallel efficiency for problem 1 (right)

by the plain aggregation, we observe for problem 2 the same as for problem 1, see Figure 5. The right diagram in this figure shows that in this case, too, the cost per iteration of the double-pairwise aggregation and the plain aggregation scheme are almost identical. It is, however, more important to observe that the usage of additional nodes with GPUs still accelerates the calculation in a reasonable manner: Remember that we have four GPUs per node, i.e. the calculation with 8 and 16 parallel processes runs on two and four nodes, respectively.

The left diagram in Figure 6 shows that the plain aggregation scheme leads to a significantly faster AMG method than the double-pairwise aggregation. For the GPU calculations the portion of the computing time spent in the setup

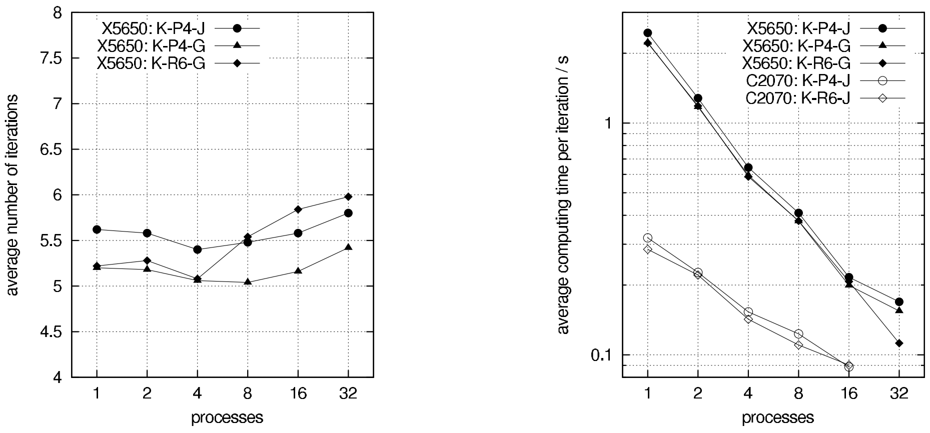


Fig. 5. Cumulative iteration count of various aggregation AMG algorithms (left) and average time per iteration of various AMG algorithms on CPU and GPU (right)

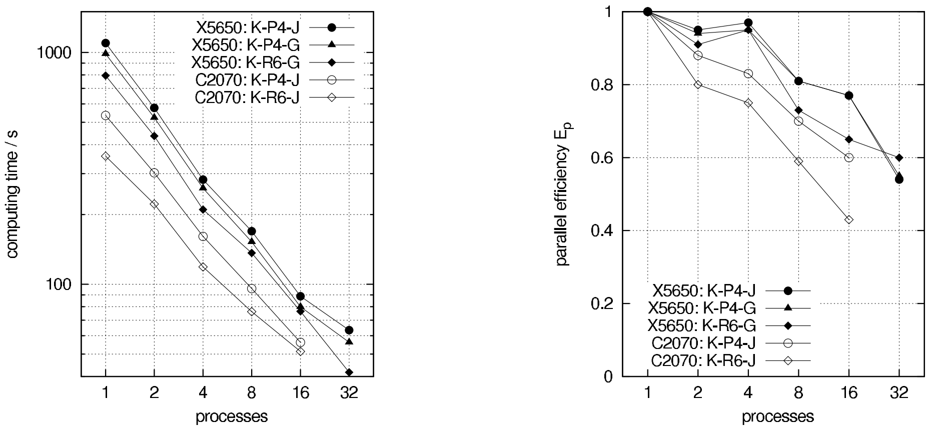


Fig. 6. Cumulative computing times: dashed line, empty symbols: AMG setup phase

is larger than for the CPU calculations. The reduction of the total computing time by substituting the double-pairwise aggregation by the plain aggregation is therefore for the GPU computations relatively large (30 %) whereas for the CPU computations it is only around 10 %. In total, i.e. including the setup on the CPU, the GPU implementation on the Tesla C2070 is around twice as fast as the fastest conventional implementation. Computations for problem 2 on computer 2 could not be carried out since the memory of the GeForce GTX480 graphics boards of computer 2 was not sufficient and only one machine with graphics boards of this type was available.

5 Conclusions

We have presented a parallel k-cycle AMG for GPUs. The conventional double-pairwise aggregation, implemented on the CPU, contributes significantly to the total computing time of the k-cycle AMG on GPU-accelerated hardware. It has been shown that it can be replaced by a more efficient plain aggregation algorithm. We have tested our implementation on a GPU cluster with four nodes each one equipped with four Nvidia Tesla C2070 GPUs. On a computer with four of the faster GeForce GTX480 graphics boards we could show that the entire AMG algorithm runs up to four times faster than the fastest AMG variant on the CPU. Although the parallel efficiency is already acceptable, future effort should be directed to an improved parallel performance.

Acknowledgement. Part of the contribution of Maximilian Emans was supported by the government of Upper Austria within the strategic program "Innovatives Oberösterreich 2010plus". The research of Manfred Liebmann was supported by the Austrian Science Fund FWF project SFB F032 "Mathematical Optimization and Applications in Biomedical Sciences".

References

1. Flynn, M.: Some computer organizations and their effectiveness. IEEE Transactons on Computers C-21, 948–960 (1972)
2. Emans, M.: Performance of parallel AMG-preconditioners in CFD-codes for weakly compressible flows. Parallel Computing 36, 326–338 (2010)
3. Haase, G., Liebmann, M., Douglas, C.C., Plank, G.: A Parallel Algebraic Multigrid Solver on Graphics Processing Units. In: Zhang, W., Chen, Z., Douglas, C.C., Tong, W. (eds.) HPCA 2009. LNCS, vol. 5938, pp. 38–47. Springer, Heidelberg (2010)
4. Notay, Y.: An aggregation-based algebraic multigrid method. Electronic Transactions on Numerical Analysis 37, 123–146 (2010)
5. Vaněk, P., Mandel, J., Brezina, M.: Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Computing 56, 179–196 (1996)
6. Emans, M.: Efficient parallel AMG methods for approximate solutions of linear systems in CFD applications. SIAM Journal on Scientific Computing 32, 2235–2254 (2010)
7. Falgout, R., Jones, J., Yang, U.: Conceptual interfaces in hypre. Future Generation Computer Systems 22, 239–251 (2006)
8. Emans, M.: AMG for Linear Systems in Engine Flow Simulations. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part II. LNCS, vol. 6068, pp. 350–359. Springer, Heidelberg (2010)
9. Hanjalic, K., Popovac, M., Hadziabdic, M.: A robust near-wall elliptic-relaxation eddy-viscosity turbulence model for cfd. International Journal of Heat and Fluid Flow 25, 1047–1051 (2004)
10. Patankar, S., Spalding, D.: A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. International Journal Heat Mass Transfer 15, 1787–1806 (1972)

Accelerating 2D-to-3D Video Conversion on Multi-core Systems

Jiangbin Feng, Yurong Chen, Eric Li, Yangzhou Du, and Yimin Zhang

Intel Labs China,
Intel Corporation
{jiangbin.feng,yurong.chen,eric.q.li,
yangzhou.du,yimin.zhang}@intel.com

Abstract. While the 3D-TV becomes widely available in the market, consumers will face the problem of serious shortage of 3D video content. Since the difficulty of 3D video capturing and manufacturing, the automatic video conversion from 2D serves as an important solution for producing 3D perception. However, 2D-to-3D video conversion is a compute-intensive task and real-time processing speed is required in online playing. Nowadays, with the multi-core processor becoming the mainstream, 2D-to-3D video conversion can be accelerated by fully utilizing the computing power of available multi-core processors. In this paper, we take a typical algorithm of automatic 2D-to-3D video conversion as reference and present typical optimization techniques to improve the implementation performance. The result shows our optimization can do the conversion on an average of 36 frames per second on an Intel Core i7 2.3 GHz processor, which meets the real-time processing requirement. We also conduct a scalability performance analysis on the multi-core system to identify the causes of bottlenecks, and make suggestion for optimization of this workload on large-scale multi-core systems.

1 Introduction

3D display devices have been widely available in the electronic market, from large-screen 3D-TVs, to middle-size desktop 3D monitors, even portable 3D smart phones. As of the popularity of 3D displays, consumers start to face a problem of severe shortage of 3D video content. The 3D video can be captured with stereo cameras, active depth sensing, or produced manually from existing 2D video footages. However, 3D video capturing requires professional devices and highly skilled photographers. Manual video conversion from 2D to 3D needs a lot of labor work and takes quite long producing time. Fortunately, with the advancement of visual analysis techniques, the computer is able to automatically do the 2D- to-3D video conversion job in moderate visual quality. Nowadays such solutions have been found in some software vendors and 3D-TV manufactures.

2D-to-3D video conversion is a very time-consuming task since it usually needs complex visual analysis and the computational complexity is often proportional to the total number of pixels. Acting as a plug-in function in traditional 2D TV receiver, the 3D conversion must be done in real-time during video broadcasting. To pursue computing speed, special hardware or high-end graphics card are often used in existing solutions. For instance, Toshiba places an 8-core Cell processor in their 3D-TV for real-time 3D video conversion. Philips builds an FPGA chip in their auto-stereoscopic 3D monitor for virtual view image generation and sub-pixel interleaving. For the desktop PC, GPU can be used to accelerate 3D video processing [1].

In this paper we use a typical automatic 2D-to-3D video conversion application as reference and optimize its implementation on Intel Core i7 2.3 GHz commodity processors. We profile the whole application flow and identify the compute-intensive modules. Then both data-level and task-level parallelization are conducted for these key modules. After optimization, the implementation can convert high-definition (1920x1080 pixels) 2D video to 3D on an average of 36 frames per second, which is much faster than the original algorithm and meets the real-time processing requirement. Furthermore, to characterize the performance of the optimized application, we also conduct a performance analysis and identify the possible causes of bottlenecks.

The remainder of this paper is organized as follows. Section 2 gives the algorithm description of our 2D-to-3D video conversion application and identifies most time-consuming modules for optimization and parallelization. Section 3 describes the implementation of the algorithm with focus on the SIMD optimization and OpenMP parallelization. Section 4 shows our experimental results and performance analysis of the optimized 3D conversion implementation. Finally our work is concluded in Section 5.

2 Algorithm Description

The flowchart of our 2D-to-3D video conversion algorithm is shown in Fig.1. A key characteristic here is using different strategy to estimate depth information for static image and dynamic image. After the input ordinary 2D video is decoded as individual frames, the first step is to do scene classification. If the current image is determined as a static scene, the *occlusion analysis* (OA) module is invoked to generate depth by using the inter-object occlusion relationship. If a scene is determined to be dynamic, the *depth from motion* (DfM) module is called to estimate the object depth according to its moving speed. When the depth image is calculated, the *depth image based rendering* (DIBR) module is applied to generate two or multiple images from virtual view points. Finally, the multi-view images are fed into the 3D display to produce depth perception in human visual system.

We analyzed the time breakdown of the whole processing flow and identified three most time consuming modules marked as thick-line rectangles in Fig.1. The detailed optimization and parallelization work on these three modules will be further introduced in the following subsections.

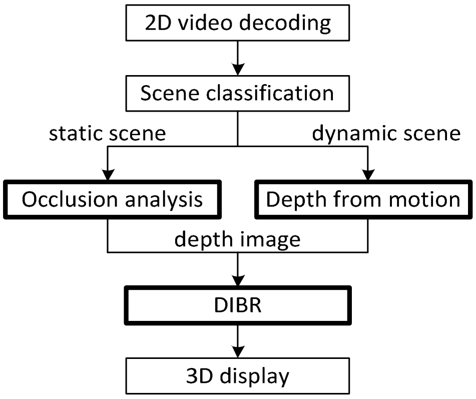


Fig. 1. The flowchart of the 2D-to-3D video conversion system

2.1 Occlusion Analysis

To generate the depth information for static images, we utilize occlusion analysis to deduce the relative depth between objects. Our knowledge is that non-occluded objects are usually closer than occluded objects. The more occlusion is found, the farther the object is located. We design a method to quantize the occlusion of each pixel and assume the depth at the pixel is promotional to its 'quantity' of occlusion. In this way, we will be able to generate a pseudo depth image for a static picture. In a single 2D image, the occlusion relationship is approximated with the accumulated color differences and the occlusion analysis for each point is shown in Fig.2. First a furthest depth boundary is defined at the upper side of the image (the blue line). For each point *p*, its shortest path (the red line) to the furthest depth boundary is calculated, taking the accumulated color difference along the path as a total cost. In this way, the depth of *p* can be estimated according to the shortest path with minimum possible cost. To speed up this procedure, the cost calculation can be done in an iterative manner by utilizing the 8-neighboring points around *p*. This is most similar to the calculation of dynamic programming. The details of occlusion analysis and depth conversion can be found in [2].

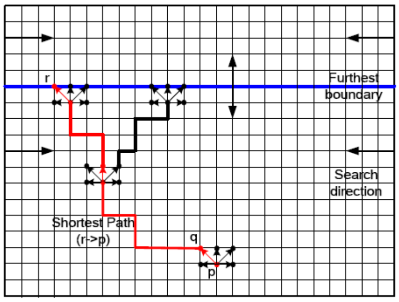


Fig. 2. Occlusion analysis for depth calculation [2]

2.2 Depth from Motion

For dynamic scenes, the depth can be estimated by the moving speed of objects. This is based on the observation that the faster moving object is generally closer to the viewer and vice versa. Fig.3 shows our implementation of the depth generation from motion. First the Mean-Shift segmentation [3] is applied so that the input 2D image is broken down into small blocks. For each block, motion vectors are calculated and their average value is taken as the moving speed of current block. Then, the depth information of the block is estimated as a reverse proportional function of its moving speed. After the depth of all blocks are calculated, they are connected to form a piece-wise smoothed depth map.

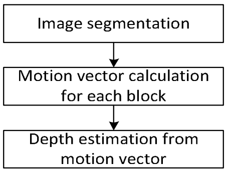


Image segmentation
Motion vector calculation for each block
Depth estimation from motion vector

Fig. 3. Generating depth from motion vector

2.3 Depth Image Based Rendering (DIBR)

After the depth map is calculated for current 2D image, the classical DIBR algorithm [4] is used to generate virtual view images. As shown in Fig.4, suppose the input 2D image is observed at the view point v_0, the virtual image from the view point v_1 can be calculated through image warping. Due to view point change, some area may suffer from disocclusion problem as indicated with the red color in Fig.4. This will cause holes in synthesized virtual images. To alleviate this effect, a simple solution is using Gaussian smoothing to remove sharp edges on the depth map as shown in Fig.4 (c). Note that the DIBR algorithm has an appealing feature that the image warping operation could be done completely independent for each row of pixels. This feature creates a great opportunity for the task-level parallelization.

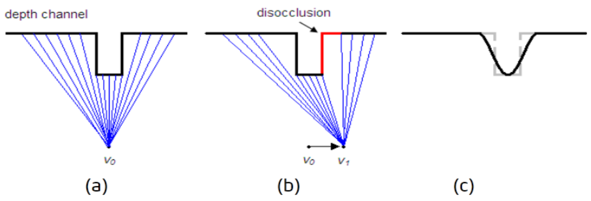


Fig. 4. The DIBR algorithm. (a) Original view point v_0; (b) New view point v_1; (c) Smoothed depth map.

3 Optimization and Parallelization

This section considers how to optimize the program's serial and parallel performance. For serial performance optimization, we use SIMD intrinsics [5] to implement the vectorization. When talking about SIMD intrinsics we refer to 128-bit wide SSE/SSE2 instructions and now we have AVX (Intel® Advanced Vector Extensions, [6]) which is 256-bit wide. In this paper, we use both SSE (SSE2, SSE3, etc) and AVX intrinsics to accelerate the serial performance. For the parallel implementation, we choose OpenMP [7] for the shared memory system.

3.1 Serial Performance Optimization

In this work, we mainly improve the performance of 2D-to-3D video conversion through SIMD intrinsics. Using SIMD technology which performs multiple arithmetic or logic operations simultaneously can effectively improve the execution efficiency. In the 2D-to-3D video conversion, all the 3 hot-spot modules mentioned in the previous section can exploit the data-level parallelism (DLP). As DIBR is the most time-consuming module, we use it as an example to describe the optimization details. Similar optimization techniques can be applied to other modules in this application.

In the serial DIBR algorithm, the results of 3 dimensions of a pixel are computed at one time. However, exploiting DLP at color space level has limited parallelism. Therefore, we reorganize the data structure of the algorithm, i.e., we process four (or eight) pixels in single channel simultaneously to utilize the four (or eight) wide SIMD capability. This process repeats 3 times to get the results from 3 different channels.

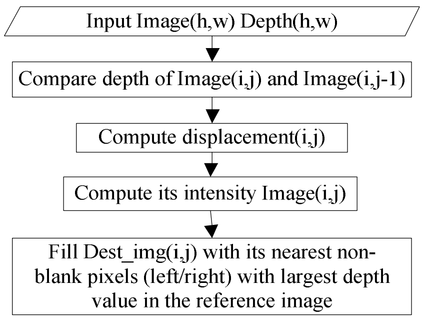

Input Image(h,w) Depth(h,w)
Compare depth of Image(i,j) and Image(i,j-1)
Compute displacement(i,j)
Compute its intensity Image(i,j)
Fill Dest_img(i,j) with its nearest non-blank pixels (left/right) with largest depth value in the reference image

Fig. 5. The flowchart of the serial DIBR

For example, in the DIBR module, the program needs to perform rendering for every pixel in each frame following the procedures below: 1) Calculate the depth difference between the current frame and the previous frame; 2) Get the distance of offset pixel according to the depth difference; 3) Rendering between minimum threshold and maximum threshold. As shown in Fig.5, the rendering needs to be executed for all the pixels in one frame [8]. We conduct the optimization based on the original DIBR as follows.

- Data reconstruction and alignment. We modify this algorithm to process 4 or 8 pixels at one time, thus the code is executed only $h*w/4$ or $h*w/8$ times to complete DIBR. We also use the 16-bit alignment for SSE and 32-bit alignment for AVX to enhance cache and memory locality.

- Branch optimization. As branch mis-prediction can break the CPU execution pipeline, it is more favorable to use SIMD instructions to reduce conditional branch operations. It will help improve the overall performance.
- Changing division into multiplication. Multiply instruction is more cost effective than division instruction. It is better to convert division instructions to multiplication to save time.
- Taking constant operation out of loop. In the outer loop there are some constant operations which are repeatedly calculated till the end. We move the constant operation out of the loop to save time.

After restructuring the data and using SSE/AVX to SIMDize DIBR, we obtain the optimized DIBR implementation. The pseudo-codes of SSE/AVX DIBR are shown in Fig.6 and Fig.7, respectively.

```
/*  depth_this : depth of this pixel
   depth_former : depth of former pixel
   src : frame input for rendering   */
for(y = 0; y < h; y++)
{
for (x = 0; x < w/4*4; x += 4)
{
mask = _mm_cmpgt(this,former);
offest1= _mm_or(_mm_and(mask,this),\
            _mm_andnot(mask,former));
offset2= _mm_or(_mm_andnot(mask,this),
            _mm_and(mask,former));
src1 = _mm_cvtps(_mm_or(_mm_and(mask,x),\
         _mm_andnot(mask,x-1)));
src2=_mm_cvtps(_mm_or(_mm_andnot(mask,x),\
        _mm_and(mask,x-1)));

mask = _mm_cmpgt(0,offset1);
i0 =_mm_or(_mm_and(mask,0),\
     _mm_andnot(mask,offset1));
mask = _mm_cmpgt(w-1,offset2);
i1=_mm_or(_mm_andnot(mask,w1),\
     _mm_and(mask,offset2));

temp0 = _mm_sub(offset2,offset1);
mask = _mm_cmpeq(temp0,_mm_setzero());
t_1 = _mm_div(1,_mm_add(,_mm_and(mask,1)));

//get this_pixel,former_pixel from the offset and src
```

```
a = _mm_mul(_mm_sub(_mm_mul(this_pixel[0],\
offset2),_mm_mul(former_pixel[0],offset1)),t_1);
a_1= _mm_mul(_mm_mul(former_pixel[0],\
        this_pixel[0]),t_1);

b = _mm_mul(_mm_sub(_mm_mul(this_pixel[1],\
offset2),_mm_mul(former_pixel[1],offset1)),t_1);
b_1= _mm_mul(_mm_mul(former_pixel[1],\
        this_pixel[1]),t_1);

c = _mm_mul(_mm_sub(_mm_mul(this_pixel[2],\
offset2),_mm_mul(former_pixel[2],offset1)),t_1);
c_1= _mm_mul(_mm_mul(former_pixel[2],\
       this_pixel[2]),t_1);

//get the arr_dstpixel from dst and offset

for (j = 0; j < 4; j++)
{
int temp_j = i1.m128_f32[j];
for (i = i0.m128i_i32[j]; i <= temp_j; \
                  i++, arr_dstpixel[j] += 3)
{
arr_dstpixel[j][0] = a.m128_f32[j] + \
                    a_1.m128_f32[j] * i;
arr_dstpixel[j][1] = b.m128_f32[j] +\
                    b_1.m128_f32[j] * i;
arr_dstpixel[j][2] = c.m128_f32[j] +\
                   c_1.m128_f32[j] * i;

}}}}
```

Fig. 6. Pseudo-code of the SSE DIBR

```
for(y = 0; y < h; y++)
{
for (x = 0; x < w/8*8; x += 8)
{
#define GT 14
mask = _mm256_cmp(this,former,GT);
offset1 = _mm256_or(_mm256_and(mask,this),\
         _mm256_andnot(mask,former));
offset2= _mm256_or(_mm256_andnot(mask,this),
          _mm256_and(mask,former));
src1 = _mm256_cvtps(_mm256_or(_mm256_and\
(mask,x),_mm256_andnot(mask,x-1)));
src2 = _mm256_cvtps(_mm256_or(\
         _mm256_andnot (mask,x),\
         _mm256_and(mask,x-1)));
#define EQ 16
temp0 = _mm256_sub(offset2,offset1);
mask =_mm256_cmp(temp0,0,EQ);
t_1=_mm256_div(1,_mm256_add(temp0,\
       _mm256_and(mask,1)));
i0= _mm256_cvtps(_mm256_max(0,\
     _mm256_floor(offset1)));
i1=_mm256_cvtps(_mm256_min(w-1,\
     _mm256_floor(offset2)));
//get this_pixel,former_pixel from the offset and
src
a = _m256_mul(_mm256_sub(\
     _mm256_mul(this_pixel[0],offset2),\
     _mm256_mul(former_pixel[0],offset1)),t_1);
```

```
a_1 = _mm256_mul(former_pixel[0],\
        this_pixel[0]),t_1);

b = _m256_mul(_mm256_sub(\
     _mm256_mul(this_pixel[1],offset2),\
_mm256_mul(former_pixel[1],offset1)),t_1);
b_1 = _mm256_mul(former_pixel[1],\
     this_pixel[1]),t_1);

c = _m256_mul(_mm256_sub(\
     _mm256_mul(this_pixel[2],offset2),\
     _mm256_mul(former_pixel[2],offset1)),t_1);
c_1 = _mm256_mul(former_pixel[2],\
        this_pixel[2]),t_1);

//get the arr_dstpixel from dst and offset

for (int j = 0; j < 8; j++)
{
temp_j = i1.m256i_i32[j];
for (int i = i0.m256i_i32[j]; i <= temp_j; i++,
dstpixel[j] += 3)
{
dstpixel[j][0] =  a.m256_f32[j] + \
                 a_1.m256_f32[j] * i;
dstpixel[j][1] =  b.m256_f32[j] + \
                 b_1.m256_f32[j] * i;
dstpixel[j][2] =  c.m256_f32[j] + \
                 c_1.m256_f32[j] * i;
}}}}
```

Fig. 7. Pseudo-code of the AVX DIBR

3.2 Parallelization

Data decomposition and Task decomposition methods are two primary decomposition methods in parallel program design. The former divides the computation among multiple threads based on different data segments. The latter operates on a set of tasks that can run in parallel. Both types of parallelism can be used in the same program and no one method is always better than the other. However, in the 2D-to-3D video conversion application, the majority of the work is conducted on 2D images, which have abundant data parallelism in the picture-level, row-level, and even pixel-level. The selection of data parallelism is a natural choice to make use of the inherent parallelism. Further, to meet the real-time processing capability for on-line video applications, it is important to extract the fine-grained parallelism within each image instead of exploiting coarse-grained parallelism at frame level.

We perform a detailed analysis of this application, and reorganize the data structure and coding flow to facilitate the use of threading models. In the following section, we use several examples to demonstrate how to design proper parallel schemes for the 2D-to-3D video conversion application.

DIBR Parallelization. Each row in the frame processed by DIBR is independent of each other. So each row can do rendering work in parallel. The pseudo-code is shown in Fig.8.A.

DfM and OA Parallelization. Besides DIBR, DfM and OA are also time-consuming which need to be parallelized. Both modules need to calculate the depth map. They divide every image into a large number of blocks and build a tree of them according to the virtual distance to camera. We calculate depth of every pixel according to its belonging block and the tree. So when we get the depth map we can parallelize pixels to run independently without disturbing each other. We not only reorganize the data in OA, but also identify some computations in the inner loop which can be moved out to the outer loop to save computation. Furthermore, we need to remove the branch operations like “break” or “goto” statement in the parallel loop, otherwise it will cause segmentation fault when directly using OpenMP parallelization. The parallel pseudo-codes are shown in Fig.8.B.

```
#pragma omp parallel
{
#pragma omp for nowait schedule(guided)
For( y = 0; y < h; y++)
{
    for (x = 0; x < w; x++)
    {
//render every pixel one by one accroding to
    //image depth
    }
}
}
```

A.

```
//depth from motion
#pragma omp parallel for schedule(dynamic)
for (i = 0 ; i < fl->nFeatures ; i++)
{
    //optical flow search right feature
    //points and judge the motion
}
//occlusion analysis
#pragma omp parallel
{
#pragma omp for nowait schedule(dynamic)
for( int i = 0; i < width * height; i++ )
{
//calculate depth map from known builded tree
}}
```

B.

Fig. 8. Pseudo-codes of the parallel DIBR, DfM and OA

3.3 Parallel Performance Optimization

After serial optimization, we continue to apply some parallel optimization techniques to enhance the performance of the parallel 2D-to-3D video conversion implementation.

Reducing Load Imbalance. The load imbalance greatly impacts the scalability performance in a parallel application. If one core spends more time than the other cores, the unbalanced load becomes a limiting factor for parallel performance. It is important to keep all the cores busy by load balancing the tasks and minimizing overhead. In the 2D-to-3D video conversion application we use several techniques to improve the load balance performance. For almost all the modules, we use the dynamic scheduling policy to minimize the load imbalance. Particularly, in the DIBR module, we manually use a "guided" scheduling policy, and the task size is chosen depending on the tasks within each parallelization loop. A guided scheduling policy helps to balance the size of tasks and scheduling overhead. Because each task is independent from each other, we use "nowait" to reduce task synchronization as shown in Fig.9.

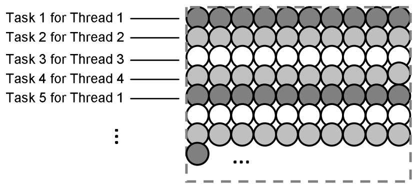


Fig. 9. The distribution of dataset for the 4 threads case

Reducing Synchronization Overhead. Often threads are not totally independent, which forces the program to add synchronization to guarantee the execution order of the threads. The frequent synchronization calls and the associated waiting operations will degrade the scaling performance on the multi-core processors. Generally the synchronization is present in the form of critical section, lock, and barrier in the OpenMP implementation. For the 2D-to-3D video conversion application, we also have to deal with some synchronization operations. For example, in OA module, it needs to calculate the depth map, all the threads push intermediate results to a shared matrix, and a critical section is necessary for synchronization. A lock is used every time when one thread pushes a value to the matrix, which consumes too much time. We replicate the shared matrix into several private matrices. Each thread operates on its local matrix to avoid the mutual access to the shared matrix. All the local matrices are merged at the end of the parallel region. This mechanism is totally lock-free and the synchronization overhead is reduced significantly.

4 Experimental Results and Performance Analysis

In this section, we evaluate and analyze the performance of the 2D-to-3D video conversion application on a 4-core laptop system, with one Intel Core i7-2820QM processor running at 2.3 GHz. Each core is equipped with a 32KB L1 data cache, a 32KB L1 instruction cache and a 256KB L2 cache. The four cores share an 8MB L3 unified cache.

We used 6 video clips in our experiments as shown in Table 1. These video clips were carefully chosen from different scene categories, including indoor and outdoor, fast motion and slow movement, people and animals with different video resolution.

Table 1. Resolution and video length of the 6 video clips

Video clips	Resolution	Video length(s)
fight.mkv	1200x512	63.035
indoor.mkv	1136x480	165.842
mountain.mkv	1280x720	61.120
seal.mkv	1280x720	59.659
sport.mkv	720x480	61.028
avatar.mkv	1920x1080	108.400

4.1 Time Breakdown

The computation time breakdown of the serial 2D-to-3D video conversion application is shown in Fig.10. From this figure, we can see that DIBR is the most compute-intensive module occupying ~60% of total computing time in all video clips except the *sport* video clip, in which DfM is the most time consuming one. OA and DfM consume almost the same time (~15% of total computing time) in most cases. These three key modules make up more than 85% (85~91%) of the whole application execution time.

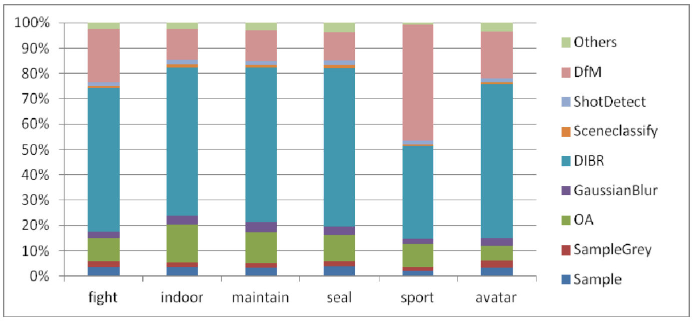


Fig. 10. The computation breakdown of the serial 2D-to-3D video conversion application

4.2 Performance Improvement

The conversion speed using frames per second (FPS) for each step of the optimization is shown in Fig.11. For each step of the experiment, we collect the computation time (excluding video decoding) of each video clip, divided them by the number of frames in the video clip and then get an average FPS from all the results.

In total, our accelerated 2D-to-3D video conversion runs ~3 times faster than the original version (a version of the original algorithm that incorporates all other serial optimizations except vectorization) on the 4-core system. The SSE optimization makes the program 63% faster than the original version on average. The AVX gains an additional 13% speedup over SSE version. Finally, the parallelization improves the speed ~60%. Furthermore, according to the conversion time shown in Table 1, we can see that after the optimization and parallelization the application can convert the high-definition (1920x1080 pixels) video clip *avatar* to 3D video on an average of 36 FPS, which is much faster than the original algorithm and meets the real-time processing requirement.

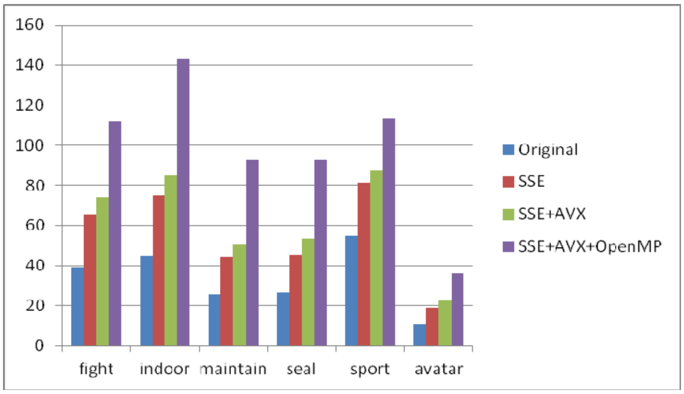


Fig. 11. Processing speed (FPS) of the 2D-to-3D video conversion application on a 4-core system

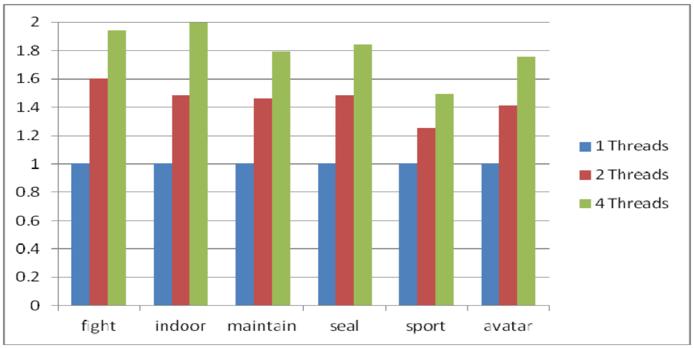


Fig. 12. Relative speedup of the optimized 2D-to-3D video conversion application

4.3 Scalability Performance

Fig.12 shows the scaling performance of the optimized 2D-to-3D video conversion application on the 4-core system. From the figure, we can find that the program scales well for 2 and 4 threads. It achieves ~1.5x speedup on two cores and ~1.8x speedup on four cores. The scalability performance is not so good, but quite respectable. Considering that we just parallelized ~85% region of the whole application, the large serial region (~15%) is one of the bottlenecks of the scaling performance. In addition, some inherent serial properties in the application such as tree building in OA and DfM also prevent us from obtaining good scalability performance.

5 Conclusion

There is a rising demand on new techniques for automatically converting 2D video content to stereoscopic 3D video display. In this paper, we optimize and parallelize a typical 2D-to-3D video conversion application on multi-core systems. As a result, the accelerated program can convert high-definition (1920x1080 pixels) 2D video to 3D on an average of 36 FPS on a 4-core laptop. The speed is much faster than the original implementation and meets the real-time processing requirement. Besides, we also analyze the scalability performance of the optimized application, and find that the sequential regions and some inherent serial properties in the program are the main limiting factors of the scaling performance of the application. To improve the performance on a large-scale multi-core system, we need to further extract parallelisms in some small serial regions and improve the parallel method for some key modules for much better scaling performance.

Acknowledgements. Our thanks go to Prof. Qionghai Dai from Tsinghua University for collaborating with us on the reference application development and for providing some reference code for our optimization and analysis.

References

1. Tsai, S.-F., et al.: A real-time 1080p 2D-to-3D video conversion system. In: IEEE International Conference on Consumer Electronics, ICCE (2011)
2. Zhang, J., Yang, Y., Dai, Q.: A novel 2D-to-3D scheme by visual attention and occlusion analysis. In: 3DTV Conference (2011)
3. Comaniciu, D., Meer, P.: Mean shift: a robust approach toward feature space analysis. IEEE Transactions on Pattern Analysis and Machine Intelligence 24(5) (May 2002)
4. Fehn, C.: Depth-image-based rendering (DIBR), compression, and transmission for a new approach on 3D-TV. In: SPIE 2004 (2004)
5. Intel Corporation, Intel C/C++ Compiler Intrinsics and Functional Equivalents, IA Software Developer's Manual, vol. 3, Appendix C
6. Intel Corporation, Intel® Advanced Vector Extensions Programming Reference (319433-011) (June 2011)
7. OpenMP Application Program Interface, Version 2.5 (May 2005)
8. You, Z., et al.: Parallel depth image-based rendering algorithm for the next generation 3D-TV applications. In: The 1st Workshop on Emerging Applications and Many Core Architecture (June 2008)

Solution of Multi-Objective Competitive Facility Location Problems Using Parallel NSGA-II on Large Scale Computing Systems

Algirdas Lančinskas and Julius Žilinskas

Institute of Mathematics and Informatics, Vilnius University,
Akademijos 4, 08663, Vilnius, Lithuania
{algirdas.lancinskas,julius.zilinskas}@mii.vu.lt
http://www.mii.vu.lt

Abstract. The multi-objective firm expansion problem on competitive facility location model, and an evolutionary algorithm suitable to solve multi-objective optimization problems are reviewed in the paper. Several strategies to parallelize the algorithm utilizing both the distributed and shared memory parallel programing models are presented. Results of experimental investigation carried out by solving the competitive facility location problem using up to 2048 processing units are presented and discussed.

Keywords: Multi-objective Optimization, Parallel Pareto Ranking, Parallel Non-dominated Sorting Genetic Algorithm, Competitive Facility Location Problem.

1 Introduction

The classical Competitive Facility Location (CFL) problem deals with locating a new facility in an area where a number of preexisting facilities are competing with each other for the market share. The goal of solving CFL problem is to locate new facilities (one or several) on purpose to maximize market share of them.

The CFL problem was first introduced by Hotelling in 1929 [1] who considered a competition on a particular segment such as a main street. There are references in literature i.e. [2–4], describing various CFL models which may vary on different properties. For instance, the location space may be the plane or a discrete set. We may want to locate just one or more than one new facility. The competition may be static (competitors are already in the market and their characteristics are known), or with foresight (the competitors are not in the market yet but they will be soon after locating of new facility). In the latter case it is necessary to make decisions with foresight about this competition, what leads to a Stackelberg-type [5] model. Furthermore, if the competitors can change their decisions, then we have a dynamic model, in which the existence of equilibrium situations is of major concern. Besides these properties very important is the behavior of customers when choosing the facility to purchase a service [6]. Knowing the behavior of customers helps the decision maker to estimate the demand captured by an existing or prospective facility.

Behavior of the customers has been studied in several disciplines such as geography, economics and marketing, however it is hard to describe the mathematical model correspondent to the behavior of customers in real-life situations. One of the simplest model of the behavior is based on the assumption that customers patronize the closest (or cheapest) facility, although this model is quite far from reality. More realistic model of the behavior of customers has been introduced by Huff [7, 8] who suggested the assumption that customer's patronage is divided among the competing facilities according to an attractiveness of the facility and the distance between customer's demand point and the facility.

A lot of research focuses on a location decision for an entering firm assuming the competing facilities to be known. Even the simplest case of CFL problem when locating a single facility in continuous space, may lead to a hard global optimization problem. The CFL problem for locating a single facility using attraction functions of gravity type has been studied in [9, 10], and in [11] using different kinds of attraction functions. The sequential decisions for two firms which are competing in a leader-follower Stackelberg situation have been considered in [5, 12].

Like most of real-life optimization problems, CFL problems are often multi-objective. Besides the main goal – to maximize the market share of new facilities, additional factors such as costs of maintenance of prospective facility or potential influence of hazards to the citizens, should be taken into account when locating new facilities.

In this paper we will focus on firm expansion problem CFL model. Let us consider two competing firms: A and B which already have preexisting facilities providing some goods or services in certain area in which demand of customers is assumed to be known. Let as assume that behavior of customers is based on the simplest model as they patronize the facility which is the closest. In the case of choosing between two or more facilities that are equidistant, customers distribute equally among all of them. Suppose that firm A wants to open a set consisting of one or several new facilities in order to increase the total demand. Naturally the expansion of firm A cannot reduce the total market share of the firm, however it may happen that some of new facilities can attract customers from the preexisting facilities belonging to the firm A thus involving the effect of "cannibalism". Therefore firm A faces multi-objective optimization problem with the following two objectives:

(1) to maximize total demand captured by new facilities,
(2) while minimizing the effect of cannibalism.

In the next section we will present the main principles of multi-objective optimization. Later we will describe an evolutionary algorithm for multi-objective optimization and several approaches to parallelize it using distributed memory as well as shared memory parallel programming models. Finally the results of the experimental investigation of performance of the parallel algorithm will be presented and conclusions will be formulated.

2 Multi-objective Optimization

In general, a Multi-objective Optimization Problem (MOP) with d variables and m objectives in the objective vector

$$f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x}) \tag{1}$$

is to find decision vector

$$\mathbf{x} = (x_1, x_2, \ldots x_d), \tag{2}$$

which optimizes the objective vector

$$F(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x})). \tag{3}$$

Focusing on the problem formulated in previous section, we will consider MOP with two objectives, one of which is subject to maximize (the total captured demand), and another – to minimize (the effect of cannibalism). Since there can be conflicts between the objectives, it can be impossible to find a single solution – a decision vector which would be the best according to both the objectives – a decision vector which is the best according to increment of the market share could be unacceptable according to the effect of cannibalism. Also two decision vectors, let us call $\mathbf{x}$ and $\mathbf{y}$, may be *indifferent* (denoted by $\mathbf{x} \sim \mathbf{y}$) to each other – one is better according to the first objective while another – according to the second. However it may happen that decision vector $\mathbf{x}$ is better than decision vector $\mathbf{y}$ according to both the objectives. If so it is said that decision vector $\mathbf{x}$ *Pareto-dominates* (or simply *dominates*) decision vector $\mathbf{y}$. The dominance relation is denoted by $\mathbf{x} \succ \mathbf{y}$. In general, decision vector $\mathbf{x}$ dominates decision vector $\mathbf{y}$ if and only if

(1) decision vector $\mathbf{x}$ is not worse than $\mathbf{y}$ according to all objectives and
(2) decision vector $\mathbf{x}$ is strictly better than $\mathbf{y}$ by at least one objective.

Decision vectors which are non-dominated by any other decision vector are called *Pareto-optimal* and a set of those vectors – *Pareto-set*. Determination of this set is the main goal of multi-objective optimization. Unfortunately, in most cases, it is difficult or impossible to determine the exact Pareto-set in reasonable time. Moreover solving most of practical multi-objective optimization problems it is not necessary to obtain exact Pareto-set, rather the approximation with a reasonable precision. Therefore an approximation methods may be used, which provide approximation of exact Pareto-set.

One well known class of optimization algorithms – Evolutionary Algorithms (EAs) are well suited to solve multi-objective optimization problems as they are based on biological processes which are inherently multi-objective. Multi-Objective EAs

(MOEAs) can yield a whole set of potential solutions – which are all optimal in some sense – and give the option to assess the trade-offs between different solutions. Furthermore EAs require little knowledge about the problem being solved and they are robust and easy implementable. To solve a certain optimization problem, it is enough to require that one is able to evaluate the objective functions for a given set of input parameters. Furthermore EAs are inherently parallel what is very important in solving facility location problems which require to process a large amount of data.

3 Non-dominated Sorting Genetic Algorithm

One of the most popular MOEA is Non-dominated Sorting Genetic Algorithm which was proposed by Srinivas and Deb [13], and was one of the first MOEQ which have been applied to various problems [14, 15]. An updated version of the algorithm (NSGA-II) have been proposed by Deb et. al. in [16].

The algorithm begins with an initial parent population P, consisting of N decision vectors randomly generated over the search space. Then the algorithm continues with the following process:

(1) A new child population Q is created by applying genetic operators (selection, crossover and mutation) to the elements of the parent population. Usually a child population has the same size as the parent population.

(2) Parent and child populations are combined into one $2N$-size population

$$R = P \cup Q. \tag{4}$$

(3) The derived population R is sorted according to the number of dominators, and reduced to the size of N decision vectors by removing the most dominated elements. If two or more decision vectors are equally dominated then crowding distances estimator [16] is used to choose the most promising decision vector.

(4) Go to the 1st step by using the reduced population as a parent population P in generation of the child population.

4 Parallel NSGA-II

4.1 Parallel Computing

Parallel computing is a form of computations in which computations are performed utilizing many processing units simultaneously. Parallel computing deals with the principle that problems requiring a lot of computational recourses can be often divided into a set of independent subproblems that can be solved using different computing recourses simultaneously. The performance of parallel algorithm is often evaluated by measuring the *speed-up* S_p of the algorithm – the ratio of time T_0 required to solve the problem using the best known sequential algorithm and time T_p required to solve the problem using parallel algorithm on p processing units:

$$S_p = \frac{T_0}{T_p}. \tag{5}$$

The ideal speed-up is linear: $S_p = p$, however, often it is hard or impossible to achieve due to reasons such as a need of additional time for communication between processors or existence of parts of the algorithm which cannot be solved simultaneously. The Amdahl's law [17] states that the speed-up of a parallel algorithm cannot be larger than

$$S_{max} = \frac{1}{\alpha} \tag{6}$$

independent on the number of processing units used. Here α is the fraction of the time a sequential algorithm spends on non-parallelizable parts.

There are two general parallel programming models – shared memory and distributed memory. The main difference is the mechanisms by which parallel processing units are able to communicate with each other. In shared memory parallel programming model processing units communicate by manipulating general address space accessible by all processing units asynchronously. In a distributed memory model, parallel processing units exchange data through passing messages to each other. OpenMP (OMP) and POSIX Threads are two most widely used shared memory parallel programing interfaces, whereas Message Passing Interface (MPI) is one of the most widely used for distributed memory parallel programing.

Naturally parallel algorithms implemented under distributed memory programming model are more time consuming for communication between processors comparing with shared memory parallel programming model. On the other hand shared memory parallel programing model has more hardware limitations and requires additional management of access to the memory that enables to avoid concurrent access to the same memory space.

4.2 Parallelization of NSGA-II Using MPI

The algorithm NSGA-II can be roughly separated into 3 parts:

(1) evaluation of objective functions,
(2) Pareto ranking and genetic operations,
(3) other computations such as non-dominated sorting, memory allocation and management, etc.

Most of parallel implementations of NSGA-II are based on parallelizing the first part which is often the most computational recourse consuming part of the algorithm [18, 19]. However if even 99% of computational recourses are utilized to evaluate objective functions, the speed-up of the algorithm cannot be larger than 100 independent on processing units used (see equation (6)).

Another strategy to parallelize NSGA-II has been proposed in [20]. The parent population P is distributed among all processors which generates appropriate subpopulation Q_i of child population such as

$$Q = \bigcup_i Q_i, \tag{7}$$

where $i = 1, \ldots, p$ and p denotes the number of processing units. The partial Pareto ranking is performed while the child subpopulations are being gathered as illustrated in Figure 1, where $dom(P \otimes Q)$ denotes the procedure of counting how many dominators each vector from the set P has in the set Q. The strategy operates on the following main principles [20]:

(1) population $P \cup Q$ can be Pareto-ranked by performing four operations: $dom(P \otimes P)$, $dom(P \otimes Q)$, $dom(Q \otimes P)$, $dom(Q \otimes Q)$;
(2) operation $dom(P \otimes P)$ is necessary only in the first generation of NSGA-II.

The parallel algorithm implemented using this strategy will be denoted by ParNSGA/HR hereinafter.

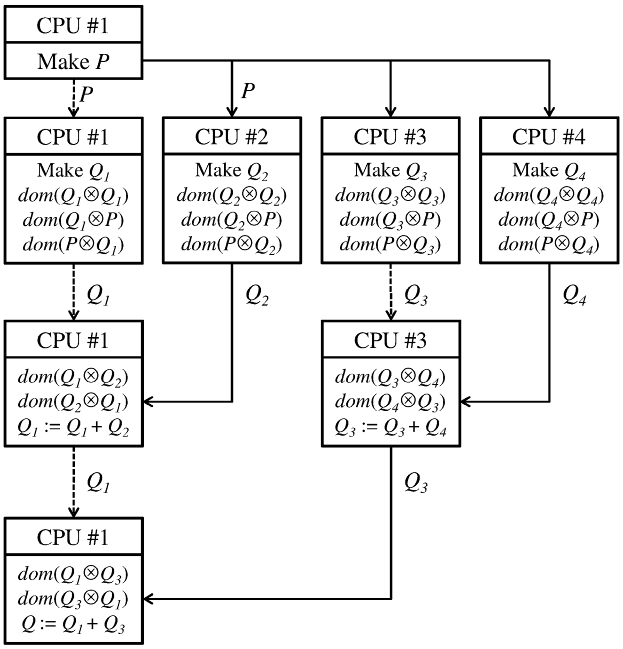


Fig. 1. Scheme of parallel NSGA-II with partially parallel Pareto ranking performed following hierarchic fashion

Although ParNSGA/HR algorithm parallelizes Pareto ranking without additional cost of communication, non-master processing units may be idle for some time. On the one hand it could be insignificant performing computations on several or several tens processing units, but on the other hand the idle time can be relatively long using several hundreds or thousands of processing units thus significantly affecting the speed-up of the algorithm. For comparison we introduce another strategy which gives an opportunity to avoid significant idle time, but increases costs for communication between processing units:

(1) The master processing unit distributes the parent population among the slaves following the hierarchic fashion.
(2) Each processing unit generates an appropriate part of child population.
(3) The master processing unit gathers subpopulations of the child subpopulations Q_i from the slaves, combines into one population Q and distributes it among the slaves.
(4) Each processing unit evaluates the corresponding subpopulation $Q_i \subset Q$ by performing $dom(Q_i \otimes Q)$, $dom(Q_i \otimes P)$ and $dom(P \otimes Q_i)$.
(5) The master processing unit gathers all the information about Pareto ranks, counts the total values of Pareto ranks of each element in $P \cup Q$ and distributes them among all slaves.
(6) Now all processing units have the whole set $P \cup Q$ including the information about Pareto ranks, and can continue with rejecting most dominated decision vector and proceeding to the next generation (2nd step).

The parallel algorithm implemented under the latter strategy will be denoted by ParNSGA/DR hereinafter. Scheme of the algorithm is given in Figure 2.

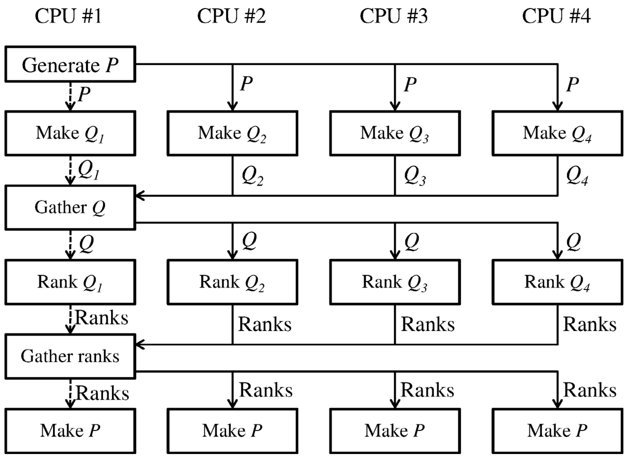


Fig. 2. Scheme of parallel NSGA-II with distributed Pareto ranking

4.3 Hybrid MPI-OpenMP Algorithm

In order to maintain the same behavior as sequential algorithm has, parallel algorithm has to initiate data exchange between processing units after every generation. Thus the main performance bottleneck in parallel algorithm is often communication latency between processing units. To reduce the communication cost required for frequent message passing we improved ParNSGA/DR by utilizing both distributed and shared memory programing models. The derived hybrid MPI-OpenMP algorithm ParNSGA/MPI-OMP is based on creating processing units consisting of a number of shared memory threads which are able to communicate through shared memory space, as illustrated in the right image in Figure 3. The communication between processing units are performed utilizing message passing interface as in ParNSGA/DR, however the number of processing units is smaller.

As illustrated in Figure 3, data from 16 processing units can be gathered within 4 steps using ParNSGA/DR (left image) and within 2 steps – using algorithm ParNSGA/MPI-OMP (right image). In general gathering of data from p_1 processing units can be done in $\log_2 p_1$ steps using ParNSGA/DR, and within $\log_2 \frac{p_1}{p_2}$ steps using algorithm ParNSGA/MPI-OMP, where p_1 denotes number of processing units per cluster.

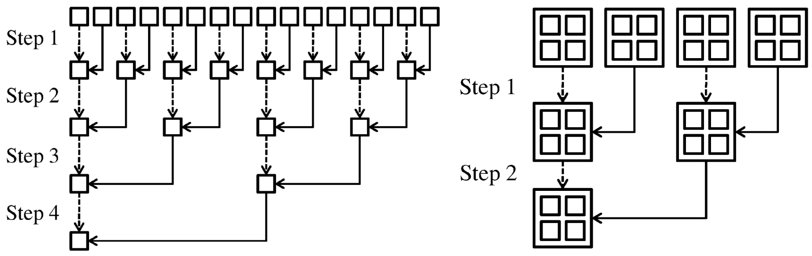


Fig. 3. Scheme of distribution of data using distributed (left) and shared (right) memory parallel programming models

5 Numerical Experiments

In the first instance the performance of two algorithms ParNSGA/DR and ParNSGA/HR which utilize distributed memory parallel programming model has been investigated. The multi-objective competitive facility location problem has been solved using two different sizes of the population: 256 and 512 decision vectors. It was expected to locate 5 new facilities therefore the number of variables of the problem has been 10 as each facility has two coordinates. Later the number of facilities expected to locate has been increased to 25 thus increasing to 50 variables. 250 generations of the genetic algorithm have been performed in each experiment. Depending on population size and the number of facilities expected to locate, duration of sequential algorithm varies from around 18 minutes (locating 5 facilities using population of 256 decision vectors) to more than 12 hours (locating 25 facilities using population of 2048 decision vectors). The number of processing units varies up to the maximum so that the workload (generation of child population and evaluation of objective functions) could be equally divided among all processing units – up to 256 processing units for population of 256 decision vectors and up to 512 processing units for population of 512 decision vectors. The results of the investigation are illustrated by two charts in Figure 4, where the horizontal axis represents the number of processing units and the vertical axis – the speed-up of the algorithm. Using population of 256 decision vectors, better performance gives algorithm ParNSGA/HR which utilizes parallelization of Pareto ranking without additional cost for communication, but forces processors to be idle (left image in Figure 4). When the size of the population has been increased to 512 decision vectors, the performance of both algorithms was similar when 256 processing units have been used, however ParNSGA/DR gives advantage against ParNSGA/HR when 512 processing units have been used (right image in Figure 4). It could be explained as overmuch large idle time of processing units using large population in ParNSGA/HR.

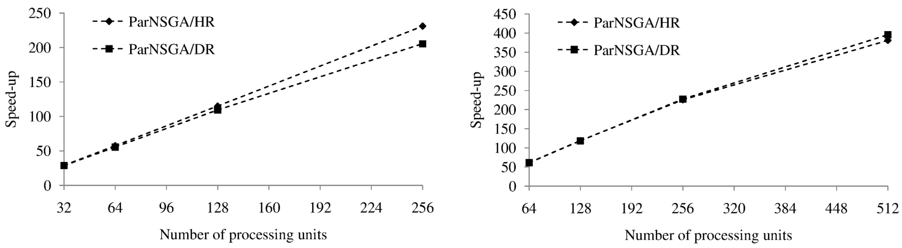


Fig. 4. Comparison of performance of parallel algorithms ParNSGA/DR and ParNSGA/HR using population of 256 (left) and 512 (right) decision vectors

Another experimental investigation has been performed to investigate the influence of utilization of shared memory parallel programing model to the performance of the algorithm. The same multi-objective optimization problem as in previous investigation has been solved using algorithms ParNSGA/DR and ParNSGA/MPI-OMP. Two different populations consisting of 1024 and 2048 decision vectors have been investigated by performing computations on different number of processing units which varies up to 1024 and 2048 depending on the size of the population. Different number of threads per shared memory processing unit (4 and 16 threads) have been also investigated. Results of the investigation are illustrated by two charts in Figure 5, where the horizontal axis represents the number of processing units and the vertical axis – the speed-up

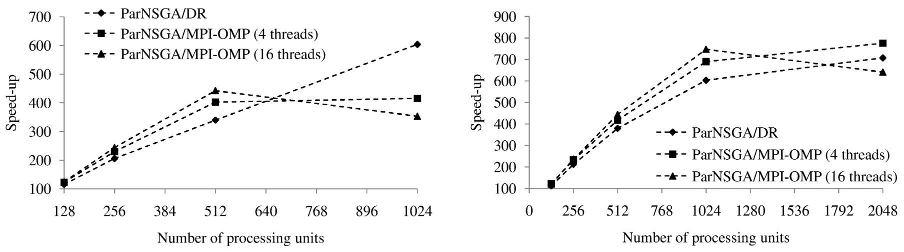


Fig. 5. Comparison of performance of parallel algorithms ParNSGA/DR and ParNSGA/MPI-OMP using population of 1024 (left) and 2048 (right) decision vectors

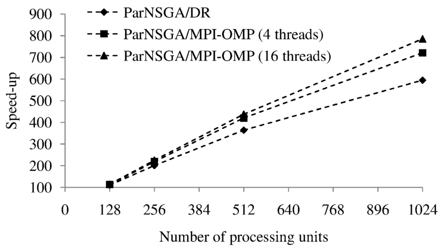


Fig. 6. Comparison of performance of parallel algorithms ParNSGA/DR and ParNSGA/MPI-OMP using population of 1024 decision vectors when number of variables of the problem has been increased to 50

of the algorithm. We can see that utilization of shared memory parallel programing model gives significant advantages to the performance of the algorithm when using up to 512 processing units for population of 1024 decision vectors (left image in Figure 5) and up to 1024 processing units for population of 2048 decision vectors (rigth image in Figure 5). Increasing the number of processing units to the maximum, the performance falls down using either 4 or 16 threads per processing unit if population of 1024 decision vectors has been used, and using 16 threads for population of 2048 decision vectors. Note that using the maximum number of processing units the workload per unit is very small comparing with initialization an synchronization costs using shared memory parallel programming subroutines. Consequently we increased the number of variables of the problem from 10 to 50 thus increasing five times the computational costs per processing unit. The size of the population has been chosen to be 1024 decision vectors and the number of processing units varies up to 1024 as before while investigating the performance of the algorithms with the population of the same size. Results of the investigation are illustrated in Figure 6, where the meanings of the vertical and horizontal axes remain the same as in Figures 4–5. From the results we can see that hybrid MPI-OpenMP algorithm ParNSGA/MPI-OMP gives significant advantage, moreover using more threads per shared memory processing unit, the hybrid algorithm was more superior comparing with the algorithm which utilizes distributed memory parallel programing model only.

6 Conclusions

Parallel evolutionary Non-dominated Sorting Genetic Algorithm has been developed in the paper. Three different versions of the parallel algorithm have been experimentally investigated utilizing both the distributed and shared memory parallel computers. The results of the investigation show that utilization of the shared memory parallel programming model gives significant advantage to the performance of the algorithm when workload per processing unit is large enough comparing with costs of communication between processing units.

Acknowledgments. This research was funded by a Grant (No. MIP-063/2012) from the Research Council of Lithuania.

The first author also wants to express his appreciation to HPC-Europa programme enabling the use of high performance computers at EPCC.

References

1. Hotelling, H.: Stability in competition. Economic Journal 39, 41–57 (1929)
2. Eiselt, H., Laporte, G.: Sequential location problems. European Journal of Operational Research 96, 217–231 (1996)
3. Eiselt, H., Laporte, G., Thisse, J.F.: Competitive location models: a framework and bibliography. Transportation Science 27(1), 44–54 (1993)
4. Plastria, F.: Static Competitive Facility Location: An Overview of Optimisation Approaches. European Journal of Operational Research 129(3), 461–470 (2001)
5. Sáiz, M.E., Hendrix, E.M.T., Fernández, J., Pelegrín, B.: On a branch-and-bound approach for a Huff-like Stackelberg location problem. OR Spectrum 31, 679–705 (2009)
6. Tóth, B., Fernández, J., Pelegrín, B., Plastria, F.: Sequential versus simultaneous approach in the location and design of two new facilities using planar Huff-like models. Journal of Operational Research 36(5), 1393–1405 (2009)
7. Huff, D.L.: Defining and estimating a trade area. Journal of Marketing 28, 34–38 (1964)
8. Huff, D.L.: A programmed solution for approximating an optimum retail location. Land Economics 42, 293–303 (1966)
9. Drezner, T.: Locating a single new facility among existing unequally attractive facilities. Journal Regional Science 34(2), 237–252 (1994)
10. Plastria, F.: Profit maximising single competitive facility location in the plane. Studies in Locational Analysis 11, 115–126 (1997)
11. Plastria, F., Carrizosa, E.: Optimal location and design of a competitive facility. Mathematical Programming 100, 247–265 (2004)
12. Redondo, J.L., Fernández, I.G., Ortigosa, P.: Heuristics for the facility location and design (1j1)-centroid problem on the plane. Computational Optimization and Applications 45(1), 111–141 (2010)

13. Srinivas, N., Deb, K.: Multi-Objective Function Optimization Using Non-dominated Sorting Genetic Algorithms. Evolutionary Computation 2(3), 221–248 (1995)
14. Mitra, K., Deb, K., Gupta, S.K.: Multiobjective Dynamic Optimization of an Industrial Nylon 6 Semibatch Reactor Using Genetic Algorithms. Journal of Applied Polymer Science 69(1), 69–87 (1998)
15. Weile, D.S., Michielssen, E., Goldberg, D.E.: Genetic Algorithm Design of pareto-optimal Broad Band Microwave Absorbers. IEEE Transactions on Electromagnetic Compatibility 38(4), 518–525 (1996)
16. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In: Deb, K., Rudolph, G., Lutton, E., Merelo, J.J., Schoenauer, M., Schwefel, H.-P., Yao, X. (eds.) PPSN 2000. LNCS, vol. 1917, pp. 849–858. Springer, Heidelberg (2000)
17. Amdahl, G.M.: Validity of the single-processor approach to achieving large scale computing capabilities. In: AFIPS Conference Proceedings, vol. 30, pp. 483–485 (1967)
18. Durillo, J.J., Nebro, A.J., Luna, F., Alba, E.: A study of master-slave approaches to parallelize NSGA-II. In: IEEE International Symposium on Parallel and Distributed Processing, pp. 14–18 (2008)
19. Coello, C.C., Lamont, G.B., Veldhuizen, D.A. (eds.): Evolutionary Algorithms for Solving Multi-Objective Problems, 2nd edn. (2007)
20. Lančinskas, A., Žilinskas, J.: Approaches to Parallelize pareto Ranking in NSGA-II Algorithm. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part II. LNCS, vol. 7204, pp. 371–380. Springer, Heidelberg (2012)

Simulation Software for Flow of Fluid with Suspended Point Particles in Complex Domains: Application to Matrix Diffusion

Jukka I. Toivanen[1], Keijo Mattila[2], Jari Hyväluoma[3], Pekka Kekäläinen[1], Tuomas Puurtinen[1], and Jussi Timonen[1]

[1] University of Jyväskylä, FI-40014 University of Jyväskylä, Finland
jukka.i.toivanen@jyu.fi
[2] Federal University of Santa Catarina, 88040-900 Florianópolis, SC, Brazil
[3] MTT Agrifood Research Finland, FI-31600 Jokioinen, Finland

Abstract. Matrix diffusion is a phenomenon in which tracer particles convected along a flow channel can diffuse into porous walls of the channel, and it causes a delay and broadening of the breakthrough curve of a tracer pulse. Analytical and numerical methods exist for modeling matrix diffusion, but there are still some features of this phenomenon, which are difficult to address using traditional approaches. To this end we propose to use the lattice-Boltzmann method with point-like tracer particles. These particles move in a continuous space, are advected by the flow, and there is a stochastic force causing them to diffuse. This approach can be extended to include particle-particle and particle-wall interactions of the tracer. Numerical results that can also be considered as validation of the LBM approach, are reported. As the reference we use recently-derived analytical solutions for the breakthrough curve of the tracer.

1 Introduction

Matrix-diffusion phenomena have been studied for over three decades [1–3], and they have attracted interest due to applications in biological and geological flows, and in the safety assessment of nuclear-waste repositories. This phenomenon arises when a pulse of tracer molecules is advected along a flow channel with porous walls. Tracer molecules diffuse into pores, which affects their breakthrough curve.

Analytical and numerical methods are available for many cases [4, 5], but there are still several features of this phenomenon that are difficult to address, such as, e.g., sorption and anion exclusion. We propose an approach based on the lattice-Boltzmann method (LBM) [6] which can be used to resolve the flow field in complicated geometries, and to track advection and diffusion of individual tracer particles suspended in the fluid. A similar approach has been used, e.g., in [7] for modeling a capillary-filling process of binary fluids that contain nanoparticles.

Before more complicated physics can be added, the approach must be validated. To this end, we consider a model transport system, for which the related mathematical model can be solved semi-analytically [8]. Although the model involves some simplifications, it has been found to be in excellent agreement with experimental results [8].

The rest of the paper is organized as follows. The matrix-diffusion problem considered is described in Section 2. A brief introduction to LBM is given in Section 3, and its parallel implementation is discussed in Section 4. Numerical results of test cases are reported in Section 5, and conclusions are drawn in Section 6.

2 Matrix Diffusion

We consider flow of solvent past a porous matrix of finite depth L_z (in the z direction), in a flow channel or fracture having aperture $2b$ (in the z direction) and width w (in the y direction). Advection of dissolved tracer along the channel is then coupled to its diffusion in the porous matrix. A schematic diagram that represents the situation is shown in Fig. 1. We assume that the fracture extends to infinity in the x direction, but we are interested in the tracer concentration at a given location $x = L$, i.e, the breakthrough curve of the tracer at that point.

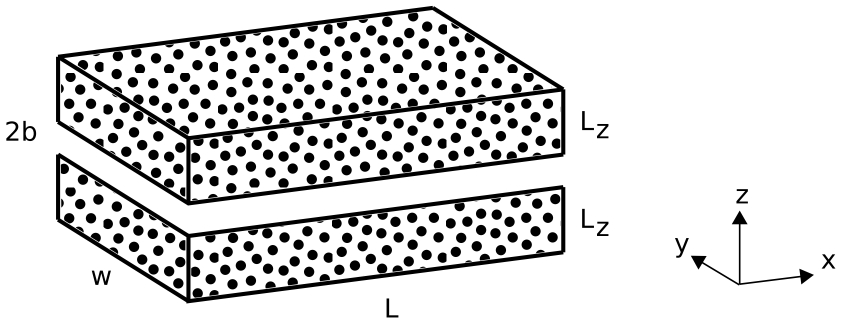


Fig. 1. Schematic diagram of a fracture surrounded by a porous matrix

When advection along the channel is much faster than transport by diffusion in the matrix, we can consider, in leading order, diffusion in the matrix as one dimensional and transverse to the channel, since longitudinal concentration gradients are small inside the matrix independent of the concentration gradients that appear within the channel. As the aperture-to-length ratios and the flow rates are small in situations in which matrix diffusion is important, we can as well assume that tracer concentration within the flow channel is constant across its aperture (a 'well-mixed' flow channel). This is a valid approximation when the time scale related to transverse diffusion of the tracer in the flow channel is short in comparison with its advection time, i.e., when $b << LD_f/\dot{q}$, where D_f is the molecular diffusion coefficient of the tracer in the solvent and $\dot{q}$ is the volumetric flow rate per channel width along the channel. With these assumptions, the tracer concentration in the flow channel, C_f, and in the porous matrix, C_m, can be described in the above transport system by the equations [2, 8]

$$\frac{\partial C_f}{\partial t}(x,t) + v\frac{\partial C_f}{\partial x}(x,t) - D_f\frac{\partial^2 C_f}{\partial x^2}(x,t) = \frac{\epsilon D}{b}\frac{\partial C_m}{\partial z}(x,0,t) \tag{1}$$

$$\frac{\partial C_m}{\partial t}(x,z,t) - D\frac{\partial^2 C_m}{\partial z^2}(x,z,t) = 0, \tag{2}$$

where D is the effective diffusion coefficient of the tracer in the matrix of porosity ϵ and v is the flow velocity in the channel.

The following boundary and initial conditions describe the situation in which a short pulse (amount M_0) of tracer is injected into the flow at the entrance ($x = 0$) of the channel at time $t = 0$:

$$C_m(x,z,0) = 0, \quad \frac{\partial C_m}{\partial z}(x,L_z,t) = 0, \quad C_m(x,0,t) = C_f(x,t) \tag{3}$$

$$C_f(x,0) = 0, \quad C_f(0,t) = \frac{M_0}{2wbv}\delta(t). \tag{4}$$

We look for a bounded solution to the above problem in the half space $x > 0$, which provides the last boundary condition needed for a unique solution. The quantity of interest is C_f at $x = L$, $C_f(L,t)$, i.e., the breakthrough curve of the tracer at that point.

Introducing the dimensionless variables

$$\xi = \frac{x}{L}, \qquad \zeta = \frac{z}{L_z}, \qquad \tau = \frac{tv}{L}, \qquad C(\xi,\tau) = \frac{2wbL}{M_0}C_f(x,t), \tag{5}$$

and solving Eq. (2) for C_m (now in the form $\frac{\partial C_m}{\partial \tau}(\zeta,\tau) - \frac{1}{\kappa^2}\frac{\partial^2 C_m}{\partial \zeta^2}(\zeta,\tau) = 0,\ \tau > 0,\ 0 < \zeta < 1$) by separating the variables, we obtain the following dimensionless form of the problem:

$$\frac{\partial C}{\partial \tau}(\xi,\tau) + \frac{\partial C}{\partial \xi}(\xi,\tau) - \mu^2\frac{\partial^2 C}{\partial \xi^2}(\xi,\tau) = -\lambda\int_0^\tau \Lambda(\tau-\sigma)\frac{\partial C}{\partial \sigma}(\xi,\sigma)\mathrm{d}\sigma. \tag{6}$$

Here

$$\Lambda(\tau) = \frac{2}{\kappa}\sum_{n=0}^{\infty} e^{-(\gamma_n^2/\kappa^2)\tau}, \qquad \gamma_n = (n+\frac{1}{2})\pi, \tag{7}$$

and the behavior of the system is controlled by three dimensionless parameters,

$$\lambda = \epsilon\frac{L}{b}\sqrt{\frac{D}{Lv}}, \qquad \kappa = \frac{L_z}{L}\sqrt{\frac{Lv}{D}}, \qquad \mu = \sqrt{\frac{D_f}{Lv}}. \tag{8}$$

The initial and boundary conditions are

$$C(\xi,0) = 0, \qquad C(0,\tau) = \delta(\tau). \tag{9}$$

3 The Lattice-Boltzmann Method

History of the lattice-Boltzmann method dates back to the lattice-gas automaton proposed in the 1980s for the solution of the Navier-Stokes equation [9], but in LBM the Boolean particle representation is replaced by a statistical one. Using the Chapman-Enskog analysis it can be shown that, in low Mach-number simulations, a lattice-Boltzmann equation (LBE) provides a second-order accurate approximation for the Navier-Stokes equation. The strengths of the method include simplicity of coding, straightforward incorporation of microscopic interactions, and suitability for parallel computing. Moreover, it is relatively straightforward to construct the computation geometry from images, since the method only requires a binary representation of the fluid and solid phases of the domain instead of triangulation of the boundaries or volumes. For this reason the method has gained popularity especially in simulations of flow through porous media.

LBM can be considered as a specific first-order-accurate finite-difference discretization of the discrete-velocity Boltzmann equation (DVBE),

$$\frac{\partial f_i}{\partial t} + \boldsymbol{c}_i \cdot \frac{\partial f_i}{\partial \boldsymbol{r}} = \mathcal{J}_i(\overrightarrow{f}), \quad i = 0, 1, \ldots, q-1, \tag{10}$$

which models the evolution of th mass-density distribution function $\overrightarrow{f}$ in the case where each particle in the system has a velocity that belongs to a discrete set $\{\boldsymbol{c}_0, \boldsymbol{c}_i, \ldots, \boldsymbol{c}_{q-1}\}$. Here $f_i \equiv f(\boldsymbol{r}, \boldsymbol{c}_i, t)$, $\boldsymbol{r}$ is the position, t the time, operator $\mathcal{J}_i$ models the collisions of particles, and $\overrightarrow{f} = (f_0, f_1, f_2, \ldots, f_{q-1})^T$.

3.1 Discretization

The degrees of freedom in LBM are real-valued distribution functions $f_i(\boldsymbol{r}, t)$, where $\boldsymbol{r}$ is now a lattice site. A family of grid models is reported in [10], denoted by DnQm. These models have m discrete velocities in a simple cubic lattice of dimension n. Roughly speaking, accuracy can be gained by increasing the number of local velocities, but this also increases the computational cost.

In the single-relaxation-time approximation, known as the BGK model [11], distribution functions are updated such that

$$f_i(\boldsymbol{r} + \Delta t c_i, t + \Delta t) = f_i(\boldsymbol{r}, t) - \underbrace{\frac{1}{\tau}\left(f_i(\boldsymbol{r}, t) - f_i^{\mathrm{eq}}(\boldsymbol{r}, t)\right)}_{\text{collision}}. \tag{11}$$

Transfer of distribution values to neighboring lattice sites is called the streaming step. In our code this is done using the swap algorithm described in [12]. The last term on the right-hand side of the equation represents the collisions through which the distribution function is relaxed towards a local equilibrium,

$$f_i^{\mathrm{eq}} = \rho(\boldsymbol{r}, t) w_i \left(1 + \frac{\boldsymbol{c}_i \cdot \boldsymbol{v}}{c_s^2} + \frac{(\boldsymbol{c}_i \cdot \boldsymbol{v})^2}{2c_s^4} - \frac{||\boldsymbol{v}||^2}{2c_s^2}\right). \tag{12}$$

Parameter τ is the relaxation time that characterizes the collision process, and is related to the kinematic viscosity of the fluid by the relation $\nu = (2\tau - 1)/6$. Constant c_s is the speed of sound, in this case $c_s = 1/\sqrt{3}$ in lattice units, and w_i are weight factors.

From the macroscopic mass and momentum densities,

$$\rho(\boldsymbol{r}, t) = \sum_i f_i(\boldsymbol{r}, t), \qquad \rho(\boldsymbol{r}, t)\boldsymbol{v}(\boldsymbol{r}, t) = \sum_i \boldsymbol{c}_i f_i(\boldsymbol{r}, t), \tag{13}$$

the macroscopic velocity $\boldsymbol{v}$ can be computed.

3.2 Point-Particle Dynamics

In this work we consider point-like tracer particles without inertia. For a review of applications and implementation techniques of LBM with colloidal particles see [13].

We assume that particles are penetrable to the fluid and exclude the volume interactions, and, assuming that the ratio of the particle mass to the drag coefficient is vanishingly small, we obtain the following stochastic differential equation for the trajectories of the tracer particles [14]:

$$d\boldsymbol{X}(t) = \boldsymbol{v}(\boldsymbol{X}(t))dt + \sqrt{2D}d\boldsymbol{W}(t) + \frac{\boldsymbol{F}(t)}{\xi}dt. \tag{14}$$

Here $\boldsymbol{X}$ is the position of the particle, and the first term on the right-hand side of the equation models advection in the flow field $\boldsymbol{v}(\boldsymbol{X}, t)$. The second term models the Brownian motion of the particle. Here D is the molecular diffusion coefficient, and $\boldsymbol{W}(t)$ is a Gaussian random variable. All other forces are represented by $\boldsymbol{F}$, and ξ is the drag coefficient. The force term can be used to model different kinds of chemical interaction, but, in the simulations discussed here, such forces are not present.

Approximation of Eq. (14) with the Euler method gives the following expression for the updated particle position:

$$\boldsymbol{X}(t + \Delta t_p) = \boldsymbol{X}(t) + \boldsymbol{v}(\boldsymbol{X}(t))\Delta t_p + \sqrt{2D}\Delta\boldsymbol{W}(t) + \frac{\boldsymbol{F}(t)}{\xi}\Delta t_{\mathrm{PD}}. \tag{15}$$

Here the random variable $\Delta\boldsymbol{W}(t)$ has variance $\langle|\Delta\boldsymbol{W}(t)|^2\rangle = 3\Delta t_{\mathrm{PD}}$ in the three-dimensional case [14], and the time step Δt_{PD} is taken to be a fraction of the LBM time step Δt. Instead of the Euler method, one could use, for example, the Heun method [15] which is a weakly second-order convergent predictor-corrector algorithm.

To compute the updated particle position using Eq. (15), the macroscopic velocity $\boldsymbol{v}$ at the position of the particle is required. Particle positions are not restricted to the lattice sites, and thus interpolation of the velocity is used to compute the velocity as follows: Each lattice cell is divided into octants. From the position of the particle we immediately obtain the lattice cell and the octant in which it is located. Next we pre-interpolate the velocity into the corners of that octant. One of the corners is a lattice site, where the velocity is already known. The other corners are located in the middle of an edge or a face of the cell, and one corner is in the cell center. In these corners the velocity is defined to be the average of those in the two, four or eight neighboring lattice sites, respectively. The velocity at the actual particle position is finally computed using trilinear interpolation of the velocities at the corners of the octant.

4 Parallel Implementation (MPI)

Our implementation of LBM uses the D3Q19 grid model. The code is parallelized following the single-program multiple-data (SPMD) paradigm using MPI. We use box-domain decomposition, where the domain can be divided along all three coordinate axes. Our code currently allows only regular decomposition, which greatly simplifies the implementation, but generally results in unbalanced workload among the processors due to varying fluid-node counts. We notice that an efficiency gain of more than 20 percentage points are reported in [16] in which optimized domain decomposition and data-transfer layout for large porous domains were used.

The present implementation is outlined in the following pseudo-code:

```
1:  $t_{\mathrm{LBM}} = 1$
2:  **while** $t_{\mathrm{LBM}} \leq t_{\mathrm{LBM}}^{\max}$ **do**
3:      stream, update density
4:      enforce boundary conditions
5:      communicate new densities
6:      compute density-dependent forces and update velocities
7:      communicate new velocities
8:      $t_{\mathrm{PAR}} = 1$
9:      **while** $t_{\mathrm{PAR}} \leq t_{\mathrm{PAR}}^{\max}$ **do**
10:         update particle positions
11:         communicate particles
12:         $t_{\mathrm{PAR}} = t_{\mathrm{PAR}} + 1$
13:     **end while**
14:     collision
15:     communicate new distributions
16:     $t_{\mathrm{LBM}} = t_{\mathrm{LBM}} + 1$
17: **end while**
```

The LBM update Eq. (11) consists of a completely local collision step that involves only the distribution values of one lattice site, and a streaming step which involves exchange of information between neighboring lattice sites. The streaming step is realized on the subdomain boundary by adding there an extra lattice layer where distribution values are acquired from the neighboring processes via MPI communication.

Tracer particles are divided among the processes according to the same domain decomposition. Since the macroscopic velocities required for the particle update are already computed for the LBM collision step, they are readily available at all lattice sites except the ones belonging to the ghost layer. Velocities at the ghost layer sites are obtained at each time step from neighboring processes via communication.

The data describing the particles are transferred between processes, when particles move between the corresponding subdomains. To this end, a list of particles residing in each cell is maintained, which allows us to communicate only those particles that reside in the overlapping region. Several particle position updates are taken inside one step of LBM, and transfer of particles takes place after each of these steps.

We notice that the choice of the same decomposition for the particle system and LBM can cause load imbalance in case the particles are not evenly distributed between processes, or if transfer rates of particle data between processes differ significantly. The latter condition can arise in heterogeneous domains due to variable fluid velocity. In some geometries, unequal domain decomposition could therefore improve the overall efficiency.

5 Numerical Results

In this Section we report results of numerical computations that have been performed with the code discussed in the previous Section. A schematic diagram of the computation geometry is shown in Fig. 3. It is a realization of the matrix-diffusion problem discussed in Section 2, except that now the flow profile in the channel will automatically be included. As discussed there, we are only concerned of the part of the parameter space, where flow profile does not play a significant role.

5.1 Scalability Tests

First we present results of scalability tests performed on a HP CP4000 BL ProLiant supercluster maintained by CSC – IT Center for Science Ltd.

In practice it is difficult to obtain good parallel performance in matrix-diffusion simulations due to the fact that the distribution of particles is very uneven in the x and z directions, especially at the initial phase of the simulation (initialization mimics a delta pulse). For this reason, in the scalability tests we made the domain decomposition only along the y axis. The following results represent an average over two computations.

In the first test we considered the strong scaling of the solver. The dimensions of the domain were $540 \times 2000 \times 90$, and the number of particles was 200000. Results of this test are shown in Table 1.

The actual efficiency of parallel programs can be better understood by analyzing the algorithm so as to determine the amount of operations it performs and the amount of data that is communicated. In [17] such an analysis has been

Table 1. Strong scaling of the solver. The speed-up (S) and efficiency coefficient (E) for different number of processors.

	12	24	36	48	60	84	108
S(n)	8.04	12.91	22.68	23.93	27.53	37.82	46.84
E(n)	0.67	0.54	0.63	0.50	0.46	0.45	0.43

done for a fluid-dynamics solver based on the finite-volume method, and in [16] an implementation of LBM has been similarly analyzed. In our case it is evident that efficiency will drop slightly when the number of processors is increased. This behavior can be explained by considering the following simple model for the workload and data transfer of the processes.

Since a slice decomposition along the y axis is used, the subdomains are effectively identical, and the number of particles is practically equal among the processes. Moreover, the amount of data communicated between two neighboring processes is independent of the number of processes. We can estimate the computation time, when using $n > 1$ processors, such that

$$T(n) \approx \frac{n_{\mathrm{it}}}{n}\left(N_{\mathrm{f}}t_{\mathrm{f}} + \frac{\Delta t}{\Delta t_{\mathrm{PD}}}Pt_{\mathrm{p}}\right) + n_{\mathrm{it}}t_{\mathrm{d}} \equiv a/n + b, \tag{16}$$

where n_{it} is the number of LBM iterations, N_{f} the number of fluid sites, t_{f} the time needed for one fluid-site update, $\Delta t/\Delta t_{\mathrm{PD}}$ the number of particle time steps within an LBM time step, P the number of particles, t_{p} the time needed for one particle update, and t_{d} the time needed for communication.

Estimates for t_{f} and t_{p} could be obtained by performing computations on a single processor. An estimate for t_{d} would be more difficult to obtain, since in addition to the exact amount of data that is transferred, it depends on the data-transfer rate and latency of the interprocessor communication network. It has been proposed in [16] to measure these quantities using specialized software. We chose, however, to fit observed speed-up data by the model Eq. (16) using a and b as the fitting parameters.

If we, furthermore, replace $T(1)$ by a constant absorbed in parameters a and b (its value would be absorbed in these parameters anyway), the speed-up can be expressed in the form

$$S(n) = \frac{T(1)}{T(n)} \approx \frac{1}{A/n + B}. \tag{17}$$

We found $A = 1.6$ and $B = 0.0070$ by making a least squares fit with this model to the measured data shown in Table 1. The fit and the measured values are shown in Fig. 2. The model predicts saturation of the speed-up, which is typical of strong scaling, with an upper bound of 143 for this particular geometry.

In the second test we considered weak scaling, where the size of the domain and the number of particles were proportional to the number of processors involved. More precisely, the size of the domain was $540 \times (50n) \times 90$ and the number of

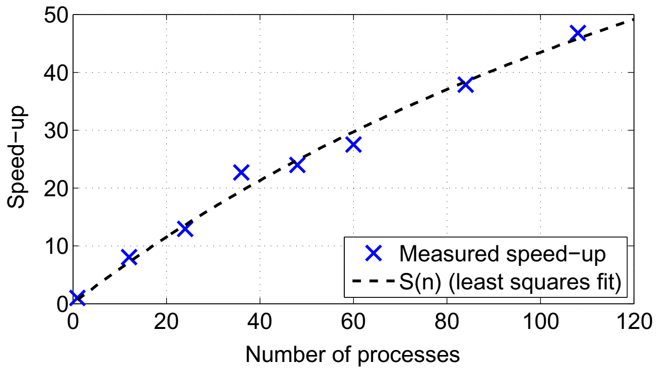


Fig. 2. Speed-up versus the number of processors. Measured values and a least squares fit by the model Eq. (17).

particles was $20000n$. Now an estimate for the computation time can be expressed in the form

$$T(n) \approx n_{\mathrm{it}} \left(N_{\mathrm{f}} t_{\mathrm{f}} + \frac{\Delta t}{\Delta t_{\mathrm{PD}}} P t_{\mathrm{p}} + t_{\mathrm{d}} \right), \tag{18}$$

where N_{f} and P are the number of fluid sites and number of particles per subdomain, respectively. This implies that $T(n)/T(1)$ should remain constant when the number of processors is increased, which is in line with the results shown in Table 2.

Table 2. Weak scaling of the solver

	12	24	36	48
T(n)/T(1)	1.69	1.49	1.58	1.51

5.2 Matrix Diffusion

Next we performed simulations of the matrix diffusion phenomenon in a physical realization of the case described by Eqs. (1)–(4). In Eq. (2) we assumed that diffusion of the tracer in the matrix takes places only in the z direction. Therefore, the matrix used in the computations was constructed so as to allow for 1D diffusion perpendicular to the channel alone, and it is shown in Figure 3.

The width of the channel, w (in the y direction), was assumed to be large, which was modelled by imposing periodic boundary conditions along the y axis. The tracer particles were removed from the simulation when they crossed the boundary of the computation domain. To prevent back-diffusion of particles through the inlet, particles were initialized at some distance downstream. Similarly, the outlet was sufficiently distant from the zone where concentration was measured.

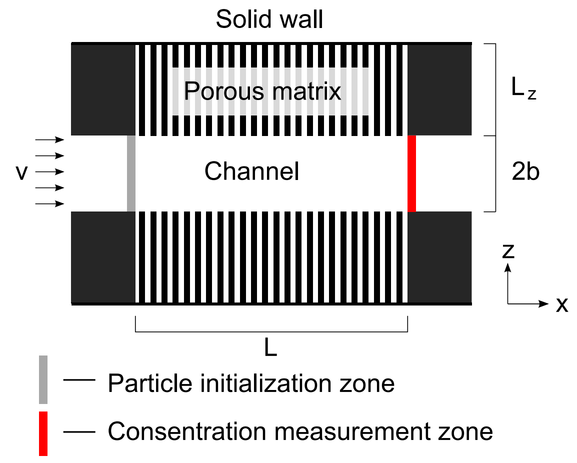


Fig. 3. A schematic diagram of the setup used in the matrix-diffusion simulations

Simulations were performed on a server equipped with 8 Intel Xeon processors, each having 8 cores running at 2.67 GHz. Typical lattice size in the simulations was about $540 \times 9 \times 90$, and the number of particles was 20000. The runtime of such a simulation with 9 processes (decomposition $1 \times 3 \times 3$) was about 48 hours.

Solutions for the system Eqs. (1)–(4) have been reported in [8] in the form of series expansions and convolution integrals. In Fig. 4 we compare the solution for this particular setup (matrix of finite depth) with our simulation results. The parameters λ, κ, and μ (L, K and M in the Figure) vary between the test cases. The simulation results are in good agreement with those of the analytical solution.

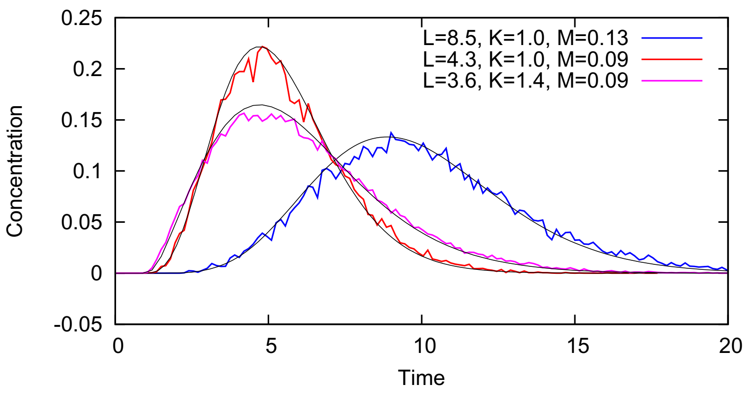


Fig. 4. Results of particle simulations and the related analytical solutions

6 Conclusions

Matrix-diffusion phenomena deal with a pulse of tracer molecules advected along a flow channel with porous walls. Tracer particles diffuse into pores, which affects the breakthrough curve of the tracer. Numerical and analytical methods are available for many cases, but there are still several features of this phenomenon that are difficult to address. To this end we propose an approach based on the lattice-Boltzmann (LB) method, which can be used to resolve the flow field in complicated geometries, and to track the advection and diffusion of individual tracer particles. In this work numerical results for validation of the LB approach were reported together with a description and performance analysis of its parallel implementation. Simulation results were in good agreement with the reference solutions.

A D3Q19 LB code was used for the fluid laden with point-like tracer particles. These particles moved in a continuous space, were advected by the fluid, and there was a stochastic force causing them to diffuse.

Acknowledgments. The scalability tests were performed using the computing resources of CSC - IT Center for Science Ltd.

References

1. Foster, S.: The Chalk groundwater tritium anomaly – A possible explanation. Journal of Hydrology 25(1-2), 159–165 (1975)
2. Neretnieks, I.: Diffusion in the rock matrix: An important factor in radionuclide retardation? Journal of Geophysical Research 85(B8), 4379–4397 (1980)
3. Bodin, J., Delay, F., de Marsily, G.: Solute transport in a single fracture with negligible matrix permeability: 1. fundamental mechanisms. Hydrogeology Journal 11(4), 418–433 (2003)
4. Barten, W., Robinson, P.: Contaminant transport in fracture networks with heterogeneous rock matrices: The PICNIC code. Technical Report NTB 01-03, NAGRA (February 2001) ISSN: 1015-2636
5. McDermott, C., Walsh, R., Mettier, R., Kosakowski, G., Kolditz, O.: Hybrid analytical and finite element numerical modeling of mass and heat transport in fractured rocks with matrix diffusion. Computational Geosciences 13(3), 349–361 (2009)
6. Succi, S.: The Lattice Boltzmann Equation for Fluid Dynamics and Beyond. Clarendon Press, Oxford (2001)
7. Ma, Y., Bhattacharya, A., Kuksenok, O., Perchak, D., Balazs, A.C.: Modeling the transport of nanoparticle-filled binary fluids through micropores. Langmuir 28(31), 11410–11421 (2012)
8. Kekäläinen, P., Voutilainen, M., Poteri, A., Hölttä, P., Hautojärvi, A., Timonen, J.: Solutions to and validation of matrix-diffusion models. Transport in Porous Media 87, 125–149 (2011)
9. Frisch, U., Hasslacher, B., Pomeau, Y.: Lattice-gas automata for the Navier-Stokes equation. Physical Review Letters 56(14), 1505–1508 (1986)

10. Qian, Y.H., D'Humiéres, D., Lallemand, P.: Lattice BGK models for Navier-Stokes equation. Europhysics Letters 17(6), 479–484 (1992)
11. Bhatnagar, P.L., Gross, E.P., Krook, M.: A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. Phys. Rev. 94(3), 511–525 (1954)
12. Mattila, K., Hyväluoma, J., Rossi, T., Aspnäs, M., Westerholm, J.: An efficient swap algorithm for the lattice Boltzmann method. Computer Physics Communications 176(3), 200–210 (2007)
13. Ladd, A.J.C., Verberg, R.: Lattice-Boltzmann simulations of particle-fluid suspensions. Journal of Statistical Physics 104(5), 1191–1251 (2001)
14. Verberg, R., Yeomans, J.M., Balazs, A.C.: Modeling the flow of fluid/particle mixtures in microchannels: Encapsulating nanoparticles within monodisperse droplets. The Journal of Chemical Physics 123(22), 224706 (2005)
15. Szymczak, P., Ladd, A.J.C.: Boundary conditions for stochastic solutions of the convection-diffusion equation. Physical Review E 68(3), 036704 (2003)
16. Vidal, D., Roy, R., Bertrand, F.: On improving the performance of large parallel lattice Boltzmann flow simulations in heterogeneous porous media. Computers & Fluids 39(2), 324–337 (2010)
17. Starikovičius, V., Čiegis, R., Iliev, O.: A parallel solver for the design of oil filters. Mathematical Modelling and Analysis 16(2), 326–341 (2011)

Part VI

HPC Interval Methods

Introduction to the "HPC Interval Methods" Minisymposium

Bartłomiej Jacek Kubica

Institute of Control and Computation Engineering, Warsaw University of Technology, Poland
bkubica@elka.pw.edu.pl

1 Introduction

Interval methods are an approach to performing numerical computations of various kinds, in the presence of errors and uncertainty. They deal with the imprecision caused by several reasons, including:

– numerical inaccuracy,
– discretization errors,
– inexact data (e.g., results of measurements),
– human-related uncertainty (e.g., precise description of decision-makers' preferences),
– ...

The origin of interval computations is attributed to Ramon E. Moore, usually [7]; precursors include Norbert E. Wiener and Mieczysław Warmus.

Initial considerations were related to numerical inaccuracy, mostly. However, as for floating-point computations, the traditional Wilkinson's error analysis [11] is sufficient, usually (not always, see e.g. the Rump's example in [2]), there are several other kinds of imprecision, where using intervals is inevitable.

Particular applications of interval methods include:

– solving systems of linear equations with uncertain parameters,
– seeking solutions of nonlinear equations and systems of equations,
– optimization of nonconvex functions,
– approximating Pareto sets of multi-criterion problems,
– solving ordinary and partial differential equations,
– ...

2 Basics of Interval Arithmetic

Now, we shall define some basic notions of intervals and their arithmetic. We follow a widely acknowledged standards (cf. e.g. [2], [4], [6], [7], [8], [10]).

We define the (closed) interval $[\underline{x}, \overline{x}]$ as a set $\{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}$.

Following [5], we use boldface lowercase letters to denote interval variables, e.g., $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, and $\mathbb{IR}$ denotes the set of all real intervals.

We design arithmetic operations on intervals so that the following condition was fulfilled: if we have $\odot \in \{+, -, \cdot, /\}$, $a \in \mathbf{a}$, $b \in \boldsymbol{b}$, then $a \odot b \in \mathbf{a} \odot \boldsymbol{b}$. The actual formulae for arithmetic operations (see e.g., [2], [4], [6], [10]) are as follows:

$$\begin{aligned}
[\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] &= [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \ , \\
[\underline{a}, \overline{a}] - [\underline{b}, \overline{b}] &= [\underline{a} - \overline{b}, \overline{a} - \underline{b}] \ , \\
[\underline{a}, \overline{a}] \ \cdot \ [\underline{b}, \overline{b}] &= [\min(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}), \max(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b})] \ , \\
[\underline{a}, \overline{a}] \ / \ [\underline{b}, \overline{b}] &= [\underline{a}, \overline{a}] \cdot \left[1 / \overline{b}, 1 / \underline{b}\right] \ , \qquad 0 \notin [\underline{b}, \overline{b}] \ .
\end{aligned}$$

Other interval operations can be defined in a similar manner, e.g.:

$$\begin{aligned}
&\exp\left([\underline{a}, \overline{a}]\right) = [\exp(\underline{a}), \exp(\overline{a})] \ , \\
&\ln\left([\underline{a}, \overline{a}]\right) = [\ln(\underline{a}), \ln(\overline{a})] \ , \qquad \underline{x} > 0 \ , \\
&\text{etc.}
\end{aligned}$$

When computing interval operations, we can use *directed rounding* (see, e.g., [6]) and round the lower bound downward and the upper bound upward. This operation, called *outward rounding*, will result in an interval that will be a bit overestimated, but *guaranteed to contain the true result of the real-number operation*. The IEEE 754 standard guarantees that there is a possibility to switch the rounding mode in floating-point computations [9].

The definition of interval vector $\mathbf{x}$, a subset of $\mathbb{R}^n$ is straightforward: $\mathbb{R}^n \supset \mathbf{x} = \mathbf{x}_1 \times \cdots \times \mathbf{x}_n$. Traditionally interval vectors are called *boxes*.

Links between real and interval functions are set by the notion of an *inclusion function*, also called an *interval extension* – see, e.g., [4].

Definition 1. *A function* $\mathsf{f}\colon \mathbb{IR} \to \mathbb{IR}$ *is an* inclusion function *of* $f\colon \mathbb{R} \to \mathbb{R}$*, if for every interval* $\mathbf{x}$ *within the domain of* f *the following condition is satisfied:*

$$\{f(x) \mid x \in \mathbf{x}\} \subseteq \mathsf{f}(\mathbf{x}) \ . \tag{1}$$

The definition is analogous for functions $f\colon \mathbb{R}^n \to \mathbb{R}^m$.

3 Interval Algorithms

The objective of most interval algorithms is to enclose by a – possibly tight – interval, the solution to a problem, e.g., equation, optimization problem, etc. Most interval methods obtain it by subdividing the initial interval of possible values and discarding and/or narrowing subintervals that cannot contain the solution. We can call this class of methods, branch-and-bound type methods – and this class contains classical b&b methods (used, e.g., in optimization [4]), branch-and-prune methods [3], partitioning parameter space (PPS) [10], etc.

In some cases, we do not need branching, as we want to obtain the so-called *hull solution*, only, i.e., obtain one – possibly wide – box, containing all of the solutions. Such problems are considered for linear systems, often [10].

There are also other variants, e.g., finding the "inner solution" of a problem, i.e., the box containing solutions only (but not necessarily all of them). See [10] for a detailed discussion.

How to process boxes in the b&b type methods? Interval approach provides us several useful tools here, like various types of constraint propagation operators and – the most important one – Newton operators.

The interval Newton operator. The interval Newton operator is one of the most powerful achievements of the interval analysis. As the classical (real-valued) Newton operator, it is used to seek roots of nonlinear equations, using the linearization (first-order Taylor expansion). For the unidimensional case, it can be expressed by the following formula:

$$N(\mathbf{x}, \mathsf{f}) = \operatorname{mid}\mathbf{x} - \frac{\mathsf{f}(\operatorname{mid}\mathbf{x})}{\frac{d\mathsf{f}}{dx}(\mathbf{x})} \ . \tag{2}$$

Obviously, this operator returns an interval. There are three possible results:

– $\mathbf{x}$ and $N(\mathbf{x}, \mathsf{f})$ are disjoint – then $f(\cdot)$ has no roots in $\mathbf{x}$,
– $N(\mathbf{x}, \mathsf{f}) \subseteq \mathbf{x}$ – then $f(\cdot)$ is *verified* to have *exactly one* root in $\mathbf{x}$,
– otherwise, all roots of $f(\cdot)$ that belong to $\mathbf{x}$, have to belong to $N(\mathbf{x}, \mathsf{f}) \cap \mathbf{x}$; so we can narrow the investigated interval.

The proof of the above result (and its several variants) can be found in numerous textbooks, e.g., [4], [7], [8].

For multivariate functions, the linearization results in a *system* of linear equations with interval-valued parameters. Consequently, the Newton operator (2) has to be replaced with a more sophisticated one, solving this system. Examples are the Krawczyk operator [8] or the (most commonly used) interval Gauss-Seidel operator (see [4]). They have analogous properties as their unidimensional relative.

4 State of the Art and the Contribution of Our Workshop

4.1 Interval Linear Systems

Interval linear systems are intensively studied by several researchers. Several kinds of the solution set are being investigated – the united solution set, tolerable solution set, controlable solution set, algebraic solutions, etc. See [10] for detailed discussions.

Also, several methods are applied – some direct methods, iterative ones, algorithms based on partitioning parameter space (PPS), etc.

As obtaining a tight enclosure of the solution set is hard (actually, it is known to be an NP-hard problem), some researchers try to apply some meta-heuristics for this task. In particular, Duda and Skalna, participants of our Minisymposium, introduced a presentation "Heterogeneous multi-agent evolutionary system for solving parametric interval linear systems".

4.2 Partial Differential Equations

PDEs are widely used in several branches of science – mechanics, fluid dynamics, quantum physics, etc. Consequently, the topic of their numerical solving is intensively studied by several researchers. In our Minisymposium, we had three PDE-related presentations:

- Tomasz Hoffmann and Andrzej Marciniak "Proper vs. directed interval arithmetic in solving the Poisson equation",
- Andrzej Marciniak and Barbara Szyszka "The central-backward Difference interval method for solving the wave equation",
- Małgorzata Jankowska and Grażyna Sypniewska-Kamińska "Interval finite-difference method for solving the one-dimensional heat conduction problem with heat sources".

4.3 Interval Matrix Operations

Operations on floating-point matrices are the core of several numerical algorithms. Hence, they have been implemented extremely efficiently, using several BLAS packages – both Open Source (e.g., ATLAS BLAS or OpenBLAS) as commercial ones (e.g., Intel MKL).

Interval matrices have a similar importance in interval computations. One might be tempted to try implementing some interval analogs of BLAS libraries, but it is also possible to reduce operations on interval matrices to operations on floating-point ones and thus utilizing classical BLAS libraries.

The presentation of Hong Diep Nguyen, Nathalie Revol and Philippe Théveny "Tradeoffs between accuracy and efficiency for optimized and parallel interval matrix multiplication" summarized several efforts in this area.

4.4 Interval Computations on GPUs

The idea of GPU computing is extensively exploited in several branches of numerical sciences, in recent years. Application of this approach to interval computations was difficult, initially, as early GPU programming tools have not allowed directed rounding.

Modern versions of both most popular GPGPU languages – CUDA and OpenCL – are compatible with IEEE floating-point standards, which makes interval computations quite convenient.

In our Minisymposium, we had two presentations related to GPU interval computing: Marco Nehmeier presented his joined work with Philip-Daniel Beck, titled "Parallel interval Newton method on CUDA" and Grzegorz Kozikowski presented his joint work with Bartlomiej Kubica "Interval arithmetic and automatic differentiation on GPU, using OpenCL".

4.5 Multi-criterion Decision Making

Decision making problems with multiple criteria are ubiquitous. To allow the decision-maker choosing a proper alternative, it is useful to present them the whole set of non-dominated points, i.e., the so-called Pareto frontier.

Interval methods are well-suited to solve this sort of problems (we try to approximate a continuous set of points). However, as for other interval algorithms, optimizing the method (choosing proper interval tools, parameterizing and arranging them) is crucial for its efficiency. The presentation of Kubica and Woźniak “Tuning the interval algorithm for seeking Pareto sets of multi-criteria problems” introduces this topic for a sophisticated, efficient method.

References

1. C-XSC interval library, http://www.xsc.de
2. Hansen, E., Walster, W.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (2004)
3. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. Springer, London (2001)
4. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
5. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis (2002), http://www.mat.univie.ac.at/~neum/software/int/notation.ps.gz
6. Kulisch, U.: Computer Arithmetic and Validity – Theory, Implementation and Applications. De Gruyter, Berlin (2008)
7. Moore, R.E.: Interval Analysis. Prentice-Hall (1966)
8. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press, Cambridge (1990)
9. Pryce, J.D. (ed.): P1788, IEEE Standard for Interval Arithmetic, http://grouper.ieee.org/groups/1788/email/pdfOWdtH2mOd9.pdf
10. Shary, S.P.: Finite-dimensional Interval Analysis. XYZ (2010) (in Russian)
11. Wilkinson, J.H.: Rounding Errors in Algebraic Processes. Prentice Hall, Englewood Cliffs (1963)

Parallel Interval Newton Method on CUDA

Philip-Daniel Beck and Marco Nehmeier

Institute of Computer Science, University of Würzburg
Am Hubland, D 97074 Würzburg, Germany
nehmeier@informatik.uni-wuerzburg.de

Abstract. In this paper we discuss a parallel variant of the interval Newton method for root finding of non linear continuously differentiable functions on the CUDA architecture. For this purpose we have investigated different dynamic load balancing methods to get an evenly balanced workload during the parallel computation. We tested the functionality, correctness and performance of our implementation in different case studies and compared it with other implementations.

Keywords: Interval arithmetic, Interval Newton method, Parallel computing, Load balancing, CUDA, GPGPU.

1 Introduction

In the last years the GPU has come into focus for general purpose computing by the introduction of CUDA (Compute Unified Device Architecture) [11] as well as the open standard OpenCL (Open Computing Language) [8] to exploit the tremendous performance of highly parallel graphic devices.

Both technologies, CUDA as well as OpenCL, have a huge impact onto the world of scientific computing and therefore it is a matter of importance for the interval community to offer their algorithms and methods on these systems. One of the famous algorithms using interval arithmetic is the interval Newton method for which a parallel implementation on CUDA is discussed in this paper.

2 Interval Arithmetic

Interval arithmetic is set arithmetic working on intervals defined as connected, closed and not necessarily bounded subsets of the reals

$$X = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\} \tag{1}$$

where $\underline{x} = -\infty$ and $\overline{x} = +\infty$ are allowed. The set of all possible intervals together with the empty set is denoted $\overline{\mathbb{IR}}$. The basic arithmetic operations on intervals are based on powerset operations:

$$X \circ Y = [\min_{x \in X, y \in Y}(\bigtriangledown(x \circ y)), \max_{x \in X, y \in Y}(\bigtriangleup(x \circ y))] \tag{2}$$

With floating point numbers the value of the lower bound is rounded toward $-\infty$ (symbol $\bigtriangledown$) and the upper bound is rounded toward $+\infty$ (symbol $\bigtriangleup$) to include all possible results of the powerset operation on the real numbers[1]. Continuous functions could be defined in a similar manner [7].

The enclosure of all real results of a basic operation or a function is the fundamental property of interval arithmetic and is called *inclusion property.*

Definition 1 (Inclusion Property). *If the corresponding interval extension* $F : \overline{\mathbb{IR}} \to \overline{\mathbb{IR}}$ *to a real (continuous) function* $f : \mathbb{R} \to \mathbb{R}$ *is defined on an interval* X *it follows:*

$$f(X) \subseteq F(X)$$

2.1 Interval Newton Method

The interval Newton method in Algorithm 1 is one of the famous applications based upon interval arithmetic. Likewise the well known Newton method, it is an iterative method to compute the roots of a function. But it has the benefit that it can — for some functions — compute *all* zeros of a non linear continuously differentiable function $f : \mathbb{R} \to \mathbb{R}$ in the start interval X_0 with guaranteed error bounds and it can provide information about the existence and uniqueness of the roots[2] [5].

The iterative computation of the enclosing interval of a root is defined as

$$X_{n+1} := X_n \cap N(X_n), \qquad n = 0, 1, 2, \dots \tag{3}$$

$$N(X_n) := \mathrm{mid}(X_n) - \frac{F(\mathrm{mid}(X_n))}{F'(X_n)} \tag{4}$$

where F and F' are the corresponding interval extension and derivative to the function f. In this paper we use an extended form of the interval Newton method [5] which will return two distinct intervals for the case $0 \in F'(X_n)$ in (4) for which the computation is performed recursively, see Algorithm 1 line 7 et seq. This has the advantage that we can use the algorithm for non monotonic functions.

With the mean value theorem it can be easily shown that each root of the function f in X_n also lies in X_{n+1} [5] and therefore we have a sequence of nested intervals[3] which means that the algorithm will always converge.

Additionally a verification step could be performed after the computation of Algorithm 1 to check the uniqueness of the enclosed roots, see [5] for details.

[1] Note that monotonicity properties could be used to define the result of an interval operation or function only using the bounds of the input intervals.

[2] Note that for the reason of readability the check for the uniqueness of the roots is not included in Algorithm 1 but is included in our implementation. Basically it is a test if $N(X_n)$ is an inner inclusion of X_n, see [5] for more details.

[3] In the case of $X_{n+1} = X_n$ the interval is bisected and the computation is performed on both distinct intervals recursively, see Algorithm 1 line 13 et seq.

Algorithm 1. INewton

```
Input:
$F$ : function
$X$ : interval
$\epsilon$ : accuracy of enclosing intervals
$zeros$ : list of enclosing intervals
Output: $zeros$
begin
    /* Use Definition 1 to check possible existence of a root      */
    if $0 \notin F(X)$ then
        return $zeros$;
    $c \leftarrow \mathrm{mid}(X)$;
    /* Newton step with extended division; formula (3) and (4)      */
    $(N_1, N_2) \leftarrow F(c)/F'(X)$;
    $N_1 \leftarrow c - N_1$;
    $N_2 \leftarrow c - N_2$;
    $X_1 \leftarrow X \cap N_1$;
    $X_2 \leftarrow X \cap N_2$;
    /* Bisection in case of no improvement                          */
    if $X_1 = X$ then
        $X_1 \leftarrow [\underline{x}, c]$;
        $X_2 \leftarrow [c, \overline{x}]$;
    foreach $i = 1, 2$ do
        /* No root                                                  */
        if $X_i = \emptyset$ then
            continue;
        /* Suitable enclosure of a root                             */
        if $\mathrm{width}(X_i) < \epsilon$ then
            $zeros.append(X_i)$;
        /* Recursive call                                           */
        else
            INewton($F, X_i, \epsilon, zeros$);

    return $zeros$;
end
```

3 Dynamic Load Balancing

The main challenge in the parallelization of the interval Newton method is a good utilization of all parallel processes during the computation. As described in Sec. 2, we can have a bisection of the workload for the cases $X_{n+1} = X_n$ in (3) or $0 \in F'(X_n)$ in (4). On the other hand, a thread will become idle in the case $X_{n+1} = \emptyset$ which means that no root exists in X_n. Hence, static load balancing is probably not the best way to precompute a good distribution of the workload.

Therefore we have investigated an implementation of the parallel interval Newton method on CUDA with four different dynamic load balancing methods to get an evenly balanced workload during the computation.

Blocking Queue [3] is a dynamic load balancing method which uses one queue in the global memory for the distribution of the tasks. The access to the queue is organized by mutual exclusion using the atomic operations `atomicCAS` and `atomicExch` on an `int`-value to realize the lock and unlock functionality, see [6] for more details.

```
__device__ void lock( void ) {
    while( atomicCAS( mutex, 0, 1 ) != 0 );
}

__device__ void unlock( void ) {
    atomicExch( mutex, 0 );
}
```

Listing 1. Lock and unlock functionality

Task Stealing [1,4] is a lock-free dynamic load balancing method which uses a unique global queue for each CUDA thread block [11]. In the case of an empty queue, the thread block will steal tasks from other thread blocks to avoid idleness. To ensure a consistent dequeue functionality with atomic operations, the CUDA function `__threadfence_system` is used during the enqueue.

```
__device__ void pushBottom(T const & v) {
    int localBot = bot;
    buf[localBot] = v;
    __threadfence_system();
    localBot += 1;
    bot = localBot;
    return;
}
```

Listing 2. Enqueue functionality

Distributed Stacks is a lock-free load balancing method using local stacks in shared memory for each thread block and is almost similar to distributed queuing [12]. Dynamic load balancing is only realized between threads of a thread block. In the case of storing an element onto the stack, the atomic operation `atomicAdd` is used to increase the stack pointer. Reading from the stack is realized simultaneously without atomic operations using the thread id `threadIdx.x` to access the elements of the stack. The workload for the thread blocks is statically distributed at the beginning of the parallel computation.

Static Task List [3] is a lock-free method which uses two arrays, the in-array and the out-array, for the dynamic load balancing. In an iteration the in-array is a read-only data-structure containing the tasks. For each task in the in-array a thread is started writing their results in the out-array. After each iteration the in-array and the out-array are swapped[4], see Fig. 1 for more details.

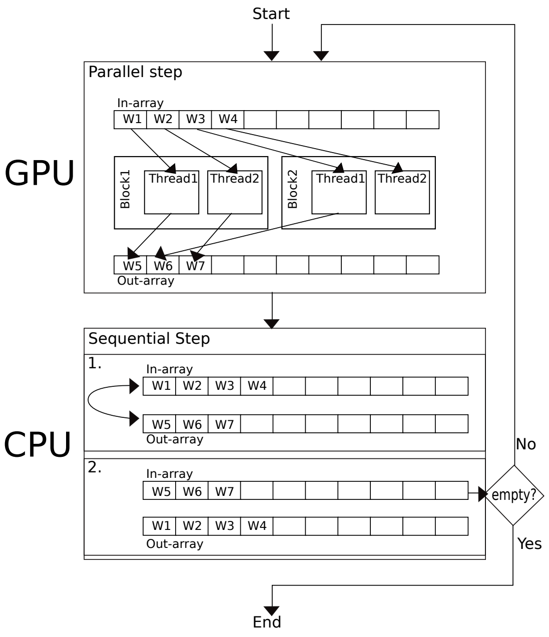


Fig. 1. Static task list

4 Implementation

In our parallel implementation of the interval Newton method on CUDA it was first of all necessary to have interval arithmetic on CUDA. For this purpose we have implemented the required data structures, interval operations, and standard functions in CUDA C.

[4] This means that the kernel is called with swapped pointers for the in-array and the out-array.

For the CUDA C implementation of the interval Newton method we simulated the recursive Algoritm 1 by an iterative CUDA kernel which uses two different concepts depending on the different load balancing methods.

Static task list is the only one of our four used load balancing methods which almost meets the common concept of lightweight threads in CUDA. In our case, this means that the threads are created at the start of the kernel and then only compute one iteration of the parallel interval Newton method. After this iteration all threads are terminated and a new kernel with new threads is started for the next iteration, see Fig. 1 for more details.

For the other three load balancing methods we use so called persistent threads [12] which means that all required threads are started at the beginning of the interval Newton method and keep alive until the algorithm terminates.

The initialization and execution of the CUDA kernels is wrapped in a C++ function which handle the data transfer between the host and the GPU. The specification of the function, which should be analyzed, and their corresponding derivative is done by functors.

5 Performance Tests

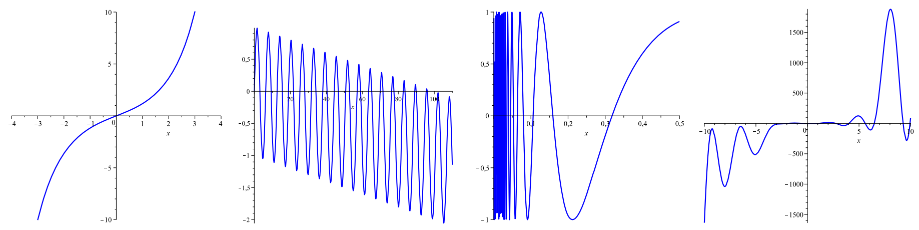
We tested our implementation on a Nvidia Tesla C2070 GPU with CUDA compute capability 2.0 hosted on a Linux Debian 64 Bit System with an Intel Xeon E5504 2.0 GHz CPU and 8 GB Memory.

For all four different implementations we have analyzed the performance for the following functions

$$\begin{aligned}
f_1(x) &= \sinh x \\
f_2(x) &= \sin x - \frac{x}{100} \\
f_3(x) &= \sin x - \frac{x}{10000} \\
f_4(x) &= \sin \frac{1}{x} \\
f_5(x) &= (3x^3 - 5x + 2) \cdot \sin^2 x + (x^3 + 5 \cdot x) \cdot \sin x - 2x^2 - x - 2 \\
f_6(x) &= x^{14} - 539.25 \cdot x^{12} + 60033.8 \cdot x^{10} - 1.77574e^6 \cdot x^8 \\
&\quad + 1.70316e^7 \cdot x^6 - 5.50378e^7 \cdot x^4 + 4.87225e^7 \cdot x^2 - 9.0e^6
\end{aligned}$$

with different thread and block configurations. Additionally we have compared our implementations with a parallel bisection algorithm on CUDA as well as with filib++ [9] and Boost [2] implementations on the CPU.

Prior to the performance tests it was necessary to analyze the best block-grid-ratio of the four different implementations on a GPU to ensure the best possible performance of each implementation. This means that we have measured the



(a) Function f_1 (b) Function f_2 (c) Function f_4 (d) Function f_5

Fig. 2. Sketches of some used functions for the performance tests

runtime with a variable number of threads per block as well as a variable number of blocks per grid on the GPU, see [6].

For our measurements we used configurations with 1 up to 56 blocks using 32, 64, or 128 threads. Thereby the number of 56 blocks is an upper bound which could be computed out of the used memory of our implementations. Table 1 shows the used memory information provided by the `nvcc` compiler using option `-ptxas-options=-v`. Note that for static task list there is no shared memory used due to the fact that there is no communication between the threads.

Table 1. Used memory

Method	Register per thread	Shared memory per Block
BlockingQueue	63	8240 Byte
TaskStealing	63	8240 Byte
DistributedStacks	63	32784 Byte
StaticTaskList	59	0 Byte

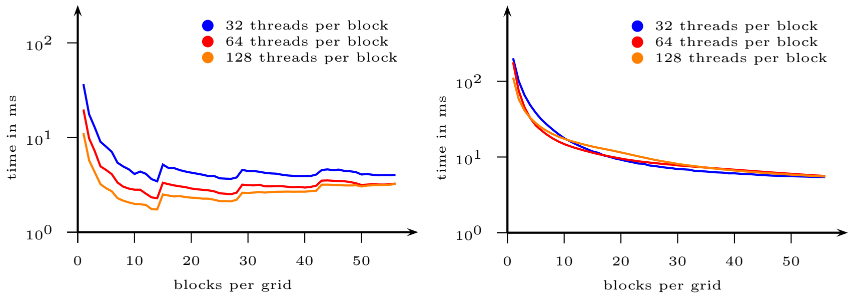
For our runtime tests we used the maximum of 128 threads per block. Additionally, a multiprocessor of a NVIDIA GPU with CUDA compute capability 2.0 is specified with 32768 registers and 48 KB shared memory [11]. Hence we can easily compute

$$\left\lfloor \frac{32768[registers/multiprocessor]}{128[threads/block] * 63[registers/thread]} \right\rfloor = 4[blocks/multiprocessor]$$

which leads to the upper bound of 56 blocks for a Nvidia Tesla C2070 GPU with 14 multiprocessors.

Figure 3 shows some sketches of the performed runtime tests with a variable number of blocks and Tab. 2 shows the best configurations for our test cases.

Note that for static task list we have not measured any difference between 32, 64, or 128 threads per block. Hence, we used 32 threads per block for the other performance tests. Furthermore, the number of blocks per grid is not specified for static task list in Tab. 2 due to the fact that the number of blocks depend on the current workload of each iteration and is adjusted automatically.



(a) Distributed stacks (b) Task stealing

Fig. 3. Sketches of the runtime measurements for function f_3 with a variable number of blocks

Table 2. Block-grid-ratio

Method	Blocks per grid	Threads per block
BlockingQueue	14	64
TaskStealing	28	64
DistributedStacks	14	128
StaticTaskList	-	32

Figure 4 shows the average runtime of 1000 runs for our test cases with double precision and an accuracy of $\epsilon = 10^{-12}$ for the enclosing intervals. It is easily visible that the additional expenses for the computation on the GPU are not worth it for simple functions like f_1 or f_2. In these cases the GPU is outperformed by filib++ or Boost on a CPU. But for harder problems like f_3 or f_4 the GPU, especially with static task list or distributed stacks, dominates filib++ and Boost.

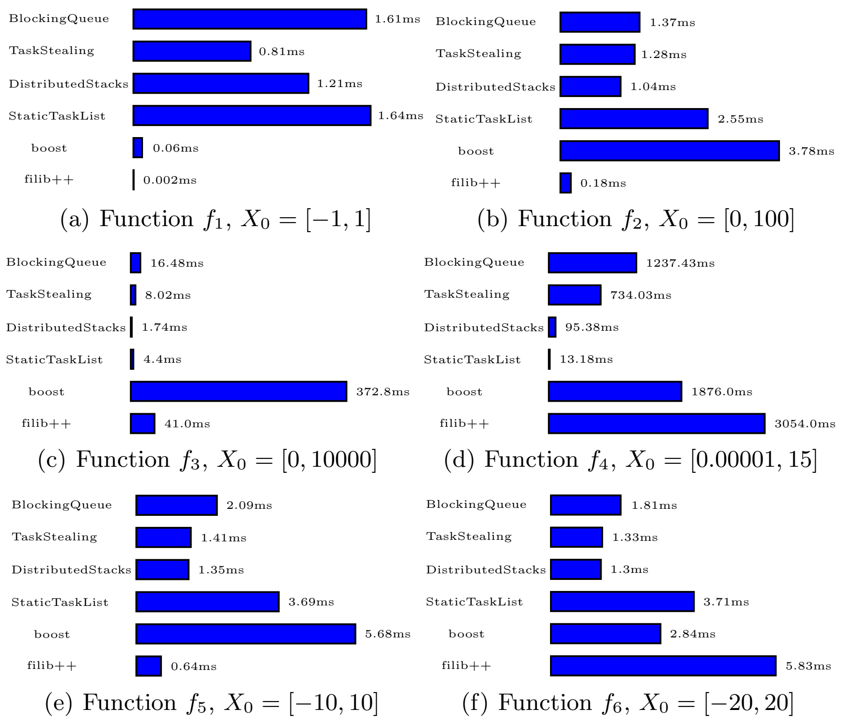
Additional performance tests have shown that there is no significant difference between the runtime with single or double precision, see Fig. 5. This results in the assumption that our implementation is mainly limited by the load balancing and not by the interval arithmetic on the GPU.

Finally we compared a bisection algorithm [5] on a GPU using the same load balancing methods with our interval Newton method. Figure 6 shows some measurements which reflect our observation that the bisection algorithm is outperformed by the interval Newton method for all our test cases.

6 Related Work

In [3] dynamic load balancing on a GPU is discussed for the task of creating an octree partitioning of a set of particles.

[5] Simply it is a branch-and-prune algortihm [7] which only uses the function value and bisection.

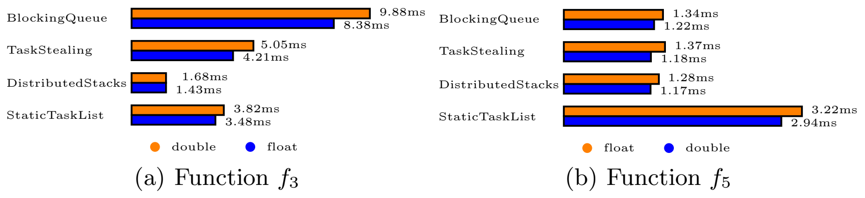


(a) Function f_1, $X_0 = [-1, 1]$ (b) Function f_2, $X_0 = [0, 100]$

(c) Function f_3, $X_0 = [0, 10000]$ (d) Function f_4, $X_0 = [0.00001, 15]$

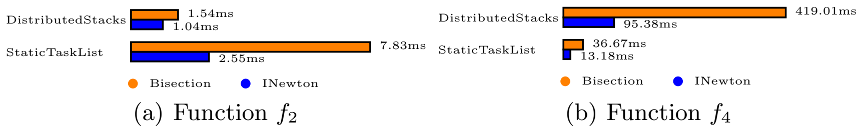
(e) Function f_5, $X_0 = [-10, 10]$ (f) Function f_6, $X_0 = [-20, 20]$

Fig. 4. Average runtime with double precision



(a) Function f_3 (b) Function f_5

Fig. 5. Runtime float vs. double



(a) Function f_2 (b) Function f_4

Fig. 6. Runtime Bisection vs. INewton

Furthermore, in [4] dynamic load balancing for an interval Newton method is analyzed for an implementation on a cluster of workstations using message passing [10].

7 Conclusion

In this paper we have discussed a parallel implementation of an interval Newton method on CUDA and especially different load balancing concepts to utilize the highly parallel CUDA architecture.

Performance analyzations of our approach showed promising results for some hard problems. Especially the two load balancing methods — static task list and distributed stacks — are well suited for complicated functions. Thereby distributed stacks should be preferred for functions with "evenly" distributed roots whereas static task list is more preferable for functions with "unevenly" distributed roots.

Further investigations in the area of parallel interval arithmetic on the GPU as well as on multicore CPU's are planned.

References

1. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1998, pp. 119–129. ACM, New York (1998)
2. Boost Interval Arithmetic Library (November 2012), http://www.boost.org/doc/libs/1_52_0/libs/numeric/interval/doc/interval.htm
3. Cederman, D., Tsigas, P.: On dynamic load balancing on graphics processors. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH 2008, pp. 57–64. Eurographics Association, Aire-la-Ville (2008)
4. Gau, C.-Y., Stadtherr, M.A.: Parallel interval-newton using message passing: dynamic load balancing strategies. In: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM), Supercomputing 2001, pp. 23–23. ACM, New York (2001)
5. Hammer, R., Hocks, M., Kulisch, U., Ratz, D.: C++ Toolbox for Verified Computing I: Basic Numerical Problems. Springer, Heidelberg (1995)
6. Jason Sanders, E.K.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Longman, Amsterdam (2010)
7. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis, 1st edn. Springer, Heidelberg (2001)
8. Khronos OpenCL Working Group. The OpenCL Specification, version 1.1.44 (June 2011)
9. Lerch, M., Tischler, G., Wolff von Gudenberg, J., Hofschuster, W., Krämer, W.: Filib++, a fast interval library supporting containment computations. ACM Trans. Math. Softw. 32(2), 299–324 (2006)

10. Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification (September 2009)
11. NVIDIA. NVIDIA CUDA reference manual, version 3.2 Beta (August. 2010)
12. Tzeng, S., Patney, A., Owens, J.D.: Task management for irregular-parallel workloads on the gpu. In: Doggett, M., Laine, S., Hunt, W. (eds.) High Performance Graphics, pp. 29–37. Eurographics Association (2010)

Heterogeneous Multi-agent Evolutionary System for Solving Parametric Interval Linear Systems

Jerzy Duda and Iwona Skalna

AGH University of Science and Technology, Krakow, Poland
jduda@zarz.agh.edu.pl, skalna@agh.edu.pl

Abstract. The problem of computing the hull, that is the tightest interval enclosure of the solution set for linear systems with parameters being nonlinear functions of interval parameters, is an NP-hard problem. However, since the problem of computing the hull can be considered as a combinatorial or as a constrained optimisation problem, metaheuristic techniques might be helpful. Alas, experiments performed so far show that they are time consuming and their performance may depend on the problem size and structure, therefore some acceleration and stabilisation techniques are required. In this paper, a new approach which rely on a multi-agent system is proposed. The idea is to apply evolutionary method and differential evolution for different agents working together to solve constrained optimisation problems. The results obtained for several examples from structural mechanics involving many parameters with large uncertainty ranges show that some synergy effect of the metaheuristics can be achieved, especially for problems of a larger size.

1 Introduction

The paper addresses the problem of solving large-scale linear algebraic systems whose elements are nonlinear functions of parameters varying within prescribed intervals. Such systems arise, e.g., in reliability and risk analysis of engineering systems. The experiments done so far (see [14]) were designed to test the most popular metaheuristics such as evolutionary algorithm (EA), tabu search (TS), simulated annealing (SA) and differential evolution (DE) for their suitability to solve such problems, especially in the case of many parameters and large uncertainty. The experiments demonstrated that for relatively small problems the considered methods give very accurate results. However, for larger problems the computation time is significant, which limits their usage. To shorten the computation time, the algorithms were run in parallel. This allowed to reduce the total computation time, but further improvements are still required. During the experiments it was found that evolutionary method and differential evolution give significantly better results than the remaining algorithms and depending on the problem characteristic and size either EA or DE was the winning strategy. This suggested to the authors to employ both methods working together as agents and exchanging the best solutions between each other, instead of using parallelised methods independently. The proposed approach is presented in Section 3.

The rest of the paper is organised as follows. Parametric interval linear systems are described in Section 2. In Section 4, computational experiments for illustrative problem instances from structural mechanics with different number of intervals as well as different uncertainty ranges are presented in order to verify the usefulness of the proposed approach. The results for homogenous agents, using either evolutionary method or differential evolution and heterogeneous agents, where half of agents use one method, whereas other half of agents use the other one are provided and compared with the results obtained by single algorithms. The paper ends with concluding remarks.

2 Parametric Interval Linear Systems

Systems of parametric linear equations arise directly in various problems (e.g., balance of forces, of electric current, etc.) Systems of linear equations arise also indirectly in engineering problems through the use of numerical methods, e.g., by discrete solution of differential equations.

A parametric linear systems is a linear system of the form:

$$A(p)x(p) = b(p), \tag{1}$$

where $A(p) = [a_{ij}(p)]$ is an $n \times n$ matrix, $b(p) = [b_j(p)]$ is n-dimensional vector, and $a_{ij}(p)$, $b_i(p)$ are general nonlinear functions of $p = (p_1, \ldots, p_k)^T$ which is a k-dimensional vector of real parameters.

Often, the parameters p_i are unknown which stems, mainly, from the scarcity or lack of data. This kind of uncertainty is recognised as *epistemic* uncertainty and can (or [5] should) be modelled using interval approach, that is using only range information. In the interval approach ([1], [8], [9]), a true unknown value of a parameter p_i is enclosed by an interval $\boldsymbol{p}_i = [\check{p} - \varDelta p, \check{p} + \varDelta p]$, where $\check{p}$ is an approximation of p_i (e.g., resulting from an inexact measurement) and $\varDelta p > 0$ is an upper bound of an approximation (measurement) error. Obviously, appropriate methods are required to propagate interval uncertainties through a calculation (see, e.g., [1], [8], [9]).

Thus, if some of the parameters are assumed to be unknown, ranging within prescribed intervals, $p_i \in \boldsymbol{p}_i$ $(i = 1, \ldots, k)$, the following family of parametric linear system, usually called *parametric interval linear system (PILS)*,

$$A(p)x(p) = b(p),\ p \in \boldsymbol{p} \tag{2}$$

is obtained, where $\boldsymbol{p} = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_k)^T$.

The set of all solutions to the point linear systems from the family (2) is called a *parametric solution set* and is defined as

$$S_{\boldsymbol{p}} = \left\{x \in \mathbb{R}^n \mid \exists\, p \in \boldsymbol{p}\ \ A(p)x = b(p)\right\}. \tag{3}$$

This set is generally of a complicated non-convex structure [2]. In practise, therefore, an interval vector $\boldsymbol{x}^*$, called the *outer solution*, satisfying $S_{\boldsymbol{p}} \subseteq \boldsymbol{x}^*$ is computed. The tightest outer solution, with respect to the inclusion property, is called a *hull solution* (or simply a hull) and is denoted by $\square S_{\boldsymbol{p}}$:

$$\square S_{\boldsymbol{p}} = \bigcap \{\boldsymbol{Y} \mid S_{\boldsymbol{p}} \subseteq \boldsymbol{Y}\}\,.$$

Computing the hull solution is in general case NP-hard [12]. However, the problem of computing the hull can be considered as the family of the following $2n$ constrained optimisation problems:

$$\begin{aligned} \underline{x}_i &= \min\{x(p)_i \mid A(p)x(p) = b(p),\, p \in \boldsymbol{p}\}, \\ \overline{x}_i &= \max\{x(p)_i \mid A(p)x(p) = b(p),\, p \in \boldsymbol{p}\}, \end{aligned} \tag{4}$$

and, therefore, heuristic approach can be used to find very good approximations of the required optima while minimising the computation overhead. Additionally, in this paper it is claimed that the usage of metaheuristic agents strategy allows additional reduction in the computational time.

Theorem 1. *Let $\underline{x}_i$ and $\overline{x}_i$ denote, respectively, the solution of the i-th minimisation and maximisation problem (4). Then, the hull solution*

$$\square S_{\mathbf{p}} = \square\{x(p) : A(p)x(p) = b(p), p \in \boldsymbol{p}\} = [\underline{x}_1, \overline{x}_1] \times ... \times [\underline{x}_n, \overline{x}_n]. \tag{5}$$

3 Methodology

3.1 Evolutionary Multi-agent System

Different strategies can be used in order to compute population-based metaheuristics in parallel. In the so called *global parallelisation model* there is one population, and computation of objective function are done in parallel on slave units [15]. This approach is particularly useful for multicore or multiprocessor architectures where communication cost is almost negligible. In the *island model* the whole population is divided into subpopulations that can be run on different heterogeneous machines. Since in this case communication time is significant, thus the subpopulations are run independently and they occasionally exchange solutions. Finally, in the *master-slave model* there is one central (master) population that communicates with other subpopulations to collect (and use) their best solutions.

When solving parametric interval linear systems, the time spent for computing the objective function significantly dominates the time spent for communication between algorithms, so the island model approach seems to be the most suitable. Agents are run independently and communicate with each other after a given time has elapsed (1-3 seconds). In the preliminary experiments, agents communicate by exchanging their best so far solutions stored in auxiliary files, but in the future more effective communication methods is planned. Three variants of island model have been considered. Two of them were homogeneous multi-agent systems based either on evolutionary method or differential evolution, while the third one was a heterogeneous system with half agents based on one method and the other half based on the other one. In the following sections metaheuristics used by agents are briefly described.

3.2 Evolutionary Optimisation

Population P consists of pop_{size} individuals characterised by k-dimensional vectors of parameters $p_i = (p_{i1}, \ldots, p_{ik})^T$, where $p_{ij} \in \boldsymbol{p}_j$, $i = 1, \ldots, pop_{size}$, $j = 1, \ldots, k$. Elements of the initial population are generated at random based on the uniform distribution. The 10% of the best individuals pass to the next generation and the rest of the population is created by the *non-uniform mutation*

$$p'_j = \begin{cases} p_j + (\overline{p}_j - p_j)\, r^{(1-t/n)^b}, & \text{if } q < 0.5 \\ p_j + (p_j - \underline{p}_j)\, r^{(1-t/n)^b}, & \text{if } q \geqslant 0.5 \end{cases} \tag{6}$$

and *arithmetic crossover*

$$p^{1\prime} = rp^1 + (1-r)p^2, \quad p^{2\prime} = rp^2 + (1-r)p^1 \tag{7}$$

It turned out from numerical experiments [14] that mutation rate r_{mut} should be close to 1 (r_{mut}=0.95), and the crossover rate r_{crs} should be less than 0.3 (r_{crs}=0.25). Population size and the number of generations n depend strongly on the problem size (usually pop_{size} should be set to at least 16 and n to 30). General outline of the EO algorithm is shown in Fig. 1.

```
Initialise $P$ of $pop_{size}$ at random
$j = 0$ /* number of generation */
**while** $(j < n)$ **do**
    Select $P'$ from $P$; Choose parents $p_1$ and $p_2$ from $P'$
    **if** $(r_{[0,1]} < r_{crs})$ **then** Offspring $o_1$ and $o_2 \longleftarrow$ Recombine $p_1$ and $p_2$
    **if** $(r_{[0,1]} < r_{mut})$ **then** Mutate $o_1$ and $o_2$
**end while**
```

Fig. 1. Outline of an evolutionary algorithm

3.3 Differential Evolution

Differential evolution (DE) has been found to be a very effective optimisation method for continuous problems [3]. DE itself can be treated as a variation of evolutionary algorithm, as the method is founded on the same principles such as selection, crossover, and mutation. However, in DE the main optimisation process is focused on the way the new individuals are created. Several strategies for constructing new individuals [11] have been defined. Basic strategy described as $/rand/1/bin$ (which means that vectors for a trial solution are selected in a random way and binomial crossover is then used) creates a mutated individual p_m as follows

$$p_m = p_1 + s \cdot (p_2 - p_3)\ , \tag{8}$$

where s is a *scale parameter* called also an *amplification factor*. After a series of experiments, the best strategy for the problem of solving large PILS appeared to be the strategy described as $/best/2/bin$ (compare [4]). In this strategy a mutated individual (trial vector) is created on the basis of the best solution p_{best} found so far and four other randomly chosen individuals

$$p_m = p_{best} - s \cdot (p_1 + p_2 - p_3 - p_4) \ . \tag{9}$$

The mutated individual p_m is then mixed with the original individual p with a probability C_R using the following *binomial crossover*

$$p'_j = \begin{cases} p_{mj}, & \text{if } r \geqslant C_R \text{ or } j = r_n \\ p_j, & \text{if } r > C_R \text{ and } j \neq r_n \end{cases} , \tag{10}$$

where $r \in [0, 1]$ is a random number and $r_n \in (0, 1, 2, ..., D)$ is a random index ensuring that p'_j is a at least an element obtained by p_{mj}. The following parameters values were taken $s = 0.8$ and $C_R = 0.9$, as the most efficient.

```
Initialise $P$ of $pop_{size}$ at random
while ($i < n$) do
   Do
      Choose at random 4 individuals $p_1$, $p_2$, $p_3$, $p_4$
      Generate mutant $p_m$ from $p_{best}$ and from $p_1$, $p_2$, $p_3$, $p_4$
   While ($p_m$ is not valid)
   $p' \longleftarrow$ Crossover($p, p_m$)
   if ($f(p') > f(p_i)$) then $p_{i+1} \longleftarrow p'$ else $p_{i+1} \longleftarrow p_i$
end while
```

Fig. 2. Outline of a differential evolution algorithm

4 Numerical Experiments

The multi-agent system proposed by the authors has been tested for the three exemplary truss structures, each of different size: four bay two floor truss, five bay six floor truss and ten bay eleven floor truss. Additionally, different levels of uncertainty for the parameters and the load have been considered: 40% for the first truss, 10% for the second, and 6% for the third truss.

Three variants of the island model have been tested for each of the test problems. Each system consisted of 8 independent agents. Number of generations n and population size pop_{size} for both evolutionary computation and differential evolution were set to the same values. For the first problem $n = 300$, $pop_{size} = 30$, for the second problem $n = 100$, $pop_{size} = 20$, and for the third problem n=10, $pop_{size} = 10$.

In order to compare the proposed variants, a measure similar to the overestimation measure described by Popova [10] was used. This time, however, as the algorithms computed the inner interval of the hull solution, the overestimation measure was calculated in the relation to the tightest inner solution, i.e. the worst estimation of the hull solution. The measure can be treated as a relative increase over the tightest inner solution and will be marked as RITIS. For each variant of the multi-agent system the result of the best agent, average result of all agents and the worst result obtained by a given system or algorithm are provided. The same data are listed for the evolutionary optimisation and differential evolution ran as a single algorithm. The overestimation is computed over the worst solution coming from all experiments for a given problem size. Comparison of the results for the first test problem consisting of 15 nodes and 38 elements is presented in Table 1.

Table 1. RITIS measure for four bay two floor truss (higher is better)

Multi-agent system	Largest RITIS	Average RITIS	Smallest RITIS	Computation time [sec.]
HomEO	1.9%	1.8%	1.7%	274
HomDE	1.5%	1.4%	1.4%	289
Heter	2.0%	1.8%	1.6%	298
SingleEO	0.1%	0.0%	0.0%	318
SingleDE	1.4%	1.3%	1.3%	232

Hull approximations obtained by the homogenous agent system based only on the evolutionary optimisation method (HomEO) and the heterogeneous system (Heter) with both EO and DE agents were on average better than the approximations obtained by the other systems and algorithms. The system with heterogeneous agents achieved the best approximation of the hull solution, but on average those two systems performed the same. Algorithms running alone with the same parameters as for the multi-agent systems obtained the worst results, however, the approximations generated by differential evolution were only slightly worse than the results obtained by DE agents working together. Contrary to the agents based on the evolutionary method, the solutions provided by the agents based on DE do not sum up in a simple way.

The results for the second test problem that consisted of 36 nodes and 110 elements are collected in Table 2.

Table 2. RITIS measure for five bay six floor truss (higher is better)

Multi-agent system	Largest RITIS	Average RITIS	Smallest RITIS	Computation time [min.]
HomEO	18.3%	17.0%	14.9%	31.7
HomDE	20.7%	20.1%	19.5%	31.5
Heter	22.9%	19.6%	16.9%	34.0
SingleEO	0.1%	0.0%	0.0%	43.1
SingleDE	19.4%	19.2%	19.0%	21.8

This time the results obtained by the agents in the heterogeneous mutli-agent system (Heter) were similar to the results obtained by the homogenous system with the agents using differential evolution (HomDE). The synergy effect of EO and DE allowed to achieve the best hull approximation, however, on average the heterogeneous system performed a little worse than HomDE.

Finally, Table 3 gathers the results for the largest of all test problems consisting of 120 nodes and 420 elements.

Table 3. RITIS measure for ten bay eleven floor truss (higher is better)

Multi-agent system	Largest RITIS	Average RITIS	Smallest RITIS	Computation time [min.]
HomEO	10.4%	5.7%	1.4%	288
HomDE	31.7%	30.7%	29.8%	291
Heter	33.1%	25.3%	5.1%	289
SingleEO	0.1%	0.1%	0.0%	356
SingleDE	18.7%	18.5%	18.2%	267

For the largest problem considered the homogenous multi-agent system based on differential evolution (HomDE) and the single DE algorithm performed on average better than others systems. Evolutionary optimisation method gave much worse results than differential evolution, thus the agents based on EO could not go hand in hand with the agents using DE method and it caused that heterogeneous agents were on average worse than homogenous DE agents. It is also wort to notice that, unlike previous experiments, approximations obtained by the DE mutli-agent system were significantly better (by 66%) that those generated by the single DE algorithm.

5 Conclusions

Heterogeneous multi-agent evolutionary system for solving parametric interval linear systems has been proposed in the paper. Although some examples of evolutionary multi-agent systems can be found in literature ([7],[6]), the system proposed by the authors can use two different methods that are based on the idea of evolution: evolutionary algorithm and differential evolution. Numerical experiments performed by the authors have shown that the proposed approach can bring a synergy effect of those two metaheuristics. Despite the experiments were computed on a single multiprocessor machine the proposed muti-agent system can be easily applied in distributed computing. This would allow to use more than 8 agents and the differences in the hull approximation between multi-agent systems and single algorithms would be more significant.

Future studies should focus on finding more efficient metaheuristic algorithms for heterogeneous agents, capable to provide good results for the problems of large size, like the third test problem presented in the paper. The authors also plan to test such metaheuristics like ant colony optimisation (ACO) and artificial bee colony (ABC). Also evolutionary method might be improved by introducing some local search algorithms based e.g. on iterated local search (ILS) or variable neighbourhood search (VNS).

Acknowledgements. This work was partly financed within the statutory activity no 11/11.200.232.

References

1. Alefeld, G., Herzberger, J.: Introduction to Interval Computations (transl. by J. Rokne from the original German 'Einführung In Die Intervallrechnung'), pp. xviii–333. Academic Press Inc., New York (1983)
2. Alefeld, G., Kreinovich, V., Mayer, G.: The Shape of the Solution Set for Systems of Interval Linear Equations with Dependent Coefficients. Mathematische Nachrichten 192(1), 23–36 (2006)
3. Dréo, J., Pétrowski, A., Siarry, P., Taillard, E.: Metaheuristics for Hard Optimization. Springer (2006)
4. Duda, J., Skalna, I.: Differential Evolution Applied to Large Scale Parametric Interval Linear Systems. In: Lirkov, I., Margenov, S., Waśniewski, J. (eds.) LSSC 2011. LNCS, vol. 7116, pp. 206–213. Springer, Heidelberg (2012)
5. Ferson, S., Ginzburg, L.R.: Different methods are needed to propagate ignorance and variability. Reliability Engineering and Systems Safety 54, 133–144 (1996)
6. Hanna, L., Cagan, J.: Evolutionary Multi-Agent Systems: An Adaptive and Dynamic Approach to Optimization. Journal of Mechanical Design 131(1), 479–487 (2009)
7. 't Hoen, P.J., de Jong, E.D.: Evolutionary Multi-agent Systems. In: Yao, X., Burke, E.K., Lozano, J.A., Smith, J., Merelo-Guervós, J.J., Bullinaria, J.A., Rowe, J.E., Tiňo, P., Kabán, A., Schwefel, H.-P. (eds.) PPSN 2004. LNCS, vol. 3242, pp. 872–881. Springer, Heidelberg (2004)
8. Moore, R.E.: Interval Analysis. Prentice-Hall, Inc., Englewood Cliffs (1966)
9. Neumaier, A.: Interval Methods for Systems of Equations, pp. xvi–255. Cambridge University Press, Cambridge (1990)
10. Popova, E., Iankov, R., Bonev, Z.: Bounding the Response of Mechanical Structures with Uncertainties in all the Parameters. In: Muhannah, R.L., Mullen, R.L. (eds.) Proceedings of the NSF Workshop on Reliable Engineering Computing (REC), pp. 245–265 (2006)
11. Price, K.S., Rainer, M., Lampinen, J.A.: Differential Evolution. A Practical Approach to Global Optimization. Springer (2005)
12. Rohn, J., Kreinovich, V.: Computing exact componentwise bounds on solutions of linear systems with interval data is NP-hard. SIAM Journal on Matrix Analysis and Application, SIMAX 16, 415–420 (1995)
13. Rump, S.M.: Verification Methods for Dense and Sparse Systems of Equations. In: Herzberger, J. (ed.) Topics in Validated Computations, pp. 63–135. Elsevier Science B. V. (1994)
14. Skalna, I., Duda, J.: A Comparison of Metaheurisitics for the Problem of Solving Parametric Interval Linear Systems. In: Dimov, I., Dimova, S., Kolkovska, N. (eds.) NMA 2010. LNCS, vol. 6046, pp. 305–312. Springer, Heidelberg (2011)
15. Talbi, E.G.: Parallel combinatorial optimization. John Wiley and Sons (2006)

Interval Finite Difference Method for Solving the One-Dimensional Heat Conduction Problem with Heat Sources

Malgorzata A. Jankowska and Grazyna Sypniewska-Kaminska

Poznan University of Technology, Institute of Applied Mechanics
Jana Pawla II 24, 60-965 Poznan, Poland
{malgorzata.jankowska,grazyna.sypniewska-kaminska}@put.poznan.pl

Abstract. The one-dimensional heat conduction equation with the term concerning some heat sources, together with the mixed boundary conditions is considered. Such problems occur in the area of the bioheat transfer and their well-known example is given by the Pennes equation. The paper deals with some interval finite difference method based on the Crank-Nicolson finite difference scheme. In the approach presented, the local truncation error of the conventional method is bounded by some interval values. A method of approximation of such error term intervals is also presented.

Keywords: finite difference methods, interval methods, bioheat transfer.

1 Introduction

Interval methods that are based on some conventional finite difference schemes and take into account also the appropriate local truncation errors represent one of a few alternative approaches (see e.g. Nakao [12]) in the area of numerical methods for solving initial-bounadary value problems for partial differential equations. The most often studied problems seem to be the elliptic ones that correspond to stationary processes. We have the approach based on high-order quadrature and a high-order finite element method utilizing Taylor model methods. It is used by Manikonda, Berz and Makino [7] to obtain verified solutions of the 3D Laplace equation. On the basis of a method of Nakao [11] for elliptic problems, a numerical verification method of solutions for stationary Navier-Stokes equations is in [15]. If we consider parabolic problems that correspond to transport phenomena, then we can give as an example a numerical verification of solutions for nonlinear parabolic equations in one-space dimensional case proposed by Minamoto and Nakao [9]. Nontrivial solutions for the heat convection problems are numerically verified by Nakao, Watanabe, Yamamoto and Nishida in e.g. [13], [16]. Finally, we have hyperbolic problems that belong to wave processes with the papers on numerical verification of solutions for nonlinear hyperbolic equations as proposed by Minamoto [8] and the interval central finite difference method for solving the wave equation as formulated by Szyszka [14].

The interval finite difference methods for solving one-dimensional heat conduction problems were proposed in a number of previous papers. A final form of the interval scheme depends on finite differences used and a kind of boundary conditions that define the problem. We can distinguish the interval method of Crank-Nicolson type [10] for solving the one-dimensional heat conduction equation with the Dirichlet (or first-type) boundary conditions. Then we have the interval method of Crank-Nicolson type for solving the heat conduction equation with the mixed boundary conditions [3] with some kind of the error term approximation proposed in [4]. Evaluation of the accuracy of the interval solution obtained with this method was presented in [5].

Subsequently, we deal with the one-dimensional heat conduction equation with the term concerning heat sources given by a function linearly dependent on the unknown temperature, as defined in Section 2.1. Note that an example of the initial-boundary value problem of the form considered is the bioheat transfer problem expressed with the Pennes equation (see e.g. [17]). The conventional and interval finite difference schemes proposed in Sections 2.2 to 2.3 are based on finite differences used in the approach of Crank-Nicolson method. The modification required takes into account a function that can represent the heat sources term. The interval method proposed deals with the local truncation error of its conventional counterpart. The endpoints of the appropriate error term intervals are approximated by means of a method presented in [4] (see Section 2.4). The results of computations obtained with the interval method are compared with values of the analytical solution in Section 3. Finally, some conclusions are given in Section 4.

2 Interval Method of Crank-Nicolson Type for Solving the Heat Conduction Problem with Heat Sources

2.1 Heat Conduction Problem

We consider the one-dimensional heat conduction equation of the form

$$\frac{\partial u}{\partial t}(x,t) - \alpha^2 \frac{\partial^2 u}{\partial x^2}(x,t) = \alpha_1 + \alpha_2 u(x,t), \quad 0 < x < L,\ t > 0, \tag{1}$$

subject to the initial condition and the Robin (third-type) boundary conditions

$$u(x,0) = f(x), \quad 0 \le x \le L, \tag{2}$$

$$\frac{\partial u}{\partial x}(0,t) - Au(0,t) = \varphi_1(t), \quad t > 0, \tag{3}$$

$$\frac{\partial u}{\partial x}(L,t) + Bu(L,t) = \varphi_2(t), \quad t > 0. \tag{4}$$

Furthermore, we assume that the consistency conditions are also satisfied, i.e. for $t = 0$ we have $\partial u/\partial x\,(0,0) - Af(0) = \varphi_1(0)$ and $\partial u/\partial x\,(L,0) + Bf(L) = \varphi_2(0)$.

We deal with the mixed boundary conditions, only if $A = 0$ or $B = 0$. Note that for $\alpha_1 = 0$ and $\alpha_2 = 0$, the initial-value problem (1)-(4) reduces to the one considered in [3]- [5] and hence, it can be solved with the interval method of Crank-Nicolson type proposed there.

2.2 Conventional Finite Difference Method

In the approach based on finite differences, we first set the maximum time $T_{\max}$ and choose two integers n and m. Then we find the mesh constants h and k such as $h = L/n$ and $k = T_{\max}/m$. Hence we have the grid points (x_i, t_j), such that $x_i = ih$ for $i = 0, 1, \ldots, n$ and $t_j = jk$ for $j = 0, 1, \ldots, m$.

We express the terms of (1) at the grid points $\left(x_i, t_{j+1/2}\right)$, where $t_{j+1/2} = (j + 1/2)\, k$ and we use some finite difference formulas, together with the appropriate local truncation errors (as derived in [3]), for $\partial u/\partial t \left(x_i, t_{j+1/2}\right)$ and $\partial^2 u/\partial x^2 \left(x_i, t_{j+1/2}\right)$. We have as follows

$$\frac{\partial u}{\partial t}(x_i, t_{j+1/2}) = \frac{u(x_i, t_{j+1}) - u(x_i, t_j)}{k} - \frac{k^2}{24}\frac{\partial^3 u}{\partial t^3}\left(x_i, \eta_j^{(1)}\right), \tag{5}$$

and

$$\begin{aligned}\frac{\partial^2 u}{\partial x^2}\left(x_i, t_{j+1/2}\right) = {} & \frac{1}{2h^2}\left[u\left(x_{i-1}, t_j\right) - 2u\left(x_i, t_j\right) + u\left(x_{i+1}, t_j\right)\right. \\ & \left. + u\left(x_{i-1}, t_{j+1}\right) - 2u\left(x_i, t_{j+1}\right) + u\left(x_{i+1}, t_{j+1}\right)\right] \\ & -\frac{h^2}{24}\left[\frac{\partial^4 u}{\partial x^4}\left(\xi_i^{(1)}, t_j\right) + \frac{\partial^4 u}{\partial x^4}\left(\xi_i^{(2)}, t_{j+1}\right)\right] - \frac{k^2}{8}\frac{\partial^4 u}{\partial t^2 \partial x^2}\left(x_i, \eta_j^{(2)}\right),\end{aligned} \tag{6}$$

where $\xi_i^{(1)}, \xi_i^{(2)} \in (x_{i-1}, x_{i+1})$, $\eta_j^{(1)}, \eta_j^{(2)} \in (t_j, t_{j+1})$.

Then, we expand u in the Taylor series about $\left(x_i, t_{j+1/2}\right)$ and evaluate it at (x_i, t_j). We get

$$u\left(x_i, t_j\right) = u\left(x_i, t_{j+1/2}\right) - \frac{k}{2}\frac{\partial u}{\partial t}\left(x_i, t_{j+1/2}\right) + \frac{k^2}{8}\frac{\partial^2 u}{\partial t^2}\left(x_i, \eta_j^{(3)}\right), \tag{7}$$

where $\eta_j^{(3)} \in \left(t_j, t_{j+1/2}\right)$. If we substitute $u\left(x_i, t_{j+1/2}\right)$ from (7) to the equation (1) expressed at $\left(x_i, t_{j+1/2}\right)$, then we have

$$\lambda_1 \frac{\partial u}{\partial t}\left(x_i, t_{j+1/2}\right) - \alpha^2 \frac{\partial^2 u}{\partial x^2}\left(x_i, t_{j+1/2}\right) = \alpha_1 + \alpha_2 u\left(x_i, t_j\right) - \alpha_2 \frac{k^2}{8}\frac{\partial^2 u}{\partial t^2}\left(x_i, \eta_j^{(3)}\right), \tag{8}$$

where $\lambda_1 = 1 - (\alpha_2 k)/2$.

Finally, inserting (5)-(6) to (8) yields

$$\begin{aligned}&-\frac{\lambda_2}{2}u\left(x_{i-1}, t_{j+1}\right) + \lambda_3 u\left(x_i, t_{j+1}\right) - \frac{\lambda_2}{2}u\left(x_{i+1}, t_{j+1}\right) = \frac{\lambda_2}{2}u\left(x_{i-1}, t_j\right) \\ &\qquad + \lambda_4 u\left(x_i, t_j\right) + \frac{\lambda_2}{2}u\left(x_{i+1}, t_j\right) + \alpha_1 k + \widehat{R}_{i,j}, \\ &i = 0, 1, \ldots, n,\ j = 0, 1, \ldots, m-1,\end{aligned} \tag{9}$$

where $\lambda_2 = \left(\alpha^2 k\right)/h^2$, $\lambda_3 = \lambda_1 + \lambda_2$, $\lambda_4 = \lambda_1 - \lambda_2 + \alpha_2 k$ and

$$\begin{aligned}\widehat{R}_{i,j} = {} & \lambda_1 \frac{k^3}{24}\frac{\partial^3 u}{\partial t^3}\left(x_i, \eta_j^{(1)}\right) - \alpha^2 \frac{kh^2}{24}\left[\frac{\partial^4 u}{\partial x^4}\left(\xi_i^{(1)}, t_j\right) + \frac{\partial^4 u}{\partial x^4}\left(\xi_i^{(2)}, t_{j+1}\right)\right] \\ & -\alpha^2 \frac{k^3}{8}\frac{\partial^4 u}{\partial t^2 \partial x^2}\left(x_i, \eta_j^{(2)}\right) - \alpha_2 \frac{k^3}{8}\frac{\partial^2 u}{\partial t^2}\left(x_i, \eta_j^{(3)}\right).\end{aligned} \tag{10}$$

From the initial and boundary conditions we have

$$\begin{aligned}&u(x_i,0)=f(x_i),\ i=0,1,\ldots,n,\\&\frac{\partial u}{\partial x}(0,t_j)-Au(0,t_j)=\varphi_1(t_j),\quad \frac{\partial u}{\partial x}(L,t_j)+Bu(L,t_j)=\varphi_2(t_j),\\&j=1,2,\ldots,m.\end{aligned}$$

Note that for the finite difference scheme (9) some values of u, i.e. $u(x_{-1},t_j)$, $u(x_{n+1},t_j)$, $j=0,1,\ldots,m$ are also needed. Subsequently, we apply the formulas derived in [3]. Then, for $j=0$ we have

$$\begin{aligned}u(x_{-1},t_0)&=-3u(x_0,t_0)+5u(x_1,t_0)-u(x_2,t_0)\\&\quad+\frac{2}{3}h^3\left[\frac{1}{2}\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(1)},t_0\right)+\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(2)},t_0\right)\right], \qquad (11)\\u(x_{n+1},t_0)&=-3u(x_n,t_0)+5u(x_{n-1},t_0)-u(x_{n-2},t_0)\\&\quad-\frac{2}{3}h^3\left[\frac{1}{2}\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(3)},t_0\right)+\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(4)},t_0\right)\right], \qquad (12)\end{aligned}$$

where $\zeta^{(1)}\in(x_{-1},x_1)$, $\zeta^{(2)}\in(x_0,x_2)$, $\zeta^{(3)}\in(x_{n-1},x_{n+1})$, $\zeta^{(4)}\in(x_{n-2},x_n)$. Furthermore, for $j=1,2,\ldots,m$ we take

$$u(x_{-1},t_j)=u(x_1,t_j)-2h\left[Au(x_0,t_j)+\varphi_1(t_j)\right]-\frac{h^3}{3}\frac{\partial^3 u}{\partial x^3}\left(\xi_j^{(L)},t_j\right), \qquad (13)$$

$$u(x_{n+1},t_j)=u(x_{n-1},t_j)-2h\left[Bu(x_n,t_j)-\varphi_2(t_j)\right]+\frac{h^3}{3}\frac{\partial^3 u}{\partial x^3}\left(\xi_j^{(R)},t_j\right), \qquad (14)$$

where $\xi_j^{(L)}\in(x_{-1},x_1)$, $\xi_j^{(R)}\in(x_{n-1},x_{n+1})$.

Now if we transform (9) with (11)-(14), then we get as follows

$$\begin{aligned}&(\lambda_2hA+\lambda_3)u(x_0,t_1)-\lambda_2u(x_1,t_1)=\left(\lambda_4-\frac{3}{2}\lambda_2\right)u(x_0,t_0)+3\lambda_2u(x_1,t_0)\\&\quad-\frac{\lambda_2}{2}u(x_2,t_0)-\alpha^2\frac{k}{h}\varphi_1(t_1)-\alpha^2\frac{kh}{6}\frac{\partial^3 u}{\partial x^3}\left(\xi_1^{(L)},t_1\right)\\&\quad+\alpha^2\frac{kh}{3}\left[\frac{1}{2}\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(1)},t_0\right)+\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(2)},t_0\right)\right]+\alpha_1k+\widehat{R}_{0,0},\\&i=0,\ j=0, \qquad (15)\end{aligned}$$

$$\begin{aligned}&(\lambda_2hA+\lambda_3)u(x_0,t_{j+1})-\lambda_2u(x_1,t_{j+1})=(\lambda_4-\lambda_2hA)u(x_0,t_j)\\&\quad+\lambda_2u(x_1,t_j)-\alpha^2\frac{k}{h}\left[\varphi_1(t_j)+\varphi_1(t_{j+1})\right]\\&\quad-\alpha^2\frac{hk}{6}\left[\frac{\partial^3 u}{\partial x^3}\left(\xi_j^{(L)},t_j\right)+\frac{\partial^3 u}{\partial x^3}\left(\xi_{j+1}^{(L)},t_{j+1}\right)\right]+\alpha_1k+\widehat{R}_{0,j},\\&i=0,\ j=1,2,\ldots,m-1, \qquad (16)\end{aligned}$$

$$\begin{aligned}
&-\frac{\lambda_2}{2}u\left(x_{i-1},t_{j+1}\right)+\lambda_3 u\left(x_i,t_{j+1}\right)-\frac{\lambda_2}{2}u\left(x_{i+1},t_{j+1}\right)=\frac{\lambda_2}{2}u\left(x_{i-1},t_j\right)\\
&\quad+\lambda_4 u\left(x_i,t_j\right)+\frac{\lambda_2}{2}u\left(x_{i+1},t_j\right)+\alpha_1 k+\widehat{R}_{i,j}\\
&i=1,2,\ldots,n-1,\ j=0,1,\ldots,m-1,
\end{aligned} \tag{17}$$

$$\begin{aligned}
&-\lambda_2 u\left(x_{n-1},t_1\right)+\left(\lambda_2 hB+\lambda_3\right)u\left(x_n,t_1\right)=\left(\lambda_4-\frac{3}{2}\lambda_2\right)u\left(x_n,t_0\right)\\
&\quad+3\lambda_2 u\left(x_{n-1},t_0\right)-\frac{\lambda_2}{2}u\left(x_{n-2},t_0\right)+\alpha^2\frac{k}{h}\varphi_2\left(t_1\right)+\alpha^2\frac{hk}{6}\frac{\partial^3 u}{\partial x^3}\left(\xi_1^{(R)},t_1\right)\\
&\quad-\alpha^2\frac{hk}{3}\left[\frac{1}{2}\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(3)},t_0\right)+\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(4)},t_0\right)\right]+\alpha_1 k+\widehat{R}_{n,0},\\
&i=n,\ j=0,
\end{aligned} \tag{18}$$

$$\begin{aligned}
&-\lambda_2 u\left(x_{n-1},t_{j+1}\right)+\left(\lambda_3+\lambda_2 hB\right)u\left(x_n,t_{j+1}\right)=\lambda_2 u\left(x_{n-1},t_j\right)\\
&\quad+\left(\lambda_4-\lambda_2 hB\right)u\left(x_n,t_j\right)+\alpha^2\frac{k}{h}\left[\varphi_2\left(t_j\right)+\varphi_2\left(t_{j+1}\right)\right]\\
&\quad+\alpha^2\frac{hk}{6}\left[\frac{\partial^3 u}{\partial x^3}\left(\xi_j^{(R)},t_j\right)+\frac{\partial^3 u}{\partial x^3}\left(\xi_{j+1}^{(R)},t_{j+1}\right)\right]+\alpha_1 k+\widehat{R}_{n,j},\\
&i=n,\ j=1,2,\ldots,m-1.
\end{aligned} \tag{19}$$

The formulas (15)-(19) can be given in the following matrix representation

$$\begin{aligned}
&Cu^{(1)}=D^{(0)}u^{(0)}+\widehat{E}_C{}^{(1)}+\widehat{E}_L{}^{(1)}, \quad j=0,\\
&Cu^{(j+1)}=D^{(1)}u^{(j)}+\widehat{E}_C{}^{(j+1)}+\widehat{E}_L{}^{(j+1)}, \quad j=1,2,\ldots,m-1,
\end{aligned} \tag{20}$$

where $u^{(j)}=\left[u\left(x_0,t_j\right),\ u\left(x_1,t_j\right),\ \ldots\ ,\ u\left(x_n,t_j\right)\right]^T$.

The matrices of coefficients C, $D^{(0)}$, $D^{(1)}$ in (20) are all tridiagonal. They depend on the stepsizes h, k and the parameters α, α_1, α_2, A, B given in the initial-boundary value problem formulation (1)-(4). Note that if we multiply the equation (17) by 2, then the resultant matrix $\widetilde{C}$ corresonding to C in (20) is symmetric and we can examine its positive definiteness. For sufficiently small values of the stepsizes h and k, we can always find the region such that $\widetilde{C}$ is positive defined. The size of this region depends mainly on the quotient α_2/α^2. The matrices considered are as follows

$$C=\begin{bmatrix}
\lambda_2 hA+\lambda_3 & -\lambda_2 & 0 & \vdots & 0 & 0 & 0\\
-\frac{\lambda_2}{2} & \lambda_3 & -\frac{\lambda_2}{2} & \vdots & 0 & 0 & 0\\
0 & -\frac{\lambda_2}{2} & \lambda_3 & \vdots & 0 & 0 & 0\\
\cdots & \cdots & \cdots & \ddots & \cdots & \cdots & \cdots\\
0 & 0 & 0 & \vdots & \lambda_3 & -\frac{\lambda_2}{2} & 0\\
0 & 0 & 0 & \vdots & -\frac{\lambda_2}{2} & \lambda_3 & -\frac{\lambda_2}{2}\\
0 & 0 & 0 & \vdots & 0 & -\lambda_2 & \lambda_2 hB+\lambda_3
\end{bmatrix}, \tag{21}$$

$$D^{(0)} = \begin{bmatrix} \lambda_4 - \frac{3}{2}\lambda_2 & 3\lambda_2 & -\frac{\lambda_2}{2} & \vdots & 0 & 0 & 0 \\ \frac{\lambda_2}{2} & \lambda_4 & \frac{\lambda_2}{2} & \vdots & 0 & 0 & 0 \\ 0 & \frac{\lambda_2}{2} & \lambda_4 & \vdots & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \vdots & \lambda_4 & \frac{\lambda_2}{2} & 0 \\ 0 & 0 & 0 & \vdots & \frac{\lambda_2}{2} & \lambda_4 & \frac{\lambda_2}{2} \\ 0 & 0 & 0 & \vdots & -\frac{\lambda_2}{2} & 3\lambda_2 & \lambda_4 - \frac{3}{2}\lambda_2 \end{bmatrix}, \tag{22}$$

$$D^{(1)} = \begin{bmatrix} \lambda_4 - \lambda_2 hA & \lambda_2 & 0 & \vdots & 0 & 0 & 0 \\ \frac{\lambda_2}{2} & \lambda_4 & \frac{\lambda_2}{2} & \vdots & 0 & 0 & 0 \\ 0 & \frac{\lambda_2}{2} & \lambda_4 & \vdots & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \vdots & \lambda_4 & \frac{\lambda_2}{2} & 0 \\ 0 & 0 & 0 & \vdots & \frac{\lambda_2}{2} & \lambda_4 & \frac{\lambda_2}{2} \\ 0 & 0 & 0 & \vdots & 0 & \lambda_2 & \lambda_4 - \lambda_2 hB \end{bmatrix}. \tag{23}$$

Now let us consider the vectors of coefficients $\widehat{E_C}^{(j)}$, $j = 1, 2, \ldots, m$, in the formulas (20). They depend on the stepsizes, the problem parameters and the values of the functions φ_1, φ_2. Furthermore, they are different for each $j = 1, 2, \ldots, m$. Similarly as the vectors $\widehat{E_L}^{(j)}$, $j = 1, 2, \ldots, m$. Most importantly the components of $\widehat{E_L}^{(j)}$ represent the local truncation error terms of the conventional finite-difference method at each mesh point. We also note their dependence not only on the stepsizes and the problem parameters but also values of some derivatives of u at the midpoints. The vectors described above are of the following form

$$\widehat{E_C}^{(1)} = \begin{bmatrix} -\alpha^2 \frac{k}{h} \varphi_1(t_1) + \alpha_1 k \\ \alpha_1 k \\ \alpha_1 k \\ \cdots \\ \alpha_1 k \\ \alpha_1 k \\ \alpha^2 \frac{k}{h} \varphi_2(t_1) + \alpha_1 k \end{bmatrix}, \tag{24}$$

$$\widehat{E}_C{}^{(j)} = \begin{bmatrix} -\alpha^2 \frac{k}{h}\left[\varphi_1\left(t_{j-1}\right)+\varphi_1\left(t_j\right)\right]+\alpha_1 k \\ \alpha_1 k \\ \alpha_1 k \\ \cdots \\ \alpha_1 k \\ \alpha_1 k \\ \alpha^2 \frac{k}{h}\left[\varphi_2\left(t_{j-1}\right)+\varphi_2\left(t_j\right)\right]+\alpha_1 k \end{bmatrix}, \tag{25}$$
$$j = 2,3,\ldots,m,$$

$$\widehat{E}_L{}^{(1)} = \begin{bmatrix} -\alpha^2 \frac{kh}{6}\frac{\partial^3 u}{\partial x^3}\left(\xi_1^{(L)},t_1\right)+\alpha^2\frac{kh}{3}\left[\frac{1}{2}\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(1)},t_0\right)+\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(2)},t_0\right)\right]+\widehat{R}_{0,0} \\ \widehat{R}_{1,0} \\ \widehat{R}_{2,0} \\ \cdots \\ \widehat{R}_{n-2,0} \\ \widehat{R}_{n-1,0} \\ \alpha^2 \frac{kh}{6}\frac{\partial^3 u}{\partial x^3}\left(\xi_1^{(R)},t_1\right)-\alpha^2\frac{kh}{3}\left[\frac{1}{2}\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(3)},t_0\right)+\frac{\partial^3 u}{\partial x^3}\left(\zeta^{(4)},t_0\right)\right]+\widehat{R}_{n,0} \end{bmatrix}, \tag{26}$$

$$\widehat{E}_L{}^{(j)} = \begin{bmatrix} -\alpha^2 \frac{kh}{6}\left[\frac{\partial^3 u}{\partial x^3}\left(\xi_{j-1}^{(L)},t_{j-1}\right)+\frac{\partial^3 u}{\partial x^3}\left(\xi_j^{(L)},t_j\right)\right]+\widehat{R}_{0,j-1} \\ \widehat{R}_{1,j-1} \\ \widehat{R}_{2,j-1} \\ \cdots \\ \widehat{R}_{n-2,j-1} \\ \widehat{R}_{n-1,j-1} \\ \alpha^2 \frac{kh}{6}\left[\frac{\partial^3 u}{\partial x^3}\left(\xi_{j-1}^{(R)},t_{j-1}\right)+\frac{\partial^3 u}{\partial x^3}\left(\xi_j^{(R)},t_j\right)\right]+\widehat{R}_{n,j-1} \end{bmatrix}, \tag{27}$$
$$j = 2,3,\ldots,m.$$

Remark 1. Consider the exact formulas (15)-(19) with (10) and the corresponding matrix representation (20) with (21)-(27). Let $u_{i,j}$ approximate $u\left(x_i,t_j\right)$. If we also neglect $\widehat{R}_{i,j}$, $i=0,1,\ldots,n$, $j=0,1,\ldots,m-1$ and all terms in (15)-(19) that contain values of the derivatives of u in some midpoints (similarly as the error terms given in components of $\widehat{E}_L{}^{(j)}$, $j=1,2,\ldots,m$, in reference to (20) with (21)-(27)), then we get the conventional finite difference method with the local truncation error $O\left(h^2+k^2\right)$.

Remark 2. Since the equations (15)-(19) (or (20) with (21)-(27)) with (10) are based on the finite differences used for derivation of the conventional Crank-Nicolson method [1], [6], then we refer to the conventional finite difference method proposed in the paper as the conventional Crank-Nicolson method (for solving the heat conduction problem with heat sources given by a function linearly dependent on the temperature) (i.e. CN-LHS method). Subsequently, we call its interval counterpart formulated in Section 2.3, the interval method of Crank-Nicolson type (i.e. ICN-LHS method).

2.3 Interval Finite Difference Method

The interval approach to (20) with (21)-(27) (or (15)-(19) with (10)) requires additional assumptions about values in midpoints of some derivatives given in the terms of (26)-(27) (or (15)-(19)) with (10) of the local truncation error of the conventional method. Following [3] and [4], we assume that there exists the intervals $M_{i,j}$, $Q_{i,j}$, $N_j^{(L)}$, $N_j^{(R)}$, $P^{(L)}$ and $P^{(R)}$, such that the following relations hold

– for $i = 0, 1, \ldots, n, j = 0, 1, \ldots, m-1$,

$$\frac{\partial^3 u}{\partial t^3}\left(x_i, \eta_j^{(1)}\right) \in M_{i,j}, \tag{28}$$

$$\frac{\partial^4 u}{\partial t^2 \partial x^2}\left(x_i, \eta_j^{(2)}\right) = \frac{1}{\alpha^2}\frac{\partial^3 u}{\partial t^3}\left(x_i, \eta_j^{(2)}\right) \in \frac{1}{\alpha^2} M_{i,j}, \tag{29}$$

– for $i = 0, 1, \ldots, n$,

$$\alpha^2 \frac{\partial^4 u}{\partial x^4}\left(\xi_i^{(1)}, t_j\right) = \frac{\partial^3 u}{\partial t \partial x^2}\left(\xi_i^{(1)}, t_j\right) \in Q_{i,j},\ j = 0, 1, \ldots, m-1, \tag{30}$$

$$\alpha^2 \frac{\partial^4 u}{\partial x^4}\left(\xi_i^{(2)}, t_j\right) = \frac{\partial^3 u}{\partial t \partial x^2}\left(\xi_i^{(2)}, t_j\right) \in Q_{i,j},\ j = 1, 2, \ldots, m, \tag{31}$$

– for $j = 1, 2, \ldots, m$,

$$\alpha^2 \frac{\partial^3 u}{\partial x^3}\left(\xi_j^{(L)}, t_j\right) = \frac{\partial^2 u}{\partial t \partial x}\left(\xi_j^{(L)}, t_j\right) \in N_j^{(L)}, \tag{32}$$

$$\alpha^2 \frac{\partial^3 u}{\partial x^3}\left(\xi_j^{(R)}, t_j\right) = \frac{\partial^2 u}{\partial t \partial x}\left(\xi_j^{(R)}, t_j\right) \in N_j^{(R)}, \tag{33}$$

– for $j = 0$,

$$\alpha^2 \frac{\partial^3 u}{\partial x^3}\left(\zeta^{(1)}, t_0\right) = \frac{\partial^2 u}{\partial t \partial x}\left(\zeta^{(1)}, t_0\right) \in N_0^{(L)}, \tag{34}$$

$$\alpha^2 \frac{\partial^3 u}{\partial x^3}\left(\zeta^{(3)}, t_0\right) = \frac{\partial^2 u}{\partial t \partial x}\left(\zeta^{(3)}, t_0\right) \in N_0^{(R)}, \tag{35}$$

$$\alpha^2 \frac{\partial^3 u}{\partial x^3}\left(\zeta^{(2)}, t_0\right) = \frac{\partial^2 u}{\partial t \partial x}\left(\zeta^{(2)}, t_0\right) \in P^{(L)}, \tag{36}$$

$$\alpha^2 \frac{\partial^3 u}{\partial x^3}\left(\zeta^{(4)}, t_0\right) = \frac{\partial^2 u}{\partial t \partial x}\left(\zeta^{(4)}, t_0\right) \in P^{(R)}. \tag{37}$$

The problem that remains is how to find the intervals that contain a value of $\partial^2 u/\partial t^2\left(x_i, \eta_j^{(3)}\right)$, where $\eta_j^{(3)} \in \left(t_j, t_{j+1/2}\right)$. We assume that for $i = 0, 1, \ldots, n$, $j = 0, 1, \ldots, m-1$, we have

$$\frac{\partial^2 u}{\partial t^2}\left(x_i, \eta_j^{(3)}\right) \in S_{i,j}. \tag{38}$$

Hence, substituting (28)-(37) and (38) to (15)-(19) with (10) we get an interval method of Crank-Nicolson type of the following form

$$\begin{aligned}(\lambda_2 hA+\lambda_3)\,U_{0,1}-\lambda_2 U_{1,1} &= \left(\lambda_4-\frac{3}{2}\lambda_2\right)U_{0,0}+3\lambda_2 U_{1,0}-\frac{\lambda_2}{2}U_{2,0}\\ &\quad -\alpha^2\frac{k}{h}\Phi_1(T_1)-\frac{kh}{6}N_1^{(L)}+\frac{kh}{3}\left(\frac{1}{2}N_0^{(L)}+P^{(L)}\right)+\alpha_1 k+R_{0,0},\\ i=0,\ j=0,\end{aligned} \tag{39}$$

$$\begin{aligned}(\lambda_2 hA+\lambda_3)\,U_{0,j+1}-\lambda_2 U_{1,j+1} &= (\lambda_4-\lambda_2 hA)\,U_{0,j}+\lambda_2 U_{1,j}\\ &\quad -\alpha^2\frac{k}{h}\left[\Phi_1(T_j)+\Phi_1(T_{j+1})\right]-\frac{kh}{6}\left(N_j^{(L)}+N_{j+1}^{(L)}\right)+\alpha_1 k+R_{0,j},\\ i=0,\ j=1,2,\ldots,m-1,\end{aligned} \tag{40}$$

$$\begin{aligned}-\frac{\lambda_2}{2}U_{i-1,j+1}+\lambda_3 U_{i,j+1}-\frac{\lambda_2}{2}U_{i+1,j+1} &= \frac{\lambda_2}{2}U_{i-1,j}\\ &\quad +\lambda_4 U_{i,j}+\frac{\lambda_2}{2}U_{i+1,j}+\alpha_1 k+R_{i,j},\\ i=1,2,\ldots,n-1,\quad j=0,1,\ldots,m-1,\end{aligned} \tag{41}$$

$$\begin{aligned}-\lambda_2 U_{n-1,1}+(\lambda_2 hB+\lambda_3)\,U_{n,1} &= \left(\lambda_4-\frac{3}{2}\lambda_2\right)U_{n,0}+3\lambda_2 U_{n-1,0}-\frac{\lambda_2}{2}U_{n-2,0}\\ &\quad +\alpha^2\frac{k}{h}\Phi_2(T_1)+\frac{kh}{6}N_1^{(R)}-\frac{kh}{3}\left(\frac{1}{2}N_0^{(R)}+P^{(R)}\right)+\alpha_1 k+R_{n,0},\\ i=n,\ j=0,\end{aligned} \tag{42}$$

$$\begin{aligned}-\lambda_2 U_{n-1,j+1}+(\lambda_2 hB+\lambda_3)\,U_{n,j+1} &= \lambda_2 U_{n-1,j}+(\lambda_4-\lambda_2 hB)\,U_{n,j}\\ &\quad +\alpha^2\frac{k}{h}\left[\Phi_2(T_j)+\Phi_2(T_{j+1})\right]+\frac{kh}{6}\left(N_j^{(R)}+N_{j+1}^{(R)}\right)+\alpha_1 k+R_{n,j},\\ i=n,\ j=1,2,\ldots,m-1,\end{aligned} \tag{43}$$

where

$$R_{i,j}=\lambda_1\frac{k^3}{24}M_{i,j}-\frac{kh^2}{24}\left(Q_{i,j}+Q_{i,j+1}\right)-\frac{k^3}{8}M_{i,j}-\alpha_2\frac{k^3}{8}S_{i,j} \tag{44}$$

and

$$U_{i,0}=F(X_i),\quad i=0,1,\ldots,n. \tag{45}$$

Note that X_i, $i=0,1,\ldots,n$, T_j, $j=0,1,\ldots,m$ are intervals such that $x_i\in X_i$ and $t_j\in T_j$. Furthermore, $F=F(X)$, $\Phi_1=\Phi_1(T)$, $\Phi_2=\Phi_2(T)$ denote interval extensions of the functions $f=f(x)$, $\varphi_1=\varphi_1(t)$ and $\varphi_2=\varphi_2(t)$, respectively.

The interval method (39)-(43) with (44) has also the matrix representation given as follows

$$\begin{aligned} &CU^{(1)} = D^{(0)}U^{(0)} + {E_C}^{(1)} + {E_L}^{(1)}, \quad j = 0, \\ &CU^{(j+1)} = D^{(1)}U^{(j)} + {E_C}^{(j+1)} + {E_L}^{(j+1)}, \quad j = 1, 2, \ldots, m-1, \end{aligned} \tag{46}$$

where $U^{(j)} = [U_{0,j},\ U_{1,j},\ \ldots\ ,\ U_{n,j}]^T$ and

$${E_C}^{(1)} = \begin{bmatrix} -\alpha^2 \frac{k}{h} \varPhi_1(T_1) + \alpha_1 k \\ \alpha_1 k \\ \alpha_1 k \\ \cdots \\ \alpha_1 k \\ \alpha_1 k \\ \alpha^2 \frac{k}{h} \varPhi_2(T_1) + \alpha_1 k \end{bmatrix}, \tag{47}$$

$${E_C}^{(j)} = \begin{bmatrix} -\alpha^2 \frac{k}{h} \left[\varPhi_1(T_{j-1}) + \varPhi_1(T_j)\right] + \alpha_1 k \\ \alpha_1 k \\ \alpha_1 k \\ \cdots \\ \alpha_1 k \\ \alpha_1 k \\ \alpha^2 \frac{k}{h} \left[\varPhi_2(T_{j-1}) + \varPhi_2(T_j)\right] + \alpha_1 k \end{bmatrix}, \tag{48}$$
$j = 2, 3, \ldots, m,$

$${E_L}^{(1)} = \begin{bmatrix} -\frac{kh}{6} N_1^{(L)} + \frac{kh}{3}\left(\frac{1}{2} N_0^{(L)} + P^{(L)}\right) + R_{0,0} \\ R_{1,0} \\ R_{2,0} \\ \cdots \\ R_{n-2,0} \\ R_{n-1,0} \\ \frac{kh}{6} N_1^{(R)} - \frac{kh}{3}\left(\frac{1}{2} N_0^{(R)} + P^{(R)}\right) + R_{n,0} \end{bmatrix}, \tag{49}$$

$${E_L}^{(j)} = \begin{bmatrix} -\frac{kh}{6}\left(N_{j-1}^{(L)} + N_j^{(L)}\right) + R_{0,j-1} \\ R_{1,j-1} \\ R_{2,j-1} \\ \cdots \\ R_{n-2,j-1} \\ R_{n-1,j-1} \\ \frac{kh}{6}\left(N_{j-1}^{(R)} + N_j^{(R)}\right) + R_{n,j-1} \end{bmatrix}, \tag{50}$$
$j = 2, 3, \ldots, m.$

In order to explain the meaning of the vectors ${E_C}^{(j)}$, ${E_L}^{(j)}$, $j = 1, 2, \ldots, m$, used in (46), we denote by ${e_C}_i^{(j)}$ and ${E_C}_i^{(j)}$ the components of $\widehat{E_C}^{(j)}$ and ${E_C}^{(j)}$, respectively. Similarly, we donote by ${e_L}_i{}^{(j)}$ and ${E_L}_i{}^{(j)}$ the components of the vectors $\widehat{E}_L{}^{(j)}$ and $E_L{}^{(j)}$. Note that for $E_C{}^{(j)}$ we always have ${e_C}_i{}^{(j)} \in {E_C}_i{}^{(j)}$, $i = 0, 1, \ldots, n$, $j = 1, 2, \ldots, m$. If we further assume that the relations (28)-(38) hold, then the vectors $E_L{}^{(j)}$ are such that ${e_L}_i{}^{(j)} \in {E_L}_i{}^{(j)}$, $i = 0, 1, \ldots, n$, $j = 1, 2, \ldots, m$. Hence, as we see the interval components of the vectors $E_L{}^{(j)}$ are such that a local truncation error of the conventional finite-difference scheme at each mesh point is enclosed in.

Theorem 1. *Let us assume that the local truncation error of the CN-LHS scheme can be bounded by the appropriate intervals at each step. Then, let $F = F(X)$, $\Phi_1 = \Phi_1(T)$, $\Phi_2 = \Phi_2(T)$ denote interval extensions of the functions $f = f(x)$, $\varphi_1 = \varphi_1(t)$, $\varphi_2 = \varphi_2(t)$, given in the initial and boundary conditions (2)-(4) of the heat conduction problem (1)-(4). If $u(x_i, 0) \in U_{i,0}$, $i = 0, 1, \ldots, n$ and the linear system of equations (46) corresponding to the ICN-LHS method (39)-(43) with (44) can be solved with some direct method, then for the interval solutions obtained we have $u(x_i, t_j) \in U_{i,j}$, $i = 0, 1, \ldots, n$, $j = 1, 2, \ldots, m$.*

Remark 3. The correctness of the above theorem can be justified in a similar way as in [3]. Note that according to Theorem 1, with the theoretical formulation (46) with (47)-(50), (44) of the interval method considered, provided the assumptions (28)-(38) can be met, the exact solution of the problem belongs to the interval solution obtained.

2.4 The Error Term Approximation

The effective means of computing values of the endpoints of the error term intervals $M_{i,j}$, $Q_{i,j}$, $N_j^{(L)}$, $N_j^{(R)}$, $P^{(L)}$, $P^{(R)}$ and $S_{i,j}$ such that the relations (28)-(38) hold, is still an open problem and deserves further research. Note that except for some special cases, in practice we usually cannot give values of the endpoints of the error term intervals such as all the appropriate assumptions are met. Hence, we cannot guarantee that the interval solutions obtained are such that they include the exact solution. Nevertheless, we can try to approximate these endpoints in a way proposed in [4] for the intervals $M_{i,j}$, $Q_{i,j}$, $N_j^{(L)}$, $N_j^{(R)}$, $P^{(L)}$, $P^{(R)}$. We follow the similar procedure for the intervals $S_{i,j}$.

We assumed that the relation (38) holds for the intervals $S_{i,j}$, $i = 0, 1, \ldots, n$, $j = 0, 1, \ldots, m-1$. Hence, they are such that for $\eta_j^{(3)} \in \left(t_j, t_{j+1/2}\right)$, the relation (38) hold. We have

$$\frac{\partial^2 u}{\partial t^2}\left(x_i, \eta_j^{(3)}\right) \in S_{i,j} = \left[\underline{S}_{i,j}, \overline{S}_{i,j}\right].$$

We can choose the endpoints $\underline{S}_{i,j}$ and $\overline{S}_{i,j}$ as

$$\underline{S}_{i,j} \approx \min\left(S_{i,j}^*, S_{i,j+1/2}^*\right), \quad \overline{S}_{i,j} \approx \max\left(S_{i,j}^*, S_{i,j+1/2}^*\right), \tag{51}$$

where

$$S_{i,j}^* = \frac{\partial^2 u}{\partial t^2}\left(x_i, t_j\right). \tag{52}$$

For the finite difference approximations of the partial derivative $\partial^2 u/\partial t^2$ we can use the following formulas

$$\begin{aligned}\frac{\partial^2 u}{\partial t^2}\left(x_i,t_j\right)=\frac{1}{k^2}\left[-u\left(x_i,t_{j+3}\right)+4u\left(x_i,t_{j+2}\right)-5u\left(x_i,t_{j+1}\right)\right.\\ \left.+\,2u\left(x_i,t_j\right)\right]+O\left(k^2\right),\end{aligned} \tag{53}$$

$$\begin{aligned}\frac{\partial^2 u}{\partial t^2}\left(x_i,t_j\right)=\frac{1}{12k^2}\left[-u\left(x_i,t_{j+2}\right)+16u\left(x_i,t_{j+1}\right)-30u\left(x_i,t_j\right)\right.\\ \left.+\,16u\left(x_i,t_{j-1}\right)-u\left(x_i,t_{j-2}\right)\right]+O\left(k^4\right),\end{aligned} \tag{54}$$

$$\begin{aligned}\frac{\partial^2 u}{\partial t^2}\left(x_i,t_j\right)=\frac{1}{k^2}\left[2u\left(x_i,t_j\right)-5u\left(x_i,t_{j-1}\right)+4u\left(x_i,t_{j-2}\right)\right.\\ \left.-\,u\left(x_i,t_{j-3}\right)\right]+O\left(k^2\right).\end{aligned} \tag{55}$$

We denote by $u_{i,j}$ the approximation of $u\left(x_i,t_j\right)$ obtained with the conventional scheme and we neglect the error terms in (53)-(55). After that we can use (53)-(55) for approximation of $S^*_{i,j}$, $i=0,1,\ldots,n$, in the following way: the formula (53) for $j=0$; the formula (54) for $j=1,2,\ldots,m-2$; the formula (55) for $j=m-1$.

Remark 4. Note that to get the endpoints $\underline{S}_{i,j}$ and $\overline{S}_{i,j}$ in (51), the exact values or just approximations of $\partial^2 u/\partial t^2$ at the points $\left(x_i,t_j\right)$ and $\left(x_i,t_{j+1/2}\right)$ are required. Hence, at the beginning we can use the conventional method, as proposed in Section 2.2, for the same value of the stepsize h (as for the interval method applied) and for the time stepsize equal to $k/2$ (because the point $\left(x_i,t_{j+1/2}\right)$ is located between $\left(x_i,t_j\right)$ and $\left(x_i,t_{j+1}\right)$). Then, with the approximations $u_{i,j}$, we can apply the formulas (53)-(55) to compute the approximations of $\partial^2 u/\partial t^2$ required. Nevertheless, if the uncertainties of initial values are considerable, then we advise to use the interval realization of the conventional method instead of the conventional one, to find the endpoints $\underline{S}_{i,j}$ and $\overline{S}_{i,j}$. The interval realization produces interval values of solution, denoted by $U^C_{i,j}=\left[\underline{u}^c_{i,j},\overline{u}^c_{i,j}\right]$, such that they include errors of initial data, rounding errors and representation errors, except for the local truncation error of the conventional method. Then we can use left and right endpoints of such intervals (instead of approximations $u_{i,j}$) in the formulas (53)-(55) and choose a minimum (maximum) value of $S^*_{i,j}$, $S^*_{i,j+1/2}$ computed for both $\underline{u}^c_{i,j}$ and $\overline{u}^c_{i,j}$ to find the endpoints $\underline{S}_{i,j}$ $\left(\overline{S}_{i,j}\right)$.

3 Numerical Experiments

We consider the heat conduction problem of the form

$$u_t\left(x,t\right)-\alpha^2 u_{xx}\left(x,t\right)=\alpha_1+\alpha_2 u\left(x,t\right),\quad 0<x<L,\ t>0, \tag{56}$$

subject to the initial and boundary conditions

$$u\left(x,0\right)=3x\left(L-x\right),\quad 0\leq x\leq L, \tag{57}$$

$$u_x\left(0,t\right)-Au\left(0,t\right)=0,\quad u_x\left(L,t\right)+Bu\left(L,t\right)=0,\quad t>0. \tag{58}$$

For such a problem formulation the analytical solution can be derived. We have

$$u\left(x,t\right)=\sum_{n=1}^{\infty}u_n\left(t\right)X_n\left(x\right), \tag{59}$$

where

$$u_n\left(t\right)=\left[g_1\left(\mu_n\right)+g_2\left(\mu_n\right)\right]\exp\left(-\gamma_n t\right)-g_1\left(\mu_n\right), \tag{60}$$

$$X_n\left(x\right)=\sin\left(\mu_n x/L\right)+\left[\mu_n/\left(AL\right)\right]\cos\left(\mu_n x/L\right), \tag{61}$$

and with the notations $c_1=A+B$, $c_2=A^2L^2+\mu_n^2$, we have

$$g_1\left(\mu_n\right)=\frac{2Ac_1L^4\alpha_1\left(Ac_1L\mu_n+Bc_2\sin\mu_n\right)}{\left(L^2\alpha_2-\alpha^2\mu_n^2\right)\left[c_1^2L\mu_n^2\left(AL+c_2\right)+Bc_2^2\sin^2\mu_n\right]}, \tag{62}$$

$$g_2\left(\mu_n\right)=\frac{6Ac_1L^3\left[c_1L\mu_n\left(2AL-\mu_n^2\right)+\left(2BL-\mu_n^2\right)c_2\sin\mu_n\right]}{c_1^2L\mu_n^4\left[AL\left(1+AL\right)+\mu_n^2\right]+B\mu_n^2c_2^2\sin^2\mu_n}. \tag{63}$$

$$\gamma_n=\alpha^2\left(\mu_n/L\right)^2-\alpha_2. \tag{64}$$

Note that μ_n, $n=1,2,\ldots$ in (60)-(64) are positive roots of the equation

$$\frac{\pi}{2}-\arctan\left[-\frac{ABL^2-\mu^2}{\left(A+B\right)L\mu}\right]+\left(n-1\right)\pi=\mu. \tag{65}$$

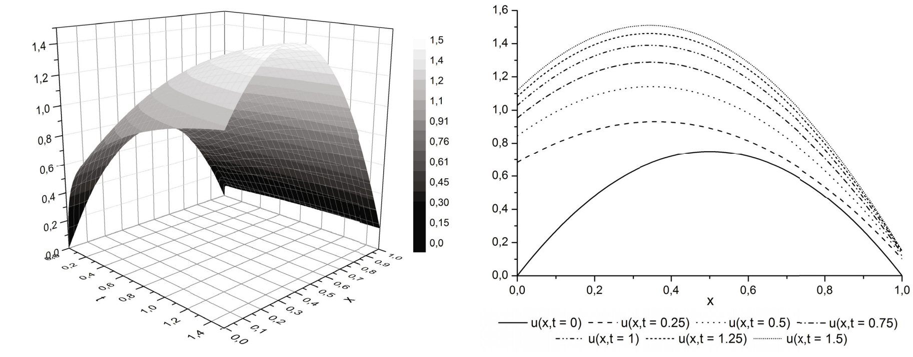
The series proposed in (59) is bounded as t approaches infinity, if the following condition is met

$$\mu_1>L\sqrt{\alpha_2/\alpha^2}. \tag{66}$$

We set $L=1$, $T_{\max}=1.5$. Furthermore, values of the constants depending on a physical problem are specified as follows

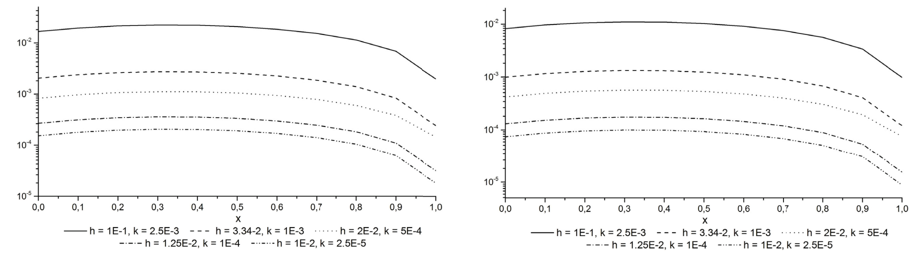
$$\alpha=\sqrt{0.6},\quad \alpha_1=2,\quad \alpha_2=1.5,\quad A=2,\quad B=25. \tag{67}$$

We use the exact solution (59) with (60)-(64) to get the temperature distribution described by (56)-(58) with (67) as shown in Fig. 1. Comparison of the widths of the interval solutions obtained for selected values of h and k is shown in Fig. 2. The widths of the interval solutions $U_{i,j}$ and the components of $E_L^{(j)}$, $j=1,2,...,m$ obtained for $h=$ 1E-2, $k=$ 2.5E-5 are presented in Fig. 3. In Tables 1-2 values of the exact and interval solutions at $t=1.0$, $t=1.5$ are given. For computations we used the C++ libraries for the floating-point conversions and the floating-point interval arithmetic dedicated for the Intel C++ compiler [2].



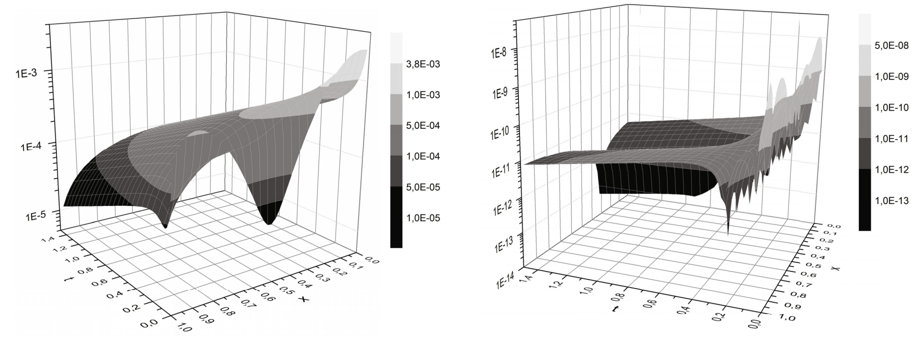
(a) **(b)**

Fig. 1. Temperature distribution described by (56)-(58) with (67) for: a) $t \in [0, 1.5]$; b) selected values of time t



(a) **(b)**

Fig. 2. Widths of the interval solution: a) $U(x, t = 1.0)$; b) $U(x, t = 1.5)$ obtained with the ICN-LHS method for the problem (56)-(58) with (67) for different values of h and k



(a) **(b)**

Fig. 3. Widths of: a) the interval solutions $U_{i,j}$; b) the components of $E_L^{(j)}$, $j = 1, 2, ..., m$ obtained with the ICN-LHS method for the problem (56)-(58) with (67) for $h = 1\text{E-}2$, $k = 2.5\text{E-}5$

Table 1. Values of the exact and interval solutions at $t = 1.0$ obtained with the ICN-LHS method for $h = 1E\text{-}2$ and $k = 2.5E\text{-}5$

x	$u\,(x, t = 1.0)$	$U\,(x, t = 1.0)$	*width*
0.0	+1.0290849E+0	[+1.02901153724382E+0,+1.02916440133155E+0]	1.528641E-04
0.1	+1.2067942E+0	[+1.20671599977288E+0,+1.20689538374457E+0]	1.793840E-04
0.2	+1.3260129E+0	[+1.32593240407807E+0,+1.32612953720827E+0]	1.971331E-04
0.3	+1.3842509E+0	[+1.38417087344458E+0,+1.38437611985331E+0]	2.052464E-04
0.4	+1.3802769E+0	[+1.38020024407569E+0,+1.38040357354486E+0]	2.033295E-04
0.5	+1.3141385E+0	[+1.31406793634956E+0,+1.31425941189625E+0]	1.914755E-04
0.6	+1.1871632E+0	[+1.18710130619728E+0,+1.18727156810621E+0]	1.702619E-04
0.7	+1.0019414E+0	[+1.00189031837399E+0,+1.00203104166227E+0]	1.407233E-04
0.8	+7.6228993E-1	[+7.62251200508054E-1,+7.62355501915610E-1]	1.043014E-04
0.9	+4.7319585E-1	[+4.73170599279248E-1,+4.73233373879811E-1]	6.277460E-05
1.0	+1.4074110E-1	[+1.40729705217429E-1,+1.40747876206010E-1]	1.817099E-05

Table 2. Values of the exact and interval solutions at $t = 1.5$ obtained with the ICN-LHS method for $h = 1E\text{-}2$ and $k = 2.5E\text{-}5$

x	$u\,(x, t = 1.5)$	$U\,(x, t = 1.5)$	*width*
0.0	+1.1183915E+0	[+1.11835154466522E+0,+1.11842581206267E+0]	7.426740E-05
0.1	+1.3116315E+0	[+1.31159315059256E+0,+1.31168030281238E+0]	8.715222E-05
0.2	+1.4412514E+0	[+1.44121523580722E+0,+1.44131101162725E+0]	9.577582E-05
0.3	+1.5042523E+0	[+1.50421881388058E+0,+1.50431853191966E+0]	9.971804E-05
0.4	+1.4991699E+0	[+1.49913960675766E+0,+1.49923839395570E+0]	9.878720E-05
0.5	+1.4261060E+0	[+1.42607931396122E+0,+1.42617234244423E+0]	9.302848E-05
0.6	+1.2867272E+0	[+1.28670429921930E+0,+1.28678702146717E+0]	8.272225E-05
0.7	+1.0842305E+0	[+1.08421166809912E+0,+1.08428003923122E+0]	6.837113E-05
0.8	+8.2327813E-1	[+8.23263209139443E-1,+8.23313884697483E-1]	5.067556E-05
0.9	+5.0989927E-1	[+5.09888175669761E-1,+5.09918675216012E-1]	3.049955E-05
1.0	+1.5136401E-1	[+1.51356399908803E-1,+1.51365228436968E-1]	8.828528E-06

4 Conclusions

The interval method of Crank-Nicolson type for solving the heat conduction problem with heat sources given by a function linearly dependent on the temperature is proposed. Such problem can be formulated with the Pennes equation of the bioheat transfer. Since the physical parameters can often take ranges of values, then the main advantage of the interval method considered is the ability to represent the uncertain values of parameters in the form of intervals. The interval solutions obtained also include the errors caused by the floating-point arithmetic used by computers, i.e. the rounding errors and the representation errors. Furthermore, we propose a method (also based on some finite differences) which can be applied for the approximation of the endpoints of the error term components. Note that such approach is not enough to guarantee the inclusion of the local truncation error in the resultant interval solutions. Nevertheless, the numerical experiments confirm that the exact solution belongs to the interval solutions obtained with the interval method considered.

References

1. Anderson, D.A., Tannehill, J.C., Pletcher, R.H.: Computational fluid mechanics and heat transfer. Hemisphere Publishing, New York (1984)
2. Jankowska, M.A.: Remarks on Algorithms Implemented in Some C++ Libraries for Floating-Point Conversions and Interval Arithmetic. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part II. LNCS, vol. 6068, pp. 436–445. Springer, Heidelberg (2010)
3. Jankowska, M.A.: An Interval Finite Difference Method of Crank-Nicolson Type for Solving the One-Dimensional Heat Conduction Equation with Mixed Boundary Conditions. In: Jónasson, K. (ed.) PARA 2010, Part II. LNCS, vol. 7134, pp. 157–167. Springer, Heidelberg (2012)
4. Jankowska, M.A.: The Error Term Approximation in Interval Method of Crank-Nicolson Type. Differential Equations and Dynamical Systems (2012), doi:10.1007/s12591-012-0144-4
5. Jankowska, M.A., Sypniewska-Kaminska, G., Kaminski, H.: Evaluation of the Accuracy of the Solution to the Heat Conduction Problem with the Interval Method of Crank-Nicolson Type. Acta Mechanica et Automatica 6(1), 36–43 (2012)
6. Lapidus, L., Pinder, G.F.: Numerical Solution of Partial Differential Equations in Science and Engineering. J. Wiley & Sons (1982)
7. Manikonda, S., Berz, M., Makino, K.: High-order verified solutions of the 3D Laplace equation. WSEAS Transactions on Computers 4(11), 1604–1610 (2005)
8. Minamoto, T.: Numerical verification of solutions for nonlinear hyperbolic equations. Applied Mathematics Letters 10(6), 91–96 (1997)
9. Minamoto, T., Nakao, M.T.: Numerical Verifications of Solutions for Nonlinear Parabolic Equations in One-Space Dimensional Case. Reliable Computing 3(2), 137–147 (1997)
10. Marciniak, A.: An Interval Version of the Crank-Nicolson Method – The First Approach. In: Jónasson, K. (ed.) PARA 2010, Part II. LNCS, vol. 7134, pp. 120–126. Springer, Heidelberg (2012)
11. Nakao, M.T.: A numerical verification method for the existence of weak solutions for nonlinear boundary value problems. Journal of Mathematical Analysis and Applications 164(2), 489–507 (1992)
12. Nakao, M.T.: Numerical verification methods for solutions of ordinary and partial differential equations. Numerical Functional Analysis and Optimization 22(3-4), 321–356 (2001)
13. Nakao, M.T., Watanabe, Y., Yamamoto, N., Nishida, T.: Some computer assisted proofs for solutions of the heat convection problems. Reliable Computing 9(5), 359–372 (2003)
14. Szyszka, B.: The Central Difference Interval Method for Solving the Wave Equation. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part II. LNCS, vol. 7204, pp. 523–532. Springer, Heidelberg (2012)
15. Watanabe, Y., Yamamoto, N., Nakao, M.T.: A Numerical Verification Method of Solutions for the Navier-Stokes Equations. Reliable Computing 5(3), 347–357 (1999)
16. Watanabe, Y., Yamamoto, N., Nakao, M.T., Nishida, T.: A numerical verification of nontrivial solutions for the heat convection problem. Journal of Mathematical Fluid Mechanics 6(1), 1–20 (2004)
17. Xu, F., Seffen, K.A., Lu, T.J.: Non-Fourier analysis of skin biothermomechanics. International Journal of Heat and Mass Transfer 51, 2237–2259 (2008)

Interval Arithmetic and Automatic Differentiation on GPU Using OpenCL

Grzegorz Kozikowski and Bartłomiej Jacek Kubica

Institute of Control and Computation Engineering, Warsaw University of Technology, Poland
g.kozikowski@stud.elka.pw.edu.pl,
bkubica@elka.pw.edu.pl

Abstract. This paper investigates efficient and powerful approach to the Gradient and the Hessian evaluation for complex functions. The idea is to apply the parallel GPU architecture and the Automatic Differentiation methods. In order to achieve better accuracy, the interval arithmetic is used. Considerations are based on sequential and parallel authors' implementation. In this solution, both the AD methods: Forward and Reverse modes are employed. Computational experiments include analysis of performance and are studied on the generated test functions with a given complexity.

Keywords: interval computations, automatic differentiation, GPGPU, OpenCL.

1 Introduction

Interval computations (see, e.g., [6], [8]) are a well-known approach to solving several decision problems, e.g., equations systems, optimization, Pareto-set seeking, etc. (see, e.g., [13]). Their significant drawback is high computational cost of the branch-and-bound (b&b) type method – the basic schema of most interval algorithms.

On the other hand, the b&b method is relatively easy to parallelize – as for the shared memory environments as for the distributed memory.

The second author, in his previous papers, (and several other researchers) investigated several multithreaded programming tools and applied them to different interval algorithms (see [13]).

A relatively new variant of multithreaded programming is computing on GPUs. The devices designed for graphical computations (image rendering, etc.) have been found useful for other kinds of floating-point computations, also. In recent years, several tools have been developed to perform such computations, including CUDA (Compute Unified Device Architecture) [2] and – more recently – OpenCL (Open Computing Language) [3].

Not much effort has been put into utilization of these tools for interval computations, up to now. CUDA has been applied in [5] and [7].

The first author used OpenCL [11] to implement not only the library of basic interval functions and arithmetic operations, but also the Automatic Differentiation algorithms for the gradient and the Hesse matrix computations according to Forward and Reverse Mode, respectively.

2 Automatic Differentiation

What Is Automatic Differentiation?

The differential calculus has been invented in the second half of the seventeenth century – independently by Isaac Newton and Gottfried Wilhelm Leibniz. Since then, it has become a basic tool of several branches of science and engineering.

Consequently, computing the derivatives – precisely and efficiently – a is very important problem for several applications. Automatic Differentiation – besides numerical and symbolical methods – is one of the approaches to perform this operation.

It is based on decomposition of complex functions into elementary ones. During such decomposition, chain-rules are created. More generally, the chain-rule consists of a sequence of arithmetic operations and calls to standard functions in which the value of each entry depends only on the input variables or previously computed values.

In general, Automatic Differentiation has two basic modes of operations: Forward Mode and Reverse Mode. First algorithm was described in 1960, by R.E. Moore [12]. The derivatives are propagated throughout the computation using the chain rule with forward order (from the elementary steps by the intermediate ones to the final result – derivative of the input function).

In the 80s, P. Werbos [12] described an efficient and revolutionary approach to computing the gradient and the Hesse matrix – the Reverse Mode. It allows to obtain the gradient immediately (for the Hessian operation – n sequences are required). The difference lies in the fact that the Reverse Mode algorithm computes derivatives for all intermediate variables backwards (reverse manner) through the computation. Nevertheless, it requires saving the entire computation trace (since the propagation is done backwards through the computation) and hence is very mmemory demanding.

Mathematical Fundamentals

To present principles of the Automatic Differentiation algorithms, let us treat each complex function as a superposition of elementary ones (the elementary function consists of a single unary or binary operator). Consider:

- $x_1, x_2, \ldots, x_n$ denote independent variables,
- $f_i \colon U \to R$, where $U \subseteq R^{n+k-1}$, for $1 \leq i \leq n+k$
- $y_{n+k} = f_{n+k}(f_1, f_2, \ldots, f_{n+k-1})$ – a differentiable function f_{n+k} (superposition of elementary $f_1, \ldots, f_{n+k-1}$),

- $f_1, \dots, f_{n+k-1}$ – differentiable, elementary functions
- y_i – intermediate results, $y_i = f_i(x_i)$ for $1 \leqslant i < n$ and $y_i = f_i(f_1, f_2, \dots, f_{i-1})$ for $i = n+1, n+2, \dots, n+k$.

Subsequent operations required to obtain the final result of f_{n+k} can be represented as a vector of continuously differentiable elementary functions (chain-rule):

$$\begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \\ y_{n+1} \\ . \\ . \\ y_{n+k} \end{bmatrix} = \begin{bmatrix} f_1(x_1) \\ f_2(x_2) \\ . \\ . \\ . \\ f_n(x_n) \\ f_{n+1}(f_1, f_2, \dots, f_n) \\ . \\ . \\ f_{n+k}(f_1, f_2, \dots, f_{n+k-1} \end{bmatrix} \tag{1}$$

This chain-rule can be simply illustrated in form of Kantorovich Graph:

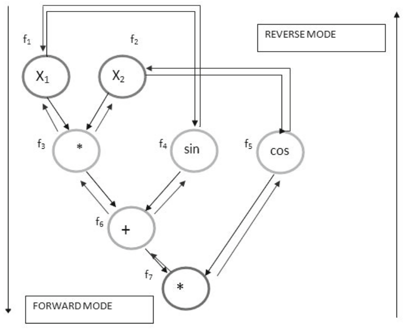


Fig. 1. The chain-rule of the example function as Kantorovich graph

Evaluation of the first-order derivatives (the gradient operation) relies on differentiation of the following elementary, functions with respect to each independent input variable. In result, we have:

$$J_f(x) = \left(\frac{\partial f_i}{\partial x_j}\right)_{(n+k)\cdot n} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ . & . & . \\ \frac{\partial f_{n+k}}{\partial x_1} & \cdots & \frac{\partial f_{n+k}}{\partial x_n} \end{bmatrix} \tag{2}$$

Each column of the above matrix (called the Jacobian matrix) denotes next elementary steps, necessary to evaluate the final partial derivative (in terms of the elementary interval operations like: +, −, ×, / and transcendental functions, such as: sin(x), cos(x), etc.).

2.1 Forward Mode – Gradient

Considering the elementary function f_{curr} from the list (1) as dependent on one or two immediately preceding functions in the Kantorovich Graph, we have: $y_{curr} = f_{curr}(f_{left})$ or $y_{curr} = f_{curr}(f_{left}, f_{right})$ for $1 \leq left \leq right < curr \leq n + k$.

Differentiating f_{curr} with respect to input variable, we obtain the results defined as follows:

– for f_{curr} that include a binary operator:

$$f_{curr}(f_{left}, f_{right}) = f_{left}\, op\, f_{right} \ , \tag{3}$$

$$\frac{\partial f_{curr}}{\partial x_i} = \frac{\partial f_{curr}}{\partial f_{left}} \cdot \frac{\partial f_{left}}{\partial x_i} + \frac{\partial f_{curr}}{\partial f_{right}} \cdot \frac{\partial f_{right}}{\partial x_i} \ , \tag{4}$$

– for f_{curr} that include an unary operator:

$$f_{curr}(f_{left}) = op\, f_{left} \ , \tag{5}$$

$$\frac{\partial f_{curr}}{\partial x_i} = \frac{\partial f_{curr}}{\partial f_{left}} \cdot \frac{\partial f_{left}}{\partial x_i} \ . \tag{6}$$

Please, note that:

$$\frac{\partial x_i}{\partial x_i} = 1 \ . \tag{7}$$

The Forward Mode algorithm is to perform the operations (4), (6) and store intermediate results for each node (starting from independent variables through intermediate functions to the final one). In this manner, we obtain the partial derivative of target function.

2.2 Forward Mode – Hessian Matrix

Due to the number of required operations, the Hessian matrix computation is more complicated. Nevertheless, the essentials are analogous to the Forward Mode algorithm for the gradient. To produce the Hessian formulae we extend the model described in the previous section. As a result of differentiation of Expression (4) by the second variable (x_j), we get the second-order derivative:

$$\begin{aligned} \frac{\partial^2 f_{curr}}{\partial x_i \partial x_j} &= \frac{\partial}{\partial x_j}\Big(\frac{\partial f_{curr}}{\partial f_{left}} \cdot \frac{\partial f_{left}}{\partial x_i} + \frac{\partial f_{curr}}{\partial f_{right}} \cdot \frac{\partial f_{right}}{\partial x_i}\Big) = \\ &= \frac{\partial^2 f_{curr}}{\partial f_{left} \partial xj} \cdot \frac{\partial f_{left}}{\partial x_i} + \frac{\partial f_{curr}}{\partial f_{left}} \cdot \frac{\partial^2 f_{left}}{\partial x_i \partial x_j} + \\ &\quad + \frac{\partial^2 f_{curr}}{\partial f_{right} \partial xj} \cdot \frac{\partial f_{right}}{\partial x_i} + \frac{\partial f_{curr}}{\partial f_{right}} \cdot \frac{\partial^2 f_{right}}{\partial x_i \partial x_j} \ , \end{aligned} \tag{8}$$

where the initial values are equal to zero: $\frac{\partial^2 x_i}{\partial x_i \partial x_j} = 0$ and $\frac{\partial^2 x_j}{\partial x_i \partial x_j} = 0$

It is worth mentioning that we utilize the gradient results for the Hessian matrix. Unfortunately, those operations are not trivial. Note, that the Hessian requires the values of those derivatives: $\frac{\partial^2 f_{curr}}{\partial f_{left}\partial x_j}$ and $\frac{\partial^2 f_{curr}}{\partial f_{right}\partial x_j}$

For simplicity and efficiency of the calculations, we should apply some abbreviations.
The following table contains appropriate derivatives depending on arithmetic operator.

Table 1. Equations for the second-order partial derivatives that are necessary for Hessian

Operation	**Left operand**	**Right operand**
addition	$\frac{\partial^2 f_{curr}}{\partial f_{left}\partial x_i} = 0$	$\frac{\partial^2 f_{curr}}{\partial f_{right}\partial x_i} = 0$
subtraction	$\frac{\partial^2 f_{curr}}{\partial f_{left}\partial x_i} = 0$	$\frac{\partial^2 f_{curr}}{\partial f_{right}\partial x_i} = 0$
multiplication	$\frac{\partial^2 f_{curr}}{\partial f_{left}\partial x_i} = \frac{\partial f_{right}}{\partial x_i}$	$\frac{\partial^2 f_{curr}}{\partial f_{right}\partial x_i} = \frac{\partial f_{left}}{\partial x_i}$
division	$\frac{\partial^2 f_{curr}}{\partial f_{left}\partial x_i} = \frac{\partial}{\partial x_i}\left(\frac{1}{\partial f_{right}}\right) = -\left(\frac{1}{f_{right}}\right)^2 \cdot \frac{\partial f_{right}}{\partial x_i}$	$\frac{\partial^2 f_{curr}}{\partial f_{right}\partial x_i} = 0$
unary	$\frac{\partial^2 f_{curr}}{\partial f_{left}\partial x_i} = \frac{\partial^2 f_{curr}}{\partial^2 f_{left}} \cdot \frac{\partial f_{left}}{\partial x_i}$	-

2.3 Reverse Mode – Gradient

Another approach to automatic differentiation is the Reverse Mode algorithm. The key is that the derivative propagation is done in reverse manner. This is often better suited to the problems with large numbers of input variables. In order to explain underlying method, we have to consider the relation below:

$$\bar{\bar{f}}_i = \frac{\partial f_{curr}}{\partial f_i} \cdot \bar{f}_{curr} \ , \tag{9}$$

where $\bar{f}_i = \frac{\partial f_{n+k}}{\partial f_i}$ and $\bar{f}_{curr} = \frac{\partial f_{n+k}}{\partial f_{curr}}$. Index *curr* corresponds to the function which is directly dependent on the operations denoted by subindex i (as could be seen in Kantorovich graph), additionally for $curr = n + k$ we assume that: $\bar{f}_{curr} = \frac{\partial f_{curr}}{\partial f_{curr}} = 1$

Taking into account the previous equation (9), we might as well derive the formulae for partial derivatives with respect to preceding functions in Kantorovich graph.

$$\frac{\partial f_{n+k}}{\partial f_{left}} = \sum \frac{\partial f_{curr}}{\partial f_{left}} \cdot \frac{\partial f_{n+k}}{\partial f_{curr}} \ , \quad \frac{\partial f_{n+k}}{\partial f_{right}} = \sum \frac{\partial f_{curr}}{\partial f_{right}} \cdot \frac{\partial f_{n+k}}{\partial f_{curr}} \ . \tag{10}$$

Having performed the above operations, the final, partial derivatives are associated with nodes of the independent variables. In result we obtain the gradient.

Formulae for the Hesse matrix computed in the Reverse Mode are analogous. Details can be found in [11].

3 Interval Arithmetic

Obviously, numerical computations are performed with limited accuracy, only. Also, the arguments which we use in our computations are often inaccurate, due to imprecise measurements, discretization error or other sources of uncertainty.

The interval calculus (see, e.g., [6], [8]) is an approach to dealing with all these kinds of errors, obtaining guaranteed results. Using the interval notation [9], we present the basic notions of interval calculus. By an interval we mean the set of numbers $\mathbf{x} = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \le x \le \overline{x}\}$, where $\underline{x}$ and $\overline{x}$ are floating-point numbers. According to the IEEE 754 standard, they can be infinite (obviously $\underline{x} = +\infty$ or $\overline{x} = -\infty$ would make little sense in practice).

Operations on intervals are defined so that the actual result of the real-number operation was verified to belong to the resulting interval. The basic arithmetic operations can be described as follows:

$$\begin{aligned}
&[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \ , \\
&[\underline{x}, \overline{x}] - [\underline{y}, \overline{y}] = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \ , \\
&[\underline{x}, \overline{x}] \cdot [\underline{y}, \overline{y}] = [\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})] \ , \\
&[\underline{x}, \overline{x}] \ / \ [\underline{y}, \overline{y}] = [\underline{x}, \overline{x}] \cdot \left[1 / \overline{y}, 1 / \underline{y}\right] \ , \qquad 0 \notin [\underline{y}, \overline{y}] \ .
\end{aligned}$$

Moreover, if using the directed rounding, we can obtain guaranteed results. This means, we can deal with numerical errors, by rounding the lower bound downwards and the upper bound – upwards and thus enclosing the correct result, rigorously.

Arithmetic operations on intervals are not the only ones to be defined. If the function $f\colon \mathbb{R} \to \mathbb{R}$ is monotonic on the interval $[\underline{x}, \overline{x}]$ then for all: $[\underline{y}, \overline{y}] \subseteq [\underline{x}, \overline{x}]$ we have:

$$\mathsf{f}([\underline{y}, \overline{y}]) = [\min\{f(\underline{y}), f(\overline{y})\}, \max\{f(\underline{y}), f(\overline{y})\}] \ .$$

Based on this assumption, we define, e.g.:

$$e^{[\underline{x}, \overline{x}]} = [e^{\underline{x}}, e^{\overline{x}}] \ , \ \log_a([\underline{x}, \overline{x}]) = [\log_a(\underline{x}), \log_a(\overline{x})] \text{ for } \underline{x} > 0 \text{ and } a > 1 \ .$$

Formulae for other transcendental functions ($\sin(\cdot)$, $\cos(\cdot)$, etc.) can be obtained, also (see, e.g., [6]). In all cases, outward rounding allows to enclose the correct result, dealing with the numerical imprecision.

4 Introduction to OpenCL Technology

4.1 GPGPU

Technological development of graphics cards had a significant impact on software development. Leading GPU manufacturers – as NVIDIA and ATI – rely on a parallel architecture in order to provide General Purpose computations to their products; not only related to rendering. This approach often gives a high speed-up.

However, multiple cores available on graphics cards have some constraints and require a suitable programming model. In particular, the decomposition of the main problem to many subroutines executed in parallel (simultaneously by many cores) should be adequate. For high performance and scalability, each thread has to perform the same computations, just on different data, to avoid the latency caused by synchronization. Such an architecture is commonly known as Single Instruction Multiple Thread. According to this model, the program includes similar instructions that are executed as threads operating on many different data elements and executed by many multiprocessors, simultaneously.

Consequently, only algorithms using data parallelism will be efficient on GPUs, usually. Fortunately, this is the case for many problems and algorithms in several fields, including compression algorithms, artificial intelligence, Monte Carlo methods or optimization.

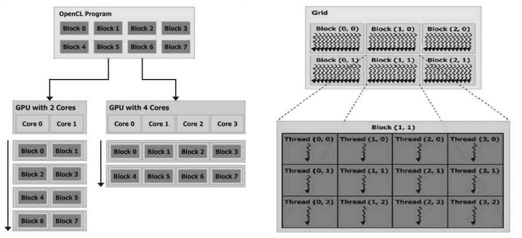
4.2 OpenCL

OpenCL (Open Computing Language) was previously developed by Apple for OS X platforms (particularly Mac OS X Snow Leopard). Soon, due to the rapid industrial progress, the project was joined by other leading companies, such as: Intel, IBM, ATI, AMD or Sun. Consequently, the Khronos consortium was established and has been managing OpenCL since 2008.

OpenCL specification provides an API, based primarily on the C99 standard for C language, for GPGPU (or other devices with independent threads). In particular, for parallel processing on GPUs, there is a specific function, called a *kernel*. It is executed many times by many different threads. Moreover, tasks are organized into many groups of 512 (GPUs supporting capability 1.1) or 1024 threads (capability 2.1) that are executed by subsequent multiprocessors. Threads and thread-groups are identified using a one-, two- or three-dimensional index. All work-groups reside on one or two-dimensional grids, so the number of all threads in a grid is the number of threads per block times the number of blocks in the grid.

OpenCL provides synchronization methods within each work-group and for the whole grid. It does not support object-oriented programming or recursion, as they might be unsuitable for some devices.

Depending on the context, OpenCL allows to use several types of memory space that differ with performance parameters and capacity. The most important of them (but not the most efficient!) is the global memory. This is an intermediary space for exchanging data between the host and other types of GPU memory. Due to its role, the global memory is available for all threads (work-items) and groups of threads (work-groups). Another one is the local memory of each multiprocessor. Because of performance and capacity features, it is often utilized in order to share data within threads in a work-group. Each thread has also its own registers to store local variables. The architecture provides also other address spaces, such as texture memory, constant memory etc. [2], but they are of less importance.



(a) Execution on GPU (b) Threads on GPU

Fig. 2. GPU Architecture

5 Architecture of Automatic Differentiation Library

5.1 Introduction

The developed library consists of the Automatic Differentiation algorithms, working on GPU. The nature of this environment implies the specific structure of implemented routines. Operator overloading was applied to transform the formulae to Kantorovich graphs. How to parallelize operations on the Kantorovich graph? For some graph's structures it would be pretty difficult, but usually, this graph contains independent nodes (at the same depth) that can be processed simultaneously.

As a simple example, let us consider the function: $y = \sin(x) + \cos(x)$. It consists of two trigonometric functions that can be evaluated in parallel – $\sin(x)$ and $\cos(x)$. These intermediate values must be stored in separate memory spaces, corresponding to its nodes. Besides the value of the intermediate function, the first and the second-order partial derivatives are associated with each node. Traversing the expressions tree and applying formulae for the Forward or Reverse Mode, the work-items compute the proper partial derivatives. Suitability for parallel processing is strictly dependent on the structure of differentiable function. Unfortunately, not all functions include maximum number of independent nodes at a fixed depth.

Obvoiously, even if we cannot parallelize the differentiation of a single function, we can still compute derivatives of several arguments in parallel.

5.2 Common Interface

Both, sequential and parallel implementations are based on the same set of functions. These are in particular the following routines:

- Evaluate(graph, i) – evaluating the function associated with the i-th node
- GradForwardStep(graph, i) and GradReverseStep(graph, i) – responsible for evaluation of partial derivative according to the equations for the gradient in Forward Mode ([4]) and Reverse Mode ([10]) respectively

– HessForwardStep(graph, i) and HessReverseStep(graph, i) – responsible for evaluation of partial derivative according to the equations for the Hessian in Forward Mode (9) and Reverse Mode respectively.

5.3 Sequential Model

Even if a formula consists a relatively large number of independent operations, the sequential architecture does not permit processing it in parallel. For this reason, it is advisable to previously sort the graph with ascending/descending order of depth for Forward/Reverse Mode and prepare it for sequential execution. Obviously, as every value and partial derivative of the function is evaluated sequentially, the next node cannot be processed until the previous one is completed. Algorithm 1 illustrates the sequential execution model.

```
input  : Array of $n$ Kanotorivich Graphs of size $s$
output: Array of intervals, representing subsequent derivatives

funcIdx $\leftarrow 1$
while funcIdx $\leq n$ do
    Sort(graph[funcIdx]) // Sorting the graph with ascending or
    descending order with respect to the depth
    graph[funcIdx][derivativeIdx].derivative $\leftarrow [1, 1]$ // Assignment of the node,
    with respect to which, the function will be differentiated

    nodeIdx $\leftarrow 1$
    while nodeIdx $\leq s$ do
        Evaluate(graph[funcIdx], nodeIdx)
        GradForwardStep(graph[funcIdx], nodeIdx)
        nodeIdx $\leftarrow$ nodeIdx $+ 1$
    end
end
```

Algorithm 1. Gradient Forward Mode – the sequential version

5.4 Parallel Model

Having in mind technical principles of OpenCL and mathematical fundamentals of the AD methods, parallel model can be designed. According to authors' assumptions, all the Kantorovich graphs and computation results are stored in local memory situated closely to cores. This space is only shared within work-items. Differentiable functions can be represented by nodes, whose amount is less or equal to the maximal number of work-items per work-group. Additionally, each work-item within a work-group corresponds to a single node of the function under differentiation it and performs computation according to the rules described in Section 2. It emphasises that each work-item is responsible for evaluation and differentiation of function associated witha particular node.

Two levels of parallelization are considered:

- parallel execution within a single function,
- parallel execution within all differentiable functions.

Algorithm 2 illustrates key points of the parallel method.

```
s    input  : Array of $n$ Kantorovich Graphs of size $s$
output: Array of intervals representing subsequent derivatives
nodeIdx $\leftarrow$ GetLocalIdx(0) // the work-item identifier (1st dimension)
funcIdx $\leftarrow$ GetGlobalIdx(0) // the work-group id (1st dimension)
derivativeIdx $\leftarrow$ GetGlobalIdx(1) // the work-group id (2nd dimension) --
differentiation with respect to the node with this index
currentDepth $\leftarrow 0$
if nodeIdx $< s$ and funcIdx $< s$ and derivativeIdx $< n$ then
    // Transfer of each graph from global to local memory
    graph[funcIdx][derivativeIdx].derivative $\leftarrow [1, 1]$ // Assignment of the node,
    wrt which the function will be differentiated
    LocalBarrier() // necessary to load all graphs into local memory
    while currentDepth $\leq$ graph[$s$].depth do
        if currentDepth $=$ graph[nodeIdx].depth then
            Evaluate(graph[funcIdx], nodeIdx)
            ForwardStep(graph[funcIdx], nodeIdx)
        end
        LocalBarrier() // after processing all nodes at a given depth
        currentDepth $\leftarrow$ currentDepth $+ 1$
    end
    // Transfer of each result from local to global memory
end
```

Algorithm 2. Gradient Forward Mode – the parallel version

6 Computational Experiments

6.1 Test Data and Test Environment

Test Data Performance of the developed procedure have been investigated for both versions: sequential and parallel. It required input functions with various structure and complexity. They have been generated by an appropriate module of the library. Three basic types of the Kantorovich graph with various complexity have been considered:

- Type 1 (a pessimistic version) – a complex function of a single variable; no binary operators, i.e., no independent operations.
- Type 2 (an expected version) – a function that contains both unary and binary operators. For n independent variables, the graph is of the size $\frac{n(n+1)}{2}$.

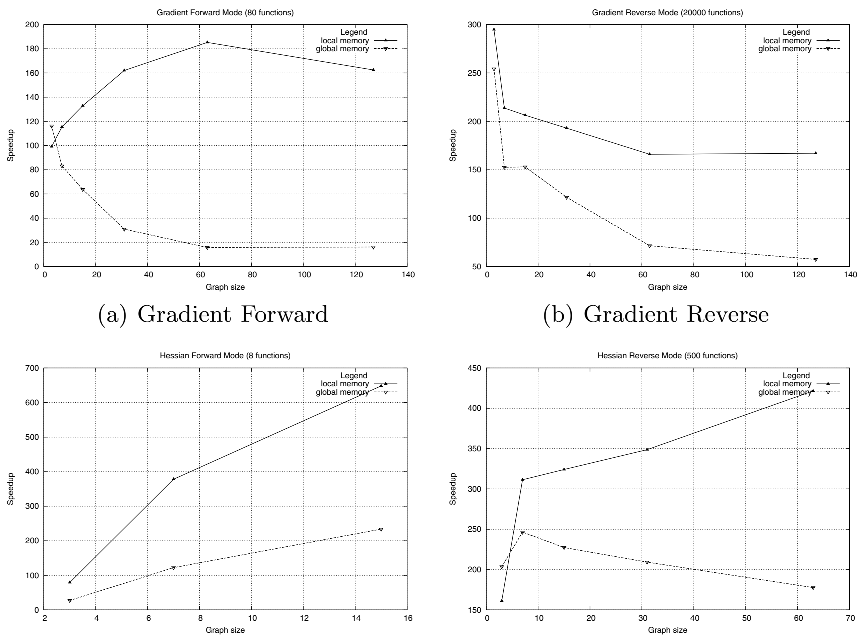
– Type 3 (an optimistic version, binary trees only) – a function consisting only of binary operators; best parallelization possibilities.

Test Environment Numerical experiments were performed on a PC equipped with:

– Dual-Core AMD Athlon X2 with 2,8Ghz clock,
– 2GB DDRAM memory (with 800Mhz frequency),
– GeForce 9800 GT 512 MB supporting CUDA/OpenCL 1.1,
– Windows XP Professional SP3 operating system.

6.2 Analysis of the Speedup with Respect to the Sequential Version

The parallel version (using local and global memory) was compared to the CPU version. Tests were performed with a given range and complexity of input functions. Along with the best properties of performance, there were selected graphs of binary tree structure (type 3). The vertical axis represents the ratio of run-time of parallel version to the sequential. The horizontal axis indicates the degree of complexity – the size of the graph assigned to the function under differentiation.



(a) Gradient Forward (b) Gradient Reverse

(c) Hessian Forward (d) Hessian Reverse

Fig. 3. Analysis of the acceleration for type 3 functions

As one could expect, the GPU version using local and even global memory is faster than the CPU version. Initially, we achieved the speedup from 100x to 120x for 80 functions (Gradient Forward). Considering the function represented by 63 nodes, we obtain a maximum ratio of execution time on the GPU compared to CPU; speedup is about 180 times. With the increasing size of the graph, we would see a steady and smooth decrease of the acceleration. Taking into account usage of global memory, plots show that a frequent data transfer from global to local memory on device is a bottleneck of GPGPU technology. For relatively simple differentiable functions, execution times of algorithm with global memory to local one, are even shorter. It comes from no need to data transfer from global memory to local one. However, greater number of nodes causes loss of acceleration and prolongs the times of algorithms on GPU. We observe that speedup has sharply fallen with increasing complexity of differentiable functions.

Much less memory requirements for Reverse Mode algorithm allow to prepare tests for a greater number of functions (5000). Both lines (global and local memory) present a declining trend. Data transfer has also a huge impact on execution time of Reverse Mode.

Forward and Reverse Mode algorithms for the Hesse Matrix are a few hundred times faster than sequential versions. Thanks to reduced memory requirements, the Reverse method features a better performance because it takes less time on data transfer. Unlike Forward Mode, Reverse Mode allows to prepare tests even for a few hundreds of functions.

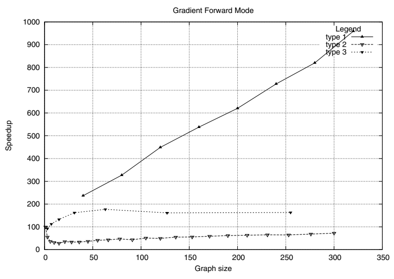
6.3 Analysis of the Impact of Input Test Function on Speedup of Differentiation

Tests were performed on various range of differentiable functions, depending on the algorithm:

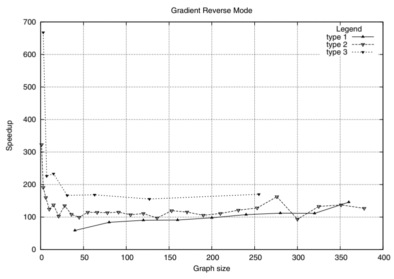
- Gradient Forward – 60 functions,
- Gradient Reverse – 10000 features,
- Hessian Forward – 6 functions,
- Hessian Reverse – 500 functions.

The vertical axis represents the ratio of time execution for parallel version to the sequential one. The horizontal axis denotes the degree of complexity – the size of the graph representing the function under differentiation.

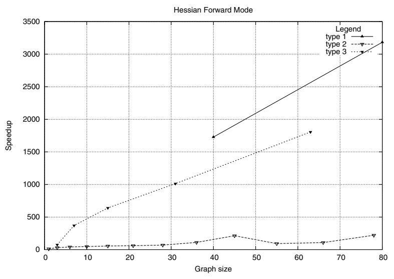
The graphs corresponding to the Forward Mode algorithm for gradient and the Hesse matrix show that the differentiation of complex functions, dependent on a single variable, is the most efficient. The speedup has a linear trend. As expected, the differentiation of functions consisting of only binary operators is more efficient than of other functions (greater number of nodes at the same depth). In comparison to CPU, we obtain even 200x shorter execution times. According to measurements for gradient and the Hesse matrix computed in reverse manner, we conclude that functions with binary operators are well-suited for parallel execution (maximal number of nodes at the same depth).



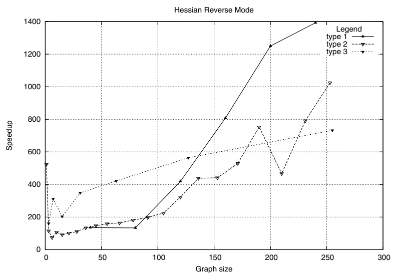
(a) Gradient Forward



(b) Gradient Reverse

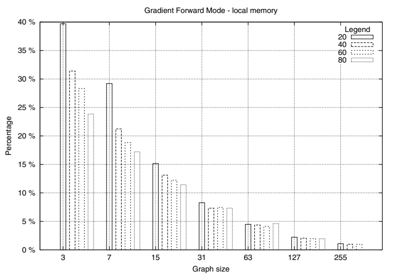


(c) Hessian Forward

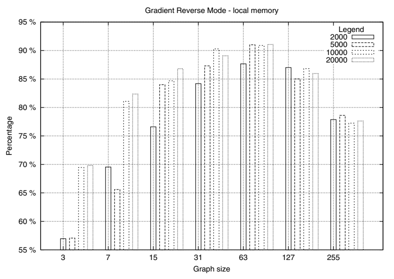


(d) Hessian Reverse

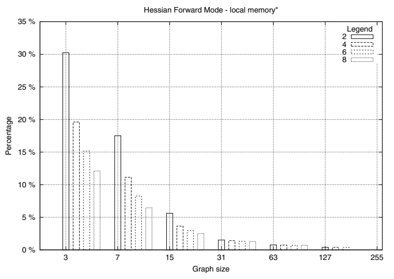
Fig. 4. Analysis of the impact of the function type and complexity on speedup of differentiation



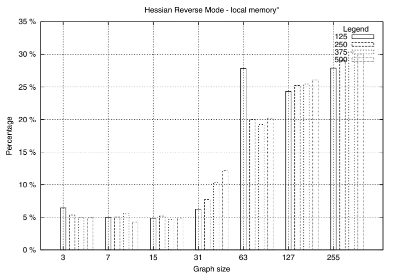
(a) Gradient Forward



(b) Gradient Reverse



(c) Hessian Forward



(d) Hessian Reverse

Fig. 5. Analysis of the impact of input test function on speedup derivative computations (local memory)

7 Future Work

Although OpenCL is an open standard, independent of the specific hardware architecture, its generality may entail a performance penalty. It is not the only interface supporting GPGPU; in particular NVIDIA offers the API set for its CUDA architecture, with similar features to OpenCL.

According to the report [10], CUDA performs slightly better than OpenCL, when transferring data between the host and the GPU. Benchmarks have shown the OpenCL can be about 13-63% slower than CUDA. Consequently, porting the kernel code can bring us a significant performance boost. Considering these factors, we may concentrate on further use of CUDA.

The second issue worth investigation is to reduce the effects of loading data to the GPU's global memory. Aiming at differentiating a large number of functions (required data exceeding the capabilities memory on each kernel, but not global memory), we would transfer data between host and device in an asynchronous way (calling non-blocking read/write functions to memory on GPU, while the kernel is being executed).

8 Conclusion

Automatic Differentiation is a powerful tool that can be useful in various fields ranging from option price evaluation in finance and optimization problems to medical imaging and geoscience. Until now, it was difficult and computationally demanding, but recent technological development allows to increase the its performance. One of the possibilities is to perform this procedure on GPUs, utilizing its potential parallelism.

As a part of this work a research regarding utilization of GPUs to AD methods has been done. We considered gradient and Hesse matrix computations in two different variants: Forward and Reverse modes. All of them have been implemented using OpenCL. Performance and memory usage of the implemented library have been investigated. Performance tests prove that the GPU architecture with multi-core processors bring us tremendous computational horsepower for Automatic Differentiation algorithms.

Acknowledgments. The paper has been done as a part of realization of the grant for statutory activity, financed by the Dean of Faculty of Electronics and Information Technology (WUT), titled "Interval methods for solving nonlinear problems".

References

1. C-XSC interval library, http://www.xsc.de.
2. CUDA homepage: http://www.nvidia.com/object/cuda_home.html
3. OpenCL homepage: http://www.khronos.org/opencl.

4. Bücker, M.: Automatic Differentiation: Applications, Theory and Implementation. Springer (1981)
5. Collange, S., Florez, J., Defour, D.: A GPU interval library based on Boost interval (2008)
6. Hansen, E., Walster, W.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (2004)
7. Jastrzebski, K., Szczap, L.: Different parallelism approaches to interval computations. Master's thesis, Faculty of Electronics and Information Technology, WUT (2009)
8. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
9. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.:Standardized notation in interval analysis (2002), http://www.mat.univie.ac.at/~neum/software/int/notation.ps.gz
10. Kamran, K., Neil, G., Firas, H.: A Performance Comparision of CUDA and OpenCL (2011), http://arxiv.org/abs/1005.2581
11. Kozikowski, G.: Implementation of automatic differentiation library using the OpenCL technology. BEng thesis, Faculty of Electronics and Information Technology, WUT (2011)
12. Werbos, P.: Backpropagation Through Time: What It Does and How to Do It
13. Kubica, B.J.: A class of problems that can be solved using interval algorithms. SCAN 2010 Proceedings. Computing 94(2-4), 271–280 (2012)

Tuning the Interval Algorithm for Seeking Pareto Sets of Multi-criteria Problems

Bartłomiej Jacek Kubica and Adam Woźniak

Institute of Control and Computation Engineering, Warsaw University of Technology, Poland
bkubica@elka.pw.edu.pl, A.Wozniak@ia.pw.edu.pl

Abstract. The paper presents the authors' effort to optimize a previously developed interval method for solving multi-criteria problems [17]. The idea is to apply heuristics presented, e.g., in [27], [22], [26] and some new ones, to the Pareto set seeking method. Numerical experiments are presented and parallelization of the algorithm is considered. Based on the tuned algorithm, we propose a new approach to interactive multiple criteria decision making.

Keywords: Pareto-set, multi-criteria analysis, interval computations, bisection, Newton operator, tuning.

1 Introduction

The general problem of multi-criteria analysis (sometimes called multi-criteria optimization) can be formulated as follows:

$$\begin{aligned} &\min_x q_k(x) \qquad k = 1, \ldots, N \ , \\ &\text{s.t.} \\ &g_j(x) \le 0 \qquad j = 1, \ldots, m \ , \\ &x_i \in [\underline{x}_i, \overline{x}_i] \qquad i = 1, \ldots, n \ . \end{aligned} \tag{1}$$

By solving the above problem we mean finding two sets. Firstly – a set of Pareto-optimal points, i.e., feasible points that are non-dominated according to criteria, and secondly – a Pareto frontier, i.e., a set of N-tuples of criteria values for Pareto-optimal points. For convenience in the sequel we will call them together Pareto sets (see, e.g., [8], [10], [25]). A feasible point x is *Pareto-optimal* (non-dominated), if there exists no other feasible point x' such that:

$$\begin{aligned} &(\forall k) \quad q_k(x') \le q_k(x) \text{ and} \\ &(\exists i) \quad q_i(x') < q_i(x) \ . \end{aligned}$$

Computing the Pareto sets – or even approximating them precisely enough – is a hard task, especially for nonlinear problems.

Interval methods (see, e.g., [11], [12], [13]) are a proper approach for this application. Indeed, several interval algorithms to compute Pareto sets have been developed – see, e.g., [3], [7], [17], [20], [28].

All of them are based on the branch-and-bound (b&b) schema that is reliable, but time-consuming and memory demanding.

It is well-known that it is crucial for efficiency of interval algorithms to design proper heuristics for choosing and parameterizing accelerating mechanisms. For example Csendes et alii considered several such accelerators for global optimization (see, e.g., [5], [27]) and Kubica – for underdetermined equations systems solving [23].

In this paper we consider tuning of a b&b method, introduced in [17], [18], [19], [21], [20].

2 Generic Algorithm

In previous papers we developed an algorithm to seek the Pareto frontier and the Pareto-optimal set. It subdivides the criteria space in a branch-and-bound manner and inverts each of the obtained sets using a variant of the SIVIA (Set Inversion Via Interval Analysis) procedure [12]. Some additional tools (like the componentwise Newton operator) are applied to speedup the computations.

Using the notation of [14], the algorithm is expressed by the following pseudocode.

```
compute_Pareto-sets ($\mathsf{q}(\cdot)$, $\mathbf{x}^{(0)}$, $\mathsf{g}(\cdot)$, $\varepsilon_y$, $\varepsilon_x$)
// $\mathsf{q}(\cdot)$ is the interval extension of the criterion function $q(\cdot) = (q_1, \ldots, q_N)(\cdot)$
// $\mathsf{g}(\cdot)$ is the interval extension of the constraint function $g(\cdot) = (g_1, \ldots, g_N)(\cdot)$
// $L$ is the list of quadruples $(\mathbf{y}, L_{\text{in}}, L_{\text{bound}}, L_{\text{unchecked}})$
$\mathbf{y}^{(0)} = \mathsf{q}(\mathbf{x}^{(0)})$;
$L = \Big\{ \big(\mathbf{y}^{(0)}, \{\}, \{\}, \{\mathbf{x}^{(0)}\}\big) \Big\}$;
while (there is a quadruple in $L$, for which $\operatorname{wid} \mathbf{y} \geq \varepsilon_y$)
    take this quadruple $(\mathbf{y}, L_{\text{in}}, L_{\text{bound}}, L_{\text{unchecked}})$ from $L$;
    bisect $\mathbf{y}$ to $\mathbf{y}^{(1)}$ and $\mathbf{y}^{(2)}$;
    for $i = 1, 2$
        SIVIA-like ($(\mathsf{q}, \mathsf{g})$, $L_{\text{unchecked}}$, $(\mathbf{y}, [-\infty, 0]^m)$, $\varepsilon_x$, $L_{\text{in}}$, $L_{\text{bound}}$);
        // where $m$ is the number of constraints $g_j$
        if (the resulting quadruple has a nonempty interior, i.e., $L_{\text{in}} \neq \emptyset$)
            delete quadruples of $L$ that are dominated by $\overline{y}^{(i)}$;
        end if
        insert the quadruple to the end of $L$;
    end for
end while
// finish the Pareto-optimal set computations
for each quadruple in $L$ do
    process boxes from $L_{\text{unchecked}}$ until all of them get to $L_{\text{in}}$ or $L_{\text{bound}}$;
end do;
end compute_Pareto-sets
```

Obviously, both loops in the above algorithm – the `while` loop and the `for each` loop can easily be parallelized (at least in a shared memory environment [?], [19]).

The `SIVIA-like` procedure is a variant of classical SIVIA, introduced in [12]. It was proposed in [17]. It computes the reverse image of a set Y, but breaks after finding an interior box (which proves that the reverse image is non-empty). The procedure can be described by the following pseudocode:

```
SIVIA-like (f(·), $L_{\text{unchecked}}$, $Y$, $\varepsilon$, $L_{\text{in}}$, $L_{\text{bound}}$)
// f(·) is the interval extension of the function
// $L_{\text{unchecked}}$ is the list of boxes to process (their sum contains the domain)
// $Y$ is the inverted set – usually an interval, but not necessarily in general
// $\varepsilon$ is the accuracy of approximation
$L_{\text{in}} = \emptyset$; // the list of boxes contained in the set $f^{-1}(Y)$
$L_{\text{bound}} = \emptyset$; // the list of boundary boxes
stack ($\mathbf{x}^{(0)}$);
while ($L_{\text{unchecked}} \neq \emptyset$) do
    pop($L_{\text{unchecked}}$, $\mathbf{x}$);
    process box $\mathbf{x}$; // perform the rejection/reduction tests
    $\mathbf{y} = \mathsf{f}(\mathbf{x})$;
    if ($\mathbf{y} \subseteq Y$)
        push ($L_{\text{in}}$, $\mathbf{x}$);
        break; // the set is sure to have a nonempty interior
    else if ($\mathbf{y} \cap Y = \emptyset$) discard $\mathbf{x}$;
    else if (wid ($\mathbf{x}$) $\leq \varepsilon$) push ($L_{\text{bound}}$, $\mathbf{x}$);
    else
        bisect $\mathbf{x}$;
        push subboxes to $L_{\text{unchecked}}$;
    end if
end while
end SIVIA-like
```

The above pseudocode does not specify details of the box processing procedure. The following techniques can be applied there:

- checking if the box $\mathbf{x}$ is contained in the reverse image of corresponding $\mathbf{y}$; possibly narrowing the box by the componentwise interval Newton operator [17],
- the monotonicity test, adapted to the multi-criteria case (this test uses the first-order information),
- checking 2nd-order Pareto-optimality conditions, using an interval Newton operator [21].

The same techniques are applied in the `for each` loop of the main `compute_Pareto-sets` procedure.

3 Improvements

Improvements, we propose in this paper, can be classified in two groups:

– improvements to the main algorithm from Section 2,
– heuristics to choose/parameterize tools to process boxes in the SIVIA-like procedure.

The corrections to the main algorithm include:

– not subdividing quadruples with empty interior,
– an additional narrowing procedure for quadruples: we compute the hull of $\mathsf{q}(\cdot)$'s of all boxes in L_{in} and L_{bound} and intersect it with $\mathbf{y}$ of the quadruple.

Considered heuristics include:

– choosing whether to apply the Newton operator or not for a specific box,
– choosing what type of the Newton operator to apply on a specific box (as in [22]),
– different procedures to choose the bisection direction for boxes.

3.1 When to Apply the Newton Operator?

In equations solving, the Newton operator is (usually) the basic tool we use. In solving other problems, it is not so – we have several forms of the monotonicity test, we can compare the objective's values (i.e., use the 0th-order information), remove regions that violate the constraints, etc.

In particular, when seeking the Pareto sets of a multi-criteria analysis problem, the Newton operator can be used to solve the system of 2nd order Pareto-optimality conditions. It is a powerful tool, but also relatively time-consuming (Hesse matrices of all criteria and all constraints have to be evaluated!), so skipping it for boxes for which it would not improve the result, would be very beneficial.

For global optimization, some interesting results have been obtained by Csendes [6] and Pal [26]. Pal investigated a heuristic based on trisection and monotonicity test – if two, out of the three subboxes get deleted by the monotonicity test, the Newton operator is applied to the third box, otherwise not. According to [6], the heuristic did not perform as expected and a simple one based on the box's diameter turned better.

In our study we adopt this simple heuristic – for boxes larger than a given threshold value, the Hesse matrices are not computed and the Newton operator is not applied; for small enough ones, they are. Csendes suggested the threshold absolute value to be 0.1, but in our experiments a smaller value, i.e., 0.025 turned out to be better.

3.2 How to Choose the Subdivision Direction?

A few approaches to branching have been proposed for different b&b algorithms. The most common one is bisection of the longest component of the box. It is used as for equations systems, as for global optimization, Pareto sets seeking and other problems.

For global optimization, a different criterion, known as *maximal smear*, is suggested by some authors (see, e.g., [13], [27]). It has been argued that bisection should reduce the diameter of objective on the subdivided box, as much as possible. Hence, we choose the variable j, for which the value $|\mathbf{g}_j(\mathbf{x})| \cdot \mathrm{wid}\,\mathbf{x}_j$ is the largest. Csendes et alii investigated this approach extensively, realizing its usefulness.

Yet another idea, for global optimization, is based on the so-called *rejection index* (see, e.g., [5])

For equations systems, the MaxSmear technique can be adapted (see [4]), by computing the maximal magnitude for all equations and all variables, but experiments performed in [22] suggested its poor performance.

In [22] a completely different heuristic has been proposed. It was based on the assumption that bisection should not reduce the diameter of interval functions by itself, but rather create subboxes suitable for reduction by the Newton operator or other rejection/reduction tests. For details, the reader is referred to [22].

In our study, we investigate the Pareto sets seeking problem and we use the following approaches for the subdivision direction choosing:

– MaxDiam, i.e., choosing the maximal diameter – as in previous papers ([17]–[21]),
– MaxSmear, based on the gradients of criteria ([13], [27]),
– the heuristic proposed in [22],
– new procedures, described below.

Inconsistency measure. In our algorithm, we invert sets from the criteria space to the decision space. So, for each box $\mathbf{x}^{(k)}$ from the decision space, we compute the vector of criteria $\mathbf{y}^{(k)} = (\mathbf{y}_1^{(k)}, \ldots, \mathbf{y}_N^{(k)})^T$ to check if they are included in the inverted set $\mathbf{y}$.

If $\mathbf{y}^{(k)} \subseteq \mathbf{y}$ or $\mathbf{y}^{(k)} \cap \mathbf{y} = \emptyset$, then no subdivision of $\mathbf{x}^{(k)}$ is needed. Otherwise, we should choose for bisection such a variable that would reduce the components of $\mathbf{y}^{(k)}$ that are the most distant (in the sense of an absolute distance) from $\mathbf{y}$.

Note, that as in [2], we consider an asymmetric distance measure here (a quasi-metric, because if one set is contained in another, the distance is considered to be zero). So, the heuristic is to find the criterion i with the highest "distance" from $\mathbf{y}_i^{(k)}$ to $\mathbf{y}_i$ and apply the MaxSmear for this single criterion, as for unicriterion optimization. Let us call this approach MaxDist.

MaxDist augmented. Our experiences from [22] suggest that basing on a single feature of the function (or box) for direction choosing results in poor performance, usually. In particular, it might be good not to bisect very short edges, i.e., not to let the boxes have too high differences between boxes sizes. This intuition leads to the MaxDistAug heuristic: choose the variable using MaxDist approach *only if* the longest edge is shorter than eight times the one chosen by MaxDist; otherwise choose the longest edge.

4 Computational Experiments

Numerical experiments were performed on a computer with 16 cores, i.e., 8 Dual-Core AMD Opterons 8218 with 2.6GHz clock. The machine ran under control of a Fedora 16 Linux operating system.ATLAS 3.9.11 was installed there for BLAS libraries. The solver was implemented in C++, using C-XSC 2.5.1 library [1] for interval computations. The GCC compiler was used, version 4.6.1.

Parallelization of the algorithm was done using POSIX threads, in a way described in [18] and [19]. This approach parallelizes the "outer loop" of the algorithm, i.e., operations on different boxes in the criteria space are done in parallel, but there is no nested parallelism on the SIVIA-like procedure applied to them. This allows larger grain-size, but makes us to execute costly operations on the list of sets in a critical section (deleting all dominated sets).

The following well-known test problems have been considered.

4.1 The Kim Problem

Our first example is a classical hard problem for multi-criteria analysis [15]:

$$\begin{aligned}
\min_{x_1,x_2} \Big(& q_1(x_1,x_2) = -\big(3\cdot(1-x_1)^2\cdot\exp(-x_1^2-(x_2+1)^2) - 10\cdot\big(\frac{x_1}{5}-x_1^3-x_2^5\big)\times \\
& \quad \times\exp(-x_1^2-x_2^2) - 3\exp(-(x_1+2)^2-x_2^2) + 0.5\cdot(2x_1+x_2)\big), \qquad (2)\\
& q_2(x_1,x_2) = -\big(3\cdot(1+x_2)^2\cdot\exp(-x_2^2-(1-x_1)^2) - 10\cdot\big(-\frac{x_2}{5}+x_2^3+x_1^5\big)\times \\
& \quad \times\exp(-x_2^2-x_1^2) - 3\exp(-(2-x_2)^2-x_1^2)\big)\Big), \\
& x_1,x_2\in[-3,3]\ .
\end{aligned}$$

This problem has a non-connected Pareto frontier and is very difficult, e.g., for evolutionary methods [15].

The solver's parameters used for this problem are: $\varepsilon_y = 0.05$ and $\varepsilon_x = 0.001$.

4.2 The Osyczka Problem

This constrained problem with 6 variables, 2 criteria and 6 constraints is taken from [3]:

$$\begin{aligned}
\min_{x_1,\ldots,x_6} \Big(& q_1(x_1,\ldots,x_6) = -\big(25(x_1-2)^2+(x_2-2)^2+(x_3-1)^2+ \\
& +(x_4-4)^2+(x_5-1)^2\big), \qquad q_2(x_1,\ldots,x_6) = \sum_{i=1}^{6} x_i^2\Big), \qquad (3)
\end{aligned}$$

$$
\begin{aligned}
&\text{s.t.}\\
&-x_1 - x_2 + 2 \le 0,\\
&x_1 + x_2 - 6 \le 0,\\
&-x_1 + x_2 - 2 \le 0,\\
&x_1 - 3x_2 - 2 \le 0,\\
&(x_3 - 3)^2 + x_4 - 4 \le 0,\\
&-(x_5 - 3)^2 - x_6 + 4 \le 0,\\
&x_1, x_2, x_6 \in [0, 10],\ x_3, x_5 \in [1, 5],\ x_4 \in [0, 6]\ .
\end{aligned}
$$

This problem has a relatively simple Pareto frontier, but the high number of constraints makes it a bit difficult for our algorithms (the `SIVIA-like` procedure has to find a feasible box).

The solver's parameters used for this problem are: $\varepsilon_y = 4.0$ and $\varepsilon_x = 0.25$.

4.3 SPH(3,3) Problem with a Nonlinear Constraint

This problem is the unconstrained problem from [29] with 3 variables and 3 criteria, augmented with a single constraint (considered also in [20]:

$$
\min_{x_1,\ldots,x_n} \Big(q_k(x_1,\ldots,x_n) = \sum_{\substack{j=1,\\ j\neq k}}^{N} x_j^2 + (x_k - 1)^2 \qquad k = 1,\ldots,N \Big) \tag{4}
$$

$$
\begin{aligned}
&\text{s.t.}\\
&-(x_1 + 2)^2 - x_2 + 1 \le 0\ ,\\
&x_i \in [-1000, 1000] \quad i = 1,\ldots,n\ .
\end{aligned}
$$

The Pareto frontier and Pareto-optimal set are three-dimensional ones and hence difficult to draw.

Obviously, $n = N = 3$, in our case. The solver's parameters used for this problem are: $\varepsilon_y = 0.2$ and $\varepsilon_x = 0.05$.

4.4 Results

In the tables below we present computational results for the described test problems and a few versions of our algorithm. Two versions of the algorithm, not applying the advanced tools have been considered:

- old – the algorithm with no improvements at all,
- simple – the quadruples with empty interior are not bisected, but put into an additional list; all quadruples that are not processed anymore are narrowed by computing the intersection of $\mathbf{y}$ and the union of criteria values for all their boxes from the decision space; no "advanced" improvements used.

The following "advanced" versions have been considered. All of them, in addition to tools from the "simple" version, switch between using or not of the Newton operator (see Subsection 3.1) and one of the heuristics to choose the coordinate for bisection (Subsection 3.2: MaxDiam, MaxSmear, Hybrid (i.e., the one presented in [22]), MaxDist or MaxDistAug.

Switching between different types of the Newton operator did not give any improvements (for the reasons that remain to be determined) and we do not present these results here. In all versions, using the 2nd order information, the GS (Gauss-Seidel) operator has been applied.

It is difficult to give a single quantity that measures the algorithm performance. The number of Hesse matrix evaluations might be a hint (as this is the most time-consuming operation), but some versions of the algorithm do not use Hesse matrices at all and they are not the most efficient ones, definitely. So, numbers of other operations (criteria evaluations, gradients evaluations, bisections, etc.) should be considered, also.

The execution time is the only general efficiency measure, but it is related to a specific machine.

As for the accuracy – Lebesgue measures of box collections seem a proper measure, but accuracies in the criteria and decision spaces might be much different; we do not have a perfect measure, again.

Numbers of boxes deleted or reduced by various tools can give us the idea on how efficient these mechanisms are.

We present all of these numbers in the tables below.

Table 1. Results for problem (2) and a single-threaded algorithm without advanced improvements

	old – 1st order	old – 2nd order	simple – 2nd order
criteria evals.	23655640	6445582	6372582
criteria grad. evals	6095650	1363678	1363678
criteria Hess. evals	—	1833418	1833418
bisecs.in crit.space	1825	1695	1371
bisecs.in dec. space	1471926	425994	425994
boxes reduced by monot.	0	1	1
boxes del. by monot.	29825	13374	13374
boxes del. by Newton I	290086	109313	109313
boxes del. by Newton II	—	24601	24601
resulting quadruples	854	737	417
quadr.with empty int.	—	—	315
internal boxes	122469	35933	35933
boundary boxes	942995	218147	219099
Lebesgue measure crit.	1.49	1.28	0.81
Lebesgue measure dec.	0.53	0.13	0.13
time (sec.)	83	81	81

We can see high improvement in the accuracy – both in the decision and criteria space. The improvement in efficiency is much smaller.

Table 2. Results for problem (2) and single-threaded algorithms with the advanced improvements

	MaxDiam	MaxSmear	Hybrid [22]	MaxDist	MaxDistAug
criteria evals.	6313992	12250258	6313992	13272688	9287190
criteria grad. evals	1499946	3297308	1499946	3724698	2377464
criteria Hesse evals	1632136	4168106	1632136	4819988	2848404
bisecs.in crit.space	1368	1351	1368	1344	1344
bisecs.in dec. space	432163	1069833	432163	1234705	741811
boxes reduced by monot.	1	1	1	1	1
boxes del. by monot.	11887	47223	11887	63995	22175
boxes del. by Newton I	113918	146350	113918	160074	136140
boxes del. by Newton II	28846	28137	28846	28043	27683
resulting quadruples	415	403	415	399	399
quadr.with empty int.	312	324	312	328	326
internal boxes	38008	278184	38008	323209	161717
boundary boxes	213483	481588	213483	527405	348493
Lebesgue measure crit.	0.80	0.77	0.80	0.78	0.78
Lebesgue measure dec.	0.13	0.11	0.13	0.11	0.11
time (sec.)	75	185	75	211	128

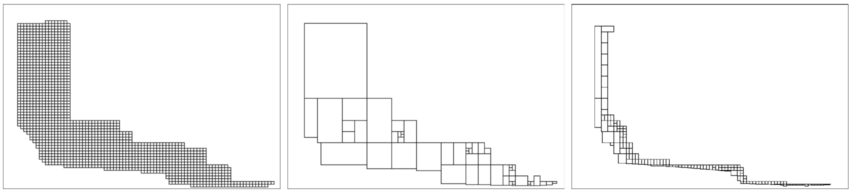


Fig. 1. Pareto frontier of the Osyczka problem (3) approximation, computed by the old, simple and MaxDistAug algorithm versions

For the Osyczka problem, we obtained a very significant improvement – as in accuracy, as in efficiency of the algorithm. Please note, that for the "simple" improvements, the accuracy is slightly decreased in the criteria space. This is because, we get a small number of relatively large boxes that cover some boxes deleted by the original algorithm (see Figure 1). Using advanced techniques, improves the accuracy (and efficiency) dramatically.

Table 3. Results for problem (3) and single-threaded algorithms without advanced improvements

	old	simple
criteria evals.	263187987	123750219
criteria grad. evals	11630028	11630028
criteria Hesse evals	0	0
constr. evals	24019414	24019414
constr. grad. evals	9014928	9014928
constr. Hesse evals	0	0
bisecs.in crit.space	1679	75
bisecs.in dec. space	2910382	2910382
boxes reduced by monot.	13306	13306
boxes del. by monot.	348936	348936
boxes del. by Newton I	8121	8121
boxes del. by Newton II	0	0
resulting quadruples	1524	3
quadr.with empty int.	—	49
internal boxes	10	10
boundary boxes	21594477	3672279
Lebesgue measure crit.	15287.62	17354.70
Lebesgue measure dec.	1210.25	195.79
time (sec.)	433	200

Table 4. Results for problem (3) and single-threaded algorithms with the advanced improvements

	MaxDiam	MaxSmear	Hybrid [22]	MaxDist	MaxDistAug
criteria evals.	105738157	n/a	105738157	n/a	46907598
criteria grad. evals	11818048	n/a	11818048	n/a	4533336
criteria Hesse evals	2	n/a	2	n/a	2
constr. evals	16115759	n/a	16115759	n/a	8115028
constr. grad. evals	9181265	n/a	9181265	n/a	4417105
constr. Hesse evals	6	n/a	6	n/a	6
bisecs.in crit.space	62	n/a	62	n/a	256
bisecs.in dec. space	2957692	n/a	2957692	n/a	1130313
boxes reduced by monot.	13423	n/a	13423	n/a	507
boxes del. by monot.	366467	n/a	366467	n/a	82417
boxes del. by Newton I	8433	n/a	8433	n/a	19324
boxes del. by Newton II	0	n/a	0	n/a	0
resulting quadruples	3	n/a	3	n/a	29
quadr.with empty int.	38	n/a	38	n/a	115
internal boxes	8	n/a	8	n/a	631
boundary boxes	3167975	n/a	3167975	n/a	975368
Lebesgue measure crit.	18038.83	n/a	18036.83	n/a	3547.30
Lebesgue measure dec.	160.70	n/a	160.70	n/a	1.22
time (sec.)	179	> 3537	180	> 1047	72

Table 5. Results for problem (4) and single-threaded algorithms with the advanced improvements

	MaxDiam	MaxSmear	Hybrid [22]	MaxDist	MaxDistAug
criteria evals.	5302025	3281457	5302025	n/a	3524538
criteria grad. evals	1535085	1057449	1535085	n/a	1198404
criteria Hesse evals	3	9	3	n/a	9
constr. evals	526237	222393	526237	n/a	221864
constr. grad. evals	392319	231366	392319	n/a	220673
constr. Hesse evals	1	1	1	n/a	1
bisecs.in crit.space	1157	1215	1157	n/a	1211
bisecs.in dec. space	244024	164247	244024	n/a	185624
boxes reduced by monot.	0	0	0	n/a	0
boxes del. by monot.	98707	53818	98707	n/a	54921
boxes del. by Newton I	8615	15845	8615	n/a	13282
boxes del. by Newton II	0	0	0	n/a	0
resulting quadruples	280	303	280	n/a	307
quadr.with empty int.	186	153	186	n/a	104
internal boxes	1289	1876	1289	n/a	6891
boundary boxes	38358	41593	38358	n/a	61511
Lebesgue measure crit.	2.37	2.28	2.37	n/a	1.82
Lebesgue measure dec.	1.43	1.37	1.43	n/a	1.07
time (sec.)	7	5	7	> 774	5

5 Interactive Decision Making Using the Pareto Frontier

Every point in the Pareto frontier and consequently in the Pareto-optimal set is an equally acceptable solution of the multi-criteria optimization problem from the formal point of view. However, making a decision means selecting one alternative (point) from the set of feasible points. So, in multi-criteria decision making (MCDM) this choice calls for the information that is not contained in criteria, that is, for the decision maker's preferences between different Pareto frontier points. This is why – compared to single criterion decision-making – a new element has to be added in MCDM. We need an active involvement of the decision-maker in doing the selection. From pioneering work of S. Gass and T. Saaty [9] it is known that this can be done by visualization of the Pareto frontier.

The improvements, we have done to our algorithm of computing Pareto sets, encourage us to begin the work on multiple criteria decision support software package (MCDSSP) designed in the framework of *a posteriori* methods for MCDM [25].

A standard *a posteriori* method includes four stages: (1) computing a Pareto frontier; (2) its visualization; (3) trade-off analysis of different points from the Pareto frontier, by the decision maker basing on her/his preferences and eventually the choice of one of the Pareto frontier points; (4) computing a Pareto-optimal point corresponding to the selected point from the Pareto frontier.

However, the designed MCDSSP to be effective, must satisfy some requirements formulated, for visualization and trade-offs calculation (cf. for example [24], [10]). These requirements are demanding and now we are working on meet all of them, especially on interactive presentation of impact of problem parameters change on Pareto sets shape.

6 Conclusions

We presented an improved and highly tuned interval algorithm for seeking the Pareto-optimal set and Pareto frontier of a multi-criteria decision problem. We also used the developed method as the basis to propose a new approach for interactive multi-criteria decision making.

Analyzing the performance of our algorithm, we made important observations. An interesting one is that quite different heuristics seem to perform well for the problem of Pareto sets seeking than for solving underdetermined nonlinear systems (see [22] for comparison). This is against our expectations, but it can be explained by several reasons:

– in [22] the Newton operator is the only rejection/reduction tool used and it is computed for almost all boxes; here a few acceleration mechanisms are used (in particular, the powerful multi-criteria version of the monotonicity test) and the Newton operator is applied only for sufficiently small boxes,
– in [22] the Newton operator is used to verify the segments of the solution manifold; here we only use the Newton operator to narrow and discard boxes, not to verify the existence of solutions,
– in the algorithm considered here, we have two kinds of variables – decision variables x and Lagrange multipliers u – used in the system of necessary conditions, solved by the Newton operator; however, only x's are bisected and not u's,
– in the algorithm considered here, significant parts are not devoted to processing the boxes, but to traversing the data structures, etc.

We can conclude that for different problems and algorithms different heuristics and policies are needed in interval computations. However, we can formulate some general remarks for them, basing on the results from both: this paper and [22]. In particular: heuristics for choosing the subdivision direction should consider several criteria of choice: assuring overall convergence, good cooperation with the rejection/reduction tests, improving the bounds on the objective(s) as much as possible, etc. For example policies, like MaxSmear, MaxDist (in contrast to MaxDistAug) seem to perform poorly.

But the choice of policies adequate to a specific problem/algorithm seems a relatively hard task and it can hardly be automated (compare also [23]). It seems to require experience and some intuition from the algorithm designer.

7 Future Work

There are still several improvements that can be done to our algorithm – both simple and sophisticated ones, e.g., improving the parallelization and dominated regions deletion by using proper data structures (see, e.g., [20], [23]). Also, it will be very interesting to compare the performance of our approach with the one of Fernandez and Toth [7]. This is going to be the subject of future investigations.

Acknowledgments. The paper has been done as a part of realization of the grant for statutory activity, financed by the Dean of Faculty of Electronics and Information Technology (WUT), titled “Interval methods for solving nonlinear problems”.

References

1. C-XSC interval library, http://www.xsc.de
2. Acioly, B.M., Bedregal, B.R.C.: A quasi-metric topology compatible with inclusion monotonicity on interval space. Reliable Computing 3, 305–313 (1997)
3. Barichard, V., Hao, J.-K.: A Population and Interval Constraint Propagation Algorithm. In: Fonseca, C.M., Fleming, P.J., Zitzler, E., Deb, K., Thiele, L. (eds.) EMO 2003. LNCS, vol. 2632, pp. 88–101. Springer, Heidelberg (2003)
4. Beelitz, T., Bischof, C.H., Lang, B.: A hybrid subdivision strategy for result-verifying nonlinear solvers. Tech.rep. 04/8, Bergische Universitiät Wuppertal (2004)
5. Csendes, T.: New subinterval selection criteria for interval global optimization. JOGO 19, 307–327 (2001)
6. Csendes, T.: Private communication
7. Fernandez, J., Toth, B.: Obtaining an outer approximation of the efficient set of nonlinear biobjective problems. Journal of Global Optimization 38, 315–331 (2007)
8. Ceberio, M., Modave, F.: Interval-based multicriteria decision making. In: Bouchon-Meunier, B., Coletti, G., Yager, R.R. (eds.) Modern Information Processing: From Theory to Applications. Elsevier (2006)
9. Gass, S., Saaty, T.: The computational algorithm for the parametric objective function. Naval Research Logistics Quarterly 2, 39 (1955)
10. Haimes, Y.Y.: Risk Modeling, Assessment and Management. J. Wiley, New York (1998)
11. Hansen, E., Walster, W.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (2004)
12. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. Springer, London (2001)
13. Kearfott, R.B.: Rigorous Global Search: Continuous Problems. Kluwer, Dordrecht (1996)
14. Kearfott, R.B., Nakao, M.T., Neumaier, A., Rump, S.M., Shary, S.P., van Hentenryck, P.: Standardized notation in interval analysis (2002), http://www.mat.univie.ac.at/~neum/software/int/notation.ps.gz
15. Kim, I.Y., de Weck, O.L.: Adaptive weighted-sum method for bi-objective optimization: Pareto front generation. Structural and Multidisciplinary Optimization 29, 149–158 (2005)
16. Kubica, B.J.: Interval methods for solving underdetermined nonlinear equations systems. In: SCAN 2008 Proceedings (2008); Reliable Computing 15, 207–217 (2011)

17. Kubica, B.J., Woźniak, A.: Interval Methods for Computing the Pareto-front of a Multicriterial Problem. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 1382–1391. Springer, Heidelberg (2008)
18. Kubica, B. J., Woźniak, A.: A multi-threaded interval algorithm for the Pareto-front computation in a multi-core environment. Presented at PARA 2008 Conference (2008); accepted for publication in LNCS 6126 (2012)
19. Kubica, B.J., Woźniak, A.: Optimization of the multi-threaded interval algorithm for the Pareto-set computation. Journal of Telecommunications and Information Technology 1, 70–75 (2010)
20. Kubica, B.J., Woźniak, A.: Computing Pareto-sets of multi-criteria problems using interval methods. Presented at SCAN 2010 (2010) (unpublished)
21. Kubica, B.J., Woźniak, A.: Using the Second-Order Information in Pareto-set Computations of a Multi-criteria Problem. In: Jónasson, K. (ed.) PARA 2010, Part II. LNCS, vol. 7134, pp. 137–147. Springer, Heidelberg (2012)
22. Kubica, B.J.: Tuning the Multithreaded Interval Method for Solving Underdetermined Systems of Nonlinear Equations. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part II. LNCS, vol. 7204, pp. 467–476. Springer, Heidelberg (2012)
23. Kubica, B.J.: A class of problems that can be solved using interval algorithms. In: SCAN 2010 Proceedings (2010); Computing 94 (2-4), 271–280 (2012)
24. McQuaid, M.J., Ong, T.-H., Chen, H., Nunamaker, J.F.: Multidimensional scaling for group memory visualization. Decision Support Systems 27, 163–176 (1999)
25. Miettinen, K.M.: Nonlinear Multiobjective Optimization. Kluwer Academic Publishers, Boston (1999)
26. Pal, L.: Global Optimization Algorithms for Bound Constrained Problems. PhD dissertation, University of Szeged (2010), http://www.emte.siculorum.ro/~pallaszlo/Disszertacio/Disszertacio/Disszertacio_PalLaszlo.pdf
27. Ratz, D., Csendes, T.: On the selection of subdivision directions in interval branch-and-bound methods for global optimization. JOGO 7, 183–207 (1995)
28. Ruetsch, G.R.: An interval algorithm for multi-objective optimization. Structural and Multidisciplinary Optimization 30, 27–37 (2005)
29. Zitzler, E., Laumanns, M., Thiele, M.: SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In: Giannakoglou, K., Tsahalis, D., Periaux, J., Papailiou, K., Fogarty, T. (eds.) Evolutionary Methods for Design Optimization and Control. CIMNE, Barcelona (2002)

A Central-Backward Difference Interval Method for Solving the Wave Equation

Andrzej Marciniak[1] and Barbara Szyszka[2]

Poznan University of Technology
[1] Institute of Computing Science
[2] Institute of Mathematics
Piotrowo 3A, 60-965 Poznan, Poland
{Andrzej.Marciniak,Barbara.Szyszka}@put.poznan.pl

Abstract. The paper is devoted to an interval difference method for solving one dimensional wave equation with the initial-boundary value problem. The method is an adaptation of the well-known central and backward difference methods with respect to discretization errors of the methods. The approximation of an initial condition is derived on the basis of expansion of a third-degree Taylor polynomial. The initial condition is also written in the interval form with respect to a discretization error. Therefore, the presented interval method includes all approximation errors (of the wave equation and the initial condition). The floating-point interval arithmetic is used. It allows to obtain interval solutions which contain all calculations errors. Moreover, it is indicated that an exact solution belongs to the interval solution obtained.

Keywords: wave equation, PDE, difference method, initial-boundary value problem, floating-point interval arithmetic, interval methods.

1 Introduction

Numerical computations for partial differential equations (PDE) are very important in scientific and technical calculations. In the approximate methods are important: the kind of discretization of the method and the size of discretization error. There is not much of works devoted to the partial differential equations in the context to interval computations. Nakao and Plum are the authors of papers related to the existence and uniqueness of these solutions. Almost all papers written by these authors are devoted to elliptic equations (see e.g. [1], [2] or [3]). The studies of partial differential equations with respect to interval methods in floating-point interval arithmetic are conducted by Marciniak for Poisson equation (see e.g. [4]) and by Jankowska (see e.g. [5]) for a heat equation. The main objective to create such methods is to take into account all numerical errors in the resulting solutions. It is obvious that we want to obtain the widths of interval solutions small enough. Difference interval methods for solving the wave equation have been presented by authors in [6], [7], [8] and [9]. The methods presented in

[6] and [7] are related to the errors of an initial condition of order $O(h)$ and estimations of errors dependent on a parameter of the wave equation (v). Another way of estimating errors in discretization method has been presented in [8] and [9]. There were presented the central-central and central-backward difference interval method and the estimations of errors independent on the parameter v. Additionally, an initial condition was considered with the local truncation error of order $O(h^4)$. But in [6], [7], [8] and [9] the interval solutions were acceptable only for $v < 1$. In this paper the central-backward (with respect to space and time, respectively) difference interval method for solving hyperbolic PDE with boundary-initial conditions is presented. The estimation error of discretization method is the same as the method presented in [8] and [9], but the initial condition is approximated (with local truncation error of order $O(h^4)$) by the new formula. Some numerical results are presented.

2 The Wave Equation

In this paper the one dimensional wave equation is considered, which corresponds to the tighten string of the length L (see [10], [11], [12] and [13]). The wave equation is given by the formula

$$v^2 \frac{\partial^2 u}{\partial x^2}(x,t) - \frac{\partial^2 u}{\partial t^2}(x,t) = 0. \tag{1}$$

The function $u = u(x,t)$ denotes the amplitude of the string displacement in the time t, where $0 < t$, $0 < x < L$ and $v = const.$

If both ends of the string are fixed at $x = 0$ and $x = L$, then the function u, for $t > 0$, satisfies the following Dirichlet conditions:

$$\begin{aligned} u(0,t) &= 0, \\ u(L,t) &= 0. \end{aligned} \tag{2}$$

The initial position and velocity of the string, for $0 < x < L$, are given by the following Cauchy conditions:

$$\begin{aligned} u(x,0) &= \varphi(x), \\ \frac{\partial u}{\partial t}(x,0) &= \psi(x), \end{aligned} \tag{3}$$

where φ and ψ are given functions.

3 The Central-Backward Difference Method

To define a difference method, the space x and time t are divided into n and m parts of the length $\Delta x = h > 0$ and $\Delta t = k > 0$, respectively (see e.g. [14]). The mesh points (x_i, t_j) of the space-time grid are set up as follows:

$$\begin{aligned} x_i &= i \cdot h, \quad h = \frac{L}{n} > 0, \quad i = 0, 1, \ldots, n, \\ t_j &= j \cdot k, \quad\quad\quad k > 0, \quad j = 0, 1, \ldots, m. \end{aligned} \tag{4}$$

Using the central difference approximation to $\frac{\partial^2 u}{\partial x^2}$ and the backward difference approximation to $\frac{\partial^2 u}{\partial t^2}$ (see e.g. [15] or [16]) we obtain

$$\begin{aligned}\frac{\partial^2 u}{\partial x^2}(x_i,t_j) &= \frac{u(x_{i+1},t_j)-2u(x_i,t_j)+u(x_{i-1},t_j)}{h^2} - \frac{h^2}{12}\cdot\frac{\partial^4 u}{\partial x^4}(\xi_i,t_j),\\ \frac{\partial^2 u}{\partial t^2}(x_i,t_j) &= \frac{u(x_i,t_j)-2u(x_i,t_{j-1})+u(x_i,t_{j-2})}{k^2} - \frac{k^2}{12}\cdot\frac{\partial^4 u}{\partial t^4}(x_i,\eta_{j-1}),\end{aligned} \tag{5}$$

where $\xi_i \in (x_{i-1},x_{i+1})$ and $\eta_{j-1} \in (t_{j-2},t_j), i = 1,2,\ldots,n-1, j = 2,3,\ldots,m$. Substituting (5) into (1) we get

$$\begin{aligned}\gamma^2 u(x_{i-1},t_j) + \gamma^2 u(x_{i+1},t_j) - (1+2\gamma^2)u(x_i,t_j) +\\ +2u(x_i,t_{j-1}) - u(x_i,t_{j-2}) = e_M,\end{aligned} \tag{6}$$

where

$$\gamma = v\frac{k}{h}, \tag{7}$$

and where the truncation error of method (e_M) is given by

$$e_M = v^2\frac{k^2h^2}{12}\cdot\frac{\partial^4 u}{\partial x^4}(\xi_i,t_j) - \frac{k^4}{12}\cdot\frac{\partial^4 u}{\partial t^4}(x_i,\eta_{j-1}) \tag{8}$$

for $\xi_i \in (x_{i-1},x_{i+1})$, $i = 1,2,\ldots,n-1$ and $\eta_{j-1} \in (t_{j-2},t_j)$, $j = 2,3,\ldots,m$. The method (6) is stable only if

$$\gamma \le 1. \tag{9}$$

The Dirichlet conditions (2), $j = 1,2,\ldots,m$, are as follows:

$$\begin{aligned}u(x_0,t_j) &= 0,\\ u(x_n,t_j) &= 0.\end{aligned} \tag{10}$$

If $u \in C^4[0,L]$ and φ'',ψ'' exist, then, for $i = 1,2,\ldots,n-1$, and $j = 0$, the Cauchy conditions (3) yield

$$\begin{aligned}u(x_i,t_0) &= \varphi(x_i),\\ u(x_i,t_1) &= \varphi(x_i) + k\psi(x_i) + \frac{k^2}{2}v^2\varphi''(x_i) + \frac{k^3}{6}v^2\psi''(x_i) + e_C.\end{aligned} \tag{11}$$

The approximation error of the second initial condition (e_C) is defined as follows:

$$e_C = \frac{k^4}{24}\cdot\frac{\partial^4 u}{\partial t^4}(x_i,\tilde{\eta}_i), \tag{12}$$

where $\tilde{\eta}_i \in (t_0,t_1)$ for $i = 1,2,\ldots,n-1$.

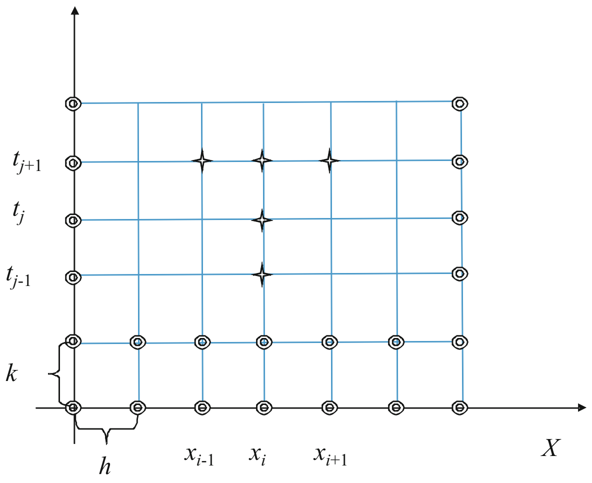


Fig. 1. The model of the mesh points grid given by the formula (6) with conditions (10) and (11)

The second equation in (11) is received by an approximation of the derivative

$$\frac{\partial u}{\partial t}(x_i, 0).$$

The third-degree Taylor polynomial for function u expanded about t_0 is used as follows:

$$u(x_i, t_1) = u(x_i, t_0) + k \cdot \frac{\partial u}{\partial t}(x_i, t_0) + \frac{k^2}{2} \cdot \frac{\partial^2 u}{\partial t^2} + (x_i, t_0) \\ + \frac{k^3}{6} \cdot \frac{\partial^3 u}{\partial t^3}(x_i, t_0) + \frac{k^4}{24} \cdot \frac{\partial^4 u}{\partial t^4}, (x_i, \tilde{\eta_i}), \tag{13}$$

where $\tilde{\eta_i} \in (t_0, t_1)$ for $i = 1, 2, \ldots, n-1$.
The first initial condition in (3) satisfies the wave equation (1). Thus, it is differentiated twice with respect to x (see [17]) and then we can write

$$\frac{\partial^2 u}{\partial t^2}(x_i, 0) = v^2 \frac{\partial^2 u}{\partial x^2}(x_i, 0) = v^2 \frac{d^2 \varphi}{dx^2}(x_i) = v^2 \varphi''(x_i).$$

As a consequence we have

$$\frac{\partial^2 u}{\partial t^2}(x_i, 0) = v^2 \varphi''(x_i). \tag{14}$$

Likewise, if ψ'' exists, then

$$\frac{\partial^3 u}{\partial t^3}(x_i, 0) = \frac{\partial}{\partial t}\left(\frac{\partial^2 u}{\partial t^2}(x_i, 0)\right) = \frac{\partial}{\partial t}\left(v^2 \frac{\partial^2 u}{\partial x^2}(x_i, 0)\right) = \\ = v^2 \frac{\partial^2}{\partial x^2}\left(\frac{\partial u}{\partial t}(x_i, 0)\right) = v^2 \frac{d^2 \psi}{dx^2}(x_i) = v^2 \psi''(x_i).$$

Finally, we obtain

$$\frac{\partial^3 u}{\partial t^3}(x_i, 0) = v^2 \psi''(x_i). \tag{15}$$

Substituting (14), (15) and the first equation of (11) into (13), the second equation in (11) is obtained.

4 A Central-Backward Difference Interval Method

In the conventional difference methods the truncation errors e_M (8) and e_C (12) are omitted. If we consider the interval method, the errors of method are involved into interval solution (see e.g. [18], [19] or [20]). The equations below are satisfied for all mesh points $(x_i, t_j) \in (X_i, T_j)$, where (X_i, T_j) are interval representations of mesh points (x_i, t_j). The functions Φ, Ψ and the values K, H, Γ are interval extensions ([21]) of the functions φ, ψ and the values k, h, γ, respectively.

Substituting (5) into (1) the central-backward difference interval method is defined by the formula

$$\begin{aligned} \Gamma^2 U(X_{i-1}, T_j) + \Gamma^2 U(X_{i+1}, T_j) - (1 + 2\Gamma^2) U(X_i, T_j) + \\ + 2U(X_i, T_{j-1}) - U(X_i, T_{j-2}) = E_M \end{aligned} \tag{16}$$

for $i = 1, 2, \ldots, n-1$ and $j = 2, 3, \ldots, m$.
The estimation of truncation error e_M (8) is defined as follows (see [8]):

$$E_M = \frac{K^2}{12}(H^2 + K^2 V^2)[-M, M], \tag{17}$$

where

$$e_M \in E_M. \tag{18}$$

The Dirichlet conditions (2) for $j = 1, 2, \ldots, m$ give

$$\begin{aligned} U(X_0, T_j) &= [0, 0], \\ U(X_n, T_j) &= [0, 0]. \end{aligned} \tag{19}$$

The Cauchy conditions (3), based on (11), for $i = 1, 2, \ldots, n-1$ yield

$$\begin{aligned} &U(X_i, T_0) = \Phi(X_i), \\ &U(X_i, T_1) = \Phi(X_i) + K\Psi(X_i) + \frac{K^2 V^2}{2}\Phi''(X_i) + \frac{K^3 V^2}{6}\Psi''(X_i) + E_C. \end{aligned} \tag{20}$$

The estimation of truncation error e_C (11) we can write as follows:

$$E_C = \frac{K^4 V^2}{24}[-M, M], \tag{21}$$

where

$$e_C \in E_C. \tag{22}$$

The value M (in formulas (17) and (21)) satisfying

$$\left| \frac{\partial^4 u(x_i,t_j)}{\partial x^2 \partial t^2} \right| \le M \tag{23}$$

is calculated by the conventional difference method as follows:

$$\begin{aligned} M \simeq \frac{s}{h^2k^2} \max_{i,j} \Big| & u(x_{i+1},t_{j+1}) - 2u(x_i,t_{j+1}) + u(x_{i-1},t_{j+1}) - \\ & -2(u(x_{i+1},t_j) - 2u(x_i,t_j) + u(x_{i-1},t_j)) + \\ & +u(x_{i+1},t_{j-1}) - 2u(x_i,t_{j-1}) + u(x_{i-1},t_{j-1}) \Big|, \end{aligned} \tag{24}$$

where $i = 1,2,\ldots,n-1, j = 1,2,\ldots,m-1$ and $s = 1.5$ (see [22]).

The central-backward difference interval method (16) leads to the system of $(m-1)$ interval linear equations (see [23]) for the $(m-1)$ unknown vectors U_j $(j = 2,3,\ldots,m)$. The system has the following form:

$$\begin{bmatrix} A & & & & \\ B & A & & & \\ C & B & A & & \\ & \ddots & \ddots & \ddots & \\ & & C & B & A \end{bmatrix} \cdot \begin{bmatrix} U_2 \\ U_3 \\ \vdots \\ \vdots \\ U_m \end{bmatrix} = \begin{bmatrix} R_2 \\ R_3 \\ \vdots \\ \vdots \\ R_m \end{bmatrix}, \tag{25}$$

with square blocks of size $(n-1)$, where the tridiagonal matrix A is given by

$$A = \begin{bmatrix} -(1+2\Gamma^2) & \Gamma^2 & & & \\ \Gamma^2 & \ddots & \ddots & & \\ & \ddots & \ddots & \Gamma^2 & \\ & & \Gamma^2 & -(1+2\Gamma^2) \end{bmatrix} \tag{26}$$

and

$$\begin{aligned} B &= 2 \cdot I, \\ C &= -I, \end{aligned} \tag{27}$$

where I is the identity matrix.
The unknown vectors U_j and the vectors of constants R_j $(j = 2,3,\ldots,m)$ occurring in (25) are as follows:

$$U_j = \begin{bmatrix} U_{1,j} \\ U_{2,j} \\ \vdots \\ U_{n-1,j} \end{bmatrix}, R_j = \begin{bmatrix} R_{1,j} \\ R_{2,j} \\ \vdots \\ R_{n-1,j} \end{bmatrix}. \tag{28}$$

Taking into account the Dirichlet (19) and the Cauchy (20) conditions, the elements $R_{i,j}$ $(i = 1, 2, \ldots, n-1, j = 2, 3, \ldots, m)$ has been derived and written in the forms as follows:

$$
\begin{aligned}
&j = 2: \\
&\quad R_{i,j} = E_M - \Phi(X_i) - 2\left(K\Psi(X_i) + \frac{K^2V^2}{2}\Phi''(X_i) + \frac{K^3V^2}{6}\Psi''(X_i) + E_C\right), \\
&j = 3: \\
&\quad R_{i,j} = E_M + \Phi(X_i) + K\Psi(X_i) + \frac{K^2V^2}{2}\Phi''(X_i) + \frac{K^3V^2}{6}\Psi''(X_i) + E_C, \\
&j = 4, 5, \ldots, m: \\
&\quad R_{i,j} = E_M.
\end{aligned}
\tag{29}
$$

On the basis of the Gaussian elimination with complete pivoting, an interval algorithm is used for solving the interval linear system of equations (25).

All calculations for the central-backward difference interval method we have performed in floating-point interval arithmetic using *IntervalArithmetic* unit written in Delphi Pascal language (see [24]). The application of *IntervalArithmetic* unit leads to interval solutions, which contain initial-data errors, data representation errors and rounding errors. Then, applying the interval method presented in this paper and floating-point interval arithmetic allow us to get the interval solutions containing all numerical errors.

5 Numerical Experiments

Let us consider an electric transmission line of the length L_x (the problem is presented in [17]), carrying alternating current of high frequency (”lossless” line). The voltage V is described by

$$\frac{\partial^2 V}{\partial x^2} = LC\frac{\partial^2 V}{\partial t^2},$$

for $0 < x < L_x$ and $0 < t$. The length of a line L_x, the inductance per unit length L and the capacitance per unit length C are as follows:

$$
\begin{aligned}
L_x &= 200[ft], \\
L &= 0.3[henries/ft], \\
C &= 0.1[farads/ft].
\end{aligned}
$$

The voltage V satisfies the following Dirichlet and Cauchy conditions:

$$
\begin{aligned}
V(0,t) = V(200,t) &= 0, && 0 < t, \\
V(x,0) &= 110\sin\frac{\pi x}{200}, && 0 \le x \le 200, \\
\frac{\partial V}{\partial t}(x,0) &= 0, && 0 \le x \le 200.
\end{aligned}
$$

The voltage V during the time $t = 0.5$ [sec.] has been calculated.
The wave propagation speed for the problem is

$$v = \frac{1}{\sqrt{LC}} \approx 5.8.$$

To obtain the interval solutions, the value of M (24) for estimations errors is needed.
For the voltage V we have

$$M = 3.3 \cdot 10^{-4}.$$

The solutions for the voltage V are presented in Table 1.

Table 1. Interval solutions and estimations of errors for the voltage V at $(x, t) = (100, 0.25)$

$n = m$	INT.left	INT.right	INT.width	E_C	E_M
10	**109.97**10803	**109.97**27505	$1.67 \cdot 10^{-3}$	$3 \cdot 10^{-9}$	$3 \cdot 10^{-5}$
20	**109.97**15332	**109.97**20302	$4.97 \cdot 10^{-4}$	$2 \cdot 10^{-10}$	$2 \cdot 10^{-6}$
30	**109.971**6293	**109.971**8774	$2.48 \cdot 10^{-4}$	$4 \cdot 10^{-11}$	$3 \cdot 10^{-7}$
40	**109.971**6630	**109.971**8230	$1.60 \cdot 10^{-4}$	$1 \cdot 10^{-11}$	$1 \cdot 10^{-7}$
50	**109.971**6766	**109.971**7994	$1.23 \cdot 10^{-4}$	$5 \cdot 10^{-12}$	$4 \cdot 10^{-8}$
60	**109.971**6804	**109.971**7903	**1.10** $\cdot 10^{-4}$	$2 \cdot 10^{-12}$	$2 \cdot 10^{-8}$
70	**109.971**6751	**109.971**7922	$1.17 \cdot 10^{-4}$	$1 \cdot 10^{-12}$	$1 \cdot 10^{-8}$
80	**109.9**659786	**109.9**774865	$1.15 \cdot 10^{-2}$	$7 \cdot 10^{-13}$	$7 \cdot 10^{-9}$
90	39.9446238	179.9988399	$1.40 \cdot 10^{2}$	$4 \cdot 10^{-13}$	$4 \cdot 10^{-9}$

In Table 1 we can see that if we have the greater value $n(= m)$, then the smaller widths of interval solutions are obtained. Such a situation is for $n(= m) \leq 60$. For $n(= m) = 60$ the width of interval solution is the smallest. For $n(= m) > 60$ the widths of interval solutions are wider and wider, despite of the values of estimations of errors e_M (8) and e_C (12) are smaller and smaller.
The solutions obtained by the conventional central-backward difference method are more accurate for the denser space-time grid (only if the stability condition (9) is satisfied). For the central-backward difference interval method implemented in floating-point interval arithmetic can be similarly (then the widths of interval solutions are smaller and smaller). But there are some positive values of n and m, for which the diameters of interval solutions increase. Then denser grid of mesh points is pointless.

The maximal widths of interval solutions for various $n(= m)$ are presented in Figure 2.

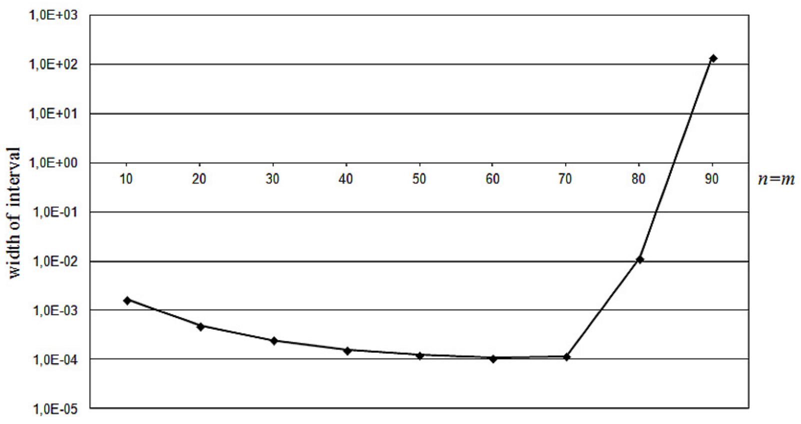


Fig. 2. The maximal widths of interval solutions for the voltage V

6 Conclusions

In the central-backward difference interval method the following values have a significant influence on the quality of obtained solutions:

- the estimation of error of the central-backward difference method (e_M (8)) – given by E_M (17),
- the way of an approximation of the second initial condition (3) – by the second formula (11),
- the estimation of approximation error(for expression e_C (12)) – the value of E_C (21),
- the estimation of M (24).

In this paper a new formula for approximation of the initial condition (3) is proposed by the expression (11). It allows us to solve the wave equation (1) for greater value of v.

The local truncation error (of order $O(h^4)$) of the second initial condition e_C (12) is a special case of the central-backward difference method truncation error e_M (8). Thus, the value of M (24) is considered as the same value in both estimations: E_M (17) and E_C (21). Additionally, the estimations of errors do not depend on the value of v.

References

1. Nakao, M.T., Watanabe, Y.: An Efficient Approach to the Numerical Verification for Solutions of Elliptic Differential Equations. Numerical Algorithms 37, 311–323 (2004)
2. Nakao, M.T., Yamamoto, N.: A simplified method of numerical verification for nonlinear elliptic equations. In: The Proceedings of International Symposium on Nonlinear Theory and its Applications (NOLTA 1995), Las Vegas, USA, pp. 263–266 (1995)

3. Plum, M.: Numerical existence proofs and explicit bounds for solutions of nonlinear elliptic boundary value problems. Computing 49, 25–44 (1992)
4. Marciniak, A.: An Interval Difference Method for Solving the Poisson Equation the First Approach. Pro Dialog 24, 49–61 (2008)
5. Jankowska, M.A.: An Interval Backward Finite Difference Method for Solving the Diffusion Equation with the Position Dependent Diffusion Coefficient. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part II. LNCS, vol. 7204, pp. 447–456. Springer, Heidelberg (2012)
6. Szyszka, B., Marciniak, A.: An interval method for solving some partial differential equations of second order. In: 80th Annual Meeting of the International Association of Applied Mathematics and Mechanics, GAMM, Proceedings on CD (2009)
7. Szyszka, B.: Central difference interval method for solving the wave equation. In: Simos, T.E., Psihoyios, G., Tsitouras, C. (eds.) International Conference on Numerical Analysis and Applied Mathematics, ICNAAM 2010, AIP Conference Proceedings, vol. 1281, pp. 2173–2176 (2010)
8. Szyszka, B.: The Central Difference Interval Method for Solving the Wave Equation. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part II. LNCS, vol. 7204, pp. 523–532. Springer, Heidelberg (2012)
9. Szyszka, B.: An Interval Difference Method for Solving Hyperbolic Partial Differential Equations. In: Mathematical Methods and Techniques in Engineering and Environmental Science, MACMESE Conference, pp. 330–334. WSEAS Press (2011)
10. Evans, L.C.: Partial Differential Equations. PWN Warszawa (2008) (in Polish)
11. Jr. Martin, R.H.: Elementary Differential Equations with Boundary Value Problems. McGraw-Hill Book Company, USA (1984)
12. Powers, D.L.: Elementary differential equations with boundary value problems. Prindle, Weber, Schmidt, Boston (1985)
13. Zill, D.G.: Differential Equations with Boundary-Value Problems. Prindle, Weber and Schmidt, Boston (1986)
14. Kacki, E.: Partial Differential Equations in physics and techniques problems. WNT Warszawa (1995) (in Polish)
15. Davis, J.L.: Finite Difference Methods in Dynamics of Continuous Media. Macmillan Publishing Company, NY (1986)
16. Kincaid, D., Cheney, W.: Numerical Analysis, Mathematics of Scientific Computing. The University of Texas at Austin (2002)
17. Burden, R.L., Faires, J.D.: Numerical Analysis. Prindle, Weber and Schmidt, Boston (1985)
18. Kalmykov, S.A., Schokin, Y.I., Yuldaschew, Z.H.: Methods of Interval Analysis. Nauka, Novosibirsk (1986)
19. Moore, R.E.: Methods and Applications of Interval Analysis. SIAM (1979)
20. Moore, R.E.: Introduction to Interval Analysis. SIAM (2009)
21. Szokin, Y.I.: Interval Analysis. Nauka, Novosibirsk (1981) (in Russian)
22. Marciniak, A.: An Interval Difference Method for Solving the Poisson Equation – the First Approach. Pro Dialog 24, 49–61 (2008)
23. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press, Cambridge (1990)
24. Gajda, K., Marciniak, A., Marlewski, A., Szyszka, B.: A Layout of an Object-Oriented System for Solving the Initial Value Problem by Interval Methods of Runge-Kutta Type. Pro Dialog 8, 39–62 (1999) (in Polish)

Part VII

Collection of Extended Abstracts

Using State-of-the-Art Sparse Matrix Optimizations for Accelerating the Performance of Multiphysics Simulations[*]

Vasileios Karakasis[1], Georgios Goumas[1], Konstantinos Nikas[1], Nectarios Koziris[1], Juha Ruokolainen[2], and Peter Råback[2]

[1] National Technical University of Athens, Greece
[2] CSC – IT Center for Science Ltd., Finland

1 Introduction

Multiphysics simulations are at the core of modern Computer Aided Engineering (CAE) allowing the analysis of multiple, simultaneously acting physical phenomena. These simulations often rely on Finite Element Methods (FEM) and the solution of large linear systems which, in turn, end up in multiple calls of the costly Sparse Matrix-Vector Multiplication (SpM$\times$V) kernel. The major—and mostly inherent—performance problem of the this kernel is its very low flop:byte ratio, meaning that the algorithm must retrieve a significant amount of data from the memory hierarchy in order to perform a useful operation. In modern hardware, where the processor speed has far overwhelmed that of the memory subsystem, this characteristic becomes an overkill [1]. Indeed, our preliminary experiments with the Elmer multiphysics package [3] showed that 60–90% of the total execution time of the solver was spent in the SpM$\times$V routine. Despite being relatively compact, the widely adopted Compressed Sparse Row (CSR) storage format for sparse matrices cannot compensate for the very low flop:byte ratio of the SpM$\times$V kernel, since it itself has a lot of redundant information. We have recently proposed the Compressed Sparse eXtended (CSX) format [2], which applies aggressive compression to the column indexing structure of CSR. Instead of storing the column index of every non-zero element of the matrix, CSX detects dense substructures of non-zero elements and stores only the initial column index of each substructure (encoded as a delta distance from the previous one) and a two-byte descriptor of the substructure. The greatest advantage of CSX over similar attempts in the past [5,4] is that it incorporates a variety of different dense substructures (incl. horizontal, vertical, diagonal and 2-D blocks) in a single storage format representation allowing high compression ratios, while its baseline performance, i.e., when no substructure is detected, is still higher than CSR's. The considerable reduction of the sparse matrix memory footprint achieved by CSX alleviates the memory subsystem significantly, especially for

[*] The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007–2013) under grant agreement n° RI-261557.

shared memory architectures, where an average performance improvement of more than 40% over multithreaded CSR implementations can be observed.

In this paper, we integrate CSX into the Elmer [3] multiphysics simulation software and evaluate its impact on the total execution time of the solver. Elmer employs iterative Krylov subspace methods for treating large problems using the Bi-Conjugate Gradient Stabilized (BiCGStab) method for the solution of the resulting linear systems. To ensure a fair comparison with CSX, we also implemented and compared a multithreaded version of the CSR used by Elmer. CSX amortized its preprocessing cost within less than 300 linear system iterations and built an up to 20% performance gain in the overall solver time after 1000 linear system iterations. To our knowledge, this is one of the first attempts to evaluate the real impact of an innovative sparse-matrix storage format within a 'production' multiphysics software.

The rest of the paper is organized as follows: Section 2 describes the CSX storage format briefly, Section 3 presents our experimental evaluation process and the performance results, and Section 4 concludes the paper and designates future work directions.

2 Optimizing SpM×V for Memory Bandwidth

The most widely used storage format for non-special (e.g., diagonal) sparse matrices is the Compressed Sparse Row (CSR) format. CSR compresses the row indexing information needed to locate a single element inside a sparse matrix by keeping only *number-of-rows* 'pointers' to the start of each row (assuming a row-wise layout of the non-zero elements) instead of *number-of-nonzeros* indices. However, there is still a lot of redundant information lurking behind the column indices, which CSR keeps intact in favor of simplicity and straightforwardness. For example, it is very common for sparse matrices, especially those arising from physical simulations, to have sequences of continuous non-zero elements. In such cases, it would suffice to store just the column index of the first element and the size of the sequence. CSX goes even further by replacing the column indices with the delta distances between them, which can be stored with one or two bytes in most of the cases, instead of the typical four-byte integer representation of the full column indices.

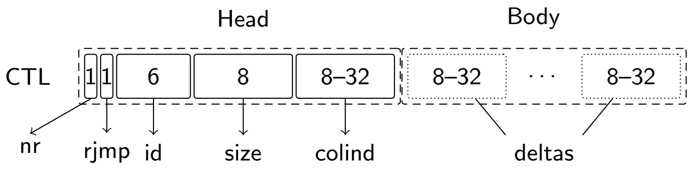


Fig. 1. The data structure used by CSX to encode the column indices of a sparse matrix

Figure 1 shows in detail the data structure (`ctl`) used by CSX to store the column indices of the sparse matrix. The main component of the `ctl` structure

Table 1. The test problems used for the experimental evaluation

Problem name	*Equations involved*	*SpM×V exec. time (%)*
fluxsolver	Heat + Flux	57.4
HeatControl	Heat	57.5
PoissonDG	Poisson + Discontinuous Galerkin	62.0
shell	Reissner-Mindlin	83.0
vortex3d	Navier-Stokes + Vorticity	92.3

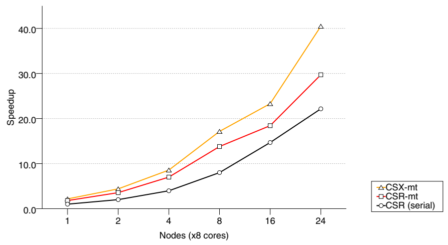
is the *unit*, which encodes either a dense substructure or a sequence of delta distances of the same type. The unit is made up of two parts: the *head* and the *body*. The head is a multiple byte sequence that stores basic information about the encoded unit. The first byte of the head stores a unique 6-bit ID of the substructure being encoded (e.g., 2×2 block) plus some metadata information for changing and/or jumping rows, the second byte stores the size of the substructure (e.g., 4 in our case), while the rest store the the initial column index of the encoded substructure as a delta distance from the previous one in a variable-length field. The body can be either empty, if the type ID refers to a dense substructure, or store the delta distances, if a unit of delta sequences is being encoded.

CSX supports all the major dense substructures that can be encountered in a sparse matrix (horizontal, vertical, diagonal, anti-diagonal and row- or column-oriented blocks) and can easily be expanded to support more. For each encoded unit, we use LLVM to generate substructure-specific optimized code in the runtime. This adds significantly to the flexibility of CSX, which can support indefinitely many substructures, provided that only 64 are encountered simultaneously in the same matrix. The selection of substructures to be encoded by CSX is made by a heuristic favoring those encodings that lead to higher compression ratios.

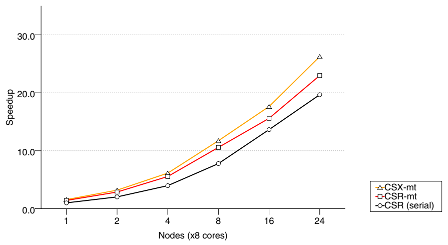
Detecting so many substructures inside a sparse matrix though, can be costly and this is not strange to CSX. Nonetheless, we have managed to considerably reduce the preprocessing cost without losing in performance by examining a mere 1% of the total non-zero elements using samples uniformly distributed all over the matrix.

3 Experimental Evaluation

The integration of the CSX storage format into the rest of the Elmer code was straightforward; Elmer can delegate the SpM×V computation to a user-specific shared library loaded at runtime, so implementing the required library interface was enough to achieve a seamless integration. The default implementation of CSR inside Elmer is single-threaded, but we also implemented a multithreaded version to perform a fair comparison with the multithreaded CSX. Our experimental platform consisted of 192 cores (24 nodes of two-way quad-core Intel Xeon E5405 [Harpertown] processors interconnected with 1 Gbps Ethernet) running Linux 2.6.38. We used GCC 4.5 for compiling both Elmer (latest version



(a) Speedup of the total execution time spent inside the SpM×V library, including the preprocessing cost in the case of CSX.



(b) Speedup of the total solver time.

Fig. 2. Average speedup of the Elmer code up to 192 cores using the CSX library (1000 linear system iterations)

from the SVN repository) and the CSX library along with LLVM 2.9 for the runtime code generation for CSX. Table 1 shows the 5 problems we selected from the Elmer test suite for the evaluation of our integration. We have appropriately increased the size of each problem to be adequately large for our system. Specifically, we opted for problem sizes leading to matrices with sizes larger than 576 MiB, which is the aggregate cache of the 24 nodes we used. Finally, we have used a simple Jacobi (diagonal) preconditioner for all the tested problems.

Figure 2 shows the average speedups achieved by simply the SpM×V code (Fig. 2(a)) and the total solver time (Fig. 2(b)) using the original Elmer CSR, our multithreaded CSR version and the CSX (incl. the preprocessing cost), respectively. In the course of 1000 linear system iterations, CSX was able to achieve a significant performance improvement of 37% over the multithreaded CSR implementation, which translates to a noticeable 14.8% average performance improvement of the total execution time of the solver. Nevertheless, we believe that this improvement could be even higher if other parts of the solver exploited parallelism within a single node as well, since the SpM×V component would become then even more prominent, allowing a higher performance benefit from the CSX optimization. Concerning the preprocessing cost of CSX, we used the typical case of exploring all the candidate substructures using matrix sampling, and yet it was able to fully amortize its cost within 224–300 linear system iterations.

4 Conclusions and Future Work

In this paper, we presented and evaluated the integration of the recently proposed Compressed Sparse eXtended (CSX) sparse matrix storage format into the Elmer multiphysics software package, being one of the first approaches of evaluating the impact of an innovative sparse matrix storage format on a 'real-life' production multiphysics software. CSX was able to improve the performance of the SpM×V component nearly 40% compared to the multithreaded CSR and offered a 15% overall performance improvement of the solver in a 24-node, 192-core SMP cluster. In the near future, we plan to expand our evaluation to NUMA architectures and even larger systems. Additionally, we are investigating ways for minimizing the initial preprocessing cost of CSX and also extensions to the CSX's interface to support efficiently problem cases where the non-zero values of the sparse matrix change during the simulation. Finally, we plan to investigate sparse matrix reordering techniques and how these affect the overall execution time of the solver using the CSX format.

References

1. Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N.: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. The Journal of Supercomputing 50(1), 36–77 (2009)
2. Kourtis, K., Karakasis, V., Goumas, G., Koziris, N.: CSX: An Extended Compression Format for SpMV on Shared Memory Systems. In: Proceedings of the 16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 2011), pp. 247–256. ACM, San Antonio (2011)
3. Lyly, M., Ruokolainen, J., Järvinen, E.: ELMER – a finite element solver for multiphysics. In: CSC Report on Scientific Computing (1999–2000)
4. Pinar, A., Heath, M.T.: Improving performance of sparse matrix-vector multiplication. In: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing. ACM, Portland (1999)
5. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. Journal of Physics: Conference Series 16(521) (2005)

Schedule Optimisation for Interactive Parallel Structure Simulations

Jovana Knežević, Ralf-Peter Mundani, and Ernst Rank

Chair for Computation in Engineering, Technische Universität München
{knezevic,mundani,rank}@bv.tum.de

Introduction

Computational effort required for the interactive parallel simulation of a structure under loading usually does not allow for results to be gained rapidly. Further, custom decomposition techniques, as in the case of long structures such as thigh bones, typically hinder the efficient exploitation of the underlying computing power. For the algebraic equations, gained by the *p*-version Finite Element Method (*p*-FEM) describing the behaviour of one such structure, $K \cdot u = d$ with K the system stiffness matrix, u the nodal displacements, and d all accumulated forces, often, due to the poor condition numbers, sophisticated iterative solvers fail to be efficient. As it is pointed out in [1, 3], applying hierarchical concepts, based on a nested dissection approach (i. e. recursive domain decomposition technique based on Schur complements, the de-facto standard of most domain decomposition approaches), allow for both the design of sophisticated direct solvers as well as for advanced parallelisation strategies, both of which are indispensable within interactive applications. Our main goal is the development of an efficient load balancing strategy for the existing structure simulation of the bone stresses with *p*-FEM organised via octrees as described in [2].

Parallelisation Strategy

Scheduling – in order to minimise the completion time of a parallel application by properly allocating the tasks to the computing resources – typically involves trade-off between the uniform work load distribution among all processors as well as keeping both the communication and optimisation costs minimal. For hierarchically organised tasks with the bottom-up dependencies, such as in our generated octree structure, for the classical scheduling based on the task locality, i. e. the property of belonging to the particular sub-tree, the number of processors which can be simultaneously exploited is decreasing by the factor of eight in each level closer to the root of the tree. Although dynamic load balancing strategies using hybrid parallelisation patterns perform excellent for the nested dissection solver [1], in interactive applications which assume aforementioned frequent updates from user's side, those rapid changes of the simulation and tasks' state favour static load balancing strategies. Furthermore, in our case, it would have to be taken into consideration that certain modifications performed by a user may involve major changes of the computational model. Consequently, for repeatedly achieving the optimal amount of work being assigned to each process for each new user update, the overhead-prone scheduling step has to be executed each time. Therefore, we need an efficient, nevertheless simple to compute scheduling optimization approach, not damming the intended interactive process, which is, according to our best knowledge, out of the focus of the already existing sophisticated optimisation strategies.

Scheduling of Tasks

The prerequisites we assume known a priori for a certain input data are the sizes of the initial tasks, data dependencies, and synchronisation requirements. Traversing the tree bottom-up, we estimate first the number of operations needed for processing each node, i. e. the necessary floating-point operations for performing a partial Gaussian elimination. This is considered to be the *weight* of that node, i. e., a rough estimation of the amount of work to be done in the nested dissection solver for that particular node, representing one task. In our approach, we avoid common assumptions “which are apt to have restricted applicability in real environments” [4], such as the target parallel architecture or uniform task execution times. However, in this stage of the project, the data locality is neglected and the load balance highly prioritized. Moreover, since the sizes of the tasks vary notably and in order to avoid single processors becoming bottlenecks having to finish considerably big tasks, at any point during the execution we allow to split a single task among several processors when mapping all tasks to the processors. The splitting is done based on the comparison of a task’s estimated work with a ‘unit’ task, i. e. the one referring to an octree leaf or in terms of FEM to a single element.

Since the scheduling problem can be solved by polynomial-depth backtrack search, thus, is NP complete for most of its variants, efficient heuristics have to be devised. In our case, in addition to the sizes of the tasks, the dependencies among them play a significant role, thus, before we make a decision how to assign tasks to processors two aspects have to be considered. The first is *the level of the task dependency* in the tree hierarchy. Namely, children nodes have to be processed before their parent nodes. Second, among the tasks of the same dependency level we distinguish between different levels in the tree hierarchy, calling this property the *processing order*. Assuming the depth of the tree is N, the tasks from level M in the hierarchy have the processing order of $N - M - 1$.

Then we form lists of priorities, based on finding the task with both the lowest level of dependency and the lowest order of processing. In this way, we make sure that the tasks inside very long branches of the tree with an estimated bigger load are still given a priority. Within the tasks having exactly the same priority according to this classification, inspired by state-of-art heuristics [5] and provided that we have more tasks in the same priority list than we have processors, we resort to a so-called *Max-min* order, making sure that big tasks, in terms of their estimated quantity of computations they involve, are the first ones assigned to the processors. Although we follow the same principle, in order to avoid additional sorting of the tasks, only partial sorting is applied, i. e. filling in our priority lists of by adding those bigger than a certain predetermined limit to the front of the list and the smaller ones to the end.

This way we end up with arrays of tasks, so-called 'phases', each phase consisting of as many tasks or their parts as there are computing resources. Namely, taken from the priority lists, tasks will be assigned to the phases in round-robin manner, while splitting tasks if necessary among several processors. Those phases refer exactly to the mapping which will be done. In the case of bulky tasks, i. e. those whose size divided by the number of available resources exceeds the size of one smallest processing unit, the task is just equally split among all the processors. The results presented in the Figure 1 illustrate that our goal of having the capacity of each phase as "full" as possible, i. e. all the processors busy with the approximately equal amount of work throughout the solver execution, is achieved.

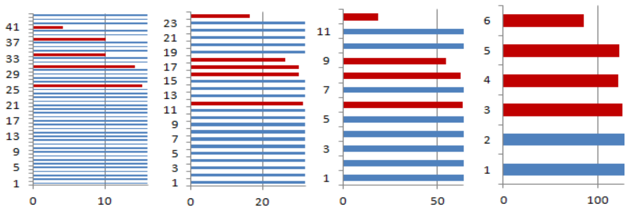


Fig. 1. "Scheduling phases": vertical axis describes the so-called phases, horizontal axis the number of processors involved in the particular phase. The balance is optimal (in terms of the number of phases) for 64 and 128 processors (two diagrams on the right). For 16 processors it results in 44 phases (left-hand side) where 43 is an optimum (last 3 phases refer to bulky tasks) and for 32 processors in 24 phases whereas optimum is 23.

Communication Pattern

At the beginning, all the working processes will receive all the tasks they are in charge of and start processing them in a sequential way. Here, we distinguish between tasks without dependencies, which can be processed immediately, and those which need information from other processors. For those which can be processed immediately, the assembly step is done and the required data is sent to the processors which will need it for the assembly of their own tasks. Otherwise, all the related data has to be received before the assembly. In our implementation, not only data necessary for a certain step, but also all the pending data which will be needed later, is received, in order to reduce the latencies caused by the MPI internal default decisions. When processing of its task or a part of the task is finished, the latter being the case, all the processors are supposed to exchange data with others having a part of the same task. Finally, only the first processor, from the list of processors in charge of the task sends the result to those not being in charge of the same task, which will, however, need this data later. The pattern is repeated until the root node is reached, when, due to its size, often all the available computing resources become involved.

We show the speedup results for the described, message passing (MPI) based, parallel scenario (Figure 2, left). The satisfactory speedup is observed for different polynomial degrees of the basis functions in finite element approximation, where higher polynomial degrees correspond to bigger problem sizes. For the reason of completeness, we show also the results for shared memory based (OpenMP) parallelisation (Figure 2, right). Results for a larger number of distributed memory computational resources are part of imminent testing. In the case of still achieving the expected speedup, according to the tendency observed for up to 7 processes, engagement of a larger number of them would result even for $p \geq 6$ in the desired rate of at least several updates per second for the calculated bone stresses in our interactive environment while, e.g., magnitude of the applied forces or the position of an implant are being modified.

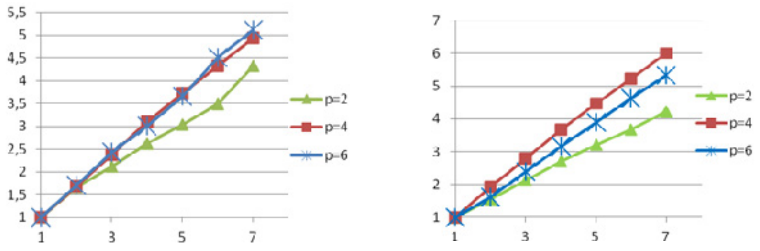


Fig. 2. Speedup results for up to 7 processors (Intel Xeon, 3.33 GHz). Left: distributed memory – the strategy described in the paper; right: shared memory parallelisation.

Conclusions

We have presented a load balancing strategy estimated as optimal for our interactive distributed environment, which exploits excellently the available processing units throughout the execution of the simulation program. The speedup results are very promising, however, testing it for a larger number of processes will show if the desired update rate for the stresses of the bone under the load is going to be achieved, even in the case of intensive user interaction. In future work, we will consider reducing communication times by taking into account also the data locality, trying to find a balance between its benefits and inevitable computational costs.

References

1. Mundani, R.-P., Düster, A., Knežević, J., Niggl, A., Rank, E.: Dynamic Load Balancing Strategies for Hierarchical *p*-FEM Solvers. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 305–312. Springer, Heidelberg (2009)
2. Knežević, J., Mundani, R.-P., Rank, E.: Interactive Computing–Virtual Planning of Hip Joint Surgeries with Real-Time Structure Simulations. International Journal of Modeling and Optimization 1(4), 308–313 (2011)

3. Mundani, R.-P., Bungartz, H.-J., Rank, E., Niggl, A., Romberg, R.: Extending the p-version of finite elements by an octree-based hierarchy. In: Domain Decomposition Methods in Science and Engineering XVI. LNCSE, vol. 55, pp. 699–706. Springer (2007)
4. Kwok, Y.-K., Ahmad, I.: Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Computing Surveys 31(4) (1999)
5. Izakian, H., Abraham, A., Snášel, V.: Comparison of Heuristics for Scheduling Independent Tasks on Heterogeneous Distributed Environments. In: International Joint Conference on Computational Sciences and Optimization, April 24-26, vol. 1, pp. 8–12 (2009)

A Shared Memory Parallel Implementation of the IRKA Algorithm for $\mathcal{H}_2$ Model Order Reduction

(Extended Abstract)

Martin Köhler and Jens Saak

Computational Methods in Systems and Control Theory
Max Planck Institute for Dynamics of Complex Technical Systems, Sandtor-Str. 1,
39106 Magdeburg, Germany
{koehlerm,saak}@mpi-magdeburg.mpg.de

Dealing with large scale dynamical systems is important in many industrial applications. In design and optimization, it is often impossible to work with the original large scale system due to the necessary time for simulation. In order to make this process economically acceptable one has to replace these large scale models by smaller ones which preserve the essential properties and dynamics of the original one. After the computation of a reduced order model a fast simulation is possible. The reduced order model can be obtained by different techniques minimizing the reduction error with respect to different system norms. One of the most common techniques in this application area is balanced truncation [8] which approximates with respect to the $\mathcal{H}_\infty$-norm. A parallel implementation of this is available in PLiCMR [2]. In this contribution we focus on the parallel implementation of the IRKA algorithm employing the $\mathcal{H}_2$-norm [1,11] for measuring the error.

1 Mathematical Background

We consider a single-input single-output (SISO) linear time invariant (LTI) system:

$$\Sigma : \begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t) \end{cases}, \tag{1}$$

with $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times 1}$ and $C \in \mathbb{R}^{1 \times n}$. The transfer function of the LTI system

$$H(s) = C\left(sI - A\right)^{-1} B, \tag{2}$$

is the frequency domain representation of the input to output map corresponding to (1). Employing (2) we can define the $\mathcal{H}_2$-norm of the system, as

$$||H||^2_{\mathcal{H}_2} = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \operatorname{tr}\left(H(i\omega)^H H(i\omega)\right) \, \mathrm{d}\omega, \tag{3}$$

where $\operatorname{tr}(\cdot)$ denotes the trace of a matrix. There exist other equivalent representations of the $\mathcal{H}_2$-norm for example using the controllability or observability gramians [1,5] that are also the key ingredient in the balanced truncation method.

The main idea behind the IRKA algorithm [5] is the computation of a reduced order model Σ_r for Σ

$$\Sigma_r : \begin{cases} \dot{x}_r(t) = A_r x_r(t) + B_r u(t) \\ y_r(t) = C_r x_r(t), \end{cases} \tag{4}$$

with $A_r = W^T A V \in \mathbb{R}^{r\times r}$, $B_r = W^T B \in \mathbb{R}^{r\times 1}$ and $C_r = CV \in \mathbb{R}^{1\times r}$, $r \ll n$, by interpolating the corresponding transfer function (2). The matrices V and W are left and right projectors in a Petrov-Galerkin framework. Gugercin, Antoulas and Beattie [5] proposed a set of optimality conditions for the interpolation points. These conditions are equivalent to older results like the Wilson conditions [11] or the Bernstein-Hyland conditions [4].

The IRKA algorithm iteratively computes an optimal set of interpolation points. The interpolation is performed by projecting the original system Σ down to the reduced order model Σ_r using the oblique projector VW^T. The matrices V and W have to be computed such that

$$\operatorname{span}\{V\} = \operatorname{span}\left\{(\mu_1 I - A)^{-1} B, \ldots, (\mu_r I - A)^{-1} B\right\}, \tag{5}$$

$$\operatorname{span}\{W\} = \operatorname{span}\left\{(\mu_1 I - A)^{-H} C^H, \ldots, (\mu_r I - A)^{-H} C^H\right\}, \tag{6}$$

where μ_i are the interpolation points and $W^T V = I$, i.e., the columns of W and V are biorthonormal. The interpolation points μ_i are determined as the mirror images of the poles of the reduced order model from the previous iteration step. The overall algorithm looks like the following:

```
Input: $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times 1}$, $C \in \mathbb{R}^{1\times n}$ and initial interpolation points
    $\mu = \{\mu_i\}_{i=1}^r$
Output: $A_r \in \mathbb{R}^{r\times r}$, $B_r \in \mathbb{R}^{r\times 1}$, $C_r \in \mathbb{R}^{1\times r}$ satisfying the optimality conditions
    from [5].
1: while not converged do
2:    Compute $V$ and $W$ according to (5) and (6)
3:    Find biorthonormal bases of $V$ and $W$
4:    $A_r := W^T A V$
5:    Compute the Eigenvalues $\lambda := \{\lambda_i\}_{i=1}^r$ of $A_r$
6:    $\mu := -\lambda$
7: end while
8: $A_r := W^T A V$, $B_r := W^T B$ and $C_r := CV$
```

The stopping criterion can be chosen in different ways. Cheap variants are a maximum iteration number or the change in the interpolation points. Expensive variants are the evaluation of the $\mathcal{H}_2$-error or further analysis of the computed subspaces.

A generalization for multiple-input multiple-output (MIMO) systems was given by Kubalinska in [7]. In the case of generalized state space systems, i.e., with an additional invertible E matrix in front of the $\dot{x}(t)$, the changes in the algorithm are small.

2 Analysis of the Algorithm

An efficient implementation requires a stepwise analysis of the algorithm with respect to the capabilities of modern multicore cpus. Due to the overall algorithm, presented in the previous section, being sequential, we need to check the single steps of the algorithm for possible acceleration by shared memory parallelization. The main focus lies on the computation of the subspaces V and W in Step 2. We easily compute them using

$$V = \left[(\mu_1 I - A)^{-1} B, \dots, (\mu_r I - A)^{-1} B \right], \tag{7}$$

$$W = \left[(\mu_1 I - A)^{-H} C^H, \dots, (\mu_r I - A)^{-H} C^H \right]. \tag{8}$$

The naive parallelization could be done using an OpenMP *for* construct. However, the properties of the interpolation parameter set allow us to safe up to half of the linear system solves. This prevents the direct use of the OpenMP *for* construct. We develop different ideas using OpenMP and PThreads for computing the columns of V and W. We will discuss and compare: a thread pool based implementation, OpenMP *for* with preprocessing and OpenMP *parallel sections.*

The special structure of the linear systems

$$(\mu_i I - A)X = Y \tag{9}$$

can be exploited to reduced the memory usage drastically and help to develop a memory efficient shared memory LU-decomposition [6]. The efficient solution of such systems plays an important role in many other algorithms in model order reduction, e.g., the TSIA algorithm [3], an alternative to IRKA for $\mathcal{H}_2$ model order reduction, the ADI algorithm [10,9], for computing the system Gramians in balanced truncation, or the solution of a class of Sylvester equations [3].

The other steps of the algorithm only allow trivial parallelism already implemented in common software packages like threaded BLAS libraries.

3 Further Parallel Components

IRKA depends on several further algorithms that strongly contribute to the efficient implementation of the method. We will shortly discuss the evaluation of the $\mathcal{H}_2$-error and the sampling of a transfer function. The $\mathcal{H}_2$-error can be determined by computing the $\mathcal{H}_2$-norm of the error system Σ_e:

$$\Sigma_e : \begin{cases} \dot{x}_e(t) = \begin{bmatrix} A & 0 \\ 0 & A_r \end{bmatrix} x_e(t) + \begin{bmatrix} B \\ B_r \end{bmatrix} u_e(t) \\ y_e(t) = \begin{bmatrix} C & -C_r \end{bmatrix} x_e(t) \end{cases} \tag{10}$$

The underlying large scale Lyapunov equation is solved using the ADI algorithm [10,9], which, as mentioned above, can be accelerated using similar ideas as in the IRKA algorithm.

The sampling of the transfer function is the evaluation of

$$H(s_i) = C(s_i I - A)^{-1} B, \tag{11}$$

or respectively

$$||H(s_i)||_2 = ||C(s_i I - A)^{-1} B||_2, \tag{12}$$

for a given set of sampling points s_i. If the inner linear system is solved directly this is very expensive regarding the memory usage. In some cases it is not possible to use as many threads as available cpu cores, because the computer does not have enough memory. Normally this problem is solved by swapping parts of the data to slow memory. We present an automatic thread number adjustment for the parallel solution of (9) to prevent unnecessary swapping.

References

1. Antoulas, A.C.: Approximation of Large-Scale Dynamical Systems. SIAM Publications, Philadelphia (2005)
2. Benner, P., Quintana-Ortí, E., Quintana-Ortí, G.: Balanced truncation model reduction of large-scale dense systems on parallel computers. Math. Comput. Model. Dyn. Syst. 6(4), 383–405 (2000)
3. Benner, P., Köhler, M., Saak, J.: Sparse-Dense Sylvester equations in $\mathcal{H}_2$-model order reduction. Tech. Rep. MPIMD/11-11, Max Planck Institute Magdeburg Preprints, submitted to Journal of Computational and Applied Mathematics (December 2011)
4. Bernstein, D., Hyland, D.: Numerical solution of the optimal model reduction equations. In: Proc. AIAA Dynamics Specialists Conf. (1984)
5. Gugercin, S., Antoulas, A.C., Beattie, C.: $\mathcal{H}_2$ model reduction for large-scale dynamical systems. SIAM J. Matrix Anal. Appl. 30(2), 609–638 (2008)
6. Köhler, M., Saak, J.: Efficiency improving implementation techniques for large scale matrix equation solvers. Chemnitz Scientific Computing Prep. CSC Preprint 09-10, TU Chemnitz (2009)
7. Kubalinska, D.: Optimal interpolation-based model reduction. Ph.D. thesis, Universität Bremen (2008)
8. Moore, B.C.: Principal component analysis in linear systems: controllability, observability, and model reduction. IEEE Trans. Automat. Control AC 26(1), 17–32 (1981)
9. Penzl, T.: Numerische Lösung großer Lyapunov-Gleichungen. Logos–Verlag, Berlin, Germany (1998); Dissertation, Fakultät für Mathematik, TU Chemnitz (1998)
10. Saak, J.: Efficient Numerical Solution of Large Scale Algebraic Matrix Equations in PDE Control and Model Order Reduction. Ph.D. thesis, TU Chemnitz (July 2009), available from http://nbn-resolving.de/urn:nbn:de:bsz:ch1-200901642
11. Wilson, D.: Optimum solution of model-reduction problem. In: Proceedings of the Institution of Electrical Engineers, vol. 117(6), pp. 1161–1165 (June 1970)

Parallel Block Preconditioning by Using the Solver of Elmer

Mika Malinen[1], Juha Ruokolainen[1], Peter Råback[1], Jonas Thies[2], and Thomas Zwinger[1]

[1] CSC – IT Center for Science Ltd.
P.O. Box 405, FI-02101 Espoo, Finland
mika.malinen@csc.fi
http://www.csc.fi

[2] Centre for Interdisciplinary Mathematics
Department of Mathematics
Uppsala University
Sweden

Abstract. We describe our recent attempts to produce effective parallel block preconditioners by employing the open source finite element package Elmer. Two example problems corresponding to the computational simulation of land ice flow and the high-fidelity modelling of small acoustical devises are considered. We present features which make these problems challenging from the viewpoint of the preconditioner design and also demonstrate how the requirement of achieving scalability can even lead to rethinking the overall formulation of the problem. The utility of the preconditioners implemented has been explored experimentally.

Keywords: preconditioning, generalized Stokes equation, variable viscosity, parallel linear solver, Elmer software.

1 Introduction

An option to simplify the Navier–Stokes flow equations by neglecting some of the inertia-force terms arises naturally in many applications. In this paper, we focus on the development of modern computational solution techniques for two non-standard flow models which arise from such simplification. The first problem considered originates from the finite element modelling of the flow of an ice sheet. The second problem, which we also discretize by using finite elements, relates to the simulation of the propagation of sound waves in small acoustical devises. A common platform for implementing these methods has been the open source software Elmer [1].

Both the models considered easily lead to solving large linear systems, the treatment of which necessitates the utilization of parallel computation. Although the two cases discussed may seem quite different at the first glance, the key concepts which we have adapted in order to solve the corresponding linear algebra problems are in common and relate to the idea of accelerating the convergence of Krylov solvers by using the block preconditioning (cf., for example, the monographs [2] and [3]).

We note that obtaining ideal linear solver performance via the block preconditioning generally relies on attaining the following two conditions simultaneously. First, the preconditioner should be such that the iteration counts needed for the convergence of the preconditioned Krylov method do not depend on the size of the discrete problem. This quality is essential in that we may then seek for better resolution by mesh refinement without ruining the ability of the preconditioner to produce quickly converging solution iterates. The second performance requirement relates to the actual way how the preconditioner can be applied. Ideally, optimal complexity solvers for performing linear solves associated with the diagonal blocks of the preconditioner should be available. This has naturally guided the preconditioner design, so that designs that enable the reuse of existing solvers for standard models have been sought. We shall widen this view here by demonstrating that, in some cases, it may be as beneficial even to reformulate the original problem, so that splittings leading to easily solvable subproblems become a reality.

Our first example case which corresponds to the computational solution of land ice flow described by the full Stokes equation has gained significant attention recently. Although highly efficient solvers for the standard Stokes system have already been devised [3], extending these solvers to handle the systems arising from the glaciology may not be straightforward. Additional questions arise as the viscosity of ice depends on the flow and can thus be highly place-dependent. In addition, it is natural in this connection to employ the stress-divergence formulation due to the easiness of imposing traction boundary conditions and, hence, utilizing the standard splittings which enable the componentwise solution of the velocity subproblem becomes less natural. The third question we want to rise here is related to a characteristic feature that ice flow problems are posed on thin domains. Finite elements with high aspect ratios are hence difficult to avoid and maintaining the stability of the associated mixed finite element approximation may become problematic. Weakening the stability of the approximation method may have an adverse effect on the efficiency of the preconditioner. We shall consider all these issues in more detail in our presentation.

The second example which describes the use of a coupled Stokes-like model in acoustics simulations has also received increasing attention recently. Nevertheless, the development of practical solution methods that enable the effective use of parallel computation has not been described yet, and we shall overview our recent developments to produce such solvers. In this case, the implementation of theoretically optimal preconditioners based on the Schur complement reduction via the elimination of the velocities is complicated by the fact that the associated Schur complement problem is still of complicated nature. Therefore we shall consider alternate formulations of the problem that avoid handling the Schur complement in this form.

References

1. Elmer finite element software homepage, http://www.csc.fi/elmer
2. Benzi, M., Golub, G.H., Liesen, J.: Numerical solution of saddle point problems. Acta Numerica 14, 1–137 (2005)
3. Elman, H.C., Silvester, D.J., Wathen, A.J.: Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics. Oxford University Press (2005)

Distributed High-Performance Parallel Mesh Generation with ViennaMesh

Jorge Rodríguez[1], Josef Weinbub[1], Dieter Pahr[2], Karl Rupp[1,3], and Siegfried Selberherr[1]

[1] Institute for Microelectronics, TU Wien
[2] Institute for Lightweight Design and Structural Biomechanics, TU Wien
[3] Institute for Analysis and Scientific Computing, TU Wien
Vienna, Austria

Abstract. The ever-growing demand for higher accuracy in scientific simulations based on the discretization of equations given on physical domains is typically coupled with an increase in the number of mesh elements. Conventional mesh generation tools struggle to keep up with the increased workload, as they do not scale with the availability of, for example, multi-core CPUs. We present a parallel mesh generation approach for multi-core and distributed computing environments based on our generic meshing library ViennaMesh and on the Advancing Front mesh generation algorithm. Our approach is discussed in detail and performance results are shown.

1 Introduction

A mesh as partitioning of a physical domain is required to model continuous phenomena by means of discretized equations in the discrete domain of a computer [6]. The continually growing demand for increased accuracy and the ability to simulate on more and more complex geometries introduces the need to increase the number of the mesh elements. Today meshes with around 10^9 vertices are utilized in large-scale scientific computations [9]. Increasing the mesh size, however, intensifies the role of mesh generation, as conventional non-parallel mesh generation tools simply take too long to compute the partitioning [4]. As a remedy to this problem, a domain decomposition technique can, for example, be utilized, where basically the initial domain is partitioned, distributed, locally (re-)meshed, and utilized [5]. Merging the mesh generation step with the simulation on distributed computing nodes significantly increases the overall efficiency, as the communication overhead is minimized. In this work we investigate a self-consistent volume mesh generation approach for multi-core CPUs and distributed computing environments based on the Message Passing Interface (MPI). We show that our concise approach achieves a considerable scaling behavior, thus we may conclude that our approach is a means to significantly accelerate the generation of large volume meshes.
This work is organized as follows: Section 2 puts the work into context. Section 3 introduces our approach, and Section 4 validates the work by depicting performance results.

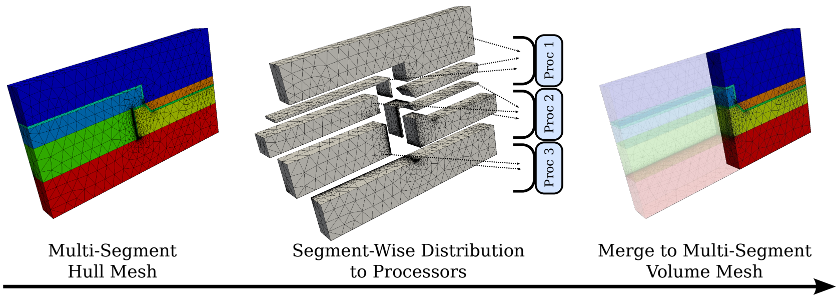

Proc 1
Proc 2
Proc 3
Multi-Segment Hull Mesh
Segment-Wise Distribution to Processors
Merge to Multi-Segment Volume Mesh

Fig. 1. The input multi-segment hull mesh (**left**) is distributed segment-wise to the individual processes (**middle**) and finally the multi-segment volume mesh is merged from the partial segment volume meshes (**right**)

2 Related Work

Different methods for parallel mesh generation are available, which offer various advantages and disadvantages [3]. For example, Delaunay based methods can be used, where basically the input domain is initially just roughly meshed and then gradually refined. The refinement process is parallelized which is quite challenging due to required synchronization steps between the individual point insertions. Another example would be Advancing Front based methods which start the volume meshing from an initial surface and gradually attach new elements to this surface. Hence, it can informally be seen as growing the volume mesh from an initial surface towards the interior. Different parallelization approaches are used, for instance, so-called partially coupled methods, where parallelizable regions in the various mesh sub-domains are identified prior to the mesh generation process. A self-consistent parallel volume meshing approach based on a shared-memory model has already been investigated previously [7]. This approach utilizes the Advancing Front technique for partial volume meshing steps, but obviously can not scale beyond a multi-core CPU.

3 Our Approach

Our approach is based on the Advancing Front meshing technique, in which the algorithm preserves the input hull mesh during the volume meshing process. Therefore, the communication overhead is minimized, as interface changes do not have to be communicated through the parallelized meshing environment. We utilize the ViennaMesh library which offers a unified interface to various mesh related tools [8]. We use the generic interface of ViennaMesh to utilize the Netgen volume mesh generation tool [1]. The input mesh is expected to be partitioned, which in our case is a reasonable assumption due to the availability of CAD tools, for example, the Synopsys Structure Editor can be utilized in the field of semiconductor device simulation [2]. We refer to each partition as a segment of the mesh, thus we call the whole mesh a multi-segment mesh. Furthermore, the individual segments of the input multi-segment mesh are hull meshes, meaning that each segment contains the surface of the sub-domain which has to be meshed. Figure 1 depicts the schematic principle of our approach. The individual hull segments are transmitted to the processes, where they are meshed. The root process is used for driving the overall parallelized mesh generation, whereas the other available processes are solely used for the partial volume meshing tasks. The partial volume mesh results are then sent back to the root node, where they are merged in a final step to the resulting multi-segment volume mesh.

4 Performance

In the following we present the performance of our approach. Our test environment consists of three workstations, namely two AMD Phenom II X4 965 with 8 GB of memory, and one INTEL i7 960 with 12 GB of memory, connected by a gigabit Ethernet network. We investigate two different types of meshes. First, two artificial test hull meshes containing 96 segments with $\sim$150k and $\sim$590k vertices, respectively (Figure 2). The number of vertices per segment is constant, allowing to investigate the optimal case, where each segment represents a constant workload for a process. Second, a hull mesh from the field of semiconductor device simulation is investigated, containing $\sim$110k vertices, 8 segments, and a varying number of vertices per segment (Figure 3). This mesh is used to outline the decrease in efficiency for small numbers of segments. The results depict that the meshing step, which includes the volume mesh generation on the nodes and the related MPI communications, scales reasonably well for meshes with approximately a ten times larger number of segments than the number of cores. Figure 2 depicts an efficiency of about 80% for 10 cores and different mesh

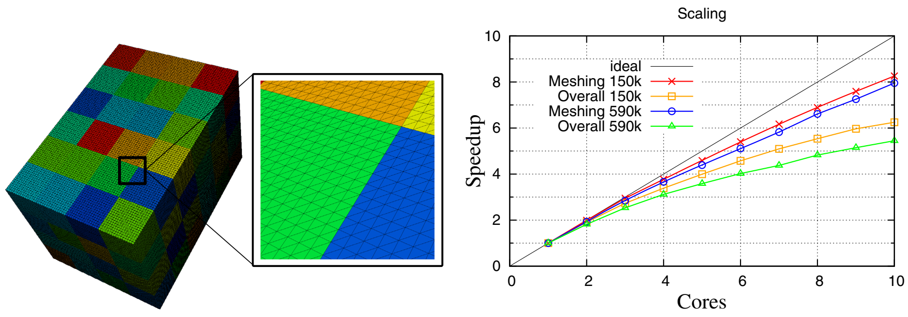


Fig. 2. Left: An artificial test mesh is analyzed in two different ways. The mesh offers 96 segments, 110k, and 590k vertices, and an equal number of vertices per segment. The colors indicate different segments. **Right:** The meshing step offers reasonable scalability for both mesh sizes (80% efficiency for 10 cores). However, the speedup decreases due to the overhead of mesh merging.

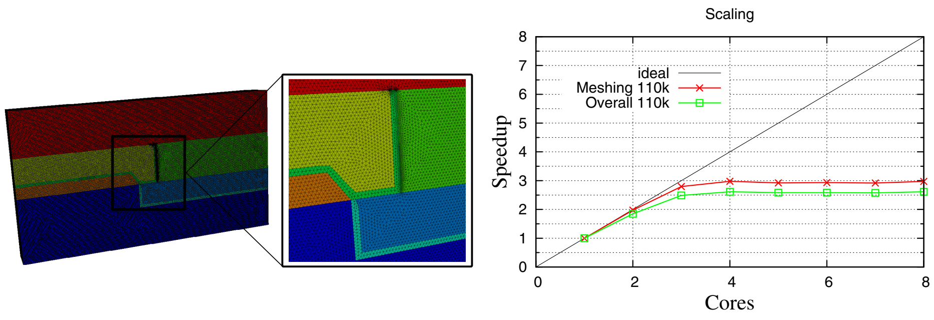


Fig. 3. Left: A mesh from the field of semiconductor device simulation is analyzed. The mesh offers 8 segments, 110k vertices, and a varying number of vertices per segment. The colors indicate different segments. **Right:** Excellent scaling can be achieved for up to 3 cores. However, due to the small number of segments and differently sized segments the scaling saturates at a speedup of 3.

sizes. However, the efficiency is reduced for larger core numbers, as the final step of merging the partial mesh results on the root node becomes large relative to the overall execution time. For example, for the 150k mesh and 2 cores, 7.7% of the overall execution time is used for the final mesh merging, where with 10 cores it is already 25%. Figure 3 outlines that our approach achieves a considerable speedup of 3 for meshes offering a small number of 8 segments. However, the scaling saturates for 4 cores and a speedup of 3, due to the small number of segments relative to the number of cores and due to the varying number of vertices in the different segments.

5 Conclusion

Our approach offers good scalability for meshes with approximately ten times larger number of segments than the number of cores. Even for meshes with approximately three times the number of segments than the number of cores, we achieve a considerable speedup. Therefore we conclude, that our presented self-consistent parallel mesh generation approach is indeed a meaningful way to significantly accelerate the volume mesh generation. However, a flexible mesh partitioning approach is required to enable an improved speedup for larger distributed environments. The mesh merging step in the root process has to be further improved, to achieve higher efficiency for large-scale meshes.

Acknowledgments. This work has been supported by the Google Summer of Code 2011 and by the European Research Council through the grant #247056 MOSILSPIN. Karl Rupp gratefully acknowledges support by the Graduate School PDETech at the TU Wien.

References

1. Netgen, http://sourceforge.net/projects/netgen-mesher/
2. Synopsys, http://www.synopsys.com/
3. Chrisochoides, N., et al.: A Survey of Parallel Mesh Generation Methods. Brown University (2005)
4. Ito, Y., et al.: Parallel Unstructured Mesh Generation by an Advancing Front Method. Mathematics and Computers in Simulation 75(5-6) (2007)
5. Magoules, F.: Mesh Partitioning Techniques and Domain Decomposition Methods. Saxe-Coburg Publications (2008)
6. Shewchuk, J.R.: Unstructured Mesh Generation. In: Combinatorial Scientific Computing. pp. 259–297. CRC Press (2012)
7. Stimpfl, F., et al.: High Performance Parallel Mesh Generation and Adaptation. In: Proc. PARA (2008)
8. Weinbub, J., et al.: High-Quality Mesh Generation Based on Orthogonal Software Modules. In: Proc. SISPAD (2011)
9. Zhou, M., et al.: Tools to Support Mesh Adaptation on Massively Parallel Computers. Engineering with Computers (2011)

Computational Chemistry Studies of LIGNOLs

Thomas Sandberg[1,2], Patrik Eklund[3], and Matti Hotokka[2]

[1] Centre of Excellence for Functional Materials
[2] Physical Chemistry, Åbo Akademi University, Porthansg. 3, 20500 Turku, Finland
[3] Organic Chemistry, Åbo Akademi University, Biskopsg. 8, 20500 Turku, Finland

1 Introduction

The forest industries are developing new innovative products in addition to the traditional bulk products. A promising raw material for added-value products consists of lignans that are extracted in chemical pulping from residual knots. The anticarcinogenic and antioxidative lignan hydroxymatairesinol (HMR) is found in large amounts in the knots of Norway spruce (Picea abies). It has been used in the synthesis of TADDOL-like α-conidendrin-based chiral 1,4-diols (LIGNOLs) [1] with the same functionality as TADDOLs [5] or BINOLs [2], which are often used as ligands for transition metal catalysed asymmetric synthesis. They have hindered structures containing two adjacent stereocenters, resulting in a fixed angle between the metal-complexing hydroxyl groups.

The structures of the LIGNOLs included in this study have been quantum chemically optimized [3] by a multi-level deterministic method, and studied by molecular dynamics simulations [4] to explore the conformational changes of the LIGNOLs in aqueous solution. As solvents play a role in the structures of many molecules, solvation effects has been included using the implicit solvation model COSMO (the COnductorlike Screening MOdel).

2 Methods

In this study the following chiral 1,4-diols (LIGNOLs) have been investigated: 1,1-diphenyl (**2Ph**), two diastereomers of 1,1,4-triphenyl (**3PhR**, **3PhS**), 1,1,4,4-tetraphenyl (**4Ph**) and 1,1,4,4-tetramethyl (**4Met**) 1,4-diol. In order to find the minimum energy structure of the LIGNOLs, initial torsional analyses were performed for each of the chiral 1,4-diols on the three single bonds ($\alpha - \gamma$) denoted in Figure 1 using a step size of 60 degrees at molecular mechanics (MM) level using the Tripos Force Field as implemented in SYBYL 8.0.

The energetically most favourable structures were optimized with the program GAMESS version 22 Feb 2006 by using Hartree-Fock (HF) theory with the basis set 6-31G*. Altogether this resulted in 6 conformers for **3PhR**, 10 for **3PhS** as well as for **4Met**, 13 for **4Ph** and 18 for the quite flexible **2Ph** structure. All of the conformers were then reoptimized for vibrational analysis using the TURBOMOLE program package version 6.1 and density functional theory (DFT) with the B3LYP hybrid exchange-correlation functional in combination with the multipole accelerated resolution of identity (MARI-J) approximation and the TZVP basis set for all atoms, as implemented in TURBOMOLE. Frequencies were scaled with a factor of 0.9614, when calculating the entropy contributions at 25 °C. To investigate the solvation effects the molecules were placed in water-like continuum solvent (ε_r = 78.39) using COSMO in TURBOMOLE. The minimum energy structure for each LIGNOL is shown in Figure 2. The code for the conformations is adopted from [3].

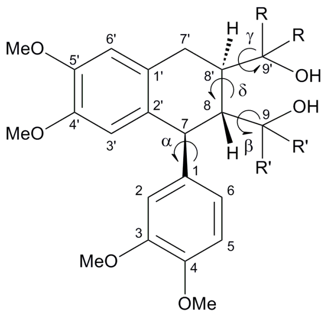


Fig. 1. The four most relevant torsional angles

The molecular dynamics (MD) simulations were performed using GROMACS version 4.5.3 software. Water was described using the TIP4P model, and the LIGNOLs were modeled with the OPLS-AA force field implemented in GROMACS. The topologies of the LIGNOLs were constructed by hand, and they comprised 415 (**2Ph**), 474 (**3Ph**), 533 (**4Ph**) and 369 (**4Met**) internal coordinates, respectively. In order to get reasonable atomic charges to help for choosing suitable atom types to the topologies, electrostatic potential fit (ESP) charges were studied with GAMESS at HF/6-31G* level. The three quantum chemically most stable conformers of each of the LIGNOLs (three per stereoisomer for **3Ph**) were chosen as starting structures for the molecular dynamics study. Each conformation was placed at the center of a cubic box with the dimension between 5.2–5.6 nm (volume = 144–174 nm^3) and solvated by 4802–5795 water molecules. Each system was first energy minimized $<$ 2000 kJ mol^{-1} nm^{-1} using steepest descent for 3–121 steps. Then the system was shaken at 398 K for 50 ps, and finally the production simulation was run for 10 ns with the temperature maintained at 298 K using the Berendsen thermostat. The pressure was maintained at 1 atm using the Berendsen barostat. A 1 fs time step was used in all simulations. A cutoff of 0.9 nm was applied to short-range nonbonded interactions, and for long-range electrostatic interactions the particle mesh Ewald (PME) method was used with grid spacing of 0.12 nm and fourth-order interpolation. In all simulations system snapshots were collected every 500 steps, i.e. 0.5 ps, for subsequent analysis. The four most relevant torsional angles in the LIGNOLs (Figure 1) were properly analyzed during the simulations.

In order to understand the hydration effect more properly the g_hbond analysing program implemented in GROMACS was used to study the number of hydrogen bonds for the oxygen atoms O9 and O9', and totally for each LIGNOL conformer, as well as the average lifetime of the uninterrupted hydrogen bonds. Tetramethyl 1,4-diol was found to be more likely to form hydrogen bonds to TIP4P, and tetraphenyl less, mainly due to the small tendency of O9 to form hydrogen bonds to TIP4P. A correllation could be seen to the number of hydrogen bonds as the lifetimes were longer for tetramethyl 1,4-diol and shorter

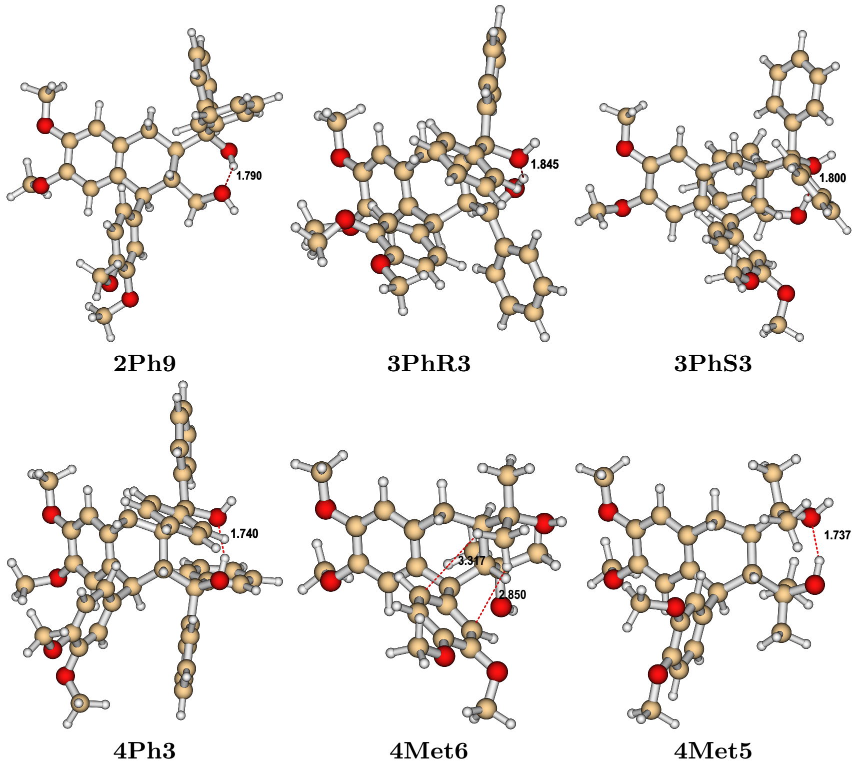


Fig. 2. The minimum energy structure for each LIGNOL

for tetraphenyl. A shorter lifetime for a large average number of hydrogen bonds implies that they are quite weak, meaning that the hydrogen bonds from O9' in **2Ph1** and **2Ph2** are strong, as they had a remarkably longer life times with the same average number of hydrogen bonds. This again could be very important for the application of these LIGNOLs as transition metal catalysts, as the bonding to a metal would be similar to the hydrogen bonding to TIP4P water. Diphenyl 1,4-diol is the only LIGNOL in this study with phenyls at C9' and not at C9, so the reason for this phenomenon is probably the electronic effects of the phenyl rings at C9'.

3 Aim of the Study

One object of this study was to observe how the angle between the hydroxyl groups in these LIGNOLs behaves in the optimized structures. The angle δ was then picked out and compared to the crystallographic data of TADDOLs. According to that the angle δ should be close to 270°. Another property that makes the structures more stable seemed to be the boat conformation of the aliphatic six-membered ring, which is perfectly formed if δ is 240°. The value of

δ should, consequently, be between 240° and 270°. If the OH groups furthermore points to the same direction, an intramolecular hydrogen bond forms between them, and the bridging hydrogen atom falls at the same place as a chelate-bonded metal ion would be situated. These factors are fulfilled in the triphenyl conformers: **3PhS3**, **3PhR3** and **3PhR7**. For the energetically more favourable conformers, a $\pi - \pi$ interaction was formed between the phenyl ring at C7 and one of the phenyl rings at C9'.

4 Conclusions

The diphenyl 1,4-diol is hardly hindered at all resulting in a wide range of almost equally stable conformers. The stability of the other phenylated 1,4-diols is mainly determined by the ability to form $\pi - \pi$ interactions between phenyl rings and the possibility for the aliphatic six-membered ring to be in boat conformation, i.e. the torsional angle δ to be between 240° and 270°. The most stable triphenyl 1,4-diols according to the DFT calculations: **3PhS3**, **3PhR3** and **3PhR7** are also the ones that could work as catalysts. The most stable tetraphenyl 1,4-diols according to the DFT calculations: **4Ph3**–**4Ph4** and **4Ph7**–**4Ph8** also seems to be possible catalysts. Those conformers of the tetramethyl 1,4-diol that have the OH groups pointing to the same direction are almost 12 kJ/mol less stable than the most favourable one, when the entropy contributions are taken into account. Moreover, they do not have the aliphatic six-membered ring in the prefered boat conformation.

In MD simulations on the LIGNOLs, the conformations preferred were the energetically most favourable ones according to quantum chemical DFT calculations in gas phase, almost irrespective of the dipole moment. The four most relevant torsional angles $\alpha - \delta$ varied quite much in accordance with their symmetry. The torsional angle δ defined in Figure 1 was generally more preferred at the stabilizing value 255° than what was seen in the gas phase optimizations. No strong correlation patterns were found, but in the last simulation of **2Ph9**, α and δ changed simultaneously, while β either initialized a confomational change or lagged behind. In the hydration studies **2Ph1** and **2Ph2** were found to have strong hydrogen bonds from O9', which could be very important for the application of these LIGNOLs as metal-binding agents.

Acknowledgements. Computing resources provided by CSC are kindly acknowledged.

References

1. Brusentsev, Y., Sandberg, T., Hotokka, M., Sjöholm, R., Eklund, P.: Synthesis and structural analysis of sterically hindered chiral 1,4-diol ligands derived from the lignan hydroxymatairesinol. Accepted in Tetrahedron Lett. (2012)
2. Brussee, J., Jansen, A.C.A.: A highly stereoselective synthesis of s(-)-[1,1'-binaphthalene]-2,2'-diol. Tetrahedron Lett. 24, 3261–3262 (1983)

3. Sandberg, T., Brusentsev, Y., Eklund, P., Hotokka, M.: Structural analysis of sterically hindered 1,4-diols from the naturally occurring lignan hydroxymatairesinol. a quantum chemical study. Int. J. Quantum Chem. 111, 4309–4317 (2011)
4. Sandberg, T., Eklund, P., Hotokka, M.: Conformational solvation studies of lignols with molecular dynamics and conductor-like screening model. Int. J. Mol. Sci. 13, 9845–9863 (2012)
5. Seebach, D., Beck, A.K., Schiess, M., Widler, L., Wonnacott, A.: Some recent advances in the use of titanium reagents for organic synthesis. Pure Appl. Chem. 55, 1807–1822 (1983)

Investigation of Load Balancing Scalability in Space Plasma Simulations

Ata Turk[1], Gunduz V. Demirci[1], Cevdet Aykanat[1], Sebastian von Alfthan[2], and Ilja Honkonen[2,3]

[1] Bilkent University, Computer Engineering Department, 06800 Ankara, Turkey
{atat,gunduz,aykanat}@cs.bilkent.edu.tr
[2] Finnish Meteorological Institute, PO Box 503, FI-00101, Helsinki, Finland
{Sebastian.von.Alfthan,ilja.honkonen}@fmi.fi
[3] Department of Physics, University of Helsinki, PO Box 64, 00014, Helsinki, Finland

Abstract. In this study we report the load-balancing performance issues that are observed during the petascaling of a space plasma simulation code developed at the Finnish Meteorological Institute (FMI). The code models the communication pattern as a hypergraph, and partitions the computational grid using the parallel hypergraph partitioning scheme (PHG) of the Zoltan partitioning framework. The result of partitioning determines the distribution of grid cells to processors. It is observed that the initial partitioning and data distribution phases take a substantial percentage of the overall computation time. Alternative (graph-partitioning-based) schemes that provide better balance are investigated. Comparisons in terms of effect on running time and load-balancing quality are presented. Test results on Juelich BlueGene/P cluster are reported.

Keywords: partitioning, petascaling, space plasma simulation.

1 Introduction

The dynamics of near Earth space environment have gained immense importance since many mission critical global technological systems depend on spacecraft that traverse this space and even small dynamical events can cause failures on the functionalities of these spacecraft. Hence performing accurate space weather forecasts are of utmost importance. Space weather forecasting is performed by modeling the electromagnetic plasma system within the near Earth space including the ionosphere, magnetosphere, and beyond.

At the Finnish Meteorological Institute (FMI), two simulation models are being developed to tackle this issue: a magnetohydrodynamic simulation code for real-time forecasting and a hybrid Vlasov simulation code for very accurate space weather forecasting. In a hybrid Vlasov model, electrons are modeled as a fluid and ions as six-dimensional distribution functions in ordinary and velocity space, enabling the description of plasma without noise.

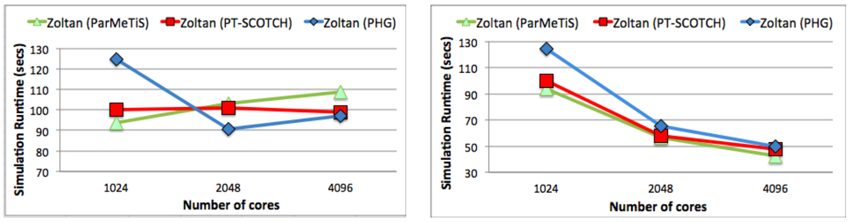
Both codes need to exhibit excellent parallel scalability to reach the required level of performance. The simulation models are designed to run on a parallel grid. In the hybrid-Vlasov code, the parallel grid contains cells in ordinary space, and each spatial grid cell contains a three-dimensional velocity distribution function, which is implemented as a simple block-structured grid. A major bottleneck for this code is the need for efficient load balancing at scale, as the target is to run it on more than 10.000 cores. For determining the distribution of spatial cells to processors, the grid uses the PHG partitioning mode of the Zoltan partitioning framework.

2 Investigations in Jugene

In this study the effects of load balancing tools in the performance of hybrid Vlasov simulation code have been analyzed in detail. The scalability of the code itself is also tested to some extent. The reported findings can be listed as follows:

– Porting of the hybrid Vlasov code to Juelich BlueGene/P (Jugene) system is performed and profiling of the performance of the code up to 10^4 cores (previous tests were performed for less than 10^3 cores due to limited resources) is achieved to reveal that it successfully scales up to 10^4 cores.
– Analysis of the load-balancing (partitioning) scheme is performed. It is observed that the time spent on preprocessing constitutes a significant portion of the overall runtime. Further analysis revealed that when the number of cores reach to 10^4, the determinant factor in simulation runtime tends to be the balancing performance instead of the overall communication cost.
– An alternative load-balancing scheme (based on graph-partitioning), which is known to have better load balancing performance, is embedded in the code. Experiments show that the alternative scheme has simulation time performances that are more scalable than PHG.

We should note here that graph partitioning models cannot exactly model the communication overheads associated with the communication patterns in the hybrid Vlasov code and the usage of the hypergraph modeling scheme is more correct theoretically. However, as a general observation we can state that, although the communication metrics optimized by graph partitioning schemes are not exact, if the problem domain is regular enough, the error made by graph partitioning method for estimating the communication overhead of a partition is more or less the same for all possible partitions in the solutions space. This property enables the graph partitioning schemes to improve its solutions over regular computational domains successfully since the error made while moving through different partitions in the solution space cancel each other. Since the subject problem domain exhibits such features, we believe that the usage of graph partitioning tools might yield good results as well and thus investigate such alternatives.



(a) Weak Scaling (b) Strong Scaling

Fig. 1. Simulation runtimes of the weak and strong scaling experiments for hybrid Vlasov code utilizing different partitioning libraries available in Zoltan

Table 1. Communication overheads and balancing performances under weak scaling experiments for various parallel partitioning libraries that are called within Zoltan

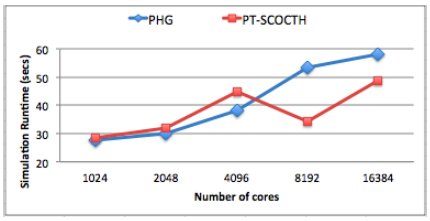
	Zoltan (PHG)			Zoltan (ParMeTiS)			Zoltan (PT-SCOTCH)		
# of cores	Total volume	Comp. imb.(%)	Comm. imb.(%)	Total volume	Comp. imb.(%)	Comm. imb.(%)	Total volume	Comp. imb.(%)	Comm. imb.(%)
1024	70.206	99,9	44,1	78.538	50,0	34,2	77.824	0,1	5,3
2048	143.538	99,9	42,9	161.318	31,3	41,0	159.744	0,1	2,6
4096	289.292	99,9	60,0	327.812	62,5	40,0	323.568	6,2	12,8

3 Experiments

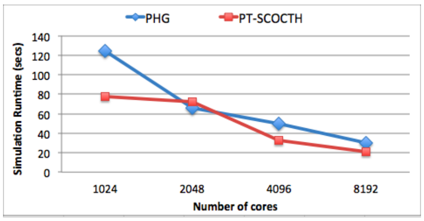
In our experiments, we first compared the performance of calling Zoltan PHG [1] with the performance of calling ParMeTiS [2] and PT-SCOTCH [3] within Zoltan in the initial load balancing step of the simulation code to see which of these three possible partitioning options available in Zoltan is best for this particular code. In these experiments we measured the weak and strong scaling performance of these three schemes. In the experiments for weak scaling, 3D grid size is arranged such that under perfect load balance, each process would have to process 16 spatial cells, and for strong scaling, total number of spatial cells is set to $16 \times 32 \times 32$. Since the memory in a Jugene node is small, we could only perform weak-scaling experiments up-to 4K cores with 16 spatial cells per core.

In Table 1, we present the total communication volume, computational imbalance, and communication imbalance values observed in the weak scaling experiments. As seen in the table, in terms of total communication volume, PHG performs the best, whereas in terms of communication and computation load balancing, PT-SCOTCH performs the best. We should note here that strong scaling experiments provided similar communication and computation balancing results but are not reported due to space constraints.

As seen in Table 1 and Fig. 1, even though PHG produces lowest overall communication overheads, the graph-based partitioning libraries produce as good as, if not better, running time results. This is probably due to PHG's poor load-balancing performance. After these observations we decided to remove the



(a) Weak scaling



(b) Strong scaling

Fig. 2. Weak scaling simulation runtimes of hybrid Vlasov code using PT-SCOTCH and Zoltan PHG in preprocessing.

overhead of calling Zoltan for running the PT-SCOTCH library, which has nice load-balancing features and rewrote the initial load balancing code such that it calls PT-SCOTCH library directly.

Table 2. Communication overheads and balancing performances under weak scaling experiments for Zoltan PHG and PT-SCOTCH

	Zoltan (PHG)			PT-SCOTCH		
# of cores	Total volume	Comp. imb.(%)	Comm. imb.(%)	Total volume	Comp. imb.(%)	Comm. imb.(%)
1024	27.726	99,6	76,9	28.672	0,9	14,4
2048	59.608	99,8	92,9	61.440	0,9	6,7
4096	123.294	99,9	73,3	126.976	1,5	6,7
8192	250.718	99,9	26.1	258.048	1,2	6,7
16384	505.008	99,9	93,3	520.192	1,4	6,7

In Fig. 2 we compare the performance results of directly calling PT-SCOTCH and Zoltan PHG in the preprocessing step of simulation. In the weak-scaling experiments reported, the number of grid cells per core is fixed to four and in the strong-scaling experiments the total number of spatial cells is set to 16×32×32. As seen in Fig. 2(a), as the number of cores reaches to 8K and beyond, PT-SCOTCH starts to perform considerably better then PHG, probably again due to its better load-balancing capability as noted in Table 2. We again note here that strong scaling experiments provided similar communication and computation balancing results but are not reported due to space constraints. Similarly, as seen in Fig. 2(b), PT-SCOTCH generally produces better results in strong-scaling experiments as well.

4 Conclusions

We showed that the hybrid Vlasov simulation code developed at FMI can scale up to 16K cores. We also showed that by replacing the initial load balancing scheme based on Zoltan parallel hypergraph partitioning tool with PT-SCOTCH parallel graph partitioning tool increases the overall communication volume but still improves the simulation runtime since PT-SCOTCH produces better balanced partitions. These results indicate that for the hybrid Vlasov code, minimizing imbalance is as important as, if not more important than, minimizing the overall communication volume.

References

1. Devine, K., Boman, E., Heaphy, R., Hendrickson, B., Vaughan, C.: Zoltan Data Management Services for Parallel Dynamic Applications. Computing in Science and Engineering 4(2), 90–97 (2002)
2. Schloegel, K., Karypis, G., Kumar, V.: Parallel static and dynamic multi-constraint graph partitioning. Concurrency and Computation: Practice and Experience 14(3), 219–240 (2002)
3. Chevalier, C., Pellegrini, F.: PT-Scotch: A tool for efficient parallel graph ordering. Parallel Computing 34(6-8), 318–331 (2008)

A Lightweight Task Graph Scheduler for Distributed High-Performance Scientific Computing

Josef Weinbub[1], Karl Rupp[1,2], and Siegfried Selberherr[1]

[1] Institute for Microelectronics, TU Wien
[2] Institute for Analysis and Scientific Computing, TU Wien, Vienna, Austria

Abstract. The continually growing demand for increased simulation complexity introduces the need for scientific software frameworks to parallelize simulation tasks. We present our approach for a task graph scheduler based on modern programming techniques. The scheduler utilizes the Message Passing Interface to distribute the tasks among distributed computing nodes. We show that our approach does not only offer a concise user-level code but also provides a high degree of scalability.

1 Introduction

The ever-growing demand of increased simulation complexity to better model physical phenomena requires, among other things, the combination of different simulation components [11]. This combination can be, for example, realized, by using the output of one tool as an input for another one. From a software point of view, this problem can be modelled as a task graph [10], which is governed by a software framework [5]. The individual simulation tools can be seen as vertices of the task graph, which are therefore executed based on the individual task dependencies. To improve the efficiency and therefore reduce the overall run-time of the framework, a parallelized approach for the task execution is required. A high-degree of flexibility is provided by a distributed approach based on the Message Passing Interface (MPI), as the execution can be spread among the nodes of a large-scale cluster environment as well as on the cores of a single workstation. In general, the distribution of parallelizable tasks among distributed [7] and shared computing [4] resources is a typical way to improve the overall run-time performance of a task graph. In this work we investigate a lightweight approach to implement a scheduler based on modern programming techniques, in particular, generic [12] and functional [8] programming in C++. By utilizing these techniques and external libraries we are able to achieve a highly concise user-level code, by simultaneously obtaining excellent scalability with regard to the execution performance.

This work is organized as follows: Section 2 introduces our approach and Section 3 validates the work by depicting performance results.

2 Our Approach

Our approach for distributing tasks on distributed computing nodes can be split into three parts. The first part is the mapping of the tasks and the corresponding dependencies on a graph datastructure; the second, the priorization based on the task dependences by utilizing a graph algorithm; and the third, the parallel and distributed execution on the computing nodes by using the Boost MPI Library [2]. Figure 1 depicts the principle of generating a task-graph and the parallel execution.

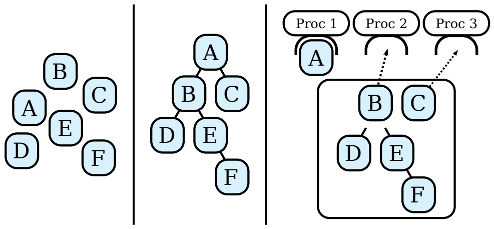


Fig. 1. Tasks are associated with vertices (**left**) and dependencies are related to edges (**middle**) in the graph. The tasks are executed on distributed processes according to their dependencies (**right**). For example, task `B` and task `C` are only executed, when task `A` is finished.

We utilize the Boost Graph Library (BGL) for the graph datastructure and the graph algorithms [1]. Each task is associated with a vertex in the graph, whereas the dependencies are mapped to edges connecting the respective vertices. Our implementation is based on the list scheduling technique, which requires a sequential list of prioritized tasks [9]. This priorization is computed by the BGL implementation of the topological sort graph algorithm [6]. According to the list scheduling approach, this prioritized list is traversed and every task is checked, whether it can be executed. This traversal is repeated until all tasks have been processed. In general, we utilize the generic and functional programming techniques. The generic programming paradigm is used to achieve a highly versatile and extendable implementation. The functional style allows to provide an intuitive user-level code and is applied by utilizing the Boost Phoenix Library (BPL) [3]. The utilization ot these programming paradigms enables to implement the following concise user-level code, which depicts the scheduling traversal of the prioritized tasks.

```
std::for_each(prioritized.begin(), prioritized.end(),
   if_(is_executable)[execute(arg1,ref(process_manager))]);
```

The set of prioritized tasks (`prioritized`) is traversed. `if_(is_executable)[..]` checks if a task is ready for execution, which is done by testing the state of the immediate predecessors. If so, `execute(..)` tries to assign the task to a process

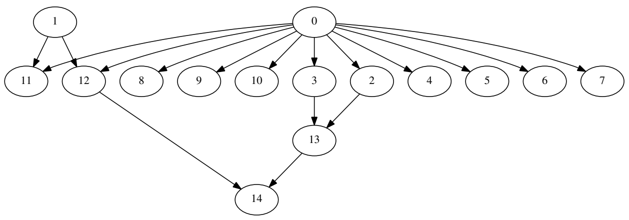
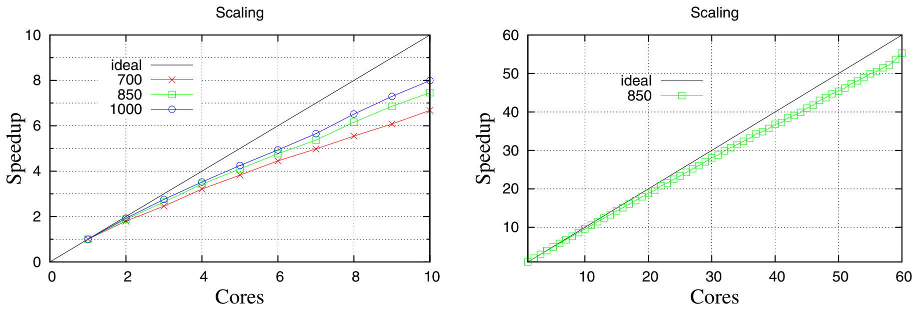


Fig. 2. An exemplary task graph is shown containing a maximum of 11 parallelizable tasks (task 2-12).



(a) Task-Graph with Dependencies **(b)** Task-Graph without Dependencies

Fig. 3. Left: The scaling for different task problem sizes is depicted based on a task graph with dependencies. The scaling efficiency for 10 cores is improved from 68% for a problem size of 700 to 80% for a problem size of 1000. **Right:** The scaling for a task problem size of 850 is shown based on a task graph without dependencies. A scaling efficiency of 91% is achieved for 60 cores.

by utilizing a process manager facility. Note that `arg1` and `ref(..)` are BPL expressions which enable access to the traversal object and the reference to an existing object, respectively.

3 Performance

In this section we present the scalability of our approach. Each task computes the dense matrix-matrix product for different problem sizes to model a computational load. Note that in this work we do not investigate the data transfer between the individual tasks, as we solely focus on the scheduling and the execution of the tasks. Our approach is evaluated based on two different test cases. First, we evaluate the speedup of a task graph with various dependencies. For this investigation we basically use the same graph layout as depicted in Figure 2. However, instead of a maximum number of 11 tasks on the second level of the graph, we use a problem offering a maximum number of 100 parallelizable tasks. Furthermore, we investigate different problem sizes with respect to the dense matrix-matrix product. The scalability is investigated for up to 10 cores. The hardware environment for this investigation consists of three workstations, two AMD Phenom II X4 965 with 8 GB of memory, and one INTEL i7 960 with 12 GB of memory, connected by a gigabit Ethernet network. Figure 3a depicts the gained performance results. A scaling efficiency of 68% for a problem size of 700 is improved to 80% for a problem size of 1000. Second, we investigate the speedup for 600 tasks without task dependencies, to investigate the optimal parallelization capabilities. This hardware environment is based on our computing cluster, where the nodes offer four six-core AMD Opteron 8435, 128 GB of system memory, and an Infiniband DDR network connection each. Figure 3b shows the speedup for this test. A scaling efficiency of around 91% for 60 cores is achieved.

4 Conclusion

Our approach based on modern programming techniques provides not only concise user-level code but also offers excellent scalability for up to 60 cores. Furthermore, the scalability improves for larger problems, which underlines the suitability of our scheduling approach for large-scale simulations.

Acknowledgments. This work has been supported by the European Research Council through the grant #247056 MOSILSPIN. Karl Rupp gratefully acknowledges support by the Graduate School PDETech at the TU Wien.

References

1. The Boost Graph Library, http://www.boost.org/libs/graph/
2. The Boost MPI Library, http://www.boost.org/libs/mpi/
3. The Boost Phoenix Library, http://www.boost.org/libs/phoenix/
4. Agrawal, K., et al.: Executing Task Graphs Using Work-Stealing. In: Proc. IPDPS (2010)
5. Carey, J.O., Carlson, B.: Framework Process Patterns (2002)
6. Cormen, T.H., et al.: Introduction to Algorithms (2009)
7. Dutot, P.F., et al.: Scheduling Parallel Task Graphs on (Almost) Homogeneous Multicluster Platforms. IEEE Trans. Parallel Distrib. Syst. 20 (2009)
8. Hughes, J.: Why Functional Programming Matters. The Comput. J. 32(2) (1989)
9. Kwok, Y.K., et al.: Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Comput. Surv. 31(4) (1999)
10. Miller, A.: The Task Graph Pattern. In: Proc. ParaPLoP (2010)
11. Quintino, T.: A Component Environment for High-Performance Scientific Computing. Ph.D. thesis, Katholieke Universiteit Leuven (2008)
12. Reis, G.D.: et al.: What is Generic Programming? In: Proc. LCSD (2005)

Author Index

Airola, Antti 280
Aittokallio, Tero 280
Alava, Mikko 43
Aochi, Hideo 107
Arbenz, Peter 361
Asif, Muhammad 43
Åström, J.A. 27
Aykanat, Cevdet 558

Bała, Piotr 115
Beck, Philip-Daniel 454
Belonosov, Mikhail A. 153
Benkner, Siegfried 328
Bethune, Iain 43
Blaheta, R. 375
Bodin, François 328
Bugajev, Andrej 387

Canova, Filippo Federici 3
Cardoso, Vitor 43
Carter, Adam 27, 43
César, Eduardo 328
Chen, Yurong 410
Chertkow, Merek A. 92
Čiegis, Raimondas 387
Civario, Gilles 328

Deadman, Edvin 171
Demirci, Gunduz V. 558
Dłubacz, Wojciech 183
Du, Yangzhou 410
Duda, Jerzy 183, 465
Dupros, Fabrice 107

Eklund, Patrik 553
Emans, Maximilian 398
Enkovaara, Jussi 63

Feng, Jiangbin 410

Gerndt, Michael 328
Goumas, Georgios 531

Haitof, Houssam 328
Hakala, Samuli 3, 63
Hapla, Václav 192
Harju, Ari 3, 141
Harris, J. Austin 92
Havu, Ville 63
He, Xin 206
Hetherington, J. 27
Higham, Nicholas J. 171
Holm, Marcus 206
Honkonen, Ilja 297, 558
Horák, David 192
Hotokka, Matti 553
Hyväluoma, Jari 434

Ioakimidis, K. 27

Jakl, O. 375
Jankowska, Malgorzata A. 473
Johnson, Chris 43
Juszczyszyn, Krzysztof 77

Kågström, Bo 250
Karakasis, Vasileios 531
Kekäläinen, Pekka 434
Knežević, Jovana 536
Köhler, Martin 541
Kolaczek, Grzegorz 77
Kostov, Clement 153
Kozikowski, Grzegorz 489
Koziris, Nectarios 531
Kubica, Bartłomiej Jacek 449, 489, 504
Kumar, Pawan 220
Kuznetsov, Sergey V. 235
Kwiatkowski, Jan 77

Lančinskas, Algirdas 422
Lehto, Olli-Pekka 309
Li, Eric 410
Liebmann, Manfred 398
Lindahl, E. 27
Linde, Mattias 319

Malinen, Mika 545
Marciniak, Andrzej 518
Mattila, Keijo 434
Meerbergen, Karl 220

Merta, Michal 192
Messer, O.E. Bronson 92
Miceli, Renato 328
Michéa, David 107
Mikkelsen, Carl Christian Kjelgaard 250
Morin, Laurent 328
Mozdzynski, G. 27
Mundani, Ralf-Peter 536
Myllykoski, Mirko 265

Nash, R.W. 27
Navarrete, Carmen 328
Nehmeier, Marco 454
Neytcheva, Maya 206
Nieminen, Risto 63
Nikas, Konstantinos 531
Nowicki, Marek 115

Okser, Sebastian 280

Pahikkala, Tapio 280
Pahr, Dieter 548
Palmroth, Minna 297
Parete-Koon, Suzanne 92
Pouderoux, Joachim 107
Puurtinen, Tuomas 434

Råback, Peter 531, 545
Ralha, Rui 171
Rank, Ernst 536
Rantalaiho, Teemu 3, 126
Reshetova, Galina V. 153
Rodríguez, Jorge 548
Roose, Dirk 220
Rossi, Tuomo 265
Ruokolainen, Juha 531, 545
Rupp, Karl 548, 563

Saak, Jens 541
Salakoski, Tapio 280
Sandberg, Thomas 553
Sandrieser, Martin 328
Schlatter, P. 27
Schuberth, Bernhard S.A. 43
Selberherr, Siegfried 548, 563
Signell, A. 27
Sikora, Anna 328
Siro, Topi 3, 141
Skalna, Iwona 465
Soloviev, Sergey A. 153
Starý, J. 375
Statford, Kevin 43
Sypniewska-Kaminska, Grazyna 473
Szyszka, Barbara 518

Tcheverda, Vladimir A. 153
Thies, Jonas 545
Timonen, Jussi 434
Toivanen, Jari 265
Toivanen, Jukka I. 434
Turan, Erhan 361, 375
Turk, Ata 558

von Alfthan, Sebastian 297, 558

Weinbub, Josef 548, 563
Weinzierl, Tobias 43
Westerholm, J. 27
Woźniak, Adam 504

Zhang, Yimin 410
Žilinskas, Julius 422
Zumbusch, Gerhard 343
Zwinger, Thomas 545