LNCS 7203

Roman Wyrzykowski
Jack Dongarra
Konrad Karczewski
Jerzy Waśniewski (Eds.)

Parallel Processing and Applied Mathematics

9th International Conference, PPAM 2011
Torun, Poland, September 2011
Revised Selected Papers, Part I

1 Part I

Springer

Roman Wyrzykowski Jack Dongarra
Konrad Karczewski Jerzy Waśniewski (Eds.)

Parallel Processing and Applied Mathematics

9th International Conference, PPAM 2011
Torun, Poland, September 11-14, 2011
Revised Selected Papers, Part I

Springer

Volume Editors

Roman Wyrzykowski
Czestochowa University of Technology, Poland
E-mail: roman@icis.pcz.pl

Jack Dongarra
University of Tennessee, Knoxville, TN, USA
E-mail: dongarra@cs.utk.edu

Konrad Karczewski
Czestochowa University of Technology, Poland
E-mail: xeno@icis.pcz.pl

Jerzy Waśniewski
Technical University, Kongens Lyngby, Denmark
E-mail: jw@imm.dtu.dk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-31463-6 e-ISBN 978-3-642-31464-3
DOI 10.1007/978-3-642-31464-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012941360

CR Subject Classification (1998): D.2, H.4, D.4, C.2.4, D.1.3, H.3, F.2

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

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

Printed on acid-free paper

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

Preface

This volume comprises the proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics – PPAM 2011, which was held in Toruń, Poland, September 11–14, 2011. It was organized by the Department of Computer and Information Science of the Częstochowa University of Technology, with the help of the Nicolaus Copernicus University in Toruń, Faculty of Mathematics and Computer Science. The main organizer was Roman Wyrzykowski.

PPAM is a biennial conference. Eight previous events have been held in different places in Poland since 1994. The proceedings of the last five conferences have been published by Springer in the *Lecture Notes in Computer Science* series (Nałęczów, 2001, vol. 2328; Częstochowa, 2003, vol. 3019; Poznań, 2005, vol. 3911; Gdańsk, 2007, vol. 4967; Wrocław, 2009, vols. 6067 and 6068).

The PPAM conferences have become an international forum for exchanging ideas between researchers involved in scientific and parallel computing, including theory and applications, as well as applied and computational mathematics. The focus of PPAM 2011 was on models, algorithms, and software tools which facilitate efficient and convenient utilization of modern parallel and distributed computing architectures, as well as on large-scale applications, and cloud computing.

This meeting gathered more than 200 participants from 33 countries. A strict refereeing process resulted in acceptance of 130 contributed presentations, while approximately 45% of the submissions were rejected. Regular tracks of the conference covered such important fields of parallel/distributed/grid computing and applied mathematics as:

– Parallel/distributed architectures and mobile computing
– Numerical algorithms and parallel numerics
– Parallel non-numerical algorithms
– Tools and environments for parallel/distributed/grid computing
– Applications of parallel/distributed computing
– Applied mathematics, neural networks and evolutionary computing
– History of computing

The plenary and invited talks were presented by:

– David A. Bader from the Georgia Institute of Technology (USA)
– Paolo Bientinesi from the RWTH Aachen (Germany)
– Christopher Carothers from the Rensselaer Polytechnic Institute (USA)
– Ewa Deelman from the University of Southern California (USA)
– Jack Dongarra from the University of Tennessee and Oak Ridge National Laboratory (USA)
– Geoffrey Ch. Fox from the Indiana University (USA)
– Fred Gustavson from the Umeå University (Sweden) and emeritus from the IBM T.J. Watson Research Center (USA)

- Tony Hey from the Microsoft Research
- Bo Kågström from the Umeå University (Sweden)
- Jakub Kurzak from the University of Tennessee (USA)
- Jarek Nabrzyski from the University of Notre Dame (USA)
- Raymond Namyst from the University of Bordeaux & INRIA (France)
- Victor Pankratius from the University of Karlsruhe (Germany)
- Markus Pueschel from the ETH Zurich (Switzerland)
- Eugen Schenfeld from the IBM T.J. Watson Research Center (USA)
- Robert Strzodka from the Max Planck Institut für Informatik (Germany)
- Bolesław Szymański from the Rensselaer Polytechnic Institute (USA)
- Richard W. Vuduc from the Georgia Institute of Technology (USA)
- Jerzy Waśniewski from the Technical University of Denmark (Denmark)

Important and integral parts of the PPAM 2011 conference were the workshops:

- Minisymposium on GPU Computing organized by José R. Herrero from the Universitat Politecnica de Catalunya (Spain), Enrique S. Quintana-Ortí from the Universitat Jaime I (Spain), and Robert Strzodka from the Max Planck Institut für Informatik (Germany)
- Minisymposium on Autotuning organized by Richard W. Vuduc from the Georgia Institute of Technology (USA) and Roman Wyrzykowski from the Częstochowa University of Technology (Poland)
- Workshop on Memory and Data Parallelism on Multi- and Manycore Platforms organized by Michael Bader from the University of Stuttgart (Germany), Carsten Trinitis, and Josef Weidendorfer from the TU München (Germany)
- Workshop on Models, Algorithms and Methodologies for Hierarchical Parallelism in New HPC Systems organized by Giulliano Laccetti and Marco Lapegna from the University of Naples Federico II (Italy) and Raffaele Montella from the University of Naples "Parthenope" (Italy)
- Workshop on Scheduling for Parallel Computing—SPC 2011—organized by Maciej Drozdowski from the Poznań University of Technology (Poland)
- The 4th Workshop on Language-Based Parallel Programming Models—WLPP 2011—organized by Ami Marowka from the Bar-Ilan University (Israel)
- The Second Workshop on Scalable Computing in Distributed Systems and the 7th Workshop on Large-Scale Computations on Grids—ScoDiS-LaSCoG 2011—organized by Dana Petcu from the West University of Timisoara (Romania) and Marcin Paprzycki from WSM and the Systems Research Institute of the Polish Academy of Sciences (Poland)
- The Third Workshop on Performance Evaluation of Parallel Applications on Large-Scale Systems organized by Jan Kwiatkowski from the Wrocław University of Technology (Poland)
- Workshop on Parallel Computational Biology—PBC 2011—organized by David A. Bader from the Georgia Institute of Technology (USA), Jarosław Żola from the Iowa State University (USA), and Scott Emrich from the University of Notre Dame (USA)

– Minisymposium on Applications of Parallel Computations in Industry and Engineering organized by Raimondas Čiegis from the Vilnius Gediminas Technical University (Lithuania) and Julius Žilinskas from the Vilnius University (Lithuania)
– Minisymposium on High-Performance Computing Interval Methods organized by Bartłomiej J. Kubica from the Warsaw University of Technology (Poland)
– Workshop on Complex Colective Systems organized by Paweł Topa and Jarosław Wąs from the AGH University of Science and Technology in Cracow (Poland)
– The First Workshop on Service-Oriented Architecture in Distributed Systems—SOADS 2011—organized by Jan Kwiatkowski from the Wrocław University of Technology (Poland) and Dariusz Wawrzyniak from the Poznań University of Technology (Poland)

The PPAM 2011 meeting began with five tutorials:

– Scientific Computing with GPUs, by Dominik Göddeke from the University of Dortmund (Germany), Jakub Kurzak from the University of Tennessee (USA), Jan-Philipp Weiss from the Karlsruhe Institute of Technology (Germany), as well as André Heidekrüger from AMD, and Tim Schröder from NVIDIA
– StarPU System for Heterogeneous Multicore Architectures, by Raymond Namyst from the University of Bordeaux and INRIA (France)
– Tutorial on the 100th Anniversary of Cholesky's Algorithm, by Fred Gustavson from the Umeå University (Sweden) and emeritus from the IBM T.J. Watson Research Center (USA) and Jerzy Waśniewski from the Technical University of Denmark (Denmark)
– FutureGrid, by Geoffrey Ch. Fox from the Indiana University (USA)
– Best Practices to Run Applications in HPC Environments, by the POWIEW Project team (Poland)

The PPAM Best Poster Award is granted to the best poster on display at the PPAM conferences, and was established at PPAM 2009. This Award is bestowed by the Program Committee members to the presenting author(s) of the best poster. The selection criteria are based on the scientific content and on the quality of the poster presentation.

The PPAM 2011 winners were Damian Wóicik, Marcin Kurowski, Bogdan Rosa, and Michał Ziemiański from the Institute of Meteorology and Water Management in Warsaw, who presented the poster "A Study on Parallel Performance of the EULAG F90/95 Code."

The Special Award was bestowed to Andrzej Jarynowski from the Jagiellonian University and Przemysław Gawroński, Krzysztof Kułakowski from the AGH University of Science and Technology in Kraków, who presented the poster "How the Competitive Altruism Leads to Bistable Homogeneous States of Cooperation or Defection."

Automated Performance Tuning ("Autotuning") of Software: The complexity of modern machines makes performance tuning a tedious and time-consuming task. The goal of *autotuning* techniques is to automate the process of selecting the highest-performing program implementation from among a space of candidates, guided by experiments. An experiment is the execution of a benchmark and observation of its performance; such experiments may be used directly to test a candidate implementation, or may be used to calibrate a model that is then used to select such an implementation. Roughly speaking, autotuning research considers questions of how to identify and generate the space of candidate program implementations as well as how to find (or search for) the best implementation given such a space. A system that implements an autotuning process is an *autotuner*. An autotuner may be a stand-alone code generation system or may be part of a compiler.

The Minisymposium on Autotuning featured a number of invited and contributed talks covering recent and diverse advances, including:

- A new high-level rewrite system for linear algebra computations, with applications to computational physics and biology (by P. Bientinesi)
- Novel uses of machine learning to facilitate searching (M. Püschel)
- The extension of autotuning ideas into general software engineering processes, such as tuning the software architecture (V. Pankratius)
- New code generation and search space pruning techniques for dense linear algebra targeted at GPU architectures (J. Kurzak and H.H.B. Sørensen)
- Reducing tuning time for high-performance LINPACK using novel performance models (P. Łuszczek)

The organizers are indebted to the PPAM 2011 sponsors, whose support was vital to the success of the conference. The main sponsor was the Intel Corporation. The other sponsors were: IBM Corporation, Hewlett-Packard Company, Microsoft Corporation, and AMD. We thank all members of the International Program Committee and additional reviewers for their diligent work in refereeing the submitted papers. Finally, we thank all of the local organizers from the Częstochowa University of Technology, and the Nicolaus Copernicus University in Toruń, who helped us to run the event very smoothly. We are especially indebted to Grażyna Kołakowska, Urszula Kroczewska, Łukasz Kuczyński, and Marcin Woźniak from the Częstochowa University of Technology; and to Andrzej Rozkosz, and Piotr Bała from the Nicolaus Copernicus University.

We hope that this volume will be useful to you. We would like everyone who reads it to feel invited to the next conference, PPAM 2013, which will be held during September 8–11, 2013, in Warsaw, the capital of Poland.

February 2012

Roman Wyrzykowski
Jack Dongarra
Konrad Karczewski
Jerzy Waśniewski

Organization

Program Committee

Jakl, Ondrej	Institute of Geonics, Czech Academy of Sciences
Janciak, Ivan	University of Vienna, Austria
Jeannot, Emmanuel	INRIA, France
Kalinov, Alexey	Cadence Design System, Russia
Kamieniarz, Grzegorz	A. Mickiewicz University, Poznań, Poland
Kiper, Ayse	Middle East Technical University, Turkey
Kitowski, Jacek	Institute of Computer Science, AGH, Poland
Korbicz, Józef	University of Zielona Góra, Poland
Kozielski, Stanislaw	Silesia University of Technology, Poland
Kranzlmueller, Dieter	Ludwig Maximilian University, Munich, and Leibniz Supercomputing Centre, Germany
Krawczyk, Henryk	Gdańsk University of Technology, Poland
Krzyżanowski, Piotr	University of Warsaw, Poland
Kwiatkowski, Jan	Wrocław University of Technology, Poland
Laccetti, Giulliano	University of Naples Federico II, Italy
Lapegna, Marco	University of Naples Federico II, Italy
Lastovetsky, Alexey	University College Dublin, Ireland
Maksimov, Vyacheslav I.	Ural Branch, Russian Academy of Sciences
Malyshkin, Victor E.	Siberian Branch, Russian Academy of Sciences
Margalef, Tomas	Universitat Autonoma de Barcelona, Spain
Margenov, Svetozar	Bulgarian Academy of Sciences, Sofia
Marowka, Ami	Bar-Ilan University, Israel
Meyer, Norbert	PSNC, Poznań, Poland
Nabrzyski, Jarek	University of Notre Dame, USA
Oksa, Gabriel	Slovak Academy of Sciences, Bratislava
Olas, Tomasz	Czestochowa University of Technology, Poland
Paprzycki, Marcin	WSM & IBS PAN, Warsaw, Poland
Petcu, Dana	West University of Timisoara, Romania
Quintana-Ortí, Enrique S.	Universitat Jaime I, Spain
Robert, Yves	Ecole Normale Superieure de Lyon, France
Rokicki, Jacek	Warsaw University of Technology, Poland
Rutkowski, Leszek	Częstochowa University of Technology, Poland
Seredyński, Franciszek	Polish Academy of Sciences and Polish-Japanese Institute of Information Technology, Warsaw, Poland
Schaefer, Robert	Institute of Computer Science, AGH, Poland
Silc, Jurij	Jozef Stefan Institute, Slovenia
Sloot, Peter M.A.	University of Amsterdam, The Netherlands
Sosonkina, Masha	Ames Laboratory and Iowa State University, USA
Sousa, Leonel	Technical University of Lisbon, Portugal
Stroiński, Maciej	PSNC, Poznań, Poland
Talia, Domenico	University of Calabria, Italy
Tchernykh, Andrei	CICESE, Ensenada, Mexico

Table of Contents – Part I

Table of Contents – Part II

Minisymposium on Applications of Parallel Computation in Industry and Engineering

A Look Back: 57 Years of Scientific Computing

Jerzy Waśniewski[1,2]

[1] Department of Informatics & Mathematical Modeling,
Technical University of Denmark,
DTU, Bldg. 305,
DK-2800 Lyngby, Denmark
jw@imm.dtu.dk

[2] UNI●C Danish IT Centre for Research and Education

Abstract. This document outlines my 57-year career in computational mathematics, a career that took me from Poland to Canada and finally to Denmark. It of course spans a period in which both hardware and software developed enormously. Along the way I was fortunate to be faced with fascinating technical challenges and privileged to be able to share them with inspiring colleagues. From the beginning, my work to a great extent was concerned, directly or indirectly, with computational linear algebra, an interest I maintain even today.

Professional Curriculum Vitae

1950 - 1952: My original plan was to study astronomy at the University of Wroclaw. One could do this after two years of either mathematics or physics, and I chose mathematics.

1952 - 1955: As a continuation of my first two years, I decided to give up astronomy for applied mathematics in technical problems. This line of study was not available at the University of Wroclaw, but it was available at the University of Warsaw. I transferred to the University of Warsaw and graduated there after three years, having specialized in computational mathematics.

1956 - 1958: I was employed by the Department of Mechanical Engineering at the Technical University of Warsaw (TUW) as an Assistant Professor of Mathematics. My initial duties were teaching and collaboration with engineers.

1956: I collaborated with the Institute of Geodesy, which had to solve symmetric positive definite linear systems of equations of orders ranging from 50 to several hundred. They solved these systems using desk calculators, the larger systems requiring several weeks.

While awaiting the completion of a new electronic digital computer being constructed by the Institute of Mathematics at the Polish Academy of Sciences, Gerard Kudelski at the Institute of Geodesy decided to exploit temporarily the equipment at that time in use in many statistics and large bookkeeping offices.

I was active in this effort. We adapted the LDL algorithm to a computing environment based on the following:

– Paper cards for storing data
– A tabulator for additions
– A multiplier for multiplications
– A card sorter
– A device for duplicating cards with changing columns of indexes.
– An electrical calculator for division.

See: L.J. Comrie et al., The application of Hollerith equipment to an agricultural investigation, J. Roy. Statist. Soc. Suppl. 4(2), 210-224 (1937).

Even though this equipment was available only at night, we could solve in several nights systems that had required several weeks with desk calculators.

1957: The new computer, designated XYZ, began operations in the autumn of this year. It had the following features:

– 512 36-bit words
– 100 operations per second
– The binary number system
– Fixed-point arithmetic
– Input / output on paper cards
– Machine language programming

I was asked to organize a computing center in connection with this computer.

1957 - 1959: I established three groups:

– A group to solve symmetric positive definite linear systems on the XYZ. The first program was made by Krzysztof Moszyński and Jerzy Świaniewicz. Krzysztof was my deputy.
– A group to calculate on desk calculators.
– A group to develop a numerical library for the XYZ. A collection of basic routines had already been provided by the machine constructors.

1959 - 1961: I moved to the Institute of Mathematics at TUW. I had to put together a new computing group and organize lectures and seminars for TUW personnel and industrial engineers. My close collaborators were A. Wakulicz, J. Hallay, M. Łącka, and T. Kornatowski.

Z. Pawlak and A. Wakulicz at the Department of Electronics at TUW developed a digital computer, designated EMC-1, with a negative-base number system! See: Donald E. Knuth, The Art of Computer Programming, Vol. 2, p. 171. N. Metropolis, J. Howlett and Gian-Carlo Rota (eds.), "A History of Computing in the Twentieth Century", Academic Press, 1980.

Jerzy Połoński at the Department of Electronics continued the development of the EMC-1 and produced a new computer, the UMC-1 (on tubes) and later UMC-10 (on transistors). My group developed the program library for the UMC-1 and assisted some of the users with their applications in such areas as hydraulics, meteorology and ship construction. Some calculations required 96 continuous hours of machine time.

1961 - 1965: I was inducted into the army and later promoted to the rank of captain. I was responsible for organizing a computing center based on the Russian Computer URAL-2 at the Military Technical University. As far as I recall, the machine parameters were the following:

– 2048 words of core memory
– 2 drums of 8192 words each
– Hardware floating-point arithmetic
– The octal number system
– Machine language programming
– Input was punched film tape and the output device was a primitive printer.

Initially there was no software at all. Since there was a strong need for making calculations without excessive delay, we supplemented the URAL-2 with a UMC-1. The UMC-1 was slower than the URAL-2 but we were familiar with it!

At the same time that the UMC-1 was being used for daily computing, we developed software for basic numerical operations on the URAL-2 and provided for input with paper tape and output with both paper tape and a teleprinter. In addition, Jerzy Hallay developed a compiler for the URAL-2 based on Polish notation. Jacek Moszczynski developed a compiler for assembly-language programs. Jerzy Hallay, Jerzy Wilczkowski, and Frania Jarosinska wrote programs for the numerical libraries.

Next, we started to run applications in the areas of hydraulics, meteorology, operations research and military science. Software was developed for solving ordinary and partial differential equations.

1965: I was moved by the Minister of War to the Ministry of Computers and Informatics and appointed Deputy Director General of Computing in Poland. My group organized a series of provincial computing centers based on the Polish Computers ZAM-2, ZAM-41, UMC-1 and UMC-10, the Russian computers URAL-2 and MINSK, and various Western computers produced by ELLIOTT, GIER, ICT and IBM. It also organized a computer factory in Wroclaw, designated ELWRO. The first computer made by ELWRO was based on the computer made by Z. Pawlak at the Technical University of Wroclaw. It was designated ODRA.

1966 - 1968: I returned to the Institute of Mathematics at TUW. Actually, I had had a close relationship with TUW during my time at the Military Technical University and the Ministry of Computers and Informatics. I taught Algol 60 (in particular GIER Algol 60) and numerical methods. I used the Danish computer GIER at the University of Warsaw and worked on linear programming models of transportation problems.

1968: I left Poland for Toronto, where I had a cousin, and didn't return to Poland until 1997, when I made the first of several visits. Through a friend at the University of Waterloo, I obtained employment for five months in the Department of Computer Science. I simulated some formulas from mathematical logic using Fortran, WatFor and APL, all of which were new to me. My supervisor was Tomasz Pietrzykowski.

The University had a "fast terminal" based on an IBM 360. Input was punched cards and output was usually printed. For smaller jobs, the output was often waiting for the user by the time he had walked the short distance from the card reader to the printer! For longer jobs, the user left his card deck in the machine room and the results were available either a few hours later or the next day. There was also an online terminal to the IBM 360 for the use of APL. The contract ended at the end of 1968. However, by then I had found another job.

1969: On January 1 I began work at Computel Systems Ltd. in Ottawa, where I was assigned to the Users' Support Group. The company sold computing time by telephone. We served as consultants to the users and I was concerned with linear programming algorithms and transportation problems. The company had two computers, a UNIVAC 1108 (Exec 2) and an IBM 360. Three linear programming packages were available: ILONA & FMPS on the 1108 and MPS on the 360. I was mostly busy with an oil company.

1970 - 1971: A friend of mine, Dr. Józef Lityński worked at the Trois Riviers branch of the University of Quebec. He introduced me to the head of the Computing Center there, Dr. Jean Lapointe, who offered me a position as consultant and researcher. One of the conditions for the job was that I should speak French. The University paid me to study French at the local Berlitz school full time the first six months. At the end of this period I gave a lecture in French for scientific personnel on Algol 60 .

In addition to the Trois Riviers branch, the University of Quebec had nine other provincial branches, each with its own computing center. The University also had a main computing center in Quebec City with a CDC 6600. The computing center of each branch had a CDC 3075, or a Terminal-200 connected to the CDC 6600. (The CDC 3075 could also function as a terminal to the CDC 6600.) In addition, some branches had teletype terminals connected to an IBM 360 at Laval University in Quebec City for APL jobs.

My duties were to teach programming and assist various departments with their computational problems. In particular, I gave support in the use of Fortran, Algol 60 and assembly language.

1971 - 1986: I left Canada for Denmark in June 1971. In August I began employment at the Regional Computing Center at the University of Copenhagen (RECKU). My duties were consulting and research. RECKU, which was new at that time, had a UNIVAC 1106, a computer I knew well.

RECKU was temporarily placed in the same building as the Niels Bohr Institute (the University's department of theoretical physics), which became one of RECKU's most important users. The Niels Bohr Institute had used the IBM 360 computer at the Danish Technical University (DTU) via an IBM 1130 Terminal. The Niels Bohr Institute also had offices at the Research Center at Risoe and used a Burroughs computer there. Now the Institute was to use the UNIVAC 1106 at RECKU.

Theoretically, a Fortran program should be able to run on any machine with a Fortran compiler, and a similar statement could be made about an Algol 60 program. In practice, however, differences exist, and that was the case here. One of my tasks was to make a program to convert IBM 360 Fortran code to UNIVAC Fortran code. I also made a program to convert Burroughs Algol 60 code to UNIVAC Algol 60 code.

1971 - 1973: Numerical software was needed for computing special functions and for solving eigenproblems, linear and nonlinear systems, and optimization problems. RECKU had only the standard UNIVAC packages UNIVAC MATH-PACK and UNIVAC STAT-PACK. Unfortunately, these were of poor quality and unpopular with the users. I therefore installed a number of new libraries, including the IBM SSP Library, the CERN Library and the Harwell Library, the latter being obtained from Lund University. A number of routines were obtained by copying program code from the Communications of the ACM and the Computer Journal. In addition, some users contributed their own software. I developed software for computing the Bessel, Gamma, and error functions and for solving eigenproblems.

We collaborated with Axel Hunding of the Department of Chemistry, University of Copenhagen on a few optimization routines, and with Kaj Madsen of the Department of Numerical Analysis, DTU on software for interval arithmetic.

1973: I attended for the first time the biennial Dundee Conference on Numerical Analysis in Dundee, Scotland. I met there two prominent members of the Department of Numerical Analysis at DTU, Hans Bruun Nielsen and Per Grove Thomsen. I discussed with them the status of numerical software at RECKU, and they introduced me to some of the Harwell Library staff attending the conference. Hans and Per also invited me to the numerical seminars organized by their department in Denmark. Some of the staff of the Department of Numerical Analysis were very knowledgeable about the Harwell Library. In particular, Kaj Madsen spent a year at Harwell, where he worked with Mike Powell on optimization problems. He contributed a few optimization routines to the Harwell Library.

Kaj Madsen was going to spend a leave of absence from DTU at the University of Copenhagen and wanted to collaborate with RECKU because of our UNIVAC computer. Together we installed a newer version of the Harwell Library, this time obtained directly from Harwell, where we had access to contributors to the Library.

1973 - 1975: I collaborated with: Per Grove Thomsen of the Department of Numerical Analysis, DTU on ordinary differential equations and graphics; Axel Hunding on optimization algorithms in medicine; Kjeld Schaumburg of the Department of Chemistry, University of Copenhagen on spectroscopic problems and ordinary differential equations; Erik Kirsbo of RECKU on map-making in oil exploration; Christian de Polignac of the University of Grenoble on multiple precision software for UNIVAC computers; and with Zahari Zlatev and Kjeld Schaumburg on a book about the routine Y12M for solving sparse linear systems of equations and ordinary differential equations for spectroscopic problems. In connection with some of these projects I was co-author of several publications a year on sparse matrix techniques and the numerical solution of ordinary differential equations.

In this period I also became familiar with the new versions of the linear programming packages ILONA and FMPS, and I implemented MATLAB on our UNIVAC.

1975: I attended the 1975 Conference on Numerical Analysis in Dundee and gave a talk on computerized map-drawing. While there, I discussed numerical libraries with people from Harwell and the Numerical Algorithms Group (NAG). On the way back to Copenhagen, I visited the NAG office at Oxford University. RECKU wanted the NAG library for its UNIVAC users, but NAG didn't have a UNIVAC version. I agreed with NAG to make a UNIVAC implementation. This was the beginning of a NAG Library Project, which became an important part of my work.

1975 - 1986: The NAG Library Project involved me in the NAG Fortran Library, the NAG Algol 60 Library, the NAG SIMULA Library, NAG Graphics, NAG GENSTAT, and NAG GLIM. It was agreed that RECKU would obtain free licenses for all implemented NAG Software and that RECKU would be a distributor of NAG-UNIVAC software. We had about 500 customers worldwide.

I collaborated closely with a number of NAG people, including Jeremy Du Croz, Steve Hague, Sven Hammarling, Ian Houman, Mick W. Point, and David Sayers. I was invited to many places in Europe and the USA and gave a number of talks.

1982: I was invited by Jack Dongarra to visit the Argonne National Laboratory in Argonne, Illinois, and when Jack moved to the University of Tennessee in Knoxville and Oak Ridge National Laboratory in Oak Ridge, Tennessee he invited me to visit that institutions. I was invited by Fred Gustavson to visit the IBM Research Center at Yorktown Heights, New York, and by Susanne Balle to gave a talk at HP Research Center in Boston. I also visited the National Bureau of Standards in Washington, D.C. and other institutions, usually giving a talk wherever I went.

1982 - 1986: I was much involved with the UNIVAC 1100 series computers. I was invited to collaborate on the new UNIVAC Virtual system. Jeremy Du Croz and I tested the UNIVAC Integrated Scientific Processor (ISP), a vector processor.

1986: The Northern Europe Computing Center (NEUCC) at DTU and the Regional EDP Center at Aarhus University (RECAU) combined with RECKU to form The Danish IT Centre for Research and Education (UNI•C). UNI•C acquired an Amdahl 1100 (Fujitsu 100) and an Alliant vector computer. My office was moved to DTU. I implemented the NAG Library on the Amdahl. The University of Manchester had got the Amdahl Computer, exactly the same as UNI•C. They used our implementation. I visited the department and gave talks a few times.

1988: I was invited to the Amdahl headquarters in Sunnyvale, California to assist in improving the level-2 BLAS. We improved these by a factor of 2 or more and the new BLAS were incorporated into the Amdahl compilers. This project was a collaboration with Jeremy Du Croz.

1989 - 1990: I moved to New Haven, Connecticut, USA, where I took an employment with Multiflow Computer Corporation as a Senior Numerical Analyst. Multiflow was a manufacturer and seller of minisupercomputer hardware and software based on VLIW (Very Long Instruction Word) technology. My duties were to implement numerical software for Multiflow computers and assist users. I worked on the BLAS, the NAG Library, and the routines Y12M for solving sparse linear systems of equations. I invited Kaj Madsen, Zahari Zlatev and Steve Hague (NAG) to Multiflow to give a talk.

1990 - 1994: I returned to UNI•C in Denmark, which had acquired supercomputers of the Connection Machine, MasPar and KSR types. I assisted Zahari Zlatev of the National Environmental Research Institute of Ministry of Environment in implementing a large-scale program for air pollution analysis. I made several visits to Thinking Machine Corporation (the manufacturer of the Connection Machine) and KSR Computer Corporation, both in Boston, and had several papers published.

1994 - 1996: Jack Dongarra and I initiated a series of conferences on applied parallel computing with the generic name PARA. The first three of these, PARA'94, PARA'95 and PARA'96 were held at UNI•C and the Department of Mathematical Modelling (IMM) at DTU. (IMM was the result of the fusion of the Department of Numerical Analysis with a number of other departments at DTU.) Kaj Madsen, the head of IMM, was a co-organizer of these conferences.

I became involved in a project to create an interface between the linear algebra package LAPACK, written in Fortran 77, and Fortran 95. The purpose was to enable the Fortran 95 user to call an LAPACK routine in his Fortran 95 program code. V. Allan Barker of IMM participated in this effort, as did a number of others. The new package was entitled LAPACK95.

I also became involved in the NetSolve project, an ambitious scheme to implement the solution of large-scale computational problems by finding and exploiting diverse computational resources connected by computer networks. The project was later abandoned.

1997 - 2000: It was decided that the PARA conferences should henceforth be held in various Scandinavian cities. Thus PARA'96 was followed by PARA'98 in Umeå, Sweden and PARA 2000 in Bergen, Norway.

Starting in 1997, I was an invited speaker at the Polish Computing Science conferences, "Parallel Processing and Applied Mathematics (PPAM)". The conferences were held in odd-numbered years.

This period saw the start of the LAWRA project (Linear Algebra with Recursive Algorithms), a collaboration between UNI●C and the IBM Research Center in Yorktown Heights.

1997: I was active in a workshop on scientific computing, where the LAWRA project was discussed.

The LAWRA team:

- Bjarne Stig Andersen (UNI●C)
- Fred Gustavson, co-coordinator (IBM)
- Alexander Karaivanov (University of Rousse, Bulgaria)
- Ivan Lirkov (Academy of Sciences, Bulgaria)
- Minka Marinova (University of Rousse, Bulgaria)
- Jerzy Waśniewski, co-coordinator (UNI●C)
- Plamen Yalamov (University of Rousse, Bulgaria)

Some Results:

- The case of symmetric matrices and full storage: The recursive algorithms were about 1/3 faster.
- The case of symmetric matrices and packed storage: The recursive algorithms were several times faster then the old packed algorithms. They were also faster than some algorithms based on full storage.

A one-day LAWRA workshop was organized at the end of 1999. It ended with a reception sponsored by IBM. The Workshop and reception were attended by some prominent numerical analysts, including Jack Dongarra, Petter Bjorstad, Bo Kågström, Fred Gustavson, and Zahari Zlatev.

LAPACK95 was updated to use LAPACK version 3. Further, test problems for the various routines in LAPACK95 were made a part of LAPACK95. The LAPACK95 Users' Guide was published by SIAM in 2001.

2001 - 2012: I retired on January 16, 2001.

Yes, I retired, but I haven't stopped working! I have, however, limited my work to: LAPACK95 support, the PARA and PPAM conferences (in addition to participating in these conferences I have done much organizational work), and the LAWRA project.

The following PARA conferences have been held during my "retirement": PARA 2002 in Espoo, Finland; PARA 2004 at DTU; PARA 2006 in Umeå, Sweden; PARA 2008 in Trondheim, Norway; PARA 2010 in Reykjavik, Iceland; PARA 2012 in Espoo, Finland. PARA 2014 will be held in Denmark.

I still have a desk at DTU. However, I work mostly from home.

The LAWRA project is documented by the following papers:

- J. Waśniewski, B.S. Andersen, and F. Gustavson. Recursive Formulation of Cholesky Algorithm in Fortran 90. In Proceedings of the 4-th

International Workshop, Applied Parallel Computing, Large Scale Scientific and Industrial Problems, PARA'96, B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski (Eds.). Umeå, Sweden, June 1996. Springer, LNCS Number 1541, pp. 574-578.
- F. Gustavson, A. Karaivanov, J. Waśniewski, and P. Yalamov. A columnwise recursive perturbation based algorithm for symmetric indefinite linear systems. In Proc. PDPTA'99, Las Vegas, 1999.
- B.S. Andersen, F. Gustavson, A. Karaivanov, J. Waśniewski,and P. Yalamov. "LAWRA, Linear Algebra With Recursive Algorithms". In Proceedings of the Third International Conference on Parallel Processing and Applied Mathematics. Editors: R. Wyrzykowski, B. Mochnacki, H. Piech, and J. Szopa. Kazimierz Dolny, Poland, 14-17.09.1999.
- Bjarne S. Andersen, Fred G. Gustavson, and Jerzy Waśniewski. "A recursive formulation of Cholesky factorization of a matrix in packed storage". ACM Transactions on Mathematical Software, 27(2):214-244, June 2001
- B.S. Andersen, J.A. Gunnels, F. Gustavson, J.K. Reid, and J. Waśniewski. "A Fully Portable High Performance Minimal Storage Hybrid Format Cholesky Algorithm". ACM Transactions on Mathematical Software, 31 (2005), 201-227.
- Gustavson, F.G., Reid, J. K., and Waśniewski, J., (2007). "Algorithm 865: Fortran 95 Subroutines for Cholesky Factorization in Blocked Hybrid Format". ACM Transactions on Mathematical Software, 33, 1 (March), 5.
- Fred G. Gustavson, Jerzy Waśniewski, Julien Langou and Jack J. Dongarra: LAPACK Working Note Nr 199 "Rectangular Full Packed Format for Cholesky's Algorithm: Factorization, Solution and Inversion", ACM Transactions on Mathematical Software, Vol. 37, No. 2, Article 18, Publication date: April 2010.
- Fred G. Gustavson, Jerzy Waśniewski, Jack J. Dongarra, Jose R. Herrero and Julien Langou: LAPACK Working Note No. 249, Level 3 Cholesky Factorization Routines as Part of Many Cholesky Algorithms. (Accepted for publication by the ACM Transactions on Mathematical Software.) "http://www.netlib.org/lapack/lawns/index.html"

2008: I was invited to join the Numerical Analysis Group of the organization European Cooperation in Science and Technology (COST), the mission of which is to promote the coordination of nationally-funded research on a European level. (See: www.cost.esf.org .) We meet twice a year to give talks on scientific computing and we organize a biennial summer school on this subject.

Acknowledgments. First I would like to thank V. Allan Barker, who suggested this professional memoir and helped me with a number of details. Thanks are also due to Roman Wyrzykowski for giving me the opportunity to present this material as a plenary lecture of the PPAM 2011 Conference and to contribute it to the Conference Proceedings. Finally, and most importantly, I must extend my heart-felt thanks to all my colleagues through the years, too numerous to mention, whose inspiration and support were the brightest part of a long professional journey.

Modeling a Leadership-Scale Storage System

Ning Liu[1], Christopher Carothers[1], Jason Cope[2], Philip Carns[2], Robert Ross[2], Adam Crume[3], and Carlos Maltzahn[3]

[1] Rensselaer Polytechnic Institute, Troy, NY 12180, USA
{liun2,chrisc}@cs.rpi.edu

[2] Argonne National Laboratory, Argonne, IL 60439, USA
{copej,pcarns,rross}@mcs.anl.gov

[3] University of California at Santa Cruz, Santa Cruz, CA 95064, USA
{adamcrume,carlosm}@soe.ucsc.edu

Abstract. Exascale supercomputers will have the potential for billion-way parallelism. While physical implementations of these systems are currently not available, HPC system designers can develop models of exascale systems to evaluate system design points. Modeling these systems and associated subsystems is a significant challenge. In this paper, we present the Co-design of Exascale Storage System (CODES) framework for evaluating exascale storage system design points. As part of our early work with CODES, we discuss the use of the CODES framework to simulate leadership-scale storage systems in a tractable amount of time using parallel discrete-event simulation. We describe the current storage system models and protocols included with the CODES framework and demonstrate the use of CODES through simulations of an existing petascale storage system.

Keywords: exascale computing, storage system design, parallel discrete-event simulation.

1 Introduction

Several challenges arise in developing reliable, high-performance exascale storage systems. In particular, the availability of hardware and system software components for these systems is still years away. System designers therefore must model and simulate these systems in order to understand potential exascale storage system designs and use cases. Simulation results can then be used to influence the design of future exascale system components. Most of the recent studies [14,13,7,9] of this type are based on massively parallel discrete-event models.

In this paper, we present our recent work developing an end-to-end storage system model of the Argonne Leadership Computing Facility's (ALCF) computing and data storage environment. This work is a prerequisite to modeling and simulating exascale storage systems because it verifies that we can accurately and quickly model an existing storage system. CODES framework leverages the Rensselaer Optimistic Simulation System (ROSS) [19,18,4]. ROSS is a parallel discrete-event simulation framework that allows simulations to be run in parallel, decreasing the run time of massive simulations. Using CODES and ROSS, we validate the storage system models against data collected from the ALCF's storage system for a variety of synthetic I/O workloads and scales. we present a model of the PVFS storage system and the I/O subsystem of the Intrepid IBM Blue Gene/P (BG/P) system in the ALCF. As an early study of the CODES project, our simulators can quickly and accurately simulate a petascale storage system using medium-fidelity hardware models and an accurate representation of the storage system software protocols.

In this paper, we describe the ALCF computing environment, present the CODES models developed for this environment, and analyze simulation results produced by these models. Section 2 of this paper describes the ALCF's Intrepid storage system architecture. Section 3 describes the CODES models that compose the end-to-end storage system. Section 4 focuses on the experimental simulation results on the standard I/O models. We discuss related work in Section 5. We conclude this paper in Section 6 with a brief discussion of future work.

2 The ALCF Computing Environment

Leadership-computing systems are on the cutting edge of computing hardware and system software. Many of the I/O hardware and software components have unique features needed by such systems. This section provides an overview of the ALCF's PVFS [5] storage system and I/O subsystem on the Intrepid IBM BG/P [1].

Figure 1 illustrates the architecture of Intrepid's storage subsystem. Like other other large HPC centers [3,17], the ALCF provides a large, parallel storage

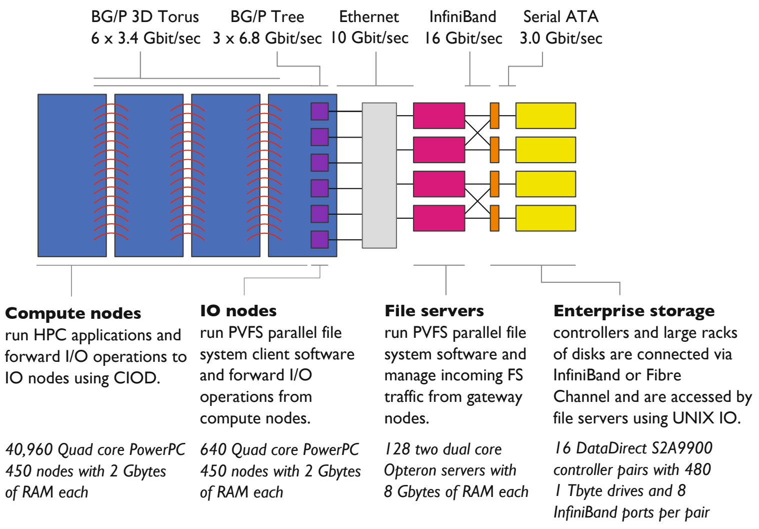

BG/P 3D Torus
6 x 3.4 Gbit/sec
BG/P Tree
3 x 6.8 Gbit/sec
Ethernet
10 Gbit/sec
InfiniBand
16 Gbit/sec
Serial ATA
3.0 Gbit/sec
Compute nodes
run HPC applications and forward I/O operations to IO nodes using CIOD.
IO nodes
run PVFS parallel file system client software and forward I/O operations from compute nodes.
File servers
run PVFS parallel file system software and manage incoming FS traffic from gateway nodes.
Enterprise storage
controllers and large racks of disks are connected via InfiniBand or Fibre Channel and are accessed by file servers using UNIX IO.
40,960 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each
640 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each
128 two dual core Opteron servers with 8 Gbytes of RAM each
16 DataDirect S2A9900 controller pairs with 480 1 Tbyte drives and 8 InfiniBand ports per pair

Fig. 1. Overview of ALCF Blue Gene/P compute and storage systems

system shared between multiple HPC resources. Intrepid is composed of several networks and several layers of computation and storage devices. The BG/P platform provides several networks that are tightly coupled with the BG/P compute nodes, so that the BG/P system can satisfy the high-performance requirements of leadership-class applications. The three-dimensional torus network is used for point-to-point communication among compute nodes (CNs), while the collective network (also known as the tree network) allow CNs to perform file I/O operations to the I/O forwarding nodes (IONs) and supports some inter-process collective operations. In the BG/P system, IONs are distinct from the storage server nodes and compute nodes. The IONs host file system clients and delegate file I/O requests on behalf of a group of compute nodes. For each group of 64 CNs on Intrepid, a single ION receives I/O requests from the CNs in that group and forwards those requests over its 10-Gigabit Ethernet network interface to PVFS and GPFS [15] storage systems. Myricom Myri-10G network infrastructure is used to translate the ION Ethernet traffic to Myrinet for the Myri-10G connected file servers. For the research described in this paper, we limited our analysis to Intrepid's PVFS storage system because a component-level study of the storage system was available [6] and the open-source nature of PVFS made it easier for us to validate our models. The PVFS file system stores data on logical units (LUNs) exported by 16 DataDirect Network (DDN) 9900 storage devices.

Intrepid provides several layers of storage software and services. The top most layer of the stack is comprised of high level I/O (HL-IO) libraries such as HDF5 or PnetCDF. These high level libraries map application data models onto conventional files and directories. The I/O middleware layer is provided by MPI-IO, which leverages both the BG/P tree network and the 3D-torus network to provide aggregate file optimizations such as two-phase I/O. Eventually, all the application I/O requests are translated into POSIX I/O requests at each CN. IBM's CIOD [10] client is used to forward the POSIX I/O requests across the BG/P tree network to the IONs. At the IONs, the CIOD server replays the forwarded I/O requests by directly issuing the POSIX file I/O requests. Each of Intrepid's IONs mounts PVFS and GPFS file system shared by all ALCF resources. PVFS file system clients on each ION communicate with PVFS servers running on the 128 storage system servers. The PVFS storage servers store file system data in LUNs exported by the DDN storage devices.

3 Modeling the ALCF Computing Environment

In this section, we describe how hardware and software components of the ALCF computing environment are modeled in CODES. The end-to-end storage system model is composed of several component models that capture the interactions of the system software and hardware interactions for I/O operations. Figure 2 illustrates the networks, hardware components, and software protocols modeled in our simulations. The storage system model also provides several configuration parameters that dictate the execution behavior of application I/O requests.

We abstracted the common features of each Blue Gene/P hardware component into CN, ION, file server, and DDN models. These models are the logical

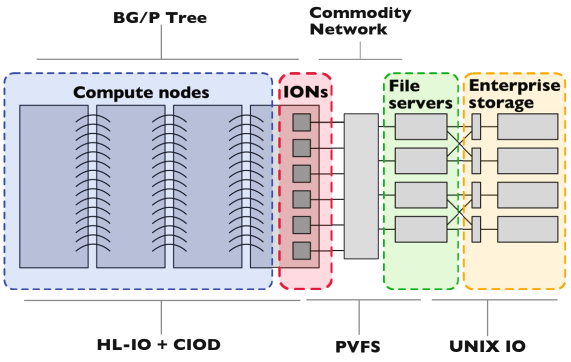


Fig. 2. CODES models for the ALCF computing environment. The models include networks (top labels), hardware components (middle labels), and software protocols (bottom labels).

processes (LPs) in our end-to-end storage system model, which are the most basic physical units of our parallel discrete-event model. The various BG/P networks are modeled as the links connecting each LP. Each LP is composed of three buffers. The incoming buffer is used to model the queuing effects from multiple LPs trying to send messages to the same LP. The outgoing buffer is used to model queuing effects when an LP tries to send multiple messages to different LPs. The processing buffer is used to model queuing effects caused by a processing unit, such as CPU, DMA engine, storage controller, or router processors. The units process incoming messages in FIFO order. Each LP also has a local hash table for recording the connections between LPs. The hash table can be viewed as a routing table from the perspective of network modeling.

The BG/P tree network is modeled by the network links that connect each CN LP with its parent and child network nodes. The root CN LP in the tree network connects to the ION LP. Network connection between two LPs are modeled as messages transmitted between the two LPs, where each LP's incoming buffer is connected to the other LP's outgoing buffer. Furthermore, the commodity networks (Ethernet and Myrinet networks) are modeled by the links connecting the IONs with the storage servers. If we increase the fidelity of our models in the future, additional network components, such as routers and switches, can be modeled as LPs.

The hardware models consist of several configurations parameters. We model the throughput and latency of the network links interconnecting distributed hardware models using Equation 1.

$$T = T_L \frac{D_P}{D_P + D_O} \tag{1}$$

Equation 1 computes the perceived throughput of a network operation (T) based on the size of the data payload (D_P), the maximum link throughput (T_L), and the size of non-payload data associated with the transfer (D_O). The data throughput and access latency of the DDN storage devices are modeled as a simple, constant function. Parameters for both models were obtained by using micro-benchmarks that measured the observed throughput between the various devices in the ALCF computing environment.

Multiple software layers are involved in Intrepid's I/O path. Our software models approximate the interfaces, protocols, and interactions of the software components deployed in the ALCF computing environment. The software models and interfaces sit on top of the hardware LPs and trigger hardware events for I/O operations. At the application layer, our models provide a POSIX-like I/O interface. Our application-level models translate application I/O requests into CIOD client requests using a series of CN and ION events. These CN and ION events reflect the interaction between the CIOD clients and servers. The CIOD server receives the CIOD client requests and generates a series of ION and storage server hardware requests that approximate the interaction of the CIOD server and the PVFS file system. The PVFS file system then generates a series of storage server and DDN events that approximate the interactions between the storage server and the DDN storage devices. The number and types of events generated by our models depend upon the complexity of the I/O system software protocol for a specific I/O layer.

Several parameters are associated with the software models. The most important parameters are the CIOD transfer size and the PVFS stripe size. CIOD limits the amount of data that can be transferred in a single I/O operation (4 MiB default value on Intrepid). CIOD requires multiple operations to transfer requests larger than 4 MiB. The PVFS stripe size dictates the block size distributed to the PVFS file servers (4 MiB default value on Intrepid) and file alignment. Requests that are not aligned on a 4 MiB boundary or exceed a 4 MiB capacity require access to multiple PVFS servers per I/O operation.

The CODES storage system simulator implements the necessary protocols to provide application-level file open, close, read, and write using the ALCF hardware and software models. Figure 3 depicts the PDES model used for application write operations. Application open, close, and read operation models have different implementation details from that of the write; they are not covered in this paper because of limited space.

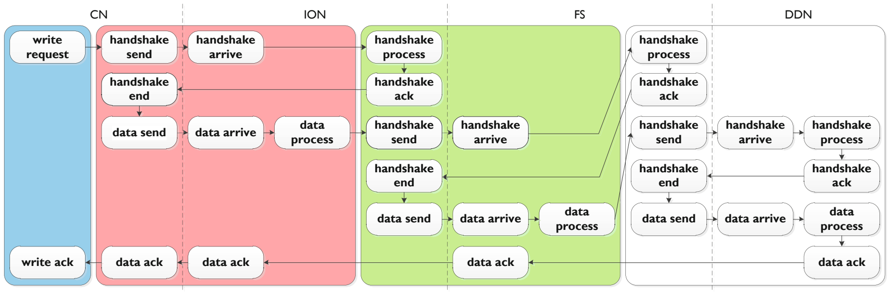


Fig. 3. CODES file write request model for Intrepid

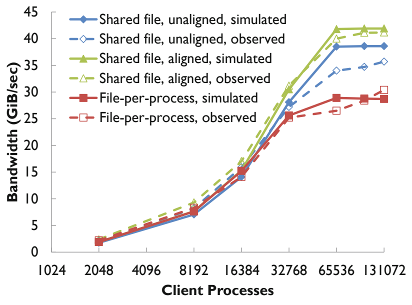
4 Model Validation and Discussion

The goal of our initial investigation using this storage system model was to validate the model's performance against data collected from Intrepid and the ALCF's PVFS storage system. Our validation of these models focused on replicating the behavior of the IOR [2] benchmark. IOR is a flexible, robust, and well-understood I/O benchmark. Validating our model against IOR gives confidence that our model is operating correctly for common I/O patterns and can be used as a springboard for future investigations into application-specific I/O patterns.

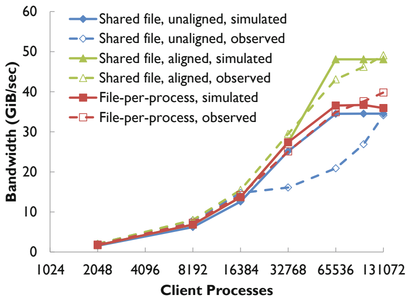
In prior work [6], we presented a thorough evaluation of the ALCF's PVFS storage system for a variety of I/O workloads and application scales. We leveraged the data generated from that past investigation to validate our simulator. We configured our simulator to be as similar as possible to the PVFS storage system described in our prior work. The experiments documented in our prior work were performed during Intrepid's acceptance testing period before the machine and file system were available for production users. During the previous investigation, the experiments used PVFS 2.8.0 with a file system configuration that consisted of 123 file servers. Each PVFS server handled both data and metadata operations. The PVFS file servers were configured with a stripe unit of 4 MiB and the CIOD maximum buffer size was set to 4 MiB.

We evaluated the model using a variety of IOR workloads, including shared file, file-per-process, stripe-aligned, stripe-unaligned, read, and write file access patterns. The shared file experiments forced each process to concurrently store data into a single file. The file-per-process experiments allowed each process to store its data into a unique file that was inaccessible by other processes. The file-per-process tests required the file system to perform additional metadata operations, such as file creations, that are not required in the shared file tests. The stripe-aligned tests used 4 MiB (4×2^{20} bytes) accesses for a total of 64 MiB per process. Stripe-aligned accesses caused each processes file requests to align with the stripe of the file. This allowed 4 MiB accesses to be made directly to the DDN LUN. The stripe-unaligned tests used 4 MB (4×10^6 bytes) accesses for a total of 64 MB per process. The stripe-unaligned accesses spanned multiple file stripes and required most requests to processed by more than one file server.

The results of our IOR write validation experiments closely follows the results observed during our previous study. The results for these experiments are illustrated in Figure 4a. The overall file system performance trend for write requests is correctly captured by our simulator. Like the results reported in our previous study, the simulator performance for write requests levels off at 64K processes and remains constant at larger scales. Our simulated results capture the performance variations from 2K to 128K client processes at roughly a 10% error rate. Specifically, the model is able to capture the extra overhead for both the stripe-unaligned experiments and file-per-process experiments. In the prior study, we observed network contention within the storage system network has caused file system performance degradation. We believe that we can capture this behavior within our models through increasing model and simulation fidelity. Specifically,



(a) write



(b) read

Fig. 4. Comparison of simulated and observed IOR performance. Figure 4a illustrates results using IOR write workloads. Figure 4b illustrates results using IOR read workloads.

we can increase network fidelity by developing and integrating commodity network hardware components, such as routers or switches, and commodity network protocols, such as Myrinet, into our storage system simulator. With these additions, we expect the total number of LPs to grow less than 100% compared to the current models size. The overall simulation runtime will stay at same level. We believe this adjustment will improve the error rate of the stripe-unaligned tests.

Figure 4b illustrates the IOR benchmark throughput for observed and simulated read operations. Like the write experiments, the stripe-aligned and stripe-unaligned accesses were investigated using 4 MiB and 4 MB PVFS stripe sizes. Our results show that the stripe-aligned read throughput closely follows the observed performance of Intrepid's PVFS storage systems. Our model is able to capture most of the performance variations for stripe-aligned read and file-per-process read experiments. It yields more error in stripe-unaligned read tests. The discrepancy is attributed to the low fidelity of our network model. We believe that the previously mentioned network contention modifications will correct this behavior as well.

All the experimental studies ran on an SMP system with a configuration of 8 cores (Intel Xeon x5430, 2.67 GHz) and 32 GiB memory. The largest test case with 128K client processes (represented as LPs in the simulation) finished within a couple of minutes, showing that our tools are capable of simulating interesting storage system designs while using modest resources. In prior work [19], we demonstrated ROSS's high efficiency attributes when modeling large-scale TCP networks. With the aid of a supercomputer resource, such as Intrepid, and by exploiting the efficiency of ROSS, we believe the simulator will be able to run an exascale storage system model in a reasonable amount of time, achieving our project goal of simulating one week worth of exascale storage system activity in $O(days)$ runtime. On Intrepid, we are currently preparing our simulator for evaluating larger scale storage system and network models. Evaluation of these large scale simulations will be a focus of our future work with CODES.

5 Related Work

As part of the exascale co-design process, there is significant interest in understanding how parallel system software such as MPI/MPI-IO and the associated supercomputing applications will scale on future architectures. For example, Perumalla's $\mu\pi$ system [13] will allow MPI programs to be transparently executed on top of the MPI modeling layer and simulate the MPI messages. In particular, $\mu\pi$ has executed an MPI job that contained over 27 million tasks and was executed on 216,000 Cray XT5 cores. A number of universities and national labs have joined together to create the Structural Simulation Toolkit (SST) [14]. SST includes a collection of hardware component models including processors, memorys and networks at different accuracy. These models use parallel component-based discrete event simulation based on MPI. The users are able to leverage multi-scale nature of SST by trading off between accuracy, complexity, and time to solution. BigSim [20] focused on the model and prediction of sequential execution blocks of large scale parallel applications. The model is based on trace-driven and it uses the scalable trace gained from machine learning for predicting overall performance. While our simulator accurately captures the large-scale storage system characteristics, these systems are more focused on providing accurate, large-scale computational performance models.

Researchers have also developed a number of parallel file system simulators. The IMPIOUS simulator [9] was developed for fast evaluation of parallel file system designs. It simulates PVFS, PanFS, and Ceph file systems based on user-provided file system specifications, including data placement strategies, replication strategies, locking disciplines, and caching strategies. The HECIOS simulator [16] is an OMNeT++ simulator for PVFS. HECIOS was used to evaluate scalable metadata operations and file data caching strategies for PVFS. PFSsim [8] is an OMNeT++ PVFS simulator that allows researchers to explore I/O scheduling algorithm design. PVFS and ext3 file systems have been simulated using colored Petri nets [12,11]. This simulation method yielded low simulation error, with less than 10% error reported for some simulations. The focus of CODES sets it apart from these related simulation tools. One of the goals of CODES is to accurately and quickly simulate large-scale storage systems. To date, CODES has been used to simulate up to 131,072 application processes, 512 PVFS file system clients, and 123 PVFS file servers. The existing simulators limited their simulations to smaller parallel systems (up to 10,000 application processes and up to 100 file servers).

6 Conclusions and Future Work

In this paper we presented an early work using our storage system framework to simulate an existing leadership-class storage system. The presented parallel discrete-event model is able to capture most tests cases of an existing, large-scale storage system with less than 10% error rate. The tests are experimented on an eight-core workstation in $O(minutes)$ runtime. Our initial simulation results are encouraging.

There are several areas of future work for this storage system simulator. We will develop standard, application-level I/O interfaces. Using these I/O interfaces, we will develop and evaluate application I/O workloads. Our plan also includes the construction of a burst buffer model and different parallel file system models. Moreover, we will construct a platform where application users can study the I/O effects and system designers can evaluate the best design points for exascale storage systems. We plan to incorporate our packet-level-accurate torus network model [7] into the current simulator and investigate its impact on storage system behaviors.

Acknowledgments. We thank Kevin Harms (ALCF) for his insight on Intrepid's PVFS storage system and helping us better understand its production configuration. This work was supported by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

1. Overview of the IBM Blue Gene/P project. IBM Journal of Research and Development, 52(1.2), 199–220 (January 2008)
2. IOR benchmark (October 2011)
3. Ang, J., Doerfler, D., Dosanjh, S., Koch, K., Morrison, J., Vigil, M.: The alliance for computing at the extreme scale. In: Proceedings of the Cray Users Group Meeting (2010)
4. Bauer, D.W., Carothers, C.D., Holder, A.: Scalable time warp on Blue Gene supercomputers. In: Proc. ACM/IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS 2009), Lake Placid, NY (2009)
5. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: A Parallel File System for Linux Clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, pp. 317–327 (2000)
6. Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., Allcock, W.: I/O performance challenges at leadership scale. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 40. ACM (2009)
7. Liu, N., Carothers, C.D.: Modeling billion-node torus networks using massively parallel discrete-event simulation. In: Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS), pp. 1–8. IEEE, France (2011)
8. Liu, Y., Figueiredo, R., Clavijo, D., Xu, Y., Zhao, M.: Towards simulation of parallel file system scheduling algorithms with PFSsim. In: Proceedings of the 7th IEEE International Workshop on Storage Network Architectures and Parallel I/O (May 2011)
9. Molina-Estolano, E., Maltzahn, C., Bent, J., Brandt, S.A.: Building a parallel file system simulator. Journal of Physics: Conference Series 180, 012050 (2009)

10. Moreira, J., Brutman, M., Castaños, J., Engelsiepen, T., Giampapa, M., Gooding, T., Haskin, R., Inglett, T., Lieber, D., McCarthy, P., Mundy, M., Parker, J., Wallenfelt, B.: Designing a highly-scalable operating system: the blue gene/l story. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006. ACM, New York (2006)
11. Nguyen, H.Q.: File system simulation: Hierachical performance measurement and modeling. PhD thesis, University of Arkansas (2011)
12. Nguyen, H.Q., Apon, A.W.: Hierarchical performance measurement and modeling of the linux file system. In: ICPE, pp. 73–84 (2011)
13. Perumalla, K.S.: $\mu\pi$: a scalable and transparent system for simulating MPI programs. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, SIMUTools 2010, pp. 62:1–62:6. ICST, Brussels (2010)
14. Rodrigues, A.F., Hemmert, K.S., Barrett, B.W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., Jacob, B.: The structural simulation toolkit. SIGMETRICS Perform. Eval. Rev. 38, 37–42 (2011)
15. Schmuck, F., Haskin, R.: Gpfs: A shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies (2002)
16. Settlemyer, B.W.: A Study of Client-side Caching in Parallel File Systems. PhD thesis, Clemson University, Clemson, South Carolina, USA (2009)
17. Shipman, G., Dillow, D., Oral, S., Wang, F.: The spider center wide file system: From concept to reality. In: Proceedings, Cray User Group (CUG) Conference, Atlanta, GA (2009)
18. Yaun, G., Carothers, C.D., Kalyanaraman, S.: Large-scale TCP models using optimistic parallel simulation. In: Proceedings of the Seventeenth Workshop on Parallel and Distributed Simulation (PADS 2003), San Diego, CA (June 2003)
19. Yaun, G.R., Bauer, D.W., Bhutada, H.L., Carothers, C.D., Yuksel, M., Kalyanaraman, S.: Largescale network simulation techniques: Examples of TCP and OSPF models. SIGCOMM Computer Comunications Review Special Issue on Tools and Technologies for Research and Eduction 33(5), 27–41 (2004)
20. Zheng, G., Gupta, G., Bohm, E., Dooley, I., Kale, L.V.: Simulating Large Scale Parallel Applications using Statistical Models for Sequential Execution Blocks. In: Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS 2010), Shanghai, China, pp. 10–15 (December 2010)

Combining Optimistic and Pessimistic Replication*

Marcin Bazydło, Szymon Francuzik, Cezary Sobaniec, and Dariusz Wawrzyniak

Institute of Computing Science, Poznań University of Technology, Poland

Abstract. This paper presents a concept of combining pessimistic and optimistic approach to replication. Optimistic replication allows for tentative system states, which increases availability and efficiency, but makes behaviour of the system less predictable, even if some operations seem completed. To enable more stable results, pessimistic and optimistic modes of operations are distinguished. Operations issued in the optimistic mode accept or produce tentative states, while operations issued in the pessimistic mode appear as completed in a stable state, termed committed. Orthogonally, to refine expectations of the results, modifications are specified as either synchronous or asynchronous, and reads as either synchronised or immediate.

Keywords: optimistic replication, availability, consistency.

1 Introduction

Replication is commonly used to improve performance, reliability and availability of data or services [9]. However, it causes the problem of consistency when the state of some replicas changes. A number of techniques has been proposed to hide the difference between replicas until they reach the same state. Nevertheless, consistency maintenance generates some overheads. Moreover, in a distributed environment a network breakdown may occur, which ceases communication between nodes, thereby compromises either availability or consistency [4]. These problems expose the distinction between optimistic and pessimistic approaches to replication.

Pessimistic approaches strive to mask any divergence between replicas, which leads to strict consistency, and in effect makes replication transparent to the users. This requires careful operation handling usually based on temporary blocking access to replicas. Crucial to the consistency is update processing, i.e. a proper order of applying modifications to individual replicas. To preserve strict consistency, the same or equivalent order of updates on each replica is necessary. On the one hand, strict consistency in some cases is hardly achievable, on the other hand, it is not always necessary. In other words, consistency requirements

* 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.

for some applications or services are less restrictive, and allow for access without constant synchronisation. This enables optimistic replication, which is based on the (optimistic) assumption that temporary difference between replicas remains invisible to users or will not impair correctness of applications. The expectation in this respect is the convergence of replicas formulated as eventual consistency.

However, there can be still some operations that cause too much risk when executed in optimistic manner. This is especially the case of operations whose results significantly influence subsequent behaviour of the users. Optimistic execution produces tentative system states (thereby the results), so may lead to confusion if some important operations are missed. To make the system states more predictable, we have introduced the distinction between optimistic and pessimistic mode of execution. This, in turn, entails appropriate indication of operations.

In view of these facts, the aim of this paper is to present a concept of combining pessimistic and optimistic approach to replication. To this end, when an operation is issued, it is defined as either optimistic or pessimistic. Besides the distinction between optimistic and pessimistic mode, some additional expectations are specified. For modifications, they concern the information on the completion, and for read-only operations — the scope of modifications to provide the result. So in effect two orthogonal classifications of operations are proposed: optimistic vs. pessimistic, and synchronous vs. asynchronous for modifications or synchronised vs. immediate for reads.

The coexistence of different classes of operations raises the question of mutual influence of their execution on the results and on the final system state. Optimistic replication distinguishes between committed and tentative states of replicas [6]. Operations issued in the optimistic mode accept or even produce tentative states, while operations issued in the pessimistic mode endeavour to transition to a committed state (at least as such must appear to the users). Actually, because of other, optimistic operations, they can also be executed in a tentative state.

2 Assumptions and System Model

In this paper we consider a system consisting of m autonomous replicas (denoted by $R_1, R_2, \ldots, R_m$) which are accessed by n processes (denoted by $P_1, P_2, \ldots, P_n$). The autonomy of replicas means that they may process requests to some extent independently of each other. The replicas are geographically distributed to allow for network partitioning while preserving availability.

Modifications issued by a process must be propagated to all replicas in order to keep their states consistent. Replication system can use either state transfer or operation transfer for that purpose [6]. State transfer consists in tracking changes caused by operations and propagating state updates to other replicas. Operation transfer consists in propagation of operations and their re-execution on every replica. Depending on the architecture of the system, state transfer may be more efficient because of: its ability to pack effects of many operations into one update message, and avoidance of their re-execution. However, change tracking highly depends on the system implementation, and may be inefficient or simply unfeasible. On the other hand, a system using operation transfer requires that all operations are deterministic in order to ensure consistency of replicas. The operation may come in arbitrary moments, which effectively means that the system must be *piecewise deterministic*. It ensures that all replicas on which the same operations have been executed in the same order remain consistent.

We have chosen operation transfer because of its wider applicability. There is a couple of reasons. First, it is usually feasible to intercept and subsequently disseminate operations. Second, replication system can work independently of the mechanism of state modification. This permits clean separation of responsibilities where the replication mechanism is not aware of implementation details of the base system.

Replication system intercepts and propagates modifications issued by the processes to all other replicas. Piecewise determinism and global ordering of modifications assure *eventual consistency* [1,7]. Eventual consistency allows for temporary divergence between replicas, but it ensures that eventually all replicas would achieve the same state if there were no new modifications.

Two main types of operations are distinguished: non-modifying operations (*reads* for short) and modifying operations (*modifications*). Reads are issued by a process in order to observe a partial state of a replica, they do not change state of the replica. Only modifications need to be propagated in order to preserve consistency. This distinction leads to improved overall efficiency due to load balancing of read operations between replicas. It should be noted that modifications are not simple state overwrites but rather general procedures causing state transitions and returning some results. Modifications will be denoted by $m(a)$, where m is an identifier of the operation and a is an argument. Read operations will be denoted by $r(a)$, where argument a represents the results of the read.

Our replication schema assumes the use of optimistic replication for providing improved availability. Optimistic replication allows concurrent modifications to be performed on different replicas, because they can be performed without coordination. This may lead to conflicts, i.e. concurrent execution of conflicting modifications. In general, two operations are conflicting if the result of their execution depends on the execution order. However, the system must decide on a global ordering of conflicting modifications in order to achieve eventual consistency. The only general way of coping with operations executed in wrong order is to retract these operations and re-execute them in the proper order. In this paper we assume that the system provides some means of retracting operations.

3 The Concept

States of Operation Processing. Essential for optimistic replication is the assumption that operations are processed independently on each replica. It is not known in advance whether the current state of the replica is appropriate for a given operation to be applied. There may be conflicting operations in the system not yet known to the replica. This uncertainty leads to the distinction between *tentative* and *committed* states [6]. An operation in tentative state can be either postponed or (optimistically) performed. Therefore, *accepted* and *applied* states are distinguished. If there is no demand for execution of an operation, it gets *accepted* state. Fig. 1 presents possible transitions between states of modifications. In case of read operations there are no transitions from applied state. It is worth noting that an applied modification can become accepted again when it must be retracted because of a conflict. An accepted operation becomes *scheduled* after establishing a global ordering of preceding operations. Finally the operation will be executed changing its state to *completed*. In case of lack of conflicts a modification executed tentatively (in applied state) can be finally committed changing its state to completed. The handling of the modifications end up in the completed state. Processing of read operations may end either in completed or in applied state. Only read operations issued in optimistic mode can be finished in applied state.

Contrary to synchronised mode, pessimistic immediate mode does not enforce execution of accepted modifications which may lead to delays. Instead the mode refers to the current state of the objects, and, in case there are uncommitted operations applied optimistically, they will be retracted. The key difference between synchronised and immediate pessimistic modes consists in handling of tentative operations. The synchronised mode waits for the current tentative operations to become committed and executed, while the immediate mode returns the last stable state, effectively ignoring new, tentative operations. Pessimistic immediate mode will be denoted by *pi* upper index.

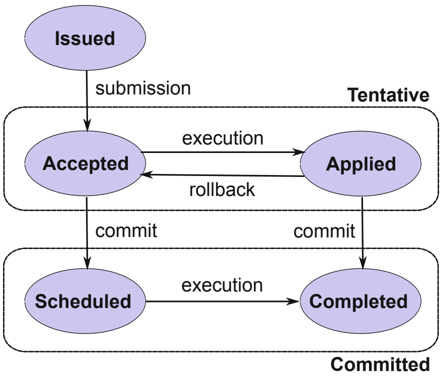


Fig. 1. States of modification processing

Table 1. Classification of read operations

	pessimistic	optimistic
synchronised	modifications accepted before the read must be committed	modifications accepted before the read must be applied
immediate	uncommitted operations must be retracted	the current state

Optimistic synchronised reads should be used for getting the best approximation of the replicated data based on the state of the replica being accesses without communicating with other replicas. It means that all accepted modifications will be applied even if they are not scheduled. This optimistic approach assumes that there are no other concurrent modifications that may interfere with the ones being executed without consulting other replicas. If this assumption is not fulfilled, the operations performed out of order will have to be retracted. The application must take into account that the data returned by the replica may be inconsistent. However, the replica will be able to generate a response even in case of communication failures which is the most important motivation for introducing this mode. Optimistic synchronised mode will be denoted by *os* upper index.

The last mode for read operations is optimistic immediate mode. In this case time plays the most important role: the application needs the data and needs it *now*. Therefore no new operations are applied, no operations are retracted: the replica returns the current state as it is. The difference between immediate and synchronised optimistic modes consists in the treatment of new accepted operations which are taken into account in synchronised mode and ignored in immediate mode. Optimistic immediate mode will be denoted by *oi* upper index.

Tab. 2 presents briefly execution modes of modifications. A pessimistic synchronous operation waits for the results of the execution, and after returning results it cannot be retracted. Therefore the operation must be committed, which means that it must be performed in cooperation with other replicas. Similarly to pessimistic synchronised reads, execution of the modification requires prior execution of all previously accepted modifications, which results from the assumption of globally consistent ordering of all modifications. Pessimistic synchronous mode will be denoted by *ps* upper index, e.g. $m^{ps}(a)$.

Table 2. Classification of modifications

	pessimistic	**optimistic**
synchronous	the modification must be committed	modifications accepted before the modification must be applied
asynchronous	modification is submitted and acknowledged	

Optimistic synchronous modifications enable applying updates even in case of communication problems between replicas. The operation will be executed on the local replica, without communicating with other replicas, but after executing all other previously accepted modifications. If there were no other conflicting modifications on other replicas the locally applied updates would become committed. However, the results will be available to the application before contacting other replicas. Optimistic synchronous mode will be denoted by *os* upper index.

Finally we have identified asynchronous mode for issuing modifications. As the name stands, such modifications are asynchronously submitted and queued for execution. The interaction with application ends before the modification completes. The system should try to optimise execution of this modifications by postponing it till it become scheduled after communication with other replicas in order to avoid execution of unordered updates. However, the modification may be executed before establishing final global ordering of updates if followed by an optimistic synchronised read or an optimistic synchronous modification. There are no differences between pessimistic asynchronous and optimistic asynchronous modes. Asynchronous mode will be denoted by a upper index.

It is worth noting that the execution of optimistic operations is not blocked by pessimistic ones. Preceding pessimistic operations may be executed tentatively without returning results to the issuing process (see Fig. 3). In fact the modes of operations focus on the view of states observed by processes rather than the states of replicas themselves.

Examples. This section covers examples of processing which illustrate differences and interactions between described operation modes. The differences result in various kinds of trade-offs between data consistency and availability, which is mostly visible when network partitions occur [4]. The following examples assume transient communication problems between replicas, which is indicated by a zigzag on the time-space diagrams between axes of replicas R_1 and R_2.

Let us consider two processes P_1 and P_2 accessing replicas R_1 and R_2 respectively as depicted in Fig. 2. At the beginning process P_1 issues an optimistic synchronous modification $m(1)$. This modification can be executed without communication with other replicas. After execution of the modification, a result is returned to the process. Then the process issues an optimistic synchronised read. Since all modifications known to the replica have already been applied, the read is executed immediately and a result is returned. Concurrently to process P_1 another process P_2 issues a pessimistic synchronous modification $m(2)$. Execution of this modification is blocked until the modification is committed. In order to commit the operation the replica must learn about all other preceding modifications, so it has to communicate with other replicas. In this and following examples it is assumed that $m(1)$ should precede $m(2)$, thus $m(2)$ is applied after execution of $m(1)$ at R_2, and then results are returned to process P_2. As can be seen optimistic operation completes quite quickly, without communicating with other replicas, while the pessimistic operation is blocked until communication is resumed.

Fig. 3 presents a slightly more complex case with additional process P_3 accessing replica R_2 concurrently with process P_2. As in the previous example process P_2 issues a pessimistic synchronous modification $m(2)$, and then process P_3 requests an optimistic synchronised read. The read forces execution of modification $m(2)$, but results of this modification are not returned to process P_2; the modification is applied tentatively and has not been committed yet. Next, replica R_2 — after resumption of communication — learns about modification $m(1)$ preceding $m(2)$, which causes withdrawal of $m(2)$. The modifications are applied in the final order, thus $m(2)$ becomes committed, and results of $m(2)$ are returned to process P_2. In this case we can see that a pessimistic operation does not observe a tentative state but at the same time an optimistic operation may reveal such state.

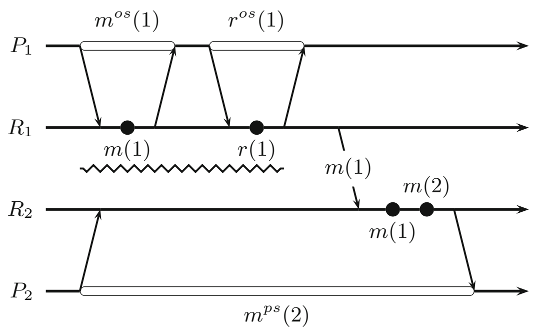


Fig. 2. Pessimistic and optimistic modifications in case of network partitioning

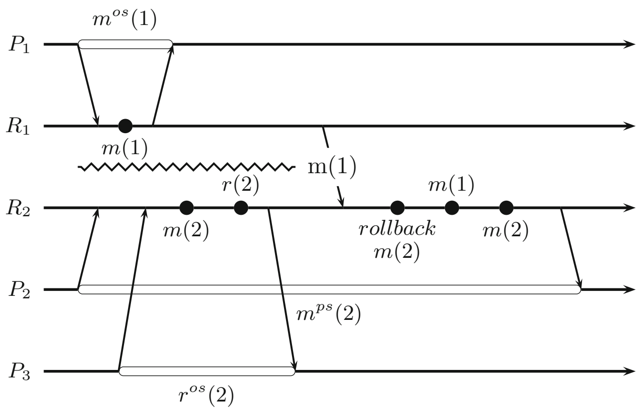


Fig. 3. Interleaving of optimistic synchronised read with pessimistic modification

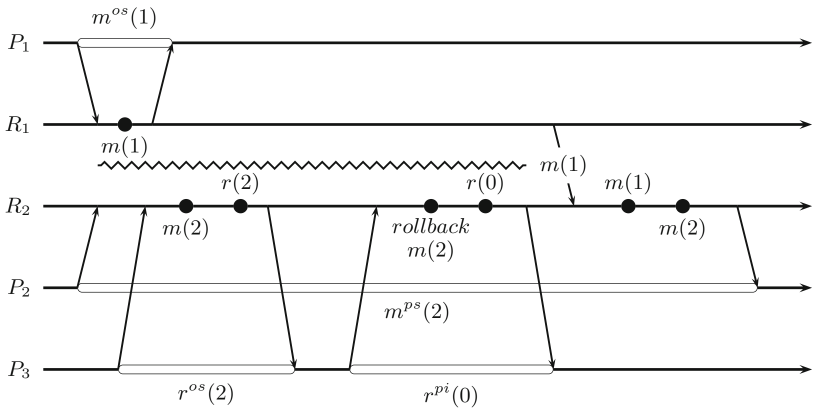


Fig. 4. Interfering of pessimistic immediate read and optimistic synchronised read

The last example depicted in Fig. 4 illustrates how operations may be executed tentatively and then reverted in order to satisfy contradicting requirements. As in the previous example process P_2 issues a pessimistic synchronous modification $m(2)$, and process P_3 issues an optimistic synchronised read after that, which causes tentative execution of modification $m(2)$ at R_2. Process P_3 gets a response to its read operation, and then it requests a pessimistic immediate read. This read operation forces withdrawal of applied and uncommitted modification $m(2)$. As a result the read operation returns the state prior to execution of $m(2)$. Later, modification $m(2)$ becomes committed after executing $m(1)$, and finally results are returned to process P_2.

4 Related Work

An overview of different optimistic replication approaches is presented in [6]. The main problems of optimistic replication are conflict resolution and ensuring eventual consistency of all replicas. We have suggested using rollback recovery as a mechanism of achieving eventual consistency. This approach is widely applied in many systems using optimistic replication, e.g.: [8], [5]. The approach proposed in [2] provides a client with multiple versions of data in response to read requests, and expects the client to reconcile them.

Conflict resolution usually requires additional information to be provided. Let us consider Bayou system which was designed for similar environment and therefore in many ways resembles the model considered in this paper. Bayou requires application developer to provide dependency checks in order to detect conflicts. If conflict is detected then merge procedure is started to resolve it. This approach requires application developer to consider possible conflicts and find proper solutions. Prerequisites for operation execution, similar to Bayou dependency checks, are also proposed in [5]: when conflict is detected transaction is omitted and issuing process is informed. We do not consider solving of conflicts. Inasmuch as there is no restriction on single modification complexity, we assume that all actions needed to resolve conflict are part of modification. Besides, we introduce pessimistic mode for critical operations in order to hide tentative states. In effect, our approach allows combining optimistic and pessimistic operations.

The idea of issuing operations in different modes appeared in hybrid consistency for distributed shared memory [3]. Essential to that approach is the distinction between strong and weak operations that apparently resemble pessimistic and optimistic ones. However, the dissimilarity between hybrid consistency and our model lies in the effect of operations on the system state. In our model, the effect is local — it concerns de facto the view of system state for the issuing process. The specification of operations in hybrid consistency determines the order of their execution on every replica, thereby influences the view for other processes.

5 Conclusions

In this paper, we have presented a concept of combining optimistic and pessimistic approaches to replication in a distributed system. Our solution provides simple way of expressing expected trade-offs between consistency and availability. We have introduced a classification of operations based on two orthogonal criteria. Generally we have distinguished between optimistic and pessimitic modes of operations. Additionally, in the case of reads we have distinguished synchronised and immediate mode, and in the case of modifications — synchronous and asynchronous mode. The modes of operations focus on the view of states observed by processes rather than the states of replicas themselves (the actual state of a replica results from the order of modifications execution). Thus the execution of optimistic operations is not blocked by pessimistic ones.

The proposed model best suits applications which accept tentative states still providing meaningful results for the users. Tangible benefits depend on the conflicts ratio and the requirements for pessimistic results. There are numerous systems that can be optimistically replicated where conflicts are rare and do not result in excessive rollback recovery. In such cases optimistic mode of operations provides improved availability, and does not introduce additional cost related to consistency management. The epitome of the application of optimistic replication is directory service. Since individual entries are rather independent of one another, their update can be usually performed optimistically. This is not the case of all modifications (e.g. new entry creation), however, in our model the problem can be avoided by means of pessimistic mode.

The concept presented in this paper has been materialised in a form of a replication protocol combining optimistic and pessimistic replication. This protocol is currently being implemented in the context of web services replication. Further investigation will be focused on evaluation of real applications using the presented model.

References

1. Birrell, A.D., Levin, R., Schroeder, M.D., Needham, R.M.: Grapevine: an exercise in distributed computing. Communications of the ACM 25(4), 260–274 (1982)
2. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. 41, 205–220 (2007)
3. Friedman, R.: Consistency Conditions for Distributed Shared Memories. Ph.D. thesis, Computer Science Department, Technion–Israel Institute of Technology (June 1994), `ftp://ftp.technion.ac.il/pub/supported/cs/thesis/roy_friedman.ps.Z`
4. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (2002)
5. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: SIGMOD 1996: Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data, pp. 173–182. ACM Press, New York (1996)

6. Saito, Y., Shapiro, M.: Optimistic replication. ACM Computing Surveys 37(1), 42–81 (2005)
7. Shapiro, M., Bhargavan, K.: The Actions-Constraints approach to replication: Definitions and proofs. Tech. Rep. MSR-TR-2004-14, Microsoft Research (March 2004)
8. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP), pp. 172–182. ACM Press (1995)
9. Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., Alonso, G.: Understanding replication in databases and distributed systems. In: Proc. of the 20th Int. Conf. on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan, R.O.C, pp. 464–474 (April 2000)

K-Resilient Session Guarantees Synchronization Protocol for Mobile Ad-Hoc Networks[*]

Jerzy Brzeziński, Dariusz Dwornikowski,
Łukasz Piątkowski, and Grzegorz Sobański

Institute of Computing Science,
Poznań University of Technology, Poland
{Jerzy.Brzezinski,Dariusz.Dwornikowski,
Lukasz.Piatkowski,Grzegorz.Sobanski}@cs.put.poznan.pl

Abstract. Session guarantees define data consistency in a distributed system as required by a single, mobile client. Consistency protocols of session guarantees are built from two components: one is aimed at providing safety (the *guarantees*) and the other at providing liveness (data synchronization). This paper presents a new k-resilient data synchronization protocol designed for mobile ad-hoc networks and other systems, where crash-stop failures occur. The application of the protocol is targeted at, but not limited to, various rescue and emergency operations.

1 Introduction

Nowadays, mobile wireless systems are rapidly gaining momentum, and every day new services are being delivered to clients. An important class of these systems are sensor and mobile ad-hoc networks (MANETs), where network nodes communicate by means of hop-by-hop packet forwarding. In some scenarios, mobile ad-hoc networks are often the only possible solution. This is the case, for example, in dynamic search, rescue or evacuation missions, where there is no time to deploy network infrastructure or the existing one may be damaged.

In such an application, services deployed in MANETs must be efficient but also available for mobile users. To meet these requirements, replication of services and their data on many nodes can be used. However, this approach leads to the replica consistency problem, among others [14,6].

This paper deals with a replica consistency model called *session guarantees* or *client centric* consistency [15], because it is especially suited for mobile systems and MANETs. In this approach, replica consistency is considered from a point of view of a single mobile and roaming client instead of a single replica, as in data centric models. In other words, the order of operations performed on replicas in the system is not important as long as the single client perceives its own operations in a required order. The system consists of servers, holding data

[*] This work has been partially supported by the Polish Ministry of Science and Higher Education within the European Regional Development Fund, Grant No. POIG.01.03.01-00-008/08 IT-SOA https://www.soa.edu.pl/

object replicas, and clients, accessing those replicas. To keep data replicas consistent, two protocols are needed: a coherence protocol, defined between clients and servers, and a synchronization protocol, defined between servers in order to propagate operations performed at one replica to the others.

This work focuses on synchronization protocols. According to our best knowledge, the solutions proposed so far have not been well suited for mobile environments. One of the first papers describing session guarantees and synchronization protocol are [15] and [13], but the problem of the synchronization protocol was considered orthogonal to the main task there. Full solution to the problem of server synchronization was introduced in [10]. Unfortunately, none of these synchronization protocols is suitable for mobile ad-hoc networks, as they assume a reliable environment. In MANETs there exist problems like unreliable message passing and network partitioning causing servers unavailability, thus requiring a dedicated solution. Some work considering unreliable environment was introduced in [7], but it still does not fit mobile systems, as it focuses only on server's recovery.

The aim of this work is to present a new synchronization protocol called SGASP – *Session Guarantees Ad-hoc Synchronization Protocol.* This protocol is suited for MANETs as it allows for errors like message retransmissions and failures. However, what is most important, it is k-resilient. SGASP assures that every client operation is known to at least k servers before the operation is returned to the client as successfully executed. This way SGASP is better suited for wireless systems, because it increases the overall availability of operations and services. What is important to note, those additional properties were achieved without using any communication primitives other than ordinary unicast and broadcast transmissions. According to our best knowledge, no k-resilient synchronization protocol was proposed earlier.

Protocols aimed at MANETs must cope with a whole set of problems, including temporary lack of communication, network partitioning and other MANET–related problems. These problems result in important consequences. Particularly, one server can accept a write operation from a client and then, immediately split into a one-node partition. The client then, can switch to another one and request replica state including the last write operation. Since the only server with that operation is gone, there is no way the new server or the client can get the required operation. In MANETs some kind of a routing protocol [12] is often run by the nodes. Since SGASP does not make any assumptions about other nodes' reachability, routing protocol is not required, although its presence may have an impact on performance of the protocol.

Despite these problems, real-life examples of application of session guarantees in MANETs exist. Let us imagine a rescue operation led by firefighters. On a flood site, the whole network infrastructure was destroyed, yet computer network can be used to coordinate and support rescue operation. Each of the fire engines on the site can form an ad-hoc network node with medium to large wireless network range. Highly mobile forces, like human squads or manned and unmanned robots, can be equipped with short to medium range wireless interfaces, extending the ad-hoc network's reach. In such a network a distributed access to remote data objects is crucial. Commanding officers need to have one log file to enable post-action analysis, all units should have best knowledge of a global state of other units' positions and sensors readings. Such a case is a real world scenario, one of many a system like Proteus project [11] is designed for. Currently, many operations, like the one mentioned above, are led and coordinated using only personal radios for direct voice communication.

2 The Session Guarantees Environment

The model of the system with session guarantees consists of two major types of participants: clients and servers. Every server S holds a full copy of the set of all objects O in the system. Clients C interact with the objects by requesting read or write operation on element O from one of servers S. Client's requirements regarding data consistency are sent to a server using a coherence protocol. This way client C can enforce the execution of locally known operations on server S. Consistency protocols use vector clocks [9,1], also called version vectors, to keep track of write operations required by a client. Those version vectors are a representation of theoretically unbounded write sets [13], which consist of all operations required by a client since the start of the consistency protocol.

Using the concept of version vectors, four types of session guarantees have been defined [15,13]. A client can request a server to fulfill one or more of them. The four guarantees are: Read Your Writes (RYW), Monotonic Writes (MW), Monotonic Reads (MR) and Writes Follow Reads (WFR) [15]. To provide particular guarantees, a server needs to know which operations are known to a client and in what order. These operations are represented as version vectors and sent to the server. A detailed description and complete formal definitions of version vector representation of required write sets, session guarantees and client protocol can be found in [15,13].

It is important to note that differences between data centric and session guarantees approaches are essential and clearly seen when we compare them. The dissimilarity is caused by a very different client cooperation and consistency models properties. Thus, trying to achieve strong data-centric consistency using session guarantees is hard or even impossible [2,3]. Also trying to achieve client centric consistency using data centric approach essentially requires usage of atomic consistency, which is very inefficient [10].

3 The SGASP Protocol

3.1 Environment Assumptions

SGASP assumes that no other communication primitives except for standard IP-like unicast and broadcast transmission, are needed. Both of them may fail to deliver a message at any time. The only additional condition is that every response message sent from a server will eventually reach the client that sent the request. The justification for this assumption is described in Section 3.3.

3.2 Data Structures

In earlier work, server version vectors [8] were used to effectively represent sets of required operations. Every element i in a version vector denoted a local view of a number of write operations performed by a server S_i. In SGASP the concept of version vectors is enhanced by a *Label* which consists of two version vectors of the same length. The first is called *timestamp* and it possesses the same semantics as in earlier protocols, except that now every position of that vector denotes a local view of a number of times a given server has tried to persist an operation (see 3.3). The second vector is called *minstate.* It describes a minimal server version vector required to execute the operation or proposition described by a label. A sample label has a form $\begin{matrix}[1\,0\,0]\\ [0\,0\,0]\end{matrix}$, where $[1\,0\,0]$ corresponds to a timestamp and $[0\,0\,0]$ to a minstate.

3.3 General Idea

The general idea of SGASP is to replicate write operations requested by clients on more than one server before executing them. Replicated operations are stored in a *Cache* set. However, they are not executed on these additional servers until they are needed by some server, especially the local one. Every operation is replicated on n server nodes, therefore even after failure of $k = n - 1$ nodes, it is still possible to find every operation. This way SGASP can counteract against network partitioning and node failures in MANETs.

SGASP is based on ODSAP protocol [10] which behavior upon receiving client's request can be loosely described by the following steps:

1. Check if the client's requirements from the request are met. If yes, go to 4.
2. Broadcast synchronization request to other servers asking for operations required to satisfy the request; wait for answers.
3. If step 2 fails, return an error message to the client and end processing.
4. Perform the operation and return a success message to the client.

The SGASP protocol works on every server in the following steps for each client request:

1. Check if the client's requirements included in the request are met or can be satisfied using operations from Cache set. If yes, go to 4.
2. Broadcast synchronization request to other servers asking for operations required to satisfy the request; wait for the answers.
3. If step 2 fails, return the error message to the client and end processing.
4. Try to persist (place a copy of the request) operation on at least n other servers by broadcasting it; wait for acknowledgements.
5. If step 4 fails, return the error message to the client and end processing.
6. Perform the operation and return a success message to the client.

3.4 Protocol Description and Examples

Every SGASP server, after receiving a request, first tries to execute all operations needed to satisfy client's requirements. Then, it tries to replicate the client write operation on at least n nodes, including itself. Only if the operation is successfully persisted, is it executed and a successful response is sent back to the client. Otherwise, the client is informed about the failure of its request. If needed, the replication step can be retried by the server after a timeout. The incoming replicated write operations are stored by every server in a *Cache* set.

If a server receives a synchronization request from another server, it replies with all locally known operations which are unknown to the requester. This synchronization step works as in ODSAP protocol [10]. If a persistence message is received, the attached operation is placed in the *Cache* set and an acknowledge message is sent in reply.

SGASP uses fairly weak communication mechanisms. As such it needs to extensively exploit labels' properties to assure correctness of the synchronization protocol. Every server S that needs to replicate (persist) a client's write request broadcasts it in a *Propose* message. Servers that receive the message cannot consider the write request a valid operation, as they don't know if the sender of the *Propose* message will get at least n acknowledgment messages from other servers. Thus, the data received in the *Propose* message is considered a *proposition*, which may or may not become a valid operation. Therefore, propositions cannot be safely executed by servers. If server S receives at least n *Ack* messages, it knows that the proposition has reached at least n servers and it is safe to execute it and respond to the client. Only then the proposition becomes an *operation*. On the other hand, if server S does not receive n *Acks*, the client's request remains forever a proposition. The important property is that labels provide sufficient information to distinguish valid operations and propositions.

In earlier protocols labels were not needed, as the relation of vector domination [10] was sufficient to achieve client centric consistency. In SGASP timestamp vector in a label is incremented every time a proposition is sent, therefore, some of its values may represent only propositions, not actual operations. Thus, the introduction of the minstate vector was necessary. Minstate shows what the last value of a version vector of the server trying to persist a client request was. This value is always assigned to the last valid (successfully persisted) operation. The difference between values of the timestamp and the minstate vector is used to distinguish between operations and propositions.

Labels are attached to every proposition and operation but they are used only in a synchronization protocol, not in a coherence protocol which remains the same as in the earlier work. Nevertheless, client's requests analysis is required to further exploit labels. A client never gets to know a version vector of a proposition. If the client receives a successful response to its request, it implies that an operation is already persisted and thus, it is not a proposition. Clients are the only source of requests received by servers. Therefore, version vectors received from clients always describe an operation, not a proposition. Using this property, a server can invalidate some propositions in it's *Cache* set and find other valid operations needed to fulfill client's requirements. Examples exploiting labels properties are shown below.

Example 1. Simple usage of labels

Let's assume we have two clients C_1 and C_2, three servers $S_1 \dots S_3$, the resilience factor $k = 1$ ($n = 2$) and all initial vector values are zero. Now, client C_1 sends a write request w_1 with vector [0 0 0] to server S_1. S_1 has a version vector state [0 0 0] that dominates client's request vector, so it tries to persist w_1. It attaches a label $\genfrac{}{}{0pt}{}{[1\,0\,0]}{[0\,0\,0]}$ to the request and broadcasts it in a *Propose* message p_1 to all servers. The label can be read as follows:

- the proposition is sent by server S_1, as the vectors differ on index 1,
- it is the first proposition sent by S_1, as the timestamp vector on index 1 is 1,
- this is the first proposition sent by server S_1 for a given client's request, as the two vectors differ by 1 on index 1,
- if this proposition becomes an operation, it will be sufficient that other servers perform the state [0 0 0].

If the *Propose* message is received by other servers, the operation w_1 is added to their *Cache* sets. If S_1 receives at least k *Ack* messages, it knows that the write replication was successful, it executes w_1 and sends a reply to the client. Otherwise, it retries to persist w_1 and sends a new proposition p_2 with a new label $\genfrac{}{}{0pt}{}{[2\,0\,0]}{[0\,0\,0]}$. As the minstate did not change, that label means it is a second trial of S_1 to persist the same client operation. If w_1 was persisted earlier, its timestamp would become a new S_1's state, so a proposition for some new operation would have updated minstate. Because that is not the case, SGASP knows it is a new proposition sent for the same operation w_1. What's more, all servers that receive both p_1 and p_2 can remove p_1 from their *Cache* sets, as it has been outdated by p_2. Now if S_1 receives *Ack* message for p_2 from the other server, it knows that there are at least $n = 2$ copies of w_1 and can successfully complete client's request. The new state of S_1 is [2 0 0], what represents a valid operation and can be safely sent in response to the client. It is also not a problem, if p_2 does not reach a server that received p_1. In this case, p_1 can be removed from *Cache* under one of the following conditions:

- A client, in our example C_1, will send a new request to that server with vector [2 0 0]. The new server has the state [2 0 0] but does not have operation with timestamp [2 0 0] in its Cache, so it has to search for the missing operation on other servers. After finding the missing operation, it sees that operation's label is $\genfrac{}{}{0pt}{}{[2\,0\,0]}{[0\,0\,0]}$, and thus, p_1 can be removed.
- Server reaches a state that dominates p_1 — if it was possible without executing p_1, would be only a proposition, not an operation. □

Example 2. A complete use case of SGASP

The situation in Figure 1 shows a more complicated scenario. Each server consists of *Cache* and *History* sets which contain requests with attached labels, the *State* column, which shows changes in server version vector value, and the *Log* column, which shows algorithm steps performed by the server. The letter "E" means local execution of an operation with the given label, "S" sending a new proposition with label, and "Y/N" right of the label value shows whether the persistence attempt was successful. The "SyncReq" entry shows that a server sent a synchronization request for missing operations. Arcs above servers show operations the server received in response. In this scenario, Client 1 with vector [2 0 0 0] and Client 2 with [0 0 2 3] have already executed some operations on Server 1, 3 and 4. The situation gets complicated when both clients meet on Server 2. Request of Client 1 is served first, gets a label $\begin{matrix}[2\,2\,0\,0]\\ [2\,0\,0\,0]\end{matrix}$ and sets server state to [2 2 0 0]. Now the request from Client 2 arrives, which forces synchronization of Server 2 with others. After executing required operations Server 2 has version vector state [2 2 2 3] and is ready to persist client's request. The persistence step is successful at the first attempt, giving the request a label $\begin{matrix}[2\,3\,2\,3]\\ [2\,2\,2\,3]\end{matrix}$ and setting server's vector to [2 3 2 3]. That version vector is also sent in response to Client 2 and it represents results of two operations originating at different servers. □

The situation presented in Figure 1 can be further extended, resulting in a directed acyclic graph of operation dependencies. One of the major tasks of SGASP is to construct that graph using operations from Cache and History sets and the knowledge derived from client's requests, which always represents a composition of timestamps of valid operations. We can find these operations by taking into account the following facts:

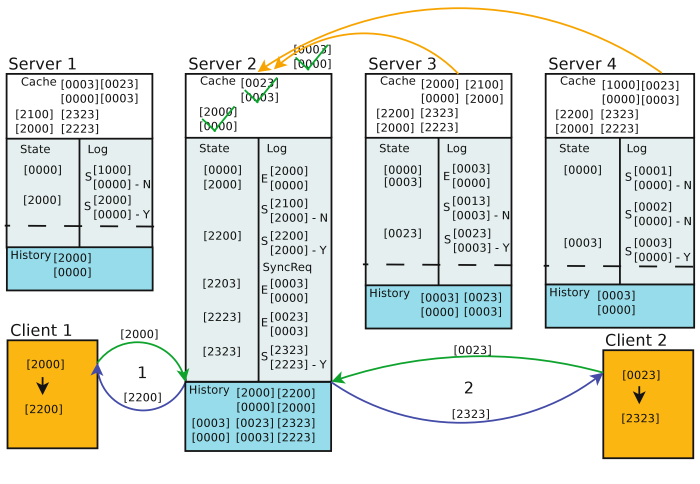


Fig. 1. Sample execution of SGASP

- client's version vector always results from a timestamp of a valid operation, or a composition of two operations,
- for each operation the originating server is known from its label,
- every position in a client's vector must come from an operation's timestamp.

In MANET it is assumed that every message transmitted on the network may be lost. The fact has been taken into consideration in SGASP but it has been assumed that server's response is never lost while being transmitted to the client. This limitation needs to be addressed as a separate problem in future work.

4 Simulation Evaluation of SGASP

The SGASP protocol was implemented and evaluated using OMNeT++ [5] simulation framework with inet–manet extension. The simulation model used was very exact. The whole environment consisting of servers and clients moving on a rectangular playground was modeled. Every network node in the simulation consisted of a mobility model and an exact full TCP/IP stack implementation. The stack includes all lower network layers, simulated with hardware radio device compatible with 802.11g, signal loss model, full MAC, IP and UDP layers. On the top of the UDP layers, the SGASP service was implemented. In order to allow a comparison of SGASP with other solutions, an ODSAP protocol was implemented and evaluated using the same parameters and environment.

While constructing the simulation, some additional assumptions were made:

1. If there is more than one server in the communication range of a client, the client uses a separate procedure to select a server. Every client keeps a list of servers, which did not reply last time to its request and chooses the closest not listed server.
2. Every server was modified to limit the length of incoming messages and requests awaiting processing. Because clients need to implement a timeout mechanism, it is useless to keep all awaiting requests and messages, as they will be processed too late, after a timeout occurs on the sender.

4.1 Simulation Parameters

In order to carry out simulation experiments, simulation parameters had to be chosen. Their values were based on typical hardware parameters as measured on real system, and in some cases, on test runs of simulation. The more important values used in the simulation were set as below:

- simulation time: 12h of virtual time,
- mobility model: mass mobility [4] with change interval randomized from normal distribution from range $(1s, 10s)$, angle change from normal distribution $(0°, 30°)$ and speed between 5 km/h and 25 km/h,
- size of the playground: 500 m x 500 m,
- wireless interface type: IEEE 802.11b with radio bit-rate 21 Mb/s and range of approximately 140 m,
- no dedicated MANET routing protocol used, broadcasts are implemented as delivering the message to all other nodes directly in range.

4.2 Basic Simulation Results

To compare the performance of protocols a measure of write success ratio was introduced. That measure is computed for every client at the end of a simulation run as a ratio of successfully completed requests to all issued requests.

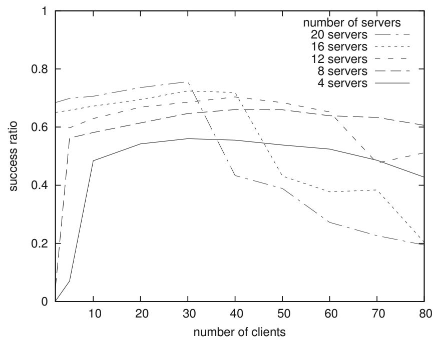


Fig. 2. Write success ratio of SGASP protocol

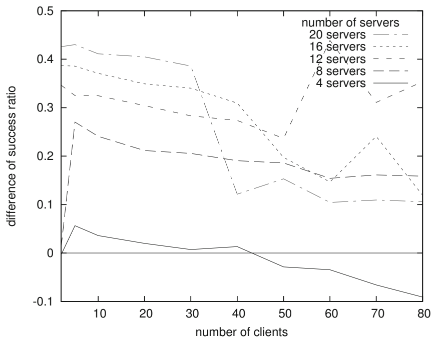


Fig. 3. Difference between write success ratio of SGASP and ODSAP protocols

Figure 2 presents how the write success ratio changes for the SGASP protocol. The chart shows that the number of servers is a critical parameter in the system and must be carefully adjusted to the number of clients. Then, a write success ratio of $\sim 80\%$ can be expected.

The comparison of SGASP and ODSAP protocols is depicted in Figure 3. The figure shows the difference between write success ratio of SGASP and ODSAP protocols in the same scenarios. Positive values show that SGASP performs better than ODSAP. It is important to note that there are only a few cases where SGASP is slightly less effective than ODSAP, while its advantage in some scenarios is up to 60 percentage. This shows that SGASP protocol is not only k-resilient but also increases performance of the system.

5 Conclusions

In this paper, a new k-resilient server synchronization protocol for session guarantees has been introduced. The protocol is dedicated to work in a mobile ad-hoc network environment. Both general idea and basic mechanisms of the protocol have been provided. The main advantage of the new protocol is a new write operation replication scheme and label based operation handling. In order to evaluate SGASP's performance and compare it with the older ODSAP protocol, both of the protocols were simulated and compared using write request success ratio measure. As it came out, SGASP does not only provide better crash resilience than ODSAP but also performs better in majority of scenarios.

Of course, there are still some important problems to address. The first one is the requirement for a client to stay in a radio range of the server until completing a request. The other is throughout performance evaluation and comparison, especially using MANET routing protocols and simulated server crashes.

References

1. Baldoni, R., Raynal, M.: Fundamentals of distributed computing: A practical tour of vector clock systems. IEEE Distributed Systems Online 3(2) (February 2002), http://dsonline.computer.org/0202/features/bal.htm
2. Brzeziński, J., Sobaniec, C., Wawrzyniak, D.: Session Guarantees to Achieve PRAM Consistency of Replicated Shared Objects. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2004. LNCS, vol. 3019, pp. 1–8. Springer, Heidelberg (2004)
3. Brzeziński, J., Sobaniec, C., Wawrzyniak, D.: From session causality to causal consistency. In: Proc. of the 12th Euromicro Conf. on Parallel, Distributed and Network-Based Processing (PDP 2004), A Coruña, Spain, pp. 152–158 (February 2004)
4. Camp, T., Boleng, J., Davies, V.: A survey of mobility models for ad hoc network research. Wireless Communication & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications 2(5), 488–502 (2002)
5. Community, O.: http://www.omnetpp.org/
6. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
7. Kobusińska, A.: Rollback-Recovery Protocols for Distributed Mobile Systems Providing Session Guarantees. Ph.D. thesis, Institute of Computing Science, Poznan University of Technology (September 2006)
8. Kobusińska, A., Libuda, M., Sobaniec, C., Wawrzyniak, D.: Version vector protocols implementing session guarantees. In: Proc. of Int. Symp. on Cluster Computing and the Grid (CCGrid 2005), Cardiff, UK, pp. 929–936 (May 2005)
9. Mattern, F.: Virtual time and global states of distributed systems. In: Cosnard, Quinton, Raynal, Robert (eds.) Proc. of the Int. Conf. on Parallel and Distributed Algorithms, pp. 215–226. Elsevier Science Publishers B. V. (October 1988)
10. Piątkowski, L., Sobaniec, C., Sobański, G.: On-demand server synchronization protocols of session guarantees. Foundations of Computing and Decision Sciences 35(4), 307–324 (2010)
11. Proteus, http://www.projektproteus.pl/
12. Royer, E.M., Toh, C.K.: A review of current routing protocols for ad hoc mobile wireless networks. IEEE Personal Communications (April 1999)
13. Sobaniec, C.: Consistency Protocols of Session Guarantees in Distributed Mobile Systems. Ph.D. thesis, Poznań University of Technology, Poznań (September 2005)
14. Tanenbaum, A.S., van Steen, M.: Distributed Systems — Principles and Paradigms. Prentice Hall, New Jersey (2002)
15. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M., Theimer, M., Welch, B.W.: Session guarantees for weakly consistent replicated data. In: Proc. of the 3rd Int. Conf. on Parallel and Distributed Information Systems (PDIS 1994), pp. 140–149. IEEE Computer Society, Austin (1994)

On Time Constraints of Reliable Broadcast Protocols for Ad Hoc Networks with the Liveness Property*

Jerzy Brzeziński, Michał Kalewski, and Dariusz Wawrzyniak

Institute of Computing Science,
Poznań University of Technology,
Piotrowo 2, 60–965 Poznań, Poland
Jerzy.Brzezinski@put.poznan.pl,
{Michal.Kalewski,Dariusz.Wawrzyniak}@cs.put.poznan.pl

Abstract. In this paper we consider a formal model of ad hoc systems and its liveness property, defined with the use of the concept of dynamic sets. In this context we analyse reliable broadcast protocols dedicated for use in this kind of networks. In solutions proposed till now it is assumed that the minimum time of direct connectivity between any neighbouring nodes is much longer than maximum message transmission time. This assumption covers, however, dependence of the required minimum time of direct communication on some system parameters. Therefore, in this paper we show precisely how the minimum time of direct connectivity depends on the total number of hosts in a network and on the total number of messages that can be disseminated by each node concurrently.

1 Introduction

Mobile ad hoc networks (MANETs) [1] are composed of autonomous and mobile hosts (or communications devices) which communicate through wireless links. Each pair of such devices, whose distance is less than their transmission range, can communicate directly with each other—a message sent by any host may be received by all the hosts in its vicinity. If hosts in MANET function as both a computing host and a router, they form a multiple hop ad hoc network. Hosts can come and go or appear in new places, so with an ad hoc network, the topology may be changing all the time and can get partitioned and reconnected in a highly unpredictable manner.

One of the fundamental communication operation in ad hoc networks is broadcast—the process of sending a message from one host to all hosts in a network. It is important for any broadcast protocol to provide some delivery guarantee beyond “best-effort”, but in case of dynamic environments this can be hard or even impossible to achieve [3]. The highly dynamic network topologies

* 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.

with partitioning and limited resources are the reasons why heuristic broadcast protocols with only probabilistic guarantees have been mainly proposed for use in MANETs. On the other hand, if it can be assumed that a group of collaborating nodes in an ad hoc network can be partitioned and that partitions heal eventually, it is possible to develop reliable dissemination (broadcast) protocols. The assumption, that any partition is not allowed to be permanently isolated, is called *MANET Liveness Property* [6–8].

In this paper we strictly define a formal model of ad hoc systems and the liveness property with the use of the concept of dynamic sets. In this context we analyse reliable broadcast protocols dedicated for use in this kind of networks. In solutions proposed till now it is assumed that the minimum time of direct connectivity between any neighbouring nodes is much longer than maximum message transmission time. This assumption covers, however, dependence of the required minimum time of direct communication on some system parameters. Therefore, in this paper we show precisely how the minimum time of direct connectivity depends on the total number of hosts in a network and on the total number of messages that can be disseminated by each node concurrently.

The paper is organised as follows. First, following [7, 8], the formal model of ad hoc systems with the liveness property is defined in Section 2. The broadcast protocols proposed in [7, 8] are presented in Section 3. Section 4 contains the analysis of time constraints of the broadcast protocols with respect to the liveness property. Finally the paper is shortly concluded in Section 5.

2 System Model

In this paper a *distributed ad hoc system* is considered. The units that are able to perform computation in the system are abstracted through the notion of a *node*, and a *link* abstraction is used to represent the communication facilities of the system. It is presumed that each link always connects two nodes in a bidirectional manner. Thereby, the topology of the distributed ad hoc system is modelled by an undirected *connectivity graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of all nodes, $p_1, p_2, \ldots, p_n$, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of links between neighbouring nodes. If a node p_i is able to communicate directly with a node p_j, then there exists a link (p_i, p_j) in the set $\mathcal{E}$. (Note that (p_i, p_j) and (p_j, p_i) denote the same link, since links are always bidirectional.) The set $\mathcal{E}$ changes with time, and thus the graph $\mathcal{G}$ can get disconnected and reconnected. Disconnection fragments the graph into isolated sub-graphs called *components* (or *partitions* of the network), such that there is a path in $\mathcal{E}$ for any two nodes in the same component, but there is not a path in $\mathcal{E}$ for any two nodes in different components.

2.1 Nodes and Communication

It is presumed that the system is composed of $N = |\mathcal{V}|$ uniquely identified nodes and each node is aware of the number of all nodes in the $\mathcal{V}$ set (that is of N). A node is either *correct* or *faulty*: a faulty node can crash (stops its processing) at any moment without any warning (*silent crash failure model*, [2]), while a correct node does not crash until processing ends. It is not assumed that a crash can be detectable with certainty. Moreover, it is also presumed that the number of faulty nodes is bound to some known value f, such that: $0 \leqslant f < \frac{1}{2}N$. Thus, the $\mathcal{V}$ set contains at least $N - f > \frac{1}{2}N$ correct nodes. A node that has not crashed is said to be *operative*.

The nodes communicate with each other only by sending messages (*message passing*). Any node, at any time can initiate the dissemination of a message m, and all nodes that are neighbours of the sender at least for the duration of a message transmission, can receive the message. More formally, the links can be described using the concept of a *dynamic set* function [5]. Let $\mathcal{E}'$ be the product set of $\mathcal{V}$: $\mathcal{E}' = \mathcal{V} \times \mathcal{V}$, and $\Gamma(\mathcal{E}')$ be the set of all subsets (power set) of $\mathcal{E}'$: $\Gamma(\mathcal{E}') = \{\mathcal{A} \mid \mathcal{A} \subseteq \mathcal{E}'\}$. Then, the dynamic set $\mathcal{E}_i$ of a node p_i is defined as follows:

Definition 1 (Dynamic Set). *The* ***dynamic set*** $\mathcal{E}_i$ *of a node* p_i *of elements from* $\mathcal{E}'$ *on some time interval* $T = [t_1, t_2]$ *is a function:*

$$\mathcal{E}_i \colon T \to \Gamma(\mathcal{E}')$$

$$\textit{such that: } \forall t \in T \; (\, \mathcal{E}_i(t) \textit{ is a set of all links of } p_i \textit{ at time } t\,).$$

Let δ be the maximum message transmission time between neighbouring nodes. Then, we introduce the abstraction of a *reliable channel*, as presented by Module 1. The interface of this module consists of two events: a *request event*, used to send a message, and an *indication event*, used to deliver the message. Reliable channels do not alter and lose (property **RC1**), duplicate (**RC2**), or create (**RC3**) messages.

Events:

Request: $\langle$ *rc.Send*, m $\rangle$: Used to send a message m.

Indication: $\langle$ *rc.Deliver*, p_s, m $\rangle$: Used to deliver the message m sent by the process p_s.

Properties:

RC1 (*Reliable Delivery*): Let p_s and p_d be any two nodes that are in wireless range of each other, and let p_s sends a message m at time t. If the two nodes remain operative at least until $t + \delta$, then the message m is delivered by p_d within δ.

RC2 (*No duplication*): No message is delivered by any node more than once.

RC3 (*No creation*): If a message m is delivered by some node p_d, then m was previously sent by p_s.

Module 1. Interface and properties of reliable channels

Finally, we can define *direct connectivity* as follows ([8, 7]):

Definition 2 (Direct Connectivity). *Let* $T = [t, t + B]$, *where* $B \gg \delta$ *is an application-specified parameter. Then, two operative nodes* p_i *and* p_j *are said to be* ***directly connected*** *iff:*

$$\forall t \in T \; (\, (p_i, p_j) \in \mathcal{E}_i(t)\,).$$

It is assumed that channels between directly connected hosts are *reliable channels.*

2.2 Network Liveness Requirement

Let $\mathcal{O}$ be the set of all operative nodes of $\mathcal{V}$ at some time t ($\mathcal{O} \subseteq \mathcal{V}$). Let $\mathcal{P}$ be a non-empty subset of $\mathcal{O}$, and $\overline{\mathcal{P}}$ be its complementary set in $\mathcal{O}$ ($\overline{\mathcal{P}}$ contains all operative nodes at time t that are not in $\mathcal{P}$). Then, the *network liveness requirement* is specified as follows ([8, 7]):

Definition 3 (Network Liveness Requirement). *A distributed ad hoc system that was initiated at t_0 satisfies the **network liveness requirement**, iff:*

$$\forall t \geqslant t_0 \; \forall \mathcal{P} \; \exists I \geqslant B \; (I \neq \infty \;\wedge\; \exists \{p_i,\, p_j\} \; (p_i \in \mathcal{P} \;\wedge\; p_j \in \overline{\mathcal{P}} \;\wedge\; (\textit{nodes } p_i \textit{ and } p_j \textit{ are directly connected within } T = [t,\, t+I]))).$$

In other words:

$$\forall t \geqslant t_0 \; \forall \mathcal{P} \; \exists I \geqslant B \; (I \neq \infty \;\wedge\; \exists \{p_i,\, p_j\} \; (p_i \in \mathcal{P} \;\wedge\; p_j \in \overline{\mathcal{P}} \;\wedge\; (\exists \{t_1,\, t_2\} \; ((t \leqslant t_1 < t_2 \leqslant t + I) \;\wedge\; (t_2 - t_1 \geqslant B) \;\wedge\; (\forall t_c \in [t_1,\, t_2] \; ((p_i,\, p_j) \in \mathcal{E}_i(t_c))))))).$$

Informally, the network liveness requirement disallows permanent partitioning to occur by requiring that reliable direct connectivity must emerge between some nodes of $\mathcal{P}$ and $\overline{\mathcal{P}}$ within some arbitrary, but unknown, amount of time I after each t.

3 Crash-Tolerant Broadcast Protocols

Broadcast protocols enable us to send a message from one host to all hosts in a network, and are a basis of communication in ad hoc networks. It is important for any broadcast protocol to provide some delivery guarantee, especially if host failures are taken into account. The properties of broadcast operations considered in this paper are described by Module 2. The interface of this module consists of two events: a *request event*, used to broadcast a message, and an *indication event*, used to deliver the broadcast message. The **BCAST1** property ensures that at least $N - f - 1$ hosts will receive each disseminating message, and **BCAST2** ensures that every node eventually discards every disseminating message and, as a result, the broadcast of every message can be eventually *terminated.* The **BCAST2** property is called *Storage Subsidence Property* and defined as follows ([8, 7]):

Definition 4 (Storage Subsidence Property). *A broadcast protocol satisfies the **storage subsidence property** if there is a finite time after which nodes permanently stop retaining the broadcast message.*

Events:

Request: $\langle$ *bcast.Broadcast*, m $\rangle$: Used to broadcast a message m.

Indication: $\langle$ *bcast.Deliver*, p_s, m $\rangle$: Used to deliver the message m broadcast by the process p_s.

Properties:

BCAST1 (*Reliable Delivery*): For a broadcast of a message m initiated at time t_b, at least $N - f - 1$ nodes (or $N - f$ including originator) receive the message within some bounded time, if the sender does not crash, or if the sender crashes and a correct node receives m.

BCAST2 (*Transmission Termination*): Each node that receives m and remains operative, including originator, discards m and stops transmitting any packets concerning the broadcast of m at some time after t_b.

Module 2. Interface and properties of the broadcast protocols

Finally, the value $N - f - 1$ in the **BCAST1** property is a consequence of the following impossibility result (the theorem is defined and proven in [7], page 14): it is impossible for any crash-tolerant reliable dissemination protocol that satisfies the storage subsidence property to guarantee that more that $N-f$ nodes (including the originator) receive a message originated by a correct node even if less than f nodes crash before the termination of the message.

We also observe that the properties **BCAST1** and **BCAST2** differ from the properties of *best-effort*, *regular reliable* and *uniform reliable* broadcasts [4], since **BCAST1** property only guarantees that at least a subset of any $N-f$ operative nodes receive every message. So, in the worst case scenario if there are f faulty nodes among the $N-f$ nodes that have received m, and if each of them crashes, eventually only $N - 2f$ operative nodes have m.

The protocols presented in [8, 7], which implement Module 2, contain four solutions: (i) *Proactive Dissemination Protocol* (PDP), (ii) *Reactive Dissemination Protocol* (RDP), (iii) *Proactive Knowledge and Reactive Message* (PKRM) and (iv) *Optimised PKRM* (PKRM$_O$).

The simplest protocol of the four, *Proactive Dissemination Protocol* (PDP), requires that each node, which has a message m, transmits it once every β seconds. The PDP protocol operates in the following manner. The originator p_i of a message m initialises a vector $K_i(m)$, as a boolean vector of N bits, to all zeros, and sets its own bit ($K_i(m)[i]$) to 1. The vector indicates the *knowledge* of the node on the propagation of m, i.e. $K_i(m)[j] = 1$ means the node p_i knows that the message m has been received by the node p_j. The message is transmitted along with the $K_i(m)$ vector. When some node p_j receives m for the first time, it initialises its $K_j(m)$ vector to be equal to the received $K_i(m)$ and sets its own bit $K_j(m)[j]$ to 1. The message is also added to the $\mathcal{U}nrealised$ set. Then, whenever the host receives the message, it merges the received vector with its own. If the node p_i has $N - f$ or more 1-bits in its $K_i(m)$ vector, it *realises* m, i.e. it cancels the periodic transmission of m and moves the message to the $\mathcal{R}ealised$ set. If a node receives a message which has been realised, then within β seconds it transmits a special realisation packet $realise_i(m)$, and a node which receives a realisation packet, realises m.

The β parameter is originally defined as a configurable parameter such that $B \geqslant 2(\beta + \delta)$, to ensure that a node both: receives a message and responds to that message during direct connectivity. This is illustrated in Figure 1, where $\beta = 4\delta$ and $B = 10\delta$. Two nodes, p_i and p_j, experience direct connectivity, but just after p_i has started sending a message (first arrow in the figure). Thus, the message cannot be received by p_j at this time. The message is sent again after next β seconds and received by p_j after δ (second arrow). Finally, the message is transmitted by p_j after next β seconds (counted by p_j), and received by p_i after δ.

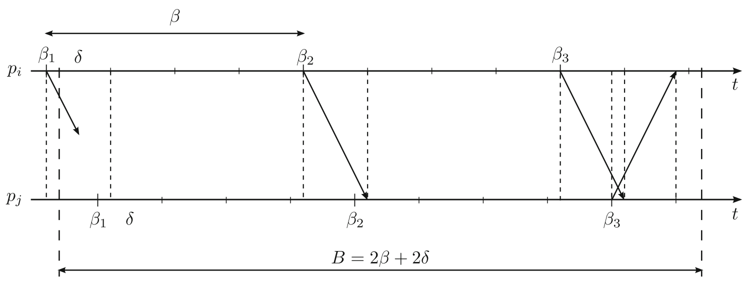


Fig. 1. An example of direct connectivity between two nodes ($\beta = 4\delta$, $B = 10\delta$ and $\forall_{i>1} : \beta_i = \beta_{i-1} + \beta$) with the use of the PDP protocol

Let us also note that the original specification of the protocol states only that a node that transmitted m once, will thereafter send it once every β seconds (until its realisation). However, no particular order of message sending is imposed when a host has more that one message to send every β seconds, except that the messages that have been received for the first time during last β seconds must be sent within β seconds.

The *Reactive Dissemination Protocol* (RDP) assumes that nodes have information about their direct neighbours, e.g. each host knows its dynamic set $\mathcal{E}_i(t)$ for all t. In this case, a node p_i propagates m only if: $\exists\{p_i, p_j\} \in \mathcal{E}_i(t)$ ($K_i(m)[j]$ $= 0$), which is evaluated once every β seconds (β is as in the PDP).

The *Proactive Knowledge and Reactive Message* (PKRM) combines the features of PDP and RDP, without requiring information about direct neighbours. In the PKRM protocol, each p_i that transmitted m once as an originator, then sends every β seconds only $K\_ptr_i(m)$ (without $KK_i(m)$) packets, and if a receiver p_j of the packets does not have m, it sends within β seconds $request_j(m)$ packet requesting m to be transmitted. Thus, if the node p_i that has m, has received $request_j(m)$ packet in the past β seconds, it sends m. As in the first two protocols, β is fixed but this time: $B \geqslant 3\beta + 2\delta$ (to ensure that nodes exchange three times all messages during direct connectivity).

Finally, the PKRM$_O$ protocol is optimised towards bandwidth reduction in the following way. First, hosts check the messages received in the past not every β seconds but every $\hat{\beta}$ seconds, where $\hat{\beta}$ is random duration distributed uniformly in $(0, \beta)$. Second, the transmission of a message can be suppressed, if the node has received at least α equivalent transmission of that message in the past $\hat{\beta}$ seconds (the parameter α is configurable *suppression threshold*).

4 Analysis of Time Constraints of the Broadcast Protocols

The time constraints of the broadcast protocols discussed in Section 3 and mentioned by the authors in [8, 7] are as follows: $B \geqslant 2(\beta + \delta)$ in case of the PDP and RDP protocols, $B \geqslant 3\beta + 2\delta$ in the case of the PKRM and PKRM$_O$ protocols, and $B \gg \delta$ in both cases; where δ is the maximum message transmission time between neighbouring nodes, B is the minimum time of direct connectivity (remarked by the authors as an application-specified parameter), and β is a *configurable* parameter. The above constraints have been set to ensure that a node both receives a message and responds to that message during direct connectivity. Let us observe, that the β parameter cannot be arbitrary, since the protocols require that during every β period, every node must be able to send *all* its unrealised messages. We denote by s the maximum number of messages that a node can disseminate *concurrently*, i.e. without being realised. Let us assume without a lost of generality that $s \geqslant 1$. In the simplest case when $s = 1$, the protocols work in a blocking manner, that is a host will start disseminating a new message providing the previous one originated by it is realised. Moreover, let $S = N \cdot s$ be the total number of messages that can be disseminated in the system by all nodes. Then, the following theorem can be proved:

Theorem 1. *If a network is composed of N nodes and if each node can disseminate concurrently at most s messages, then the PDP, RDP, PKRM and PKRM$_O$ protocols require that $\beta \geqslant N \cdot s \cdot \delta$.*

In other words, Theorem 1 states that the minimum value of the β parameter cannot be shorter than the total time of transmission of all messages that can be disseminated in the system by all nodes. To prove Theorem 1 it is necessary to show that a node can be required to send $S = N \cdot s$ messages during a β seconds period.

Proof. Let us consider a system of N hosts, denoted by $p_1, \ldots, p_N$, and let each node disseminate s messages. If all the nodes remain operative and connected with each other (complete graph), then within first β seconds each node sends its own s messages. So, at the end of this period, every other node receives $S - s$ messages, to have in total (including its own) S unrealised messages, and the following conditions are fulfilled:

- $\forall_{i \in \{1, \ldots, N\}} \left(|\mathcal{U}nrealised_i| = S \right)$,
- $\forall_{i \in \{1, \ldots, N\}} \left(\forall m \in \mathcal{U}nrealised_i \left(\sum_{j=1}^{N} K_i(m)[j] = 1 \right) \text{ iff } p_i \text{ is an originator of } m \right)$,
- $\forall_{i \in \{1, \ldots, N\}} \left(\forall m \in \mathcal{U}nrealised_i \left(\sum_{j=1}^{N} K_i(m)[j] = 2 \right) \text{ iff } p_i \text{ has received } m \text{ from other node} \right)$.

Thus, if $N - f > 2$, then after another β seconds every node is required to send S messages. Precisely, in the case of the PDP protocol, within next β seconds period every node simply sends all S unrealised messages. The same occurs in the case of the RDP protocol, because if $N - f > 2$ and nodes remain connected, then:

$$\forall_{i \in \{1, \ldots, N\}} \; (\forall m \in \mathcal{U}nrealised_i \; (\forall t \in \langle \beta, 2\beta \rangle \; (\exists \{p_i, p_j\} \in \mathcal{E}_i(t) \; (K_i(m)[j] = 0)))).$$

In the case of the PKRM and PKRM$_O$ protocols, within next β seconds period each node is required to send $K\_ptr_i(m)$ packets for all S unrealised messages (we observe that so far no two $K_i(m)$ vectors are equal, so the suppression parameter of the PKRM$_O$ protocol will remain equal to 0).

If $N - f = 2$, then all the received messages can be realised. In this case, within next β seconds period the nodes send $S - s$ $realise_i(m)$ packets and s still unrealised messages originated by them, or $K\_ptr_i(m)$ packets with the use of the PKRM and PKRM$_O$ protocols. □

We also observe that the scenario when all nodes are directly connected (as in the proof above) is not the only case when the need to send S messages during a β seconds period may occur. For example, let us assume that at the beginning N nodes in a network, each disseminating s messages, are divided into two partitions: first contains all the nodes with even numbers, and second contains all the nodes with odd numbers. Let the nodes in the two partitions exchange their messages in such a way, that every node in each partition receives all other messages in its partition. If $f \leqslant \lfloor \frac{1}{2}N \rfloor - 2$, then so far no message can be realised. Moreover, if any two nodes, one with an odd number (say p_1) and one with an even number (say p_2) directly connect (which must occur in accordance with the network liveness requirement), they exchange their messages, and each of them has now all S messages: $|\mathcal{U}nrealised_1| = |\mathcal{U}nrealised_2| = S$, $\forall m \in \mathcal{U}nrealised_1$ $(\sum_{j=1}^{N} K_i(m)[j] = \lceil \frac{1}{2}N \rceil + 1)$, $\forall m \in \mathcal{U}nrealised_2$ $(\sum_{j=1}^{N} K_i(m)[j] = \lfloor \frac{1}{2}N \rfloor + 1)$. As it can be seen, no message has so far been received by at least $N - f \geqslant N - \lfloor \frac{1}{2}N \rfloor - 2$ nodes, and thus the two nodes are required now to broadcast all S messages once every β seconds (until its realisation).

To illustrate the possibility in the scenario above, the succeeding example with the use of the simplest PDP protocol is presented.

Example 1. Assuming $f = 0$, let us consider a system of four hosts ($N = 4$), denoted by $p_1, \ldots, p_4$, and let each node sends a message m_i ($i = 1, \ldots, 4$), i.e. $s = 1$ and thus $S = 4$. According to Theorem 1, let $\beta = 4\delta$ and $B = 2(\beta + \delta) = 10\delta$, so that the nodes with direct connectivity would be able to exchange (send and respond to) four messages. For the sake of presentation simplicity we assume that direct connections always occur at the beginning of successive β seconds periods.

Let us consider the following scenario. During the period from t_0 to t_1 ($t_1 - t_0 = B$), the nodes p_1 and p_2 exchange their messages: p_1 receives m_2, and p_2 receives m_1. Simultaneously, the nodes p_3 and p_4 exchange their messages: m_3 and m_4. Next, if during the period from t_2 to t_3 ($t_3 - t_2 = B$) the nodes p_2 and p_3 experience direct connectivity, they exchange their messages. After β_4 p_2 receives for the first time m_3 and m_4 from p_3, while p_3 receives for the first time

m_1 and m_2 from p_2—as shown in Figure 2. Both nodes have now all $S = 4$ messages, but no message has so far been received by all $N - f = 4$ nodes.

Thus, since β_5 the nodes p_2 and p_3 broadcast all S messages, until its realisation (m_1 and m_2 must also be received by p_4, m_3 and m_4 must also be received by p_1). At the same time p_1 and p_4 nodes are required to periodically (once every β seconds) broadcast their two messages. □

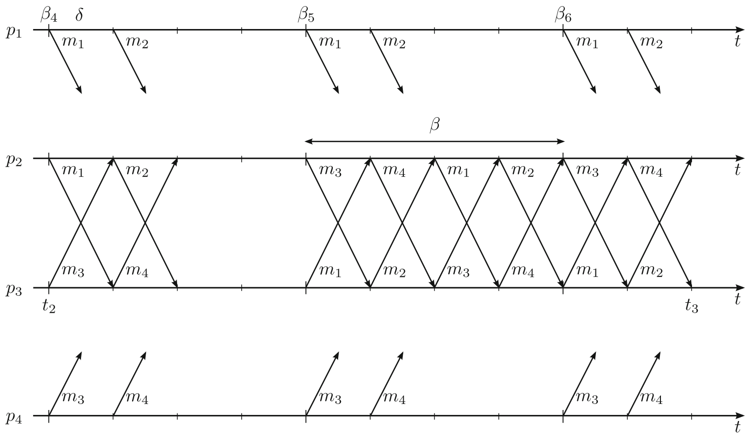


Fig. 2. Space-time diagrams illustrating the exchange of messages between nodes in Example 1 ($\beta = 4\delta$, $B = 2(\beta + \delta) = 10\delta$ and $\beta_i = \beta_{i-1} + \beta$)

One can observe that all the proactive protocols require that every node broadcasts all its unrealised messages periodically (once every β seconds), even if no other node is able to receive it—consider p_1 and p_4 nodes in Figure 2(b). The PKRM and PKRM$_O$ protocols make some optimisation to broadcast only $K\_ptr_i(m)$ packets instead of whole messages, though this requires longer time of direct connections (at least $3\beta + 2\delta$ comparing to $2(\beta + \delta)$). The PKRM$_O$ protocol uses also the suppression threshold to reduce the number of transmissions, but it only works by observing transmissions of equivalent messages, so on probability basis (and it may require some tests to set proper α parameter value in an actual environment). On the other hand, the reactive RDP protocol do not require constant broadcast of messages, since it depends on the neighbourhood of a node, but to gain information about node's neighbours it is necessary to use some mechanism of constant signalling or *beacon* messaging e.g. at the MAC layer, or similar solution. Finally, as we have shown, the minimum time of direct connectivity between nodes, in case of all four protocols, depends on the total number of messages that can be disseminated in a network simultaneously, and thus indirectly on the total number of nodes in a network. This in turn may require in practise to adjust β configurable parameter when the number of nodes changes, and in extreme cases can lead to a difficulty in obtaining direct connections that ensure the liveness property, making the correctness of the protocols dependent on its parameters.

5 Conclusions

In this paper we have defined exact model of ad hoc systems and the MANET liveness property with the use of dynamic sets, analysed the crash-tolerant protocols presented in [7, 8]. Moreover, we proved that the minimum time of direct connectivity between nodes and correctness of the protocols depend on the total number of hosts in a network and on the total number of messages that can be disseminated by each node concurrently. Thus, further investigation is required to find improved versions of the proactive protocols, which fulfil **BCAST1** and **BCAST2** properties and work correctly when the minimum time of direct connections between nodes allow them to exchange (send and respond to) at least only two messages, making the correctness of the protocols independent on system parameters.

References

1. Barbeau, M., Kranakis, E.: Principles of Ad-hoc Networking, 1st edn. John Wiley & Sons (April 2007)
2. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Computing Surveys 31(1), 1–26 (1999)
3. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (2002)
4. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming, 1st edn. Springer-Verlag New York, Inc. (April 2006)
5. Liu, S., McDermid, J.A.: Dynamic sets and their application in VDM. In: Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing (SAC 1993), pp. 187–192. ACM (February 1993)
6. Pagani, E., Rossi, G.P.: Providing reliable and fault tolerant broadcast delivery in mobile ad-hoc networks. Mobile Networks and Applications 4(3), 175–192 (1999)
7. Vollset, E.W.: Design and Evaluation of Crash Tolerant Protocols for Mobile Ad-hoc Networks. Ph.D. thesis, University of Newcastle Upon Tyne (September 2005)
8. Vollset, E.W., Ezhilchelvan, P.D.: Design and performance-study of crash-tolerant protocols for broadcasting and supporting consensus in MANETs. In: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005), pp. 166–178. IEEE Computer Society (October 2005)

Data Transfers on the Fly for Hierarchical Systems of Chip Multi-Processors

Marek Tudruj[1,2] and Łukasz Maśko[1]

[1] Institute of Computer Science of the Polish Academy of Sciences,
ul. Ordona 21, 01–237 Warsaw, Poland
[2] Polish–Japanese Institute of Information Technology,
ul. Koszykowa 86, 02–008 Warsaw, Poland
`{tudruj,masko}@ipipan.waw.pl`

Abstract. Local and global communication between computing cores is an essential problem of efficient parallel computations in many–core massively parallel systems based on many Chip Multi–Processor (CMP) modules interconnected by global networks. The paper presents new methods for data communication inside and between CMP modules. At the level of data communication between CMP modules a special network implements communication between CMP module external shared memories with simultaneous reads on the fly to L2 data caches and main memories of CMP modules. Similar mechanism improves local communication between shared memory modules and data caches inside CMPs.

1 Introduction

The interconnect-centric CMP modules design style [1, 2] is now establishing rules for industrial CMP data communication architecture. In larger CMP processors multi-hop on-chip networks [3] are embedded. Hierachical systems of CMPs where many CMP modules are interconnected by a global data network constitute interesting architectural solutions which can provide advantages for parallel computing at the functional and technology levels. At the functional level CMP module clustering can offer improvements in program parallelization efficiency and scalability. At the technology levels CMP module clustering can be a remedy for current VLSI technology limitations existing in large monolithic multicore processor chips, such as power dissipation, wire delays, signal cross talks and silicon area space limits [4]. The architectural model of CMP clusters of CMPs can be noticed in current architectural research, which however, is limited to standard SMP multicore processor architecture [5, 6].

In this paper, we discuss a hierarchical modular structure of many CMP modules with extensive architectural support for internal CMP core clustering and and external inter-CMP data communication. Inside CMP modules we propose dynamic creation of temporary core clusters to provide means for efficient transmission of highly shared data. It is done with the use of special group communication called reads on the fly, similar to cache injection [8], which was discussed in the context of CMPs in our earlier papers [9–11]. Reads on the fly reduce traffic on memory busses and provides direct cache to cache transfers, thus eliminating standard use of shared memory. The CMP-based architecture proposed in this paper offers several novel features comparing current CMP dynamic interconnect technology [7] and our previous works. At the level of internal CMP data communication, we propose reads on the fly implemented by snooping the L1-L2 data cache bus, which are much faster than those on the main memory bus. Reads on the fly are also organized on the L2–main memory busses, which offers very preloading of L2 data caches. At the level of inter–CMP data communication, we propose data reads on the fly implemented for CMP modules shared memories. We also propose to use redundant inter-CMP global networks to support multiple dynamic CMP clusters with data reads on the fly.

The paper is composed of 4 main parts. In the first part, the general system architecture and reads on the fly on local CMP networks are described. In the second part, reads on the fly on busses connecting L2 data banks with the CMP main memory are explained. In the third part, inter–CMP module reads on the fly are proposed. In the fourth part experimental results are presented.

2 System Architecture and Reads on the Fly in Local CMP Networks

The proposed parallel multi–CMP system consists of CMP modules interconnected by a global data network, Fig. 1a. A CMP module consists of a number of processor cores supplied with L1 caches (separate for data and instructions). All cores can share L2 data cache banks. L1 data caches can be connected to L2 banks through a local data communication network. The block diagram of the proposed CMP is shown in Fig. 1b. The local data communication network is a set of L2 bank busses. It allows creation of dynamic core clusters by runtime connecting/switching of L1 banks to/between the L2 busses. L1 switching and

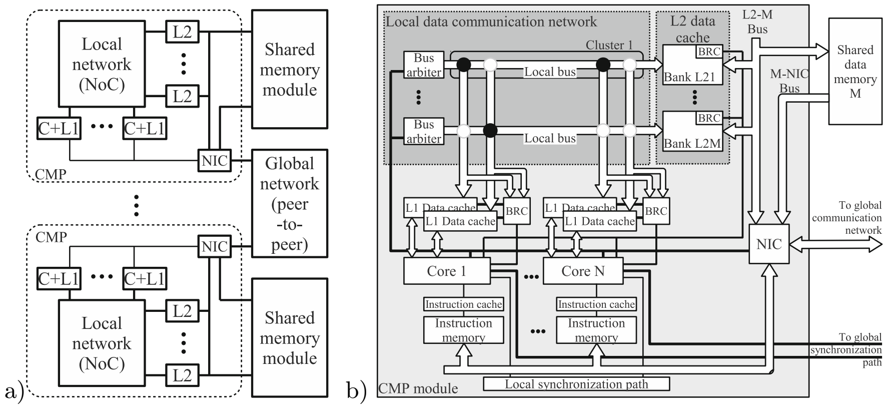


Fig. 1. General system structure (a) and the structure of a CMP module (b)

L1 to L2 accesses are done under control of bus arbiters. Dynamic core clusters enable advanced group data communication, involving a cluster of cores, especially useful for shared data transfers. Each core cooperates with several L1 banks (two in the presented experiments) which can be connected to separate L2 busses, thus enabling a core to belong to more than one dynamic cluster.

Data present on a L2 bus may be simultaneously captured to many L1 core data caches (reads on the fly) as a result of program requests issued to the L1 cache Bus Request Controller (BRC). Information about data to be captured contains the starting address, the length (measured in L1 cache blocks) of the data and the indication of the L1 bank to which data are to be read on the fly. The BRC snoops the address lines of the L1–L2 busses to which the corresponding L1 bank is connected and compares the addresses to those stored in the snooping tables. When a match of the address is discovered, the BRC captures the data from the bus to the indicated L1 bank. This operation requires synchronization of the writer with all the readers using a barrier instruction placed in the program. All L2 cache banks of a CMP module are connected through the L2-M bus to the local fragment of the distributed memory, shared by all CMP modules in the system. Programs for the proposed architecture are divided into tasks for processor cores defined according to the macro data flow graph (MDFG) representation. The tasks follow a cache-controlled macro data-flow execution paradigm. It assumes that all data have to be pre-fetched to core's L1 data cache before a task begins execution, L1 cache reloading is disabled and results may be sent from L1 to L2 caches only after a task completes.

3 Reads on the Fly on L2–Memory Busses

Data transfers between shared memory fragments attached to CMP modules can be done with the use of the global interconnection network. The global network can be any of dynamic data exchange networks such as a bus or a crossbar switch. Data transmissions are performed on requests deposed by cores in the Network Interface Controllers (NIC) of CMP modules. For a global data transfer, a connection between both CMP modules in the global network must be organized and the L2-M bus on the destination CMP must be reserved. On the source CMP the dedicated Memory-to-NIC (M-NIC) bus is used. When the complete data transfer path is prepared, the data are transmitted from a source memory to the destination with the intermediate use of NICs of both involved CMP modules. Such global data communication can be implemented as split–phase transactions with buffering of data packages in NIC controllers or it can be done using the wormhole routing without data buffering in NIC units.

When data brought over the global network are written to a CMP module shared memory, the data can be captured to many L2 banks inside this module in a single transaction using reads on the fly. This feature extends the read on the fly concept towards the level of global communication. Such elimination of many separate data transfers strongly improves pre-fetching of data for computations performed by a given core or core clusters.

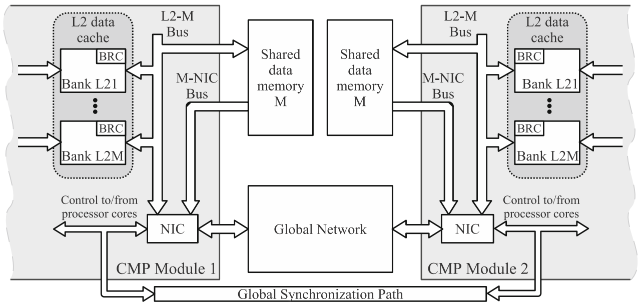


Fig. 2. Global communication infrastructure between CMP modules

To implement reads on the fly during CMPs memory to memory global data transfers, a Bus Request Controller (BRC) is provided in each L2 bank. An L2 bank BRC task is to store read on the fly requests sent by cores and then to snoop the L2 bus to capture data to L2 if the match of addresses occurs. A distant memory transaction requested for a CMP can be a read or write. The write has to be synchronized with read on the fly requests deposed in BRCs of all involved L2 banks by barriers executted using a Global Synchronization Path. If the distant memory transaction is a read, then the read on the fly to L2 takes place in the CMP of the transmission initializing core and the barrier activates the distant CMP write. If the transaction is a distant write then the read on the fly to L2 takes place in the distant CMP, which implies, that the global barrier must activate a read in the CMP memory of the initializing core.

To enable automatic design of programs for the proposed architecture with optimized data communication control an extended macro data flow graph (EMDFG) representation is proposed. Compared to the standard macro data flow graph program representation, some additional nodes have been introduced:

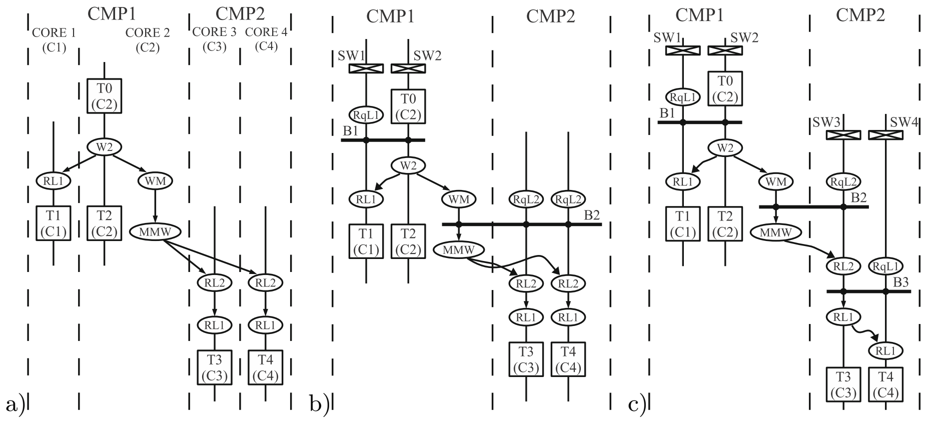


Fig. 3. Simple EMDF representation of the graph (a), EMDFG for reads on the fly: to L1 in CMP1and L2 in CMP2 (b) and to L1 in CMP1 and to L2, L1 in CMP2 (c)

W2 – write of data from core's data cache L1 to L2, WM – write of some data from the L2 to the shared memory of the CMP module, MMW – write from the shared memory of a CMP to the memory of another CMP, RqL1 – deposing read on the fly request in a BRC of a L1 bank, RL1 – read on the fly from the L1 bus, RqL2 – deposing a read on the fly request to a BRC of a L2 bank, RL2 – read on the fly from the L2-memory bus to L2, B – a barrier, SW – switching a core's data cache bank between clusters – i.e. between L2 buses.

The EMDFG will be illustrated on an example in which a macro node T0 assigned to core C2 in CMP1 module distributes data to tasks T1-T4 assigned to cores C1-C4 in modules CMP1 and CMP2 (Fig. 3). Fig. 3a represents the respective EMDFG without reads on the fly. Fig. 3b represents the respective EMDFG with reads on the fly to L1 in CMP1 and to L2 in CMP2. L1 caches of C1 and C2 get connected to the same L2 bank bus in CMP1 (SW1, SW2). Task T0 writes its results from L1 to the L2 bank in CMP1 after fulfillment of the barrier B1 by read on the fly request (RqL1) issued by task T1. T1 reads data on the fly while they are written by T0 to L2. T0 passes data to T2 in L1. These data are next sent from L2 to memory (WM). Next, some of the data are sent to the CMP2 memory via the global network (MMW) after the barrier B2 is fulfilled by read on the fly requests to L2 (RqL2) in CMP2. When the data are written to the CMP2 memory (via NIC buffer) over the "L2-M" bus in CMP2, they are read on the fly to two CMP2 L2 cache banks (RL2). After the data are read to L2 they are pre-fetched (RL1) to L1 caches of cores C3, C4 in CMP2.

A similar CMP1 to CMP2 data transfer but with data pre-fetching to L1 banks of cores C3 and C4 in CMP2 is shown in Fig 3c. When the data are written to the CMP2 memory by the "L2-M" bus, they are read on the fly to one CMP2 L2 cache bank (RL2). Next, a read on the fly to L1 of core C4 is organized. The L1 caches of C3 and C4 were earier connected to the same L2 bank bus in CMP2 (SW3, SW4). Barrier B3 conditions pre-fetching of data from L2 to L1 of core C4. When the data are read from L2 to L1 bank of core C3 they are read on the fly to L1 of core C4.

4 Data Reads on the Fly during Global Inter–CMP Transfers

Efficient data prefetching for parts of programs executed by many CMPs working in parallel is crucial for the overall performance of parallel applications. Reads on the fly applied at the level of main memory fragments of CMPs can significantly reduce application program execution time. The global network used to interconnect CMP modules (we assume – a special crossbar switch) has to be designed to enable a data posting operation for sets of CMP modules in the system, see Fig. 4. The data posting is done as a result of the memory to memory multicast or broadcast on the global network (MMM or MMB nodes in EMDFG) placed in the application program. The data are exposed by the NIC unit of a sender CMP as a series of L2 data blocks accompanied by block starting addresses. The posting has to be synchronized with all receiver CMPs before the sender

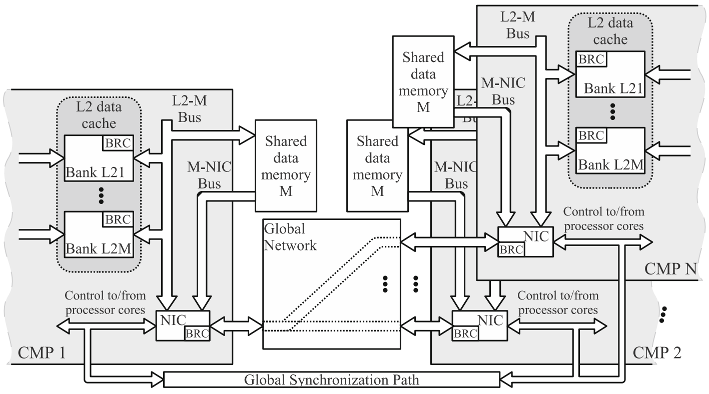


Fig. 4. System architecture for reads on the fly for many CMP modules

NIC reads data from memory and sends to the global nework. It is done by a barrier, which is fulfilled by respective read on the fly requests deposed in the BRC blocks of the NIC units. BRCs in all participating NICs snoop the address parts of packets coming from the global network and selectively capture the data for NICs to send them next to the CMPs main memories.

The EMDFG for reads on the fly to memories of a set of CMPs is shown in Fig. 5. In this figure task T1 on core C1 in CMP1 transfers data with the use of reads on the fly for tasks T2-T5 in CMP2 and CMP3. First, T1 writes its L1 contents to L2 (W2) and then to the main memory (WM). In Fig 5a, the C1 thread is synchronized (barrier B1) with the deposals to NIC BRCs of data read on the fly to memory requests (MMRq) in CMP2 and CMP3. When B1 is fulfilled, the NIC of CMP1 exposes the data on the global network (MMM – Memory to Memory Multicast). When the data reach the NICs of CMP2 and CMP3, they are read to their data memories. Next, the data are pre-fetched to L2 (RL2) and then to L1 (RL1) by separate pre-fetch operations. Fig. 5b presents the EMDFG for reads on the fly to memories of CMP2 and CMP3 combined with reads on the fly to L2 in CMP2 and CMP3, and also with a read on the fly to L1 cache for core C5 in CMP3. In this graph, data read on the fly requests to CMP2 and CMP3 data memories (MMRq) are synchronized by a common barrier B1 with the read on the fly requests to two L2 banks in CMP2 (RqL2) and one L2 bank in CMP3 (RqL2). In CMP2, some data are pre-fetched from the L2 banks to L1s in C2 and C3 cores in the standard way (RL1). In CMP3, the data are read from a bank of L2 to L1 of C4 in the standard way with a simultaneous read on the fly to L1 of C5. The L1 caches of C4 and C5 in CMP3 have been earlier connected to the source L2 bank bus (SW1, SW2).

Global data communication in the proposed multi-CMP system requires setting of link connections between many pairs of CMPs. The inter-CMP connection setting is a multi-phase operation performed under control of NIC controllers at both sides of the global network. To eliminate the connection setting time, the look-ahead reconfiguration principle can be applied with the use of redundant global networks [12]. With the look-ahead reconfiguration, it is possible to

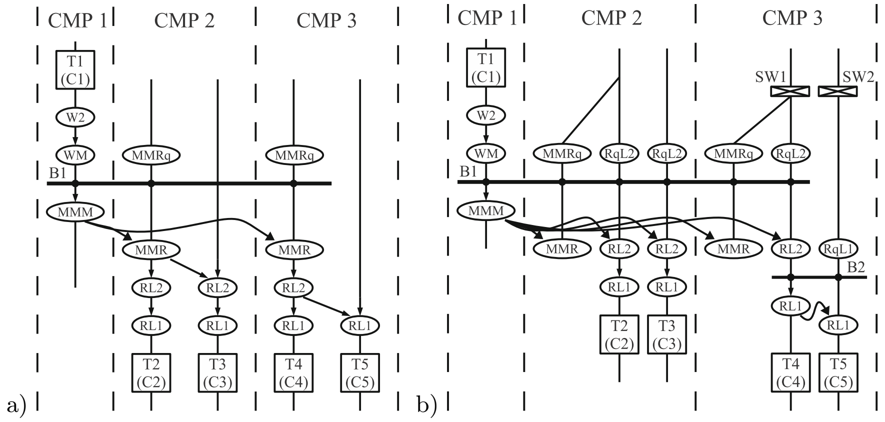


Fig. 5. EMDFG for reads on the fly to many CMPs (a) and combined reads on the fly to CMP2 and CMP3 with simultaneous reads on the fly to L2 and L1 caches (b)

preserve and use some inter-CMP connections for a longer time than that of a single global data transfer. A possible fsystem structure for organizing the look-ahead configured inter-CMP connections is shown in Fig. 6. In this structure, two global networks are used to house connections between different pairs of CMPs. A CMP Link Switch is introduced to switch each CMP between the global networks to the look-ahead prepared connections. Connections inside the global networks and the control of the CMP Link Switch are organized by a Switch Controller on requests coming to CMP NIC units from application programs. The Switch Controller performs the necessary synchronization of CMP switching with execution of programs. This architecture seems to be especially interesting for CMP-level data reads on the fly organized in dynamically created clusters of CMP modules. By increasing the number of used global switches, we can organize a number of dynamic CMP clusters in which many global data transfer transations on the fly can be performed in parallel.

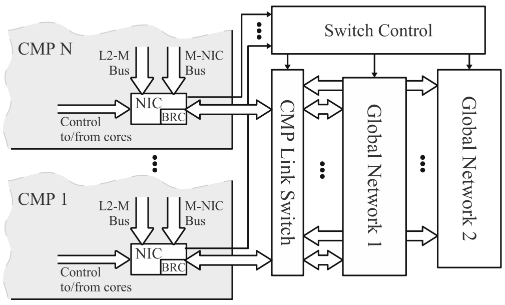
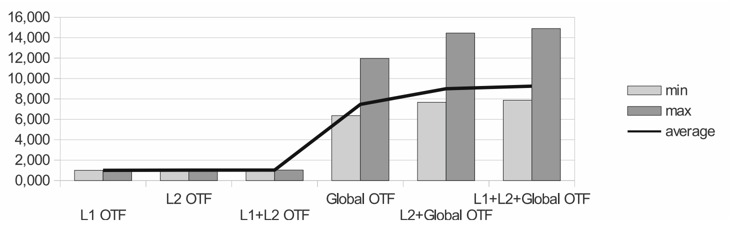


Fig. 6. Multiple global network architecture for global communication with look–ahead reconfiguration

5 Experimental Results

The influence of reads on the fly on execution time of numerical applications was verfied by simulation of square matrix parallel multiplication based on matrix decomposition into stripes. We have measured execution time of the initial phase of the algorithm, where data are distributed between SoC modules and further delivered to cores' data caches, and the whole execution time. We have compared it to execution of the same program graph in a system without any reads on the fly. We have analyzed reads on the fly (OTF) for: data transfers from L2 to L1 banks, transfers from shared memory to L2 banks and global transfers between CMP memory modules. 6 configurations were tested in total: OTF only between L2 and L1 (denoted as L1), OTF only between shared memory and L2 banks (denoted as L2), OTF only between CMP memories (denoted as Global) and their 3 combinations: L1+L2, L2+Global and L1+L2+Global. The simulations were done using a simulator written in C/C++. It was cycle-accurate for communication operations in the assumed system architecture. Matrix sizes from 16 to 1024 were considered.

Figure 7a shows minimal, maximal and average value (over all examined matrix sizes) of the obtained improvement for the data distribution phase, which consists of data transfer to shared memory of a CMP module, then from this memory to L2 cache and finally from L2 to L1 cache. We assume, that L1 cache memory is large enough to hold both parts of matrices and space for results. We can see, that data transfers on the fly on internal busses inside a CMP have marginal influence on program execution time, compared to the influence on global data transfers. It is caused by large data volumes transferred via the global network comparing to transfers on local CMP networks. In a standard system, the same parts of matrices must be transferred many times to CMPs.

a)


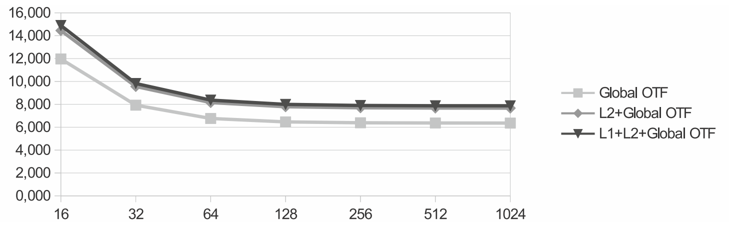
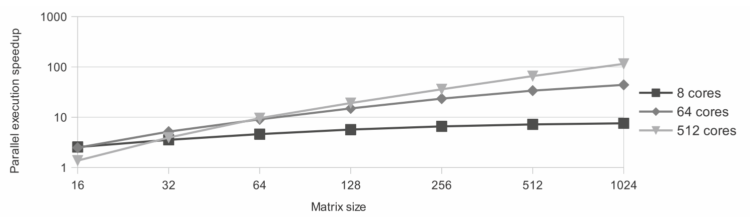
b)


Fig. 7. Reduction factor of initial data transfer time for different versions of reads on the fly, compared to standard communication

a)


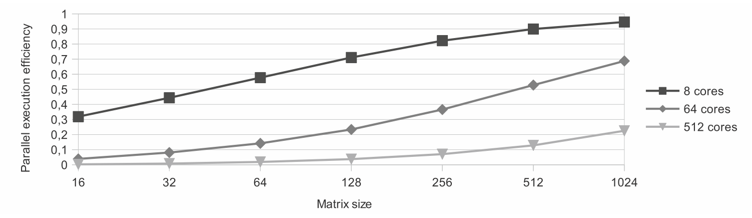
b)


Fig. 8. Parallel speedup and efficiency with global reads on the fly for different numbers of cores and matrix sizes

With reads on the fly, it needs a single network transmission. A big discrepancy between minimal and maximal improvement, when reads on the fly were used for global communication, is shown in Fig. 7b. Larger improvement for small matrix sizes results from reduction of constant transmission overheads such as startup time and confirmation after each packet, which have bigger impact on small transfers. For bigger matrix sizes the impact of these overheads for both standard and OTF transmissions decreases, therefore the speedup is smaller.

We have also compared parallel execution of the whole program graph with reads on the fly to serial execution. Three different system configurations were used, which implied the grain of computations: 8, 64 and 512 cores. The system contained 4, 16 and 64 CMPs with 2, 4 and 8 cores, respectively. The obtained parallel speedup and efficiency are presented in Fig. 8. Execution speedup depends on computation grain. For small matrix sizes, smaller configurations behave better. Increasing the number of cores decreases transferred data grain, therefore the impact of data transfer overhead increases, reducing computation efficiency. For bigger matrices we can better exploit parallelism of computations by employing a higher number of parallel cores keeping data grain at reasonable level, thus obtaining better overall speedup.

6 Conclusions

New architectural solutions for efficient data communication for a system of globally interconnected CMP modules has been presented. Communication is based on reads on the fly, applied both for local data transfers inside CMP modules between hierarchical data caches and for global data transfers between distributed shared memory modules. The proposed mechanisms allow parallel data reads on the fly to L2 memory banks during transfers to main memory. They also enable multiple parallel transfers to main memories of many CMPs and eliminate transfers of the same data. It requires a special but viable design of the global network. The simulation results show that the proposed solutions are very efficient for pre-fetching data to memories and data caches. They are especially profitable for global communication, when the volume of shared data is big and the data must be transmitted to many CMPs at the same time, for instance during initial data loading. Programs for the proposed system are built following an extended macro data flow graph representation and cache controlled execution paradigm. An automatic scheduler which transforms program standard macro data flow graphs into optimized extended versions is under construction.

References

1. Owens, J.D., et al.: Research Challenges for On-Chip Interconnection Networks. IEEE MICRO, 96–108 (September-October 2007)
2. Kundu, S., Peh, L.S.: On-Chip Interconnects for Multicores. IEEE MICRO, 3–5 (September-October 2007)
3. Ye, T.T., et al.: Packetization and routing analysis of on-chip multiprocessor networks. Journal of Systems Architecture 50, 81–104 (2004)
4. Kumar, R., Zyuban, V., Tullsen, D.M.: Interconnections in Multi–Core Architectures: Understanding Mechanisms, Overheads and Scaling. SIGARCH Computer Architecture News 33(2) (May 2005)
5. Wu, X., Taylor, V., Lively, C., Sharkawi, S.: Performance Analysis and Optimization of Parallel Scientific Applications on CMP Cluster Systems. Scalable Computing: Practice and Experience 10(1) (2009)
6. Chi, Z., Xin, Y., Srinivasan, A.: Processor affinity and MPI performance on SMP-CMP clusters. In: Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW 2010, April 19-23, pp. 1–8. IEEE CS Press (2010)
7. Shen, J.-S., Hsiung, P.-A. (eds.): Dynamic Reconfigurable Network-on-Chip Design, Innovations for Computational Processing and Communication. IGI Global (2010)
8. Milenkovic, A., Milutinovic, V.: Cache Injection: A Novel Technique for Tolerating Memory Latency in Bus-Based SMPs. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 558–566. Springer, Heidelberg (2000)
9. Tudruj, M., Maśko, Ł.: Dynamic SMP Clusters with Communication on the Fly in NoC Technology for Very Fine Grain Computations. In: ISPDC 2004, Cork, pp. 97–104. IEEE CS Press (July 2004)
10. Tudruj, M., Maśko, Ł.: Towards Massively Parallel Computations Based on Dynamic SMP Clusters wih Communication on the Fly. In: ISPDC 2005, Lille, France, pp. 155–162. IEEE CS Press (July 2005)
11. Tudruj, M., Maśko, Ł.: Dynamic SMP Clusters with Communication on the Fly in SoC Technology Applied for Medium-Grain Parallel Matrix Multiplication. In: PDP 2007, Naples, Italy, pp. 270–277. IEEE CS Press (February 2007)
12. Laskowski, E., Maśko, Ł., Tudruj, M., Thor, M.: Program Execution Control in a Multi CMP Module System with a Look-Ahead Configured Global Network. In: ISPDC 2009, Lisbon, pp. 193–204. IEEE CS Press (July 2009)

New Level-3 BLAS Kernels for Cholesky Factorization

Fred G. Gustavson[1], Jerzy Waśniewski[2], and José R. Herrero[3]

[1] IBM Research, Emeritus, Umeå University
[2] Technical University of Denmark
[3] Universitat Politècnica de Catalunya, BarcelonaTech
fg2935@hotmail.com, jw@imm.dtu.dk, josepr@ac.upc.edu

Abstract. Some Linear Algebra Libraries use Level-2 routines during the factorization part of any Level-3 block factorization algorithm. We discuss four Level-3 routines called DPOTF3, a new type of BLAS, for the factorization part of a block Cholesky factorization algorithm for use by LAPACK routine DPOTRF or for BPF (Blocked Packed Format) Cholesky factorization. The four routines DPOTF3 are Fortran routines. Our main result is that performance of routines DPOTF3 is still increasing when the performance of Level-2 routine DPOTF2 of LAPACK starts to decrease. This means that the performance of DGEMM, DSYRK, and DTRSM will increase due to their use of larger block sizes and also to making less passes over the matrix elements. We present corroborating performance results for DPOTF3 versus DPOTF2 on a variety of common platforms. The four DPOTF3 routines are based on simple register blocking; different platforms have different numbers of registers and so our four routines have different register blockings. Blocked Packed Format (BPF) is discussed. LAPACK routines for _POTRF and _PPTRF using BPF instead of full and packed format are shown to be trivial modifications of LAPACK _POTRF source codes. Upper BPF is shown to be identical to square block packed format. Performance results for DBPTRF and DPOTRF for large n show that routines DPOTF3 does increase performance for large n.

1 Introduction

We consider Cholesky block factorizations of a symmetric positive definite matrix where the data has been stored using Block Packed Format (BPF) [10, 11, 2], [13], [Algorithm 865]. We will especially examine the case where the matrix A is block factored into U^TU, where U is an upper triangular matrix. Upper BPF is also Square Block (SB) packed format; see Section 1.1 for details. We will also show in Section 1.1 that the implementation of BPF is a restructured form of the LAPACK factorization routines _PPTRF or _POTRF. For _PPTRF we will need slightly more storage [4]. Hence matrix-matrix operations that take advantage of Level-3 BLAS can be used and thereby higher performance [5, 11] is achieved. This paper focuses on the replacement of LAPACK routines _PPTF2 or _POTF2 routines, which is based on using Level-2 BLAS operations, by routines _POTF3.

_POTF3 are Level-3 Fortran routines that use forms of register blocking [12] based on machine characteristics.

The performance numbers presented in Section 3 bear out that the Level-3 based factorization Fortran routines _POTF3 for the factorization part of Cholesky factorization gives improved performance over the traditional Level-2 _POTF2 routines used by LAPACK. The gains come from the use of square block (SB) format and the use of Level-3 register blocking. Besides the gains one obtains by use of the _POTF3 routines one gets gains from using Level-3 BLAS routines _GEMM, _SYRK, _TRSM. In Fig. 1 we compare a right looking version of DPOTRF with a right looking SBPF implementation of Cholesky factorization [11]. These gains, especially for packed routines, suggests a change of direction for traditional LAPACK packed software.

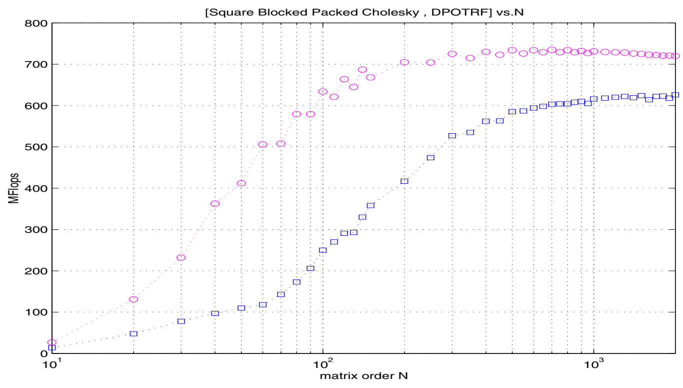


Fig. 1. Performance of Right Looking SBPF (plot symbol ○) and DPOTRF (plot symbol □) Cholesky factorization algorithms on an IBM POWER3 of peak rate 800 MFlops. DPOTRF calls DPOTF2 and ESSL BLAS. SBPF Cholesky calls DPOTF3 and BLAS kernel routines.

A main point of our paper is that the Level-3 Fortran routines _POTF3 allow one to increase the block size *nb* used by a traditional LAPACK routine such as _POTRF. Our performance numbers show that performance starts degrading at block size 64 for _POTF2. However performance continues to increase past block size 64 to 100 for our new Level-3 Fortran routines _POTF3. Such an increase in *nb* will have a good effect on the overall performance of _POTRF as the Level-3 BLAS _TRSM, _SYRK and _GEMM will perform better for two reasons. The first reason is that Level-3 BLAS perform better when the $k = nb$ dimension of _GEMM is larger. The second reason is that Level-3 BLAS are called less frequently by a ratio of increased block size of the Level-3 Fortran routines _POTF3 over the block size used by Level-2 routine _POTF2. Calling Level-3 BLAS less frequently does not mean less data copying will be done. The data copying amount remains the same. However, less overhead occurs by calling less frequently and also, the data copying can perhaps be done more efficiently. It is beyond the scope of this short paper to conclusively demonstrate this assertion. However, experimental verifications of this assertion are given by our performance results and also by the performance results in [2]. The recent paper by [20] also demonstrates that our assertions are correct; he gives both experimental and qualitative results.

One variant of our BPF, lower BPF, is not new. It was used by [4] as the basis for packed distributed storage used by ScaLAPACK. This storage layout consist of a collection of block columns; each of these has column size nb. Each block column is stored in standard column major (CM) format. In this variant one does a LL^T Cholesky factorization, where L is a lower triangular block matrix. Note that lower BPF is not a preferred format as it does not give rise to contiguous SB. Another main point of our paper is that we can transpose each block column to obtain our upper triangular formulation which is then a SB format data layout. These layouts use about the same storage as LAPACK _PPTRF routines. These layouts can also use Level-3 BLAS routines: their performance is about the same as LAPACK _POTRF routines and hence they do *not* have the performance of LAPACK _PPTRF routines. The field of new data structures dates back at least to 1997. Space does not allow a detailed listing of this large area of research. We offer a survey paper which partially covers the field up to 2004 [6]. After that there are more papers. We give two [16, 17].

1.1 Introduction to BPF

The purpose of packed storage for a matrix is to conserve storage when that matrix has a special property. Symmetric and triangular matrices are two examples. In designing the Level-3 BLAS, [5] did not specify packed storage schemes for symmetric, Hermitian or triangular matrices. The reason given at the time was "such storage schemes do not seem to lend themselves to partitioning into blocks ... Also packed storage is required much less with large memory machines available today". The BPF algorithms demonstrates that packing and Level-3 BLAS are compatible resulting in no performance loss. As memories continue to get larger, the problems that are solved get larger: there will always be an advantage in saving storage.

We pack a symmetric matrix by using BPF where each block is held contiguously in memory [11, 2]. This usually avoids the data copies, see [12], that are inevitable when Level-3 BLAS are applied to matrices held in standard column major (CM) or row major (RM) format in rectangular arrays. Note, too, that many data copies may be needed for the same submatrix in the course of a Cholesky factorization [10–12].

We show an example of lower and upper BPF in Fig. 2 with blocks of size 2 superimposed. Fig. 2 shows where each matrix element is stored within the array that holds it. The rectangles of Fig. 2 are suitable for passing to the BLAS since the stride between elements of each rectangle is uniform. In Fig. 2a we do *not* further divide each rectangle into SB as these SB are *not* contiguous as they are in Fig. 2b. BPF consists of a collection of $N = \lceil n/nb \rceil$ rectangular matrices concatenated together. The size of the i^{th} rectangle is $n - i \cdot nb$ by nb

for $i = 0, \ldots, N-1$. Consider the i^{th} rectangle. Its LDA is either $i \cdot nb$ or nb. In Figs. 2ab the LDA's are $n - i \cdot nb, nb$. The rectangles in Fig. 2a are the transposes of the rectangles in Fig. 2b and vice versa. The rectangles of Fig. 2b have a major advantage over the rectangles of Fig. 2a: the i^{th} rectangle consists of $N-i$ square blocks, SB. This gives two dimensional granularity to _GEMM for upper BPF which lower BPF *cannot* possess. We therefore need a way to get from a lower layout to an upper layout in-place. If the matrix order is n and the block size is nb, and $n = N \cdot nb$ then this rearrangement may be performed *very efficiently* in-place by a "vector transpose" routine [14, 15]. Otherwise, the rearrangement, if done directly, becomes very costly. Therefore, this condition becomes a *crucial* condition. So, when the order is *not* an integer multiple of the block size, we pad the rectangles so the i^{th} LDA is $(N-i) \cdot nb$ and hence a multiple of nb. We further assume that nb is chosen so that a block fits comfortably into a Level-1 or Level-2 cache. The LAPACK ILAENV routine may be called to set nb.

2a. Lower Blocked Packed Format

$$\begin{array}{rr|rr|rr}
0 & & & & & \\
1 & 7 & & & & \\
2 & 8 & 12 & & & \\
3 & 9 & 13 & 17 & & \\
4 & 10 & 14 & 18 & 20 & \\
5 & 11 & 15 & 19 & 21 & 23
\end{array}$$

2b. Upper Blocked Packed Format

$$\begin{array}{rr|rr|rr}
0 & 2 & 4 & 6 & 8 & 10 \\
 & 3 & 5 & 7 & 9 & 11 \\
\hline
 & & 12 & 14 & 16 & 18 \\
 & & & 15 & 17 & 19 \\
\hline
 & & & & 20 & 22 \\
 & & & & & 23
\end{array}$$

Fig. 2. Lower Blocked Column Packed and Upper Square Blocked Packed Formats

```
$l = \lceil n/nb \rceil$
do $i = 1, l$
    Call of Level-3 BLAS _SYRK $i-1$ times
    Call of LAPACK subroutine _POTF2
    Call of Level-3 BLAS _GEMM $i-1$ times
    Call of Level-3 BLAS _TRSM
end do
```

Fig. 3. LAPACK _POTRF algorithms for BPF of Fig. 2

We factorize the matrix A as laid out in Figs. 2 using LAPACK's _POTRF routines trivially modified to handle the BPF of Figs. 2; see Fig. 3. This trivial modification is shown in Fig 3 where one needs to call _SYRK and _GEMM $i-1$ times at factor stage i because the block rectangles representations do not have uniform stride across rectangles. Additionally, for upper BPF only, one can call _GEMM $(n-i-1)(i-1)$ times, as described above, where each call is a SB _GEMM update. This approach was used by a LAPACK multicore implementation [19].

2 The _POTF3 Routines

_POTF3 uses a modified version of LAPACK _POTRF. Hence it is very different from _POTF2. _POTF3 only works well on a contiguous SB that fits into a L1 or L2 cache. It needs to use a tiny block size kb. We chose $kb = 2$. Blocks of this size are called *register* blocks. This block contains four elements of A; we load them into four scalar variables `t11, t12, t21, t22`. This alerts most compilers to put and hold the small register block in registers. For a diagonal block $a_{i:i+1,i:i+1}$ we load it into four registers `t11, t12, t21, t22`, we update it with _SYRK, we factor it, and we store it back into $a_{i:i+1,i:i+1}$ as $u_{i:i+1,i:i+1}$. This combined operation is called fusion by the compiler community. For an off diagonal block $a_{i:i+1,j:j+1}$ we load it, we update it with _GEMM, we scale it with _TRSM, and we store it. This again is an example of fusion. In the scaling operation we replace divisions by $u_{i,i}, u_{i+1,i+1}$ by reciprocal multiplies. The two reciprocals are saved in two registers during the _SYRK factor fusion computation. Fusion, as used here, avoids procedure call overheads for very small computations; in effect, we replace all calls to BLAS by in-line code. See [8, 9, 7] for related remarks on this point. The key loop in the _GEMM _TRSM fusion computation is the _GEMM loop.

Note that _POTRF cannot use fusion because it must explicitly call Level-3 BLAS. However, these calls are at the nb block size level or larger area level; the calling overhead is therefore negligible.

In another _POTF3 version we accumulate into a vector block of size 1×4 in the inner _GEMM loop. Each execution of the vector loop involves the same number of floating-point operations (8) as for the 2×2 case; it requires 5 reals to be loaded from cache instead of 4.

On most of our processors, faster execution was possible by having an inner _GEMM loop that updated both $A_{i,j}$ and $A_{i,j+1}$. The scalar variables `aki` and `aki1` need only be loaded once, so we now have 6 memory accesses and 16 floating-point operations. This loop uses 14 local variables, and all 14 of them should be assigned to registers.

We found that this algorithm gave very good performance, see Section 3. The implementation of this version of _POTF3 is available in the TOMS Algorithm paper [13],[Algorithm 865].

2.1 _POTF3 Routines Can Use a Larger Block Size *nb*

The domain of A for Cholesky factorization is an upper triangle for _POTF3. This means that only half of the cache block of size nb^2 is accessed by _POTF3. Furthermore, in the outer loop of _POTF3 at stage j only locations $L(j) = j(nb - j)$ of the triangle are accessed where $0 \leq j < nb$. The function L has a maximum value of $nb^2/4$ at $j = nb/2$. Hence, during execution of _POTF3, only half of the cache is used and the maximum cache usage at any time instance is one quarter of the cache size. This means that _POTF3 can use a larger block size before its performance will start to degrade. This fact is true for all four _POTF3 computations and this is what our experiments showed: As nb increased from 64 to 100 the performance of _POTF3 increased. _POTF2 performance, on the other hand, started degrading as nb increased beyond 64.

Furthermore, and this is our main result, as nb increases so does the k dimension of _GEMM increase as $k = nb$. It therefore follows that overall performance of _POTRF increases: _GEMM performance is the key performance component of _POTRF. See the papers of [2, 20] where performance evidence of this assertion is given.

3 Experimental Results

We consider matrix orders of 40, 64, and 100 since these orders will typically allow the computation to fit comfortably in Level-1 or Level-2 caches.

We do our calculations in DOUBLE PRECISION. The DOUBLE PRECISION names of the subroutines used in this section are DPOTRF and DPOTF2 from the LAPACK library and four simple Fortran Level-3 DPOTF3 routines described below. LAPACK DPOTF2 is a Fortran routine that calls Level-2 BLAS routine DGEMV and it is called by DPOTRF. DPOTRF also calls Level-3 BLAS routines DTRSM, DSYRK, and DGEMM as well as LAPACK subroutine

Table 1. Performance in Mflop/s of the Kernel Cholesky Algorithm. Comparison between different computers and different versions of subroutines.

Mat	Ven	Recur	dpotf2		2x2 w. fma		1x4		2x4		2x2	
ord	dor	sive			8 flops		8 flops		16 flops		8 flops	
	lap	lap	lap	fac	lap	fac	lap	fac	lap	fac	lap	fac
1	2	3	4	5	6	7	8	9	10	11	12	13
Newton: SUN UltraSPARC IV+, 1800 MHz, dual-core, Sunperf BLAS												
40	759	547	490	437	1239	1257	1004	1012	1515	**1518**	1299	1317
64	1101	1086	738	739	1563	1562	1291	1295	1940	**1952**	1646	1650
100	1264	1317	1228	1094	1610	1838	1505	1541	1729	**2291**	1641	1954
Freke: SGI-Intel Itanium2, 1.5 GHz/6, SGI BLAS												
40	396	652	399	408	1493	1612	1613	1769	2045	**2298**	1511	1629
64	623	1206	624	631	2044	2097	1974	2027	2723	**2824**	2065	2116
100	1341	1906	1317	840	2790	2648	2985	3491	3238	**4051**	2796	2668
Battle: 2×Intel Xeon, CPU @ 1.6 GHz, Atlas BLAS												
40	333	355	455	461	818	840	781	799	806	815	824	**846**
64	489	483	614	620	1015	1022	996	1005	1003	1002	1071	**1077**
100	883	904	883	801	1093	1191	1080	1248	1081	1210	1110	**1284**
Nala: 2×AMD Dual Core Opteron 265 @ 1.8 GHz, Atlas BLAS												
40	350	370	409	397	731	696	812	**784**	773	741	783	736
64	552	539	552	544	925	909	1075	**1064**	968	959	944	987
100	710	686	759	651	942	1037	972	**1231**	949	1093	950	1114
Zoot: 4×Intel Xeon Quad Core E7340 @ 2.4 GHz, Atlas BLAS												
40	497	515	842	844	1380	1451	1279	1294	1487	**1502**	1416	1412
64	713	710	1143	1146	1675	1674	1565	1565	1837	**1841**	1674	1674
100	1232	1234	1327	1696	1533	2294	1503	2160	1563	**2625**	1530	2285

ILAENV. The purpose of ILAENV is to set the block size used by DPOTRF. As described above the four Fortran routines DPOTF3 are a new type of Level-3 BLAS called FACTOR BLAS.

Table 1 contain comparison numbers in Mflop/s. There are results for five computers inside the table: SUN UltraSPARC IV+, SGI - Intel Itanium2, Intel Xeon, AMD Dual Core Opteron, and Intel Xeon Quad Core.

The table has thirteen columns. The first column shows the matrix order. The second column contains results for the vendor optimized Cholesky routine DPOTRF and the third column has results for the Recursive Algorithm [1].

The columns from four to thirteen contain results when DPOTF2 and the four Fortran routines DPOTF3 are used within DPOTRF and are used by themselves. In column 4, DPOTF2 is called by DPOTRF, and in columns 6, 8, 10, 12 the four DPOTF3 routines are called by DPOTRF. We now denote these four routines by suffixes a,b,c,d. In column 5 DPOTF2 is used alone. In columns 7, 9, 11, 13 routines DPOTF3(a,b,c,d) are used alone.

There are five Fortran routines used in this study besides DPOTRF:

1. The LAPACK routine DPOTF2: The fourth and fifth columns have results of using routine DPOTRF to call DPOTF2 and routine DPOTF2 directly: these results are tabulated in the fourth and fifth columns respectively.
2. The 2×2 blocking routine DPOTF3a specialized for the operation FMA $(a \times b + c)$ using seven floating point (fp) registers (this 2×2 blocking DPOTF3a routine replaces routine DPOTF2): these results are tabulated in the sixth and seventh columns respectively.
3. The 1×4 blocking routine DPOTF3b is optimized for the case $\mathrm{mod}(n, 4) = 0$ where n is the matrix order. It uses eight fp registers. This 1×4 blocking routine DPOTF3b replaces routine DPOTF2: these results are tabulated in the eighth and ninth columns respectively.
4. The 2×4 blocking routine DPOTF3c uses fourteen fp registers. This 2×4 blocking routine DPOTF3c replaces routine DPOTF2: these results are tabulated in the tenth and eleventh columns respectively.
5. The 2×2 blocking routine DPOTF3d . It is not specialized for the FMA operation and uses six fp registers. This 2×2 blocking routine DPOTF3d replaces DPOTF2: these performance results are tabulated in the twelfth and thirteenth columns respectively.

Before continuing, we note that Level-3 BLAS will only be called in columns 4, 6, 8, 10, 12 for block size 100. This is because ILAENV has set the block size to be 64 in our study. Hence, Level-3 BLAS only have effect on our performance study in these five columns.

It can be seen that the DPOTF3c code with submatrix blocks of size 2×4, see column eleven, is remarkably successful for the Sun (Newton), SGI (Freke), and Quad Core Xeon (Zoot) computers. For all these three platforms, it significantly outperforms the compiled LAPACK code and the recursive algorithm. It outperforms the vendor's optimized codes. The 2×2 DPOTF3d code in column thirteen, not prepared for the FMA operation, is superior on the Intel Xeon

(Battle) computer. The 1×4 DPOTF3b in column nine is superior on the Dual Core AMD (Nala) platform. All the superior results are colored in red.

These performance numbers reveal a significant innovation about the use of Level-3 Fortran DPOTF3a,b,c,d codes over use of Level-2 LAPACK DPOTF2 code. We demonstrate why in the next two paragraphs.

The results of columns 10 and 11 are about the same for $n = 40$ and $n = 64$. For column 10 some additional work is done. DPOTRF calls ILAENV which sets $nb = 64$. It then calls DPOTF3c and returns after DPOTF3c completes. For column 11 only DPOTF3c is called. Hence column 10 takes slightly more time than column 11. However, in column 10, for $n = 100$ DPOTRF via calling ILAENV still sets $nb = 64$ and then DPOTRF does a Level-3 blocked computation. For example, take $nb = 100$. With $nb = 64$ DPOTRF does a sub blocking of nb sizes equal to 64 and 36. Thus, DPOTRF calls Factor(64), DTRSM(64,36), DSYRK(36,64), and Factor(36) before it returns. The two Factor calls are to the DPOTF3c routine. However, in column 11, DPOTF3c is called only once with $nb = 100$. In columns ten and eleven performance is always increasing over doing the Level-3 blocked computation of DPOTRF. This means the DPOTF3c routine is outperforming DTRSM and DSYRK.

Now, take columns four and five. For $n = 40$ and $n = 64$ the results are again about equal for the reasons cited above. For $n = 100$ the results favor DPOTRF with Level-3 blocking except for the Zoot platform. The DPOTF2 performance is decreasing relative to the blocked computation as n increases from 64 to 100. The opposite result is true for most of the columns six to thirteen, namely DPOTF3a,b,c,d performance is increasing relative to the blocked computation as n increases from 64 to 100. For Zoot in column 5 DPOTF2 is 4 way parallel. This is the reason why it is outperforming LAPACK blocked at n=100.

An essential conclusion is that the faster four Level-3 DPOTF3 Fortran routines really help to increase performance. Here is why. Take any n for DPOTRF. DPOTRF can choose a larger block size nb and it will do a blocked computation with this block size. All four subroutines of DPOTRF will perform better over calling DPOTRF with a smaller block size which must be the case if DPOTF2 is used instead. The paper [2] gives large n performance results for BPHF where nb was set larger than 64. The results for $nb = 100$ were much better. The above explanation explains why this had to be the case.

These results emphasize that LAPACK users should use ILAENV to set nb based on the speeds of Factorization, DTRSM, DSYRK and DGEMM. This information is part of the LAPACK User's guide but many users do not do this finer tuning. The code for the 1×4 DPOTF3b subroutine is available from the companion paper [13],[Algorithm 865]. The code for _POTRF and its subroutines is available from the LAPACK package [3].

4 Conclusions

We have shown that four simple Fortran codes DPOTF3i produce Level-3 Cholesky factorization routines that perform better than the Level-2 LAPACK

DPOTF2 routine. We have also shown that their use enables LAPACK routine DPOTRF to increase its block size nb. Since nb is the k dimension of the _GEMM, _SYRK and _TRSM Level-3 BLAS, their SMP performance will improve and hence the overall performance of SMP _POTRF will improve. We have provided explanations and verified them with performance studies. Our results corroborate results that were observed by [2, 20]. It was seen that DBPTRF performance was less sensitive to the choice of one nb for an entire range of n values. For DPOTRF using DPOTF2 one needed to increase nb as n increased for optimal performance whereas for DBPTRF using DPOTF3i usually a single nb value gave uniformly good performance.

We used BPF format in this paper. It is a generalization of standard packed format. We discussed lower BPF format which consisted of $N = n/nb$ rectangular blocks whose LDA's were $n = j \cdot nb$ for $0 \leq j < N$. We showed that upper packed format had the additional property that its rectangular blocks were really a multiple number of $i = N - j$ square blocks for rectangle j. In all there are $N(N+1)/2$ SB. We gave LAPACK _POTRF and _PPTRF algorithms using BPF and showed that these codes were trivial modifications of current _POTRF algorithms. In the multicore era it appears that SB format will be the data layout of choice. Thus, we think that for upper BPF format the current Cell implementations of [19] will carry over with trivial modifications.

Another purpose of our paper is to promote the new **Block Packed Data Format** storage or variants thereof; see Section 1.1. BPF algorithms are variants of the BPHF algorithm and they use slightly more computer memory than $n \times (n+1)/2$ matrix elements. They usually perform better or equal to the full format storage algorithms. The full format algorithms require additional storage of $(n-1) \times n/2$ matrix elements in the computer memory but never reference these elements. Finally, full format algorithms and their related Level-3 BLAS are no longer being used on multi-core processors. For symmetric and triangular matrices the format of choice is SBPF which is the same as upper BPF.

Acknowledgments. We would like to thank Bernd Dammann, Niels Carl W. Hansen, Paul Peltz and Barry Britt for consulting help on the systems used in our experiments. Thanks also go to the Ministerio de Educación y Ciencia of Spain for funding project grant TIN2007-60625.

References

1. Andersen, B.S., Gustavson, F.G., Waśniewski, J.: A Recursive Formulation of Cholesky Factorization of a Matrix in Packed Storage. ACM TOMS 27(2), 214–244 (2001)
2. Andersen, B.S., Gunnels, J.A., Gustavson, F.G., Reid, J.K., Waśniewski, J.: A Fully Portable High Performance Minimal Storage Hybrid Cholesky Algorithm. ACM TOMS 31(2), 201–227 (2005)
3. Anderson, E., et al.: LAPACK Users' Guide Release 3.0. SIAM, Philadelphia (1999)
4. D'Azevedo, E., Dongarra, J.J.: Packed storage extension of ScaLAPACK. ORNL Report 6190, Oak Ridge National Laboratory, 13 pages (May 1998)

5. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: Set of Level 3 Basic Linear Algebra Subprograms. TOMS 16(1), 1–17 (1990)
6. Elmroth, E., Gustavson, F.G., Jonsson, I., Kågström, B.: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM Review 46(1), 3–45 (2004)
7. Gunnels, J.A., Gustavson, F.G., Pingali, K.K., Yotov, K.: Is Cache-Oblivious DGEMM Viable? In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 919–928. Springer, Heidelberg (2007)
8. Gustavson, F.G.: Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. IBM J. R. & D 41(6), 737–755 (1997)
9. Gustavson, F.G., Jonsson, I.: Minimal Storage High Performance Cholesky via Blocking and Recursion. IBM J. R. & D 44(6), 823–849 (2000)
10. Gustavson, F.G.: New Generalized Data Structures for Matrices Lead to a Variety of High-Performance Algorithms. In: Boisvert, R.F., Tang, P.T.P. (eds.) Proceedings of the IFIP WG 2.5 Working Group on The Architecture of Scientific Software, Ottawa, Canada, October 2-4, pp. 211–234. Kluwer Academic Pub. (2000)
11. Gustavson, F.G.: High Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices. IBM J. R. & D 47(1), 31–55 (2003)
12. Gustavson, F.G., Gunnels, J., Sexton, J.: Minimal Data Copy For Dense Linear Algebra Factorization. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 540–549. Springer, Heidelberg (2007)
13. Gustavson, F.G., Reid, J.K., Waśniewski, J.: Algorithm 865: Fortran 95 Subroutines for Cholesky Factorization in Blocked Hybrid Format. ACM TOMS 33(1), 5 pages (2007)
14. Gustavson, F.G.: Cache Blocking. In: Jónasson, K. (ed.) PARA 2010, Part I. LNCS, vol. 7133, pp. 22–32. Springer, Heidelberg (2012)
15. Gustavson, F.G., Karlsson, L., Kågström, B.: Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion. ACM TOMS, 34 pages (to appear, 2012)
16. Herrero, J.R., Navarro, J.J.: Compiler-Optimized Kernels: An Efficient Alternative to Hand-Coded Inner Kernels. In: Gavrilova, M.L., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganá, A., Mun, Y., Choo, H. (eds.) ICCSA 2006. LNCS, vol. 3984, pp. 762–771. Springer, Heidelberg (2006)
17. Herrero, J.R.: New Data Structures for Matrices and Specialized Inner Kernels: Low Overhead for High Performance. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 659–667. Springer, Heidelberg (2008)
18. Knuth, D.: The Art of Computer Programming, 3rd edn., vol. 1&2. Addison-Wesley
19. Kurzak, J., Buttari, A., Dongarra, J.: Solving systems of Linear Equations on the Cell Processor using Cholesky Factorization. IEEE Trans. Parallel Distrib. Syst. 19(9), 1175–1186 (2008)
20. Whaley, C.: Empirically tuning LAPACK's blocking factor for increased performance. In: Proc. of the Conf. on Computer Aspects of Numerical Algs., 8 pages (2008)

Parallel Preconditioner for Nonconforming Adini Discretization of a Plate Problem on Nonconforming Meshes

Leszek Marcinkowski*

Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, Banacha 2, 02-097 Warszawa, Poland
L.Marcinkowski@mimuw.edu.pl

Abstract. In this paper we present a domain decomposition parallel preconditioner for a discretization of a plate problem on nonconforming meshes in 2D. The local discretizations are Adini nonconforming plate finite elements. On the interfaces between adjacent subdomains two mortar conditions are imposed. The condition number of the preconditioned problem is almost optimal i.e. it is bounded poly-logarithmically with respect to the mesh parameters.

1 Introduction

Many physical phenomena or technical problems are modeled by differential equations. Usually while constructing a discrete problem approximating the original differential equation a uniform global mesh is utilized. However quite often it is required to construct an approximation which locally in subdomains uses different types of independent methods, or meshes. A mortar method enables us to use independent meshes or discretization methods in nonoverlapping subdomains. For general presentation of a mortar method we refer to [1], [2], and [3]. We should mention that an alternative discretization method on nonconforming meshes is C^0 interior penalty method, cf. [4–6].

In this paper we focus on a parallel domain decomposition preconditioner for solving the system of equations arising from the mortar finite element discretization of a model plate problem which in subdomains uses a rectangular nonconforming Adini element, cf. [7]. Our method is based on an abstract Additive Schwarz (ASM) framework, e.g. cf. [8], i.e. we first introduce decomposition of the discrete space into a coarse space and subspaces related to interfaces. Then an application of our parallel preconditioner to a vector is based on solving local independent problems associated with this decomposition.

There are many effective domain decomposition algorithms for solving mortar discretization of second order problems, cf. e.g. [9], [10], [11], [12], [13], [14] [15], [16] and references therein. There are much smaller number of investigations devoted to effective solvers for mortar approximation of fourth order problems,

* This work was partially supported by Polish Scientific Grant: N N201 0069 33.

cf. e.g. [17], [18], [19], [20] and [21]. However to our knowledge there are no results concerning domain decomposition parallel preconditioners for mortar Adini finite element discretizations in the literature.

The remainder of this paper is organized as follows: in Section 2 we discuss the mortar discretization with local Adini finite element spaces in subdomains, Section 3 is devoted to construction of our parallel preconditioner, and in Section 4 we present condition estimates for the preconditioned problem.

2 Discrete Problem

In this section we introduce our model plate problem and its discretization on nonmatching meshes by the mortar method with the nonconforming Adini finite element, e.g. cf. [7]. Let Ω be a polygonal domain on the plane which is a sum

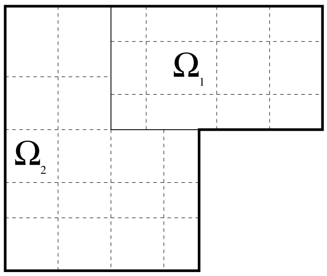


Fig. 1. L-shape domain. Nonconforming meshes in two subdomains.

of rectangles with the edges parallel to the axes. Our model differential clamped plate problem is to find $u^* \in H_0^2(\Omega)$ such that

$$a(u^*, v) = \int_\Omega fv\,dx \quad \forall v \in H_0^2(\Omega), \tag{1}$$

where u^* can be interpreted as the displacement and $f \in L^2(\Omega)$ as the body force, and

$$a(u,v) := \int_\Omega [\triangle u \triangle v + \\ +(1-\nu)\left(2\frac{\partial^2 u}{\partial x_1 \partial x_2}\frac{\partial^2 v}{\partial x_1 \partial x_2} - \frac{\partial^2 u}{\partial x_1 \partial x_1}\frac{\partial^2 v}{\partial x_2 \partial x_2} - \frac{\partial^2 u}{\partial x_2 \partial x_2}\frac{\partial^2 v}{\partial x_1 \partial x_1}\right)\Big]\,dx.$$

Here $H_0^2(\Omega) := \{v \in H^2(\Omega) :\ v = \partial_n v = 0 \ \text{ on } \ \partial\Omega\}$, and ∂_n is the normal unit derivative outward to $\partial\Omega$. The Poisson ratio ν satisfies $0 < \nu < 1/2$.

We assume that we have the following decomposition of Ω into disjoint subdomains that are either rectangles or are a sum of rectangles with edges parallel to the axes, e.g. L-shaped subdomains, cf. Figure 1:

$$\overline{\Omega} = \bigcup_{k=1}^{N} \overline{\Omega}_k$$

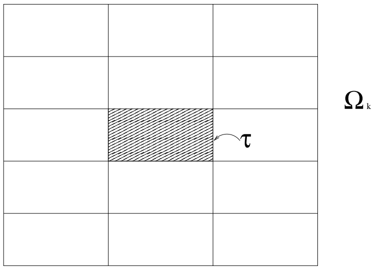


Fig. 2. Rectangular triangulation $T_h(\Omega_k)$

with

$$\partial\Omega_k \cap \partial\Omega_l = \begin{cases} \emptyset, & \\ \overline{\Gamma}_{kl} & \text{a common edge,} \\ v & \text{a common vertex.} \end{cases}$$

The decomposition forms the coarse triangulation of Ω and we assume the shape regularity of it in the sense of Section 2, p.5 in [22]. An important role is played by the global interface: $\Gamma := \bigcup_{k=1}^{N}(\partial\Omega_k \setminus \partial\Omega)$ being the sum of all edges which are not on the boundary of Ω. Let $H_k := \operatorname{diam}(\Omega_k)$.

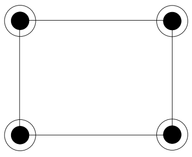


Fig. 3. Adini element

In each subdomain Ω_k we introduce an independent rectangular quasi-uniform triangulation $T_h(\Omega_k)$, cf. Figure 2, with $h_k := \max_{\tau\in T_h(\Omega_k)} \operatorname{diam}\tau$, e.g. cf. [23]. For $\Omega_k, \overline{\Omega}_k, \partial\Omega_k, \Gamma_{kl}$ let us define $\Omega_{k,h}, \overline{\Omega}_{k,h}, \partial\Omega_{k,h}, \Gamma^k_{kl,h}$ as the sets of all nodal points (nodes), i.e. of all vertices of elements of $T_h(\Omega_k)$ which are in the respective set, e.g. $\partial\Omega_{k,h}$ is the set of all nodal points from $T_h(\Omega_k)$ which are on $\partial\Omega_k$.

The local Adini finite element space $X_h(\Omega_k)$ is defined as follows, cf. Ch. 7, §49 in [7]:

$$\begin{aligned} X_h(\Omega_k) := \{v \in L^2(\Omega_k):\ & v_{|\tau} \in P_3(\tau) \oplus \operatorname{span}\{x_1^3 x_2, x_1 x_2^3\} \quad \text{for} \quad \tau \in T_h(\Omega_k), \\ & v, v_{x_1}, v_{x_2} \ \text{continuous at the nodal points of} \ \overline{\Omega}_{\mathrm{k,h}}, \\ & \text{and} \ v(p) = \frac{\partial v}{\partial x_1}(p) = \frac{\partial v}{\partial x_2}(p) = 0 \quad \text{for} \quad p \in \partial\Omega_k \cap \partial\Omega\}, \end{aligned}$$

where $\tau \in T_h(\Omega_k)$ is an rectangular element and $P_3(\tau)$ is the space of cubic polynomials on τ, cf. Figures 2 and 3.

Note that $X_h(\Omega_k) \not\subset H^2(\Omega_k)$ and in that sense it is a nonconforming finite element space, but $X_h(\Omega_k) \subset H^1(\Omega_k)$, cf. [7].

There are three degrees of freedom of the Adini finite element function $v \in X_h(\Omega_k)$ at a vertex q of an element τ, cf. Figure 3:

$$\left\{ v(q), \frac{\partial v}{\partial x_1}(q), \frac{\partial v}{\partial x_2}(q) \right\}.$$

We also introduce an auxiliary global space:

$$X_h(\Omega) := \Pi_{k=1}^N X_h(\Omega_k),$$

local bilinear forms for $u, v \in X_k(\Omega_k)$:

$$a_{h,k}(u,v) := \textstyle\sum_{\tau \in T_h(\Omega_k)} \int_\tau [\triangle u \triangle v + \\ +(1-\nu)\left(2\frac{\partial^2 u}{\partial x_1 \partial x_2}\frac{\partial^2 v}{\partial x_1 \partial x_2} - \frac{\partial^2 u}{\partial x_1 \partial x_1}\frac{\partial^2 v}{\partial x_2 \partial x_2} - \frac{\partial^2 u}{\partial x_2 \partial x_2}\frac{\partial^2 v}{\partial x_1 \partial x_1}\right)\Big]\, dx$$

and a global form for $u = (u_k)_{k=1}^N, v = (v_k)_{k=1}^N \in X_h(\Omega)$ being a sum of local forms:

$$a_H(u,v) := \sum_{k=1}^N a_{h,k}(u_k, v_k).$$

Note that each interface $\Gamma_{kl} \subset \Gamma$ which is the open common edge of two neighboring subdomains Ω_k and Ω_l inherits two independent one-dimensional triangulations from respective two dimensional triangulations of these subdomains, i.e. the h_k-one denoted further as $T_h^k(\Gamma_{kl})$ from $T_h(\Omega_k)$ and the h_l triangulation $T_h^l(\Gamma_{lk})$ obtained from $T_h(\Omega_l)$, cf. Figure 1. Thus for each interface $\Gamma_{kl} = \partial\Omega_k \cap \partial\Omega_l$, we distinguish between two sides of this edge. We introduce one side as a master (mortar) denoted by $\gamma_{kl} \subset \partial\Omega_k$ and the second one as a slave (non-mortar) $\delta_{lk} \subset \partial\Omega_l$ if $h_k \le h_l$. Naturally, both γ_{kl} and δ_{lk} geometrically share the same position, but have different meshes $T_h^k(\gamma_{kl}) := T_h^k(\Gamma_{kl})$ and $T_h^l(\delta_{lk}) := T_h^l(\Gamma_{lk})$, respectively. Finally, we introduce $\gamma_{kl,h} := \Gamma_{kl,h}^k$, i.e. the set of all nodal points (nodes) of $T_h^k(\Gamma_{kl})$ which are on γ_{kl} and $\delta_{lk,h} := \Gamma_{lk,h}^l$ as the set of all nodes of $T_h^l(\Gamma_{lk})$ which are on δ_{lk}.

We introduce additional test spaces on each slave (nonmortar) δ_{lk}. Let the first one denoted by $M_t^{h_l}(\delta_{lk})$ be the space formed by C^1 smooth functions that are piecewise cubic on $T_h^l(\delta_{lk})$ except for two elements, that touch the ends of slave, where they are piecewise linear and let $M_n^{h_l}(\delta_{lk})$ be the space formed by continuous piecewise linear functions on the h_l triangulation of δ_{lk}, which are piecewise constant on two elements which touch the ends of this slave.

We say that $u_k \in X_h(\Omega_k)$ and $u_l \in X_h(\Omega_l)$ for $\partial\Omega_l \cap \partial\Omega_k = \Gamma_{kl}$, satisfy the mortar conditions if

$$\int_{\delta_{lk}} (u_k - u_l)\psi\, ds = 0 \qquad \forall \psi \in M_t^{h_l}(\delta_{lk}), \tag{2}$$

$$\int_{\delta_{lk}} (I_{h_k}\partial_n u_k - I_{h_l}\partial_n u_l)\phi\, ds = 0 \qquad \forall \phi \in M_n^{h_l}(\delta_{lk}), \tag{3}$$

where $I_{h_s}\partial_n u_s$ is the continuous piecewise linear function interpolating $\partial_n u_s$ at nodal points of $\Gamma_{kl,h}^s$ for $s = k, l$.

The discrete space V^h is defined as the subspace of $X_h(\Omega)$ formed by functions which satisfy the mortar conditions (2) and (3) and are continuous at all crosspoints. Continuity of $u = (u_j)_{j=1}^N \in V^h$ at crosspoints means that $u_k(c_r) = u_l(c_r)$ if c_r is a crosspoint which is a common vertex of Ω_k and Ω_l. Note that the gradient ∇u is not continuous at crosspoints for $u \in V^h$ in general.

We can now introduce a discrete problem: Find $u_h^* \in V^h$ such that

$$a_H(u_h^*, v) = \int_\Omega f v \, dx \quad \forall v \in V^h. \tag{4}$$

We see that $a_H(u,u) = 0$ implies that u is a linear polynomial in all elements of $T_h(\Omega_k)$, then from the continuity of u, u_{x_1}, u_{x_2} at all vertices of $\Omega_{k,h}$ follows that u linear in Ω_k and from the mortar condition follows that u linear in Ω. Finally, the boundary conditions yield $u = 0$. Hence the form $a_H(\cdot,\cdot)$ is positive definite over V^h and we have

Proposition 1. *The problem (4) has a unique solution.*

We introduce a standard nodal basis of Adini finite element space, i.e. with each degree of freedom at all nodal points which are not in $\partial\Omega$ or in any slave δ_{ji}, we associate one nodal basis function which has this degree of freedom equal to one, and all remaining degrees of freedom are equal to zero. Then we can rewrite the discrete problem (4) into a system of linear equations. If we additionally assume that $h_i \asymp h_j$, for an interface $\overline{\Gamma}_{ij} = \partial\Omega_i \cap \partial\Omega_j$, we can see that the condition number of the resulting matrix is bounded by $C\underline{h}^{-4}$, where $\underline{h} = \inf_k h_k$ and C is a positive constant independent of any h_k and the number of subdomains.

3 Additive Schwarz Method Preconditioner

In this section, we present a preconditioner for solving the system of equations arising from the discrete problem (4) in the abstract terms of the Additive Schwarz Method (ASM), e.g. see Ch. 2 in [8].

We first decompose locally each function $u \in X_h(\Omega_i)$ into two parts orthogonal in terms of $a_{i,h}(\cdot,\cdot)$ as follows

$$u = \mathcal{P}_i u + \mathcal{H}_i u,$$

where the orthogonal projection $\mathcal{P}_i u \in X_{0,h}(\Omega_i)$ is defined by

$$a_{h,i}(\mathcal{P}_i u, v) = a_{h,i}(u, v) \quad \forall v \in X_{0,h}(\Omega_i),$$

where

$$X_{0,h}(\Omega_i) = \{u \in X_h(\Omega_i) : \ u(p) = \nabla u(p) = 0 \qquad \forall p \in \partial\Omega_{i,h}\}.$$

The discrete biharmonic part $\mathcal{H}_i u := u - \mathcal{P}_i u \in X_h(\Omega_i)$ satisfies

$$\begin{cases} a_{i,h}(\mathcal{H}_i u, v) = 0 & \forall v \in X_{0,h}(\Omega_i), \\ \mathcal{H}_i u(q) = u(q) & \forall q \in \partial\Omega_{i,h}, \\ \nabla\mathcal{H}_i u(p) = \nabla u(p) & \forall p \in \partial\Omega_{i,h}, \end{cases}$$

We also introduce the global decomposition of $u \in V^h$ into $u = \mathcal{P}u + \mathcal{H}u$ with

$$\mathcal{P}u := (\mathcal{P}_1 u, \ldots, \mathcal{P}_i u, \ldots, \mathcal{P}_N u), \qquad \mathcal{H}u := (\mathcal{H}_1 u, \ldots, \mathcal{H}_i u, \ldots, \mathcal{H}_N u).$$

We next define $W = \{\mathcal{H}u : u \in V^h\}$, i.e. the subspace of discrete biharmonic functions of V^h. Note that $u \in W$ is uniquely defined by the values of three degrees of freedom associated with all vertices of subdomains and with all nodes on mortars. We can decompose u_h^*, the solution of (4), into $u_h^* = \mathcal{P}u_h^* + \mathcal{H}u_h^*$. $\mathcal{P}u_h^*$ can be computed by solving N independent local problems. The discrete biharmonic part of u_h^* further denoted by $\tilde{u}_h^* := \mathcal{H}u_h^*$ is the solution of the following variational discrete problem

$$a_H(\tilde{u}_h^*, v) = f(v) \qquad \forall v \in W. \tag{5}$$

Rewriting this problem into matrix form in the standard nodal basis we get a Schur complement problem, e.g. cf. [8]:

$$S\tilde{\boldsymbol{u}}^* = \tilde{\boldsymbol{f}}, \tag{6}$$

where $\tilde{\boldsymbol{u}}^*$ is a vector with all degrees of freedom of $\tilde{u}_h^*$ associated with vertices of subdomains and all nodal points on mortars.

We describe ASM for solving this problem in terms of decomposition of the space W into several subspaces. Let us define a coarse space:

$$\begin{aligned} W_0 := \{u \in W : \; & v_{k|\gamma_{kl}} = v_{l|\delta_{lk}} \in P_3(\Gamma_{kl}), \\ & I_{h_k}\partial_n v_{k|\gamma_{kl}} = I_{h_l}\partial_n v_{l|\delta_{lk}} \in P_1(\Gamma_{kl}) \quad \text{and} \\ & v, \nabla v \;\; \text{are} \;\; \text{continuous} \;\; \text{at} \;\; \text{all} \;\; \text{crosspoints}\}. \end{aligned}$$

Here $P_1(\Gamma_{kl})$ is the space of linear polynomials over Γ_{kl} and I_{h_k}, I_{h_l} are piecewise linear interpolants defined over h_k and h_l meshes of master $\gamma_{kl} = \Gamma_{kl}$ and slave δ_{lk}, respectively. Note that $u \in W_0$ is uniquely defined by the value of u and ∇u at all crosspoints. Thus dimension of W_0 is equal to three times number of crosspoints. We next define $W_{x,s}$, a one-dimensional space associated with and a vertex $x \in \mathcal{V}(\Omega_k)$ and $s \in \{1, 2\}$. Here $\mathcal{V}(\Omega_k)$ is the set of the vertices of $\partial\Omega_k$ which are not on $\partial\Omega$. We distinguish between vertices of different substructures even if they occupies geometrically the position of the same crosspoint as two of three degrees of freedom of $u \in W$ at a crosspoint may be discontinuous, namely, the ones related to derivatives. Let us also introduce

$$\mathcal{V} := \bigcup_{k=1}^{N} \mathcal{V}(\Omega_k).$$

We define $W_{x,s} = \mathrm{span}\{\phi_{x,s}\}$ for $s = 1, 2$, where $\phi_{x,s} \in W$ is the locally discrete biharmonic vertex function associated with degree of freedom corresponding to x_s derivative, and a vertex $x \in \mathcal{V}$, i.e. it satisfies:

$$\frac{\partial \phi_{x,s}}{\partial x_r}(y) = \begin{cases} 1 & r = s, y = x, \\ 0 & \text{otherwise}, \end{cases} \qquad \forall\, y \in \bigcup_{\gamma_{ij} \subset \Gamma} \gamma_{ij,h} \cup \mathcal{V}, \quad r = 1, 2.$$

We next define subspaces associated with masters. Let γ_{kl} be a master and δ_{lk} its associated slave. Then let W_{kl} be a space formed by all functions $v \in W$ which has all degrees of freedom which are not associated with a nodal point in $\gamma_{kl,h}$ equal to zero, i.e.

$$W_{kl} = \{v \in W : v(x) = \frac{\partial v}{\partial x_1}(x) = \frac{\partial v}{\partial x_2}(x) = 0 \text{ for } x \in \left\{ \bigcup_{\gamma_{ij} \subset \Gamma} \gamma_{ij,h} \cup \mathcal{V} \right\} \setminus \gamma_{kl,h}\}.$$

Note that W_{kl} contains functions which have nonzero degrees of freedom only at nodes of $\gamma_{kl,h}$ and by the mortar conditions at ones of $\delta_{lk,h}$, and in $\Omega_{k,h}$ and $\Omega_{l,h}$.

We have the following decomposition of W:

$$W = W_0 + \sum_{\gamma_{kl} \subset \Gamma} W_{kl} + \sum_{x \in \mathcal{V}} \sum_{s=1,2} W_{x,s}.$$

Let $T_0 : W \to W_0$, $T_{kl} : W \to W_{kl}$ and $T_{x,s} : W \to W_{x,s}$ be orthogonal projections in terms of $a_H(\cdot, \cdot)$ onto W_0, W_{kl}, and $W_{x,s}$, respectively.

Then the operator $T : W \to W$ is defined as:

$$T := T_0 + \sum_{\gamma_{ij} \subset \Gamma} T_{ij} + \sum_{x \in \mathcal{V}} \sum_{s=1,2} T_{x,s}.$$

Next we replace problem (5) by a new one:

$$T(\tilde{u}_h^*) = g, \tag{7}$$

where

$$g := g_0 + \sum_{\gamma_{kl} \subset \Gamma} g_{kl} + \sum_{x \in \mathcal{V}} \sum_{s=1,2} g_{x,s}$$

and $g_0 = T_0 \tilde{u}_h^*$, $g_{kl} = T_{kl} \tilde{u}_h^*$ and $g_{x,s} = T_{x,s} \tilde{u}_h^*$.

Note that if we rewrite (7) into matrix form, then (7) takes the form of the preconditioned system (6), e.g. cf. [8]:

$$M^{-1} S \tilde{\boldsymbol{u}}^* = M^{-1} \tilde{\boldsymbol{f}}, \tag{8}$$

where S is the Schur complement matrix from (6) and M^{-1} is a parallel preconditioner of ASM type.

3.1 Implementation

We briefly discuss implementation some issues. In practice to solve (7) the CG iterative method is used, but for the simplicity of presentation we describe the ideas of implementation on the basis of the Richardson iterative method with a positive parameter τ: take any $u^{(0)}$ and iterate until convergence:

$$u^{(n+1)} := u^{(n)} - \tau\,(T(u^{(n)}) - g) = u^{(n)} - \tau\,T(u^{(n)} - \tilde{u}_h^*) = u^{(n)} - \tau\,r^{(n)} \quad n \geq 0.$$

To compute $r^{(n)} = T(u^{(n)}) - g$ we have to solve the following problems:

– compute $r_0 \in W_0$ such that

$$a_H(r_0, v) = a_H(T_0(u^{(n)} - \tilde{u}_h^*), v) = a_H(u^{(n)}, v) - f(v) \qquad \forall v \in W_0,$$

– compute $r_{kl} \in W_{kl}$ for all $\gamma_{kl} \subset \Gamma$ such that

$$a_H(r_{kl}, v) = a_H(T_{kl}(u^{(n)} - \tilde{u}_h^*), v) = a_H(u^{(n)}, v) - f(v) \qquad \forall v \in W_{kl},$$

– compute $r_{x,s} \in W_{x,s}$ for all $x \in \mathcal{V}$ and $s = 1, 2$ such that

$$a_H(r_{x,s}, v) = a_H(T_{x,s}(u^{(n)} - \tilde{u}_h^*), v) = a_H(u^{(n)}, v) - f(v) \qquad \forall v \in W_{x,s}.$$

Then $r^{(n)} = r_0 + \sum_{\gamma_{kl} \subset \Gamma} r_{kl} + \sum_{x \in \mathcal{V}} \sum_{s=1,2} r_{x,s}$. Note that all these problems are independent so they can be solved in parallel.

4 Condition Number Estimates

We now state the main theoretical result of this paper.

Theorem 1. *It holds that*

$$C_1\,(1 + \log(H/\underline{h}))^{-2} a_H(u, u) \leq a_H(Tu, u) \leq C_2\,a_H(u, u) \qquad \forall w \in W,$$

where C_1, C_2 are positive constants independent of mesh parameters and the number of subdomains, $\underline{h} = \min_k h_k$ and $H = \max_k H_k$.

From this theorem we get a corollary:

Corollary 1. *The condition number of $M^{-1}S$ from (8) is bounded by $C\,(1 + \log(H/\underline{h}))^2$, where C is positive constants independent of mesh parameters and the number of subdomains.*

Sketch of the Proof of Theorem 1
The proof is based on the abstract theory of ASM. We have to check three main assumptions, e.g. cf. § 2.3 in [8].

Here the assumption II (Local Stability, cf. Ass. 2.4 in [8]) is satisfied with a constant equal to one as $T_0, T_{kl}, T_{x,s}$ are orthogonal projection in terms of the form $a_H(\cdot, \cdot)$ and assumption III (Strengthened Cauchy-Schwarz inequalities cf. Ass. 2.3 in [8]) is satisfied with a constant independent of h_k and H by a standard coloring argument.

The assumption I (Stable decomposition, cf. Ass. 2.1 in [8]) requires that there exists a constant C such that for any $u \in W$ there is a decomposition:

$$u = u_0 + \sum_{\gamma_{kl} \subset \Gamma} u_{kl} + \sum_{x \in \mathcal{V}} \sum_{s=1,2} u_{x,s} \tag{9}$$

with $u_0 \in W_0, u_{kl} \in W_{kl}$, and $u_{x,s} \in W_{x,s}$ such that

$$a_H(u_0, u_0) + \sum_{\gamma_{kl} \subset \Gamma} a_H(u_{kl}, u_{kl}) + \sum_{x \in \mathcal{V}} \sum_{s=1,2} a_H(u_{x,s}, u_{x,s}) \le C_0^2 a_H(u, u). \quad (10)$$

with $C_0^2 = C(1 + \log(H/\underline{h})^2$.

We first define u_0 as a unique function in W_0 such that $u_0(c_r) = u(c_r)$ and $\nabla u_0(c_r) = \sum_{\{k: c_r \in \partial\Omega_k\}} (1/N_{c_r}) \nabla u_k(x)$ for any crosspoint c_r, where subscripts k in the last sum are taken over all substructures Ω_k which has c_r as its vertex and N_{c_r} is the number of such substructures. Then introducing $w = u - u_0$ we define $u_{x,s} := \frac{\partial w}{\partial x_s}(x)\phi_{x,s}$ for $s = 1, 2$ of length and $x \in \mathcal{V}$ and $u_{kl} \in W_{kl}$ for $\gamma_{kl} \subset \Gamma$ such that $u_{kl}(p) = w(p)$, $\frac{\partial u_{kl}}{\partial x_s}(p) = \frac{\partial w}{\partial x_s}(p)$ for $s = 1, 2$ and $p \in \gamma_{kl,h}$.

We see that (9) is satisfied and we skip the technical proof of (10). Finally, utilizing Theorem 2.7 in [8] we end the proof of Theorem 1.

Summary

In this paper we present an efficient parallel preconditioner for Adini discretization of a model clamped plate problem. The condition estimates for the preconditioned problem are almost optimal.

References

1. Bernardi, C., Maday, Y., Patera, A.T.: A new nonconforming approach to domain decomposition: the mortar element method. In: Nonlinear Partial Differential Equations and their Applications. Collège de France Seminar, vol. XI (Paris, 1989–1991). Pitman Res. Notes Math. Ser., vol. 299, pp. 13–51. Longman Sci. Tech., Harlow (1994)
2. Ben Belgacem, F.: The mortar finite element method with Lagrange multipliers. Numer. Math. 84(2), 173–197 (1999); First published as a technical report in 1994
3. Ben Belgacem, F., Maday, Y.: The mortar element method for three-dimensional finite elements. RAIRO Modél. Math. Anal. Numér. 31(2), 289–302 (1997)
4. Brenner, S.C., Sung, L.Y.: C^0 interior penalty methods for fourth order elliptic boundary value problems on polygonal domains. J. Sci. Comput. 22/23, 83–118 (2005)
5. Brenner, S.C., Wang, K.: Two-level additive Schwarz preconditioners for C^0 interior penalty methods. Numer. Math. 102(2), 231–255 (2005)
6. Brenner, S.C., Sung, L.Y.: Multigrid algorithms for C^0 interior penalty methods. SIAM J. Numer. Anal. 44(1), 199–223 (2006)
7. Ciarlet, P.G.: Basic error estimates for elliptic problems. In: Handbook of Numerical Analysis, vol. II, pp. 17–351. North-Holland, Amsterdam (1991)
8. Toselli, A., Widlund, O.: Domain decomposition methods—algorithms and theory. Springer Series in Computational Mathematics, vol. 34. Springer, Berlin (2005)
9. Achdou, Y., Kuznetsov, Y.A.: Substructuring preconditioners for finite element methods on nonmatching grids. East-West J. Numer. Math. 3(1), 1–28 (1995)
10. Achdou, Y., Maday, Y., Widlund, O.B.: Iterative substructuring preconditioners for mortar element methods in two dimensions. SIAM J. Numer. Anal. 36(2), 551–580 (1999)

11. Bjørstad, P.E., Dryja, M., Rahman, T.: Additive Schwarz methods for elliptic mortar finite element problems. Numer. Math. 95(3), 427–457 (2003)
12. Braess, D., Dahmen, W., Wieners, C.: A multigrid algorithm for the mortar finite element method. SIAM J. Numer. Anal. 37(1), 48–69 (1999)
13. Marcinkowski, L.: The mortar element method with locally nonconforming elements. BIT 39(4), 716–739 (1999)
14. Dryja, M.: A Neumann-Neumann algorithm for a mortar discetization of elliptic problems with discontinuous coefficients. Numer. Math. 99, 645–656 (2005)
15. Kim, H.H., Lee, C.O.: A preconditioner for the FETI-DP formulation with mortar methods in two dimensions. SIAM J. Numer. Anal. 42(5), 2159–2175 (2005)
16. Marcinkowski, L., Rahman, T.: Neumann-Neumann algorithms for a mortar Crouzeix-Raviart element for 2nd order elliptic problems. BIT 48(3), 607–626 (2008)
17. Xu, X., Li, L., Chen, W.: A multigrid method for the mortar-type Morley element approximation of a plate bending problem. SIAM J. Numer. Anal. 39(5), 1712–1731 (2001/2002)
18. Marcinkowski, L.: Domain decomposition methods for mortar finite element discretizations of plate problems. SIAM J. Numer. Anal. 39(4), 1097–1114 (2001)
19. Marcinkowski, L.: A Neumann-Neumann algorithm for a mortar finite element discretization of fourth-order elliptic problems in 2d. Numer. Methods Partial Differential Equations 25(6), 1425–1442 (2009), http://www.interscience.wiley.com, doi:10.1002/num.20406 Published online in Wiley InterScience on December 11, 2008
20. Marcinkowski, L.: A balancing Neumann-Neumann method for a mortar finite element discretization of a fourth order elliptic problem. J. Numer. Math. 18(3), 219–234 (2010)
21. Marcinkowski, L.: A preconditioner for a FETI-DP method for mortar element discretization of a 4th order problem in 2d. Electron. Trans. Numer. Anal. 38, 1–16 (2011)
22. Brenner, S.C.: The condition number of the Schur complement in domain decomposition. Numer. Math. 83(2), 187–203 (1999)
23. Brenner, S.C., Scott, L.R.: The mathematical theory of finite element methods, 3rd edn. Texts in Applied Mathematics, vol. 15. Springer, New York (2008)

Incomplete Cyclic Reduction of Banded and Strictly Diagonally Dominant Linear Systems

Carl Christian Kjelgaard Mikkelsen and Bo Kågström

Department of Computing Science and HPC2N,
Umeå University, Sweden
{spock,bokg}@cs.umu.se

Abstract. The ScaLAPACK library contains a pair of routines for solving banded linear systems which are strictly diagonally dominant by rows. Mathematically, the algorithm is complete block cyclic reduction corresponding to a particular block partitioning of the system. In this paper we extend Heller's analysis of incomplete cyclic reduction for block tridiagonal systems to the ScaLAPACK case. We obtain a tight estimate on the significance of the off diagonal blocks of the tridiagonal linear systems generated by the cyclic reduction algorithm. Numerical experiments illustrate the advantage of omitting all but the first reduction step for a class of matrices related to high order approximations of the Laplace operator.

Keywords: Banded or block tridiagonal linear systems, strict diagonal dominance, incomplete cyclic reduction, ScaLAPACK.

1 Introduction

Let A be a nonsingular N by N block tridiagonal matrix, i.e.

$$A = \begin{bmatrix} D_1 & F_1 & & \\ E_2 & \ddots & \ddots & \\ & \ddots & \ddots & F_{N-1} \\ & & E_N & D_N \end{bmatrix} \tag{1}$$

and consider the solution of the linear system

$$Ax = f \tag{2}$$

on a parallel machine. If $A = [a_{ij}]$ is strictly diagonally dominant by rows, i.e

$$\forall\, i \;:\; \sum_{j \neq i} |a_{ij}| < |a_{ii}|,$$

then we can use cyclic reduction to solve the linear system (2). The basic algorithm is due to R.W. Hockney [4] who worked closely with G. H. Golub on the solution of certain tridiagonal linear systems. Heller [3] showed that if A is strictly diagonally dominant, then the cyclic reduction algorithm runs to completion. Moreover, Heller introduced the incomplete cyclic reduction algorithm and estimated the truncation error.

Every banded matrix A can be partitioned as a block tridiagonal matrix (1) only the dimension of the diagonal blocks D_i may not be too small. Specifically, if A has k superdiagonals and k subdiagonals, then the dimension of each D_i must be at least k.

ScaLAPACK [2] contains a pair of routines (PDDBTRF/PDDBTRS) which can be used to solve narrow banded linear systems which are strictly diagonally dominant by rows. The algorithm is complete block cyclic reduction corresponding to a particular partitioning of the system. The odd numbered diagonal blocks are as large as possible, while the even numbered diagonal blocks are k by k.

In this paper we extend Heller's analysis to the ScaLAPACK case. Central to the analysis is the *dominance factor* ϵ given by

$$\epsilon = \max_i \left\{ \frac{1}{|a_{ii}|} \sum_{j \neq i} |a_{ij}| \right\} \geq 0.$$

The strict diagonal dominance of A implies that $\epsilon < 1$.

We review the incomplete cyclic reduction algorithm and some related results in Section 2. The new results are derived in Section 3. We have already proved Theorem 1 for tridiagonal matrices ($k = 1$) in a previous paper [7]. In this paper, we derive Theorem 2 and use it to prove Theorem 1 in the general case. In Section 4 we demonstrate the existence of a class of physically relevant linear systems for which it is numerically sound to neglect all but the first reduction step, an observation which greatly improves the strong scalability of the algorithm.

This presentation owes much to Heller's original paper [3] which we highly recommend. The ScaLAPACK implementation of block cyclic reduction of narrow banded linear systems is described in detail in the paper by Arbenz, Cleary, Dongarra and Hegland [1].

2 The Algorithm

Complete block cyclic reduction of the linear system (2) is equivalent to Gaussian elimination with no pivoting on a permuted system

$$(PAP^T)Px = Pf,$$

where P is a block permutation matrix, which reorders the vector $(1, 2, \ldots, N)$, so the odd multiples of 2^0 come first, followed by odd multiples of 2^1, etc.

The incomplete block cyclic reduction algorithm is stated as Algorithm 1. Given an integer m we obtain an approximation y of the true solution as follows. First, we execute m steps of cyclic reduction (lines 1 through 3) in order to eliminate the variables which correspond to blocks numbered with an odd

Algorithm 1. The incomplete block cyclic reduction algorithm.

```
Input: An N by N block tridiagonal linear system
        $$A^{(0)}x^{(0)} = v^{(0)}$$
    where $A^{(0)}$ is strictly diagonally dominant by rows and an integer $m \geq 0$.
Output: An approximation $y$ of the exact solution.
1: for $j = 1 : 1 : m$ do
2:     Assemble the block tridiagonal linear system
           $$A^{(j)}x^{(j)} = v^{(j)}$$
       where
           $$A^{(j)} := \left(E_i^{(j)}, D_i^{(j)}, F_i^{(j)}\right)_{\{i \in \mathbb{N} \,:\, 1 \leq i \cdot 2^j \leq N\}}, \quad \text{and} \quad \left(v_i^{(j)}\right)_{\{i \in \mathbb{N} \,:\, 1 \leq i \cdot 2^j \leq N\}}$$
       are given by
           $$D_i^{(j)} := D_{2i}^{(j-1)} - E_{2i}^{(j-1)}\left({D_{2i-1}^{(j-1)}}^{-1} F_{2i-1}^{(j-1)}\right) - F_{2i}^{(j-1)}\left({D_{2i+1}^{(j-1)}}^{-1} E_{2i+1}^{(j-1)}\right)$$
           $$E_i^{(j)} := -E_{2i}^{(j-1)}\left({D_{2i-1}^{(j-1)}}^{-1} E_{2i-1}^{(j-1)}\right)$$
           $$F_i^{(j)} := -F_{2i}^{(j-1)}\left({D_{2i+1}^{(j-1)}}^{-1} F_{2i+1}^{(j-1)}\right)$$
           $$v_i^{(j)} := v_{2i}^{(j-1)} - E_{2i}^{(j-1)}\left({D_{2i-1}^{(j-1)}}^{-1} v_{2i-1}^{(j-1)}\right) - F_{2i}^{(j-1)}\left({D_{2i+1}^{(j-1)}}^{-1} v_{2i+1}^{(j-1)}\right)$$
3: end for
4: Define the block diagonal matrix
           $$D^{(m)} := \mathrm{diag}(D_i^{(m)})_{\{i \in \mathbb{N} \,:\, 1 \leq i \cdot 2^m \leq N\}}$$
   and solve the system
           $$D^{(m)}y^{(m)} = v^{(m)}$$
   with respect to $y^{(m)}$.
5: for $j = m : -1 : 1$ do
6:     Set $y_{2i}^{(j-1)} = y_i^{(j)}$
7:     Solve
           $$D_{2i-1}^{(j-1)} y_{2i-1}^{(j-1)} = v_{2i-1}^{(j-1)} - E_{2i-1}^{(j-1)} y_{2i-2}^{(j-1)} - F_{2i-1}^{(j-1)} y_{2i+2}^{(j-1)}$$
       with respect to $y_{2i-1}^{(j-1)}$.
8: end for
9: Return $y$ given by
           $$y = ({y_1^{(0)}}^T, {y_2^{(0)}}^T, \ldots, {y_N^{(0)}}^T)^T.$$
```

multiple of $2^0, 2^1, \ldots, 2^{m-1}$. The remaining components can be found by solving the block tridiagonal linear system

$$A^{(m)}x^{(m)} = v^{(m)}. \tag{3}$$

We approximate the solution of (3) by dropping the off diagonal blocks (line 4). The remaining components of y are obtained by backward substitution (lines 5 through 8).

Heller [3] measured the significance of the off diagonal blocks using the auxiliary matrices $B^{(j)}$ given by

$$B^{(j)} = {D^{(j)}}^{-1}(A^{(j)} - D^{(j)}).$$

Heller [3] showed that

$$\|x - y\|_\infty \leq \|B^{(m)}\|_\infty \|x\|_\infty,$$

and derived the central estimate

$$\|B^{(j+1)}\|_\infty \leq \|{B^{(j)}}^2\|_\infty \leq \|B^{(j)}\|_\infty^2 < 1, \quad j = 0, 1, 2, \ldots,$$

which shows that the error decays quadratically. Recently, we have shown that

$$\|B^{(j)}\|_\infty \leq \epsilon^{2^j}$$

and this estimate is tight [7]. This general estimate carries to the ScaLAPACK case, but it does not reflect the banded structure of the odd numbered diagonal blocks. Theorem 1 shows how to integrate this information into the analysis.

3 The Main Result

Theorem 1. *If the odd numbered diagonal blocks can be partitioned as block tridiagonal matrices with q diagonal blocks, then*

$$\|B^{(1)}\|_\infty \leq \epsilon^{1+q}. \tag{4}$$

Moreover, this estimate is tight.

It is straight forward to verify that equality is achieved for block tridiagonal matrices for which $E_i = 0$, $D_i = I_k$ and $F_i = \epsilon I_k$, where I_k denotes the k by k identity matrix.

We now reduce the proof of the central inequality (4) to a single application of Theorem 2. Normally, we would illustrate the cyclic reduction algorithm using explicit block permutations, but here we use "in-place" computations in order to estimate $B^{(1)}$. Now, let

$$G = \begin{bmatrix} E_{-1} & D_{-1} & F_{-1} & & \\ & E_0 & D_0 & F_0 & \\ & & E_1 & D_1 & F_1 \end{bmatrix}$$

be a compressed representation of three consecutive block rows drawn for a block tridiagonal linear system. Similarly, let

$$A_0^{(1)} = \begin{pmatrix} E_0^{(1)} & D_0^{(1)} & F_0^{(1)} \end{pmatrix}$$

be a representation of the corresponding block row of the Schur complement. Specifically, we have

$$D_0^{(1)} = D_0 - E_0 D_{-1}^{-1} F_{-1} - F_0 D_1^{-1} F_1$$

and

$$E_0^{(1)} = -E_0 D_{-1}^{-1} E_{-1}, \quad F_0^{(1)} = -F_0 D_1^{-1} F_1.$$

Our goal is to estimate the infinity norm of the matrix $B_0^{(1)}$ given by

$$B_0^{(1)} = \left[U_0^{(1)} \middle| V_0^{(1)} \right],$$

where

$$D_0^{(1)} \left[U_0^{(1)} \middle| V_0^{(1)} \right] = \left[E_0^{(1)} \middle| F_0^{(1)} \right].$$

To this end we identify $B_0^{(1)}$ with a certain submatrix of a matrix G' which is row equivalent to G. Let $\left[U_i, V_i\right]$ be the solution of the linear system

$$D_i \left[U_i, V_i\right] = \left[E_i, F_i\right], \quad i \in \{-1, 0, 1\}.$$

Then

$$G \sim \left[\begin{array}{c|c|c|c|c} U_{-1} & I & V_{-1} & & \\ \hline & U_0 & I & V_0 & \\ \hline & & U_1 & I & V_1 \end{array}\right]$$
$$\sim \left[\begin{array}{c|c|c|c|c} U_{-1} & I & V_{-1} & & \\ \hline -U_0U_{-1} & & (I - U_0V_{-1} - V_0U_1) & & -V_0V_1 \\ \hline & & U_1 & I & V_1 \end{array}\right].$$

Moreover, it is straight forward to verify that

$$G \sim \left[\begin{array}{c|c|c|c|c} U_{-1} & I & V_{-1} & & \\ \hline U_0^{(1)} & & I & & V_0^{(1)} \\ \hline & & U_1 & I & V_1 \end{array}\right]$$
$$\sim \left[\begin{array}{c|c|c|c|c} U_{-1} - V_{-1}U_0^{(1)} & I & & & -V_{-1}V_0^{(1)} \\ \hline U_0^{(1)} & & I & & V_0^{(1)} \\ \hline -U_1U_0^{(1)} & & & I & V_1 - U_1V_0^{(1)} \end{array}\right] =: G'.$$

Theorem 2 details the structure of the matrix G' and Theorem 1 is an immediate consequence.

Theorem 2. *Let $q \in \mathbb{N}$ and let G_q be a representation of $2q + 1$ consecutive block rows of a block tridiagonal matrix A which is strictly diagonally dominant by rows with dominance factor ϵ, i.e.*

$$
G_q = \left[\begin{array}{c|cccc|c|cccc|c}
E_{-q} & D_{-q} & F_{-q} & & & & & & & & \\
 & \ddots & \ddots & \ddots & & & & & & & \\
 & & \ddots & \ddots & \ddots & & & & & & \\
 & & & E_{-1} & D_{-1} & F_{-1} & & & & & \\
\hline
 & & & & E_0 & D_0 & F_0 & & & & \\
\hline
 & & & & & E_1 & D_1 & F_1 & & & \\
 & & & & & & \ddots & \ddots & \ddots & & \\
 & & & & & & & \ddots & \ddots & \ddots & \\
 & & & & & & & & E_q & D_q & F_q
\end{array}\right].
$$

Then G_q is row equivalent to a unique matrix K_q of the form

$$
K_q = \left[\begin{array}{c|cccc|c|cccc|c}
\mathcal{U}_{-q}^{(q)} & I & & & & & & & & & \mathcal{V}_{-q}^{(q)} \\
\vdots & & \ddots & & & & & & & & \vdots \\
\vdots & & & \ddots & & & & & & & \vdots \\
\mathcal{U}_{-1}^{(q)} & & & & I & & & & & & \mathcal{V}_{-1}^{(q)} \\
\hline
\mathcal{U}_{0}^{(q)} & & & & & I & & & & & \mathcal{V}_{0}^{(q)} \\
\hline
\mathcal{U}_{1}^{(q)} & & & & & & I & & & & \mathcal{V}_{1}^{(q)} \\
\vdots & & & & & & & \ddots & & & \vdots \\
\vdots & & & & & & & & \ddots & & \vdots \\
\mathcal{U}_{q}^{(q)} & & & & & & & & & I & \mathcal{V}_{q}^{(q)}
\end{array}\right] \tag{5}
$$

where the spikes decay exponentially as we move towards the main block row. Specifically, if we define

$$
Z_i^{(q)} = \left[\begin{array}{c|c} \mathcal{U}_{-i}^{(q)} & \mathcal{V}_{-i}^{(q)} \\ \hline \mathcal{U}_{i}^{(q)} & \mathcal{V}_{i}^{(q)} \end{array}\right], \quad 0 < i \leq q,
$$

and

$$
Z_0^{(q)} = \left[\mathcal{U}_0^{(q)} \middle| \mathcal{V}_0^{(q)}\right],
$$

then

$$
\|Z_i^{(q)}\|_\infty \leq \epsilon^{1+q-i}, \quad 0 \leq i \leq q.
$$

Proof. The existence and uniqueness of the matrix K_q is immediate. Moreover, Corollary 3.2 [6] implies that

$$
\|Z_i^{(q)}\|_\infty \leq \epsilon < 1, \quad 0 \leq i \leq q.
$$

In particular, we already know that

$$
\|Z_q^{(q)}\| \leq \epsilon = \epsilon^{(1+q)-q}
$$

and we can focus on the "interior" case of $0 \leq i < q$.

We prove the decay of the spikes using the principle of mathematical induction. Let Ω be given by

$$\Omega = \{q \in \mathbb{N} \mid \forall i \leq q : \|Z_i^{(q)}\|_\infty \leq \epsilon^{1+q-i}\}.$$

Our goal is to show that $\Omega = \mathbb{N}$. We begin by showing that $1 \in \Omega$. We have

$$G_1 = \begin{bmatrix} E_{-1} & D_{-1} & F_{-1} & & \\ & E_0 & D_0 & F_0 & \\ & & E_1 & D_1 & F_1 \end{bmatrix} \sim \begin{bmatrix} U_{-1} & I & V_{-1} & & \\ & U_0 & I & V_0 & \\ & & U_1 & I & V_1 \end{bmatrix}$$
$$\sim \begin{bmatrix} U_{-1} & I & V_{-1} & & \\ \mathcal{U}_0^{(1)} & & I & & \mathcal{V}_0^{(1)} \\ & & U_1 & I & V_1 \end{bmatrix} \sim \begin{bmatrix} \mathcal{U}_{-1}^{(1)} & I & & \mathcal{V}_{-1}^{(1)} \\ \mathcal{U}_0^{(1)} & & I & \mathcal{V}_0^{(1)} \\ \mathcal{U}_1^{(1)} & & I & \mathcal{V}_0^{(1)} \end{bmatrix},$$

where

$$D_i \left[U_i \middle| V_i \right] = \left[E_i \middle| F_i \right]$$

and

$$Z_0^{(1)} = \left[\mathcal{U}_0^{(1)} \middle| \mathcal{V}_0^{(1)} \right] = -(I - U_0 V_{-1} - V_0 U_1)^{-1} \left[U_0 U_{-1} \middle| V_0 V_1 \right].$$

The exact formula for

$$\left[\mathcal{U}_i^{(1)} \middle| \mathcal{V}_i^{(1)} \right], \quad i = \pm 1$$

is irrelevant, because we already know that

$$\left\| \left[\mathcal{U}_i^{(1)} \middle| \mathcal{V}_i^{(1)} \right] \right\|_\infty \leq \epsilon, \quad i = -1, 0, 1.$$

It remains to be seen that $\|Z_0^{(1)}\|_\infty \leq \epsilon^2$. By definition

$$\begin{aligned} Z_0^{(1)} &= \left[U_0 \middle| V_0 \right] \left[\frac{V_{-1}}{U_1} \right] Z_0^{(1)} - \left[U_0 U_{-1} \middle| V_0 V_1 \right] \\ &= \left[U_0 \middle| V_0 \right] \left\{ \left[\frac{V_{-1}}{U_1} \right] \left[\mathcal{U}_0^{(1)} \middle| \mathcal{V}_0^{(1)} \right] - \left[\begin{array}{c|c} U_{-1} & 0 \\ \hline 0 & V_1 \end{array} \right] \right\} \\ &= \left[U_0 \middle| V_0 \right] \left[\begin{array}{c|c|c} V_{-1} & U_{-1} & 0 \\ \hline U_1 & 0 & V_1 \end{array} \right] \left[\begin{array}{c|c} \mathcal{U}_0^{(1)} & \mathcal{V}_0^{(1)} \\ \hline -I & 0 \\ \hline 0 & -I \end{array} \right] \end{aligned}$$

and since we already know that $\|Z_0^{(1)}\|_\infty < 1$ we can conclude

$$\|Z_0^{(1)}\|_\infty \leq \epsilon^2 \max\{\|Z_0^{(1)}\|_\infty, 1\} = \epsilon^2,$$

and we have shown that $1 \in \Omega$. Now assume that $1 < q$ and $q - 1 \in \Omega$. We must show that $q \in \Omega$. By assumption

$$G_q \sim \left[\begin{array}{c|c|cccccc|c|c}
U_{-q} & I & V_{-q} & & & & & & & \\ \hline
 & \mathcal{U}_{1-q}^{(q-1)} & I & & & & & & \mathcal{V}_{1-q}^{(q-1)} & \\
 & \vdots & & \ddots & & & & & \vdots & \\
 & \mathcal{U}_{-1}^{(q-1)} & & & I & & & & \mathcal{V}_{-1}^{(q-1)} & \\
 & \mathcal{U}_{0}^{(q-1)} & & & & I & & & \mathcal{V}_{0}^{(q-1)} & \\
 & \mathcal{U}_{1}^{(q-1)} & & & & & I & & \mathcal{V}_{1}^{(q-1)} & \\
 & \vdots & & & & & & \ddots & \vdots & \\
 & \mathcal{U}_{q-1}^{(q-1)} & & & & & & \ddots & \mathcal{V}_{q-1}^{(q-1)} & \\ \hline
 & & & & & & & U_q & I & V_q
\end{array}\right]$$

$$=: \left[\begin{array}{c|c|c|c|c}
U_{-q} & I & V'_{-q} & & \\ \hline
 & \mathcal{U}^{(q-1)} & I & \mathcal{V}^{(q-1)} & \\ \hline
 & & U'_q & I & V_q
\end{array}\right],$$

where we have made the last definition to emphasize the similarity between our current situation and the problem of showing that $1 \in \Omega$.

We continue to reduce G_q using elementary block row operations. We have

$$G_q \sim \left[\begin{array}{c|c|c|c|c}
U_{1-q} & I & V'_{1-q} & & \\ \hline
\xi^{(q)} & & I & & \nu^{(q)} \\ \hline
 & & U'_{q-1} & I & V_{q-1}
\end{array}\right]$$

$$\sim \left[\begin{array}{c|c|c|c|c}
U_{1-q} - V'_{1-q}\xi^{(q)} & I & & & -V'_{1-q}\nu^{(q)} \\ \hline
\xi^{(q)} & & I & & \nu^{(q)} \\ \hline
-U'_{q-1}\xi^{(q)} & & & I & V_{q-1} - U'_{q-1}\nu^{(q)}
\end{array}\right] = K_q,$$

where the last equality is critical, but follows immediately from the uniqueness of the matrix K_q. Now, it is straightforward to verify that

$$\left[\xi^{(q)} \middle| \nu^{(q)}\right] = -\left(I - \mathcal{U}^{(q-1)}V'_{1-q} - \mathcal{V}^{(q-1)}U'_{q-1}\right)^{-1}\left[\mathcal{U}^{(q-1)}U_{1-q} \middle| \mathcal{V}^{(q-1)}V_{q-1}\right],$$

or equivalently

$$\left[\xi^{(q)} \middle| \nu^{(q)}\right] = \left[\mathcal{U}^{(q-1)} \middle| \mathcal{V}^{(q-1)}\right] \left\{ \left[\begin{array}{c} V'_{1-q} \\ \hline U'_{q-1} \end{array}\right] \left[\xi^{(q)} \middle| \nu^{(q)}\right] - \left[\begin{array}{c|c} U_{1-q} & 0 \\ \hline 0 & V_{q-1} \end{array}\right] \right\}$$

$$= \left[\mathcal{U}^{(q-1)} \middle| \mathcal{V}^{(q-1)}\right] \left[\begin{array}{c|c|c} V'_{1-q} & -U_{1-q} & 0 \\ \hline U'_{q-1} & 0 & -V_{q-1} \end{array}\right] \left[\begin{array}{c|c} \xi^{(q)} & \nu^{(q)} \\ \hline I & 0 \\ \hline 0 & I \end{array}\right]. \quad (6)$$

It follows that equation (6) is just a compact way of expressing the fact that

$$Z_i^{(q)} = Z_i^{(q-1)} \left[\begin{array}{c|c|c} V'_{1-q} & -U_{1-q} & 0 \\ \hline U'_{q-1} & 0 & -V_{q-1} \end{array}\right] \left[\begin{array}{c|c} \xi^{(q)} & \nu^{(q)} \\ \hline I & 0 \\ \hline 0 & I \end{array}\right], \quad i < q,$$

and this relation will allow us to estimate $\|Z_i^{(q)}\|_\infty$. By assumption, $q-1 \in \Omega$, or equivalently

$$\|Z_i^{(q-1)}\|_\infty \leq \epsilon^{q-i}, \quad i \leq q-1.$$

It follows that

$$\|Z_i^{(q)}\|_\infty \leq \epsilon^{q-i}\epsilon \max\{\| \left[\xi^{(q)} \ \nu^{(q)}\right] \|_\infty, 1\} = \epsilon^{1+q-i}, \quad i < q,$$

because $\| \left[\xi^{(q)} \ \nu^{(q)}\right] \|_\infty < 1$. The extreme case of $i = q$ was settled at the beginning of the proof. This shows that $q \in \Omega$ and the proof is complete. □

4 Numerical Experiments

An incomplete cyclic reduction algorithm was implemented in Fortran 90 using MPI. We inherited the communication pattern from the ScaLAPACK implementation, but choose the memory layout of the Arbenz-Hegland implementation [1]. In reality we rewrote the ScaLAPACK implementation from scratch inserting a parameter controlling the number of reduction steps and replacing the blocking receive instructions from BLACS with nonblocking ones. As in the ScaLAPACK implementation [1] we split the algorithm into two phases: a factorization phase which exclusively references the matrix and a solve phase which references the right hand side in order to compute the solution.

Here we report on our findings for a very specific class of matrices, namely the pentadiagonal Toeplitz matrices $A \in \mathbb{R}^{n\times n}$ given by

$$A = \operatorname{diag}(\beta, \alpha, 1, \alpha, \beta), \quad \alpha = \frac{334}{899}, \quad \beta = \frac{43}{1798}.$$

These matrices occur naturally in connection with the construction of highly accurate approximations of the Laplace operator [5]. We have $\epsilon = 2(\alpha+\beta) \approx 0.7909$. The worst case estimate given by Theorem 1 is pessimistic. Specifically, in the case of $q = 25$, a direct calculation establishes that

$$\|B^{(1)}\|_\infty \approx 8.5652 \times 10^{-24} \ll \epsilon^{26} \approx 2.2431 \times 10^{-3}.$$

It follows that we can safely discard all but the first reduction step even if the banded segment assigned to each core contains as little as $\mu = 50$ rows.

We measured the performance of our implementation on the supercomputer AKKA at HPC2N. Briefly, AKKA consists of 672 nodes using an Infiniband interconnect. Each node contains two Intel Xeon quad-core L5420 CPU with a total of 16 GB RAM per node.

We used $p \in \{1, 2, 4, \ldots, 512\}$ cores, drawn from a fixed set of 64 nodes containing 512 cores. We used $n \in \{26622, 52222, 103422\}$. The dimensions were chosen so that each core would always receive a banded segment of exactly the same size regardless of the number of cores. For example $n = 26622$ corresponds to having $\mu = 50$ when using $p = 512$ cores, because $n = \mu p + k(p-1)$, where $k = 2$

is the number of super/subdiagonals. Finally, we used $r \in \{1, 10, 20, 30, 40, 50\}$. Every column of every solution was obtained with a forward normwise relative error no greater than $12u$, where $u = 2^{-53}$ is the double precision round-off error. Every experiment was programmed into a single job and submitted to the queue.

The *strong scalability efficiency* ξ_{strong} is defined as

$$\xi_{\text{strong}} = \frac{T_1/p}{T_p},$$

where p is the number of cores, T_1 is the execution time on a single core and T_p is the execution time on p cores. When $p = 1$ the parallel overhead vanishes and our implementation reduces to LAPACKs banded solver save for a few conditional statements and system calls. The measured efficiencies are displayed in Figures 1 through 3. For the sake of visual clarity we only display the solve efficiencies for $r \in \{1, 10, 50\}$. We shall now discuss the graphs in some detail.

Gaussian elimination has a very low arithmetic intensity for narrow banded linear systems. Spreading the same system across an increasing number of cores will eventually create a situation where a significant fraction of the system resides in the cache memory allowing for efficiencies greater than unity which is exactly what we have experienced.

The graphs are not monotone and are a bit erratic. By applying the law of large numbers we conclude that it is likely that the oscillations can be reduced by a factor of, say, 100 by repeating each experiment $100^2 = 10^4$ times and

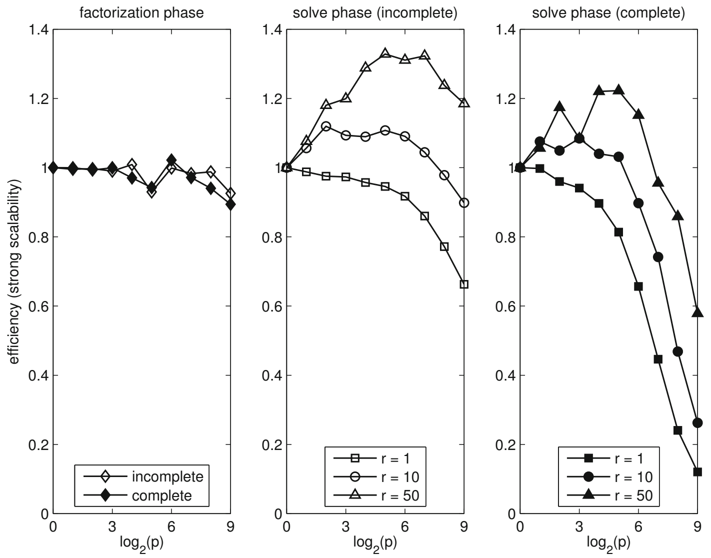


Fig. 1. Strong scalability efficiency curves for $n = 26622$. Notice the difference in the solve efficiencies between the two algorithms for large values of p.

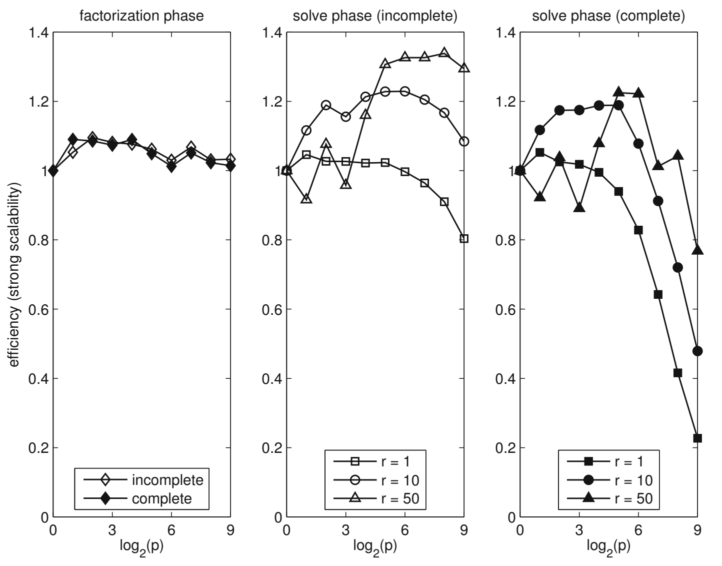


Fig. 2. Strong scalability efficiency curves for $n = 52222$

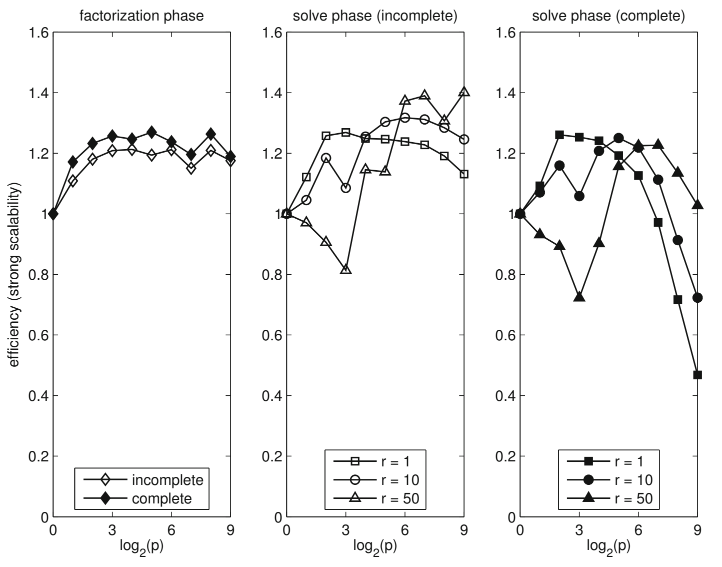


Fig. 3. Strong scalability efficiency curves for $n = 103422$. Notice that the solve efficiency for the incomplete algorithm is greater than unity for all values of r and $\log_2(p) \geq 6$.

computing the average run-time. However, even 25 repetitions required more than 250 machine hours, so 10^4 repetitions was not feasible for this paper.

For the factorization phase we were surprised to see that the experiments are inconclusive as the efficiencies are very similar. However, for the solve phase we see that the efficiency is improved considerably by neglecting all but the first reduction step. This effect is present regardless of n and r, but it most pronounced for $n = 26666$ and $r = 1$, where the ratio of the parallel work to the communication cost is minimal.

For both algorithms we see the efficiency of the solve phase increase with the number of right hand sides. This effect is present for all n and is due to the fact that the number of messages is independent of r, while the parallel work is a monotone increasing function of r.

Acknowledgments. 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 Foundation for Strategic Research under the frame program A3 02:128 and the EU Mål 2 project UMIT. This research was conducted using the resources of High Performance Computing Center North (HPC2N).

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., Ruiz, D., Duff, I., Frayssé, V., Giraud, L. (eds.) Euro-Par 1999. LNCS, vol. 1685, pp. 1078–1087. Springer, Heidelberg (1999)
2. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK User's Guide. SIAM, USA (1997)
3. Heller, D.: Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. SIAM J. Numer. Anal. 13, 484–496 (1976)
4. Hockney, R.W.: A fast direct solution of Poisson's equation using Fourier analysis. J. Assoc. Comput. Mach. 12, 95–113 (1965)
5. Lele, S.K.: Compact finite difference schemes with spectral-like resolution. J. Comput. Phys. 103, 16–42 (1992)
6. Mikkelsen, C.C.K., Manguoglu, M.: Analysis of the truncated SPIKE algorithm. SIAM J. Matrix Anal. Appl. 30, 1500–1519 (2008)
7. Mikkelsen, C.C.K., Kågström, B.: Parallel Solution of Narrow Banded Diagonally Dominant Linear Systems. In: Jónasson, K. (ed.) PARA 2010. LNCS, vol. 7134, pp. 280–290. Springer, Heidelberg (2012)

Fast and Small Nonlinear Pseudorandom Number Generators for Computer Simulation

Samuel Neves and Filipe Araujo

CISUC, Dept. of Informatics Engineering
University of Coimbra, Portugal
{sneves,filipius}@dei.uc.pt

Abstract. In this paper we present Tyche, a nonlinear pseudorandom number generator designed for computer simulation. Tyche has a small 128-bit state and an expected period length of 2^{127}. Unlike most nonlinear generators, Tyche is consistently fast across architectures, due to its very simple iteration function derived from ChaCha, one of today's fastest stream ciphers.

Tyche is especially amenable for the highly parallel environments we find today, in particular for Graphics Processing Units (GPUs), where it enables a very large number of uncorrelated parallel streams running independently. For example, 2^{16} parallel independent streams are expected to generate about 2^{96} pseudorandom numbers each, without overlaps.

Additionally, we determine bounds for the period length and parallelism of our generators, and evaluate their statistical quality and performance. We compare Tyche and the variant Tyche-i to the XORWOW and TEA_8 generators in CPUs and GPUs. Our comparisons show that Tyche and Tyche-i simultaneously achieve high performance and excellent statistical properties, particularly when compared to other nonlinear generators.

Keywords: ChaCha, GPU, PRNG, random number generation, SIMD, Tyche, Tyche-i.

1 Introduction

Pseudorandom numbers are often used for testing, simulation and even aesthetic purposes. They are an integral part of Monte Carlo methods, genetic and evolutionary algorithms, and are extensively used in noise generation for computer graphics.

Monte Carlo methods were first used for computing purposes by Ulam and von Neumann, while attempting to solve hard problems in particle physics [1]. Monte Carlo methods consist of iterated random sampling of many inputs in some probability distribution, followed by later processing. Given enough inputs, it is possible to obtain an approximate solution with a reasonable degree of certainty. This is particularly useful for problems with many degrees of freedom, where analytical or exact methods would be far too inefficient. Monte Carlo methods are not, however, very computationally efficient — typically, to reduce the error margin by half, one has to quadruple the amount of sampled inputs [2]. Today, Monte Carlo methods are used in the most various fields, such as particle physics, weather forecasting, financial analysis, operations research, etc.

Current general-purpose processors typically have 2 or 4 cores. Graphics processing units have tens to hundreds [3]; future architectures are slated to scale up to hundreds and thousands of cores [4]. This development entails some consequences: silicon real estate is limited, and the increase in processing units decreases the total fast memory available on-chip. Thus, it becomes an interesting problem to design a random number generator that can take advantage of massively parallel architectures and still remain small, fast and of high quality. With these goals in mind, we introduce Tyche, a fast and small-state pseudorandom number generator especially suited for current parallel architectures. The iteration function of Tyche, derived from the stream cipher ChaCha's *quarter-round* function [5], is nonlinear and fast; it uses a very small amount of state (4 32-bit registers) and, yet, it has a very large average period.

In Section 2 we review the state of the art in theory and practice of random number generation. In Section 3 we describe the Tyche function. In Section 4 we provide an analysis of several important features of the algorithm, such as the expected period, statistical quality and parallelism. We then introduce a variant of Tyche with higher instruction-level parallelism in Section 5. In Section 6, we experimentally evaluate and compare Tyche. Section 7 concludes the paper.

2 Related Work

There is an enormous body of work in the literature regarding pseudorandom number generation. One of the first and most popular methods to generate pseudorandom numbers in digital computers was Lehmer's linear congruential method, consisting of a linear recurrence modulo some integer m. Since then, researchers have proposed numerous other linear algorithms, most notably lagged Fibonacci generators, linear feedback shift registers, Xorshift generators and the Mersenne Twister [6,7,8]. The statistical properties of linear generators are well known and widely studied; several empirical tests are described in [6, Chapter 3]. One of the drawbacks of such linear generators, in particular linear congruential generators, is their very regular lattice structure [6, Section 3.3.4]. This causes the usable amount of numbers in a simulation to be far less than the full period of the generator [9].

Nonlinear pseudorandom generators, like the *Inversive congruential generator* and *Blum Blum Shub* [10,11], avoid the drawbacks of linearity. However, nonlinear generators often require more complex arithmetic than linear ones and have smaller periods, rendering them impractical for simulations.

The generation of pseudorandom numbers in parallel environments is also a well studied subject [12,13,14]. The main problem is to enable multiple concurrent threads of execution to get random numbers in parallel. Two main solutions exist for this problem: *parametrization* and *cycle splitting*. Parametrization consists in creating a slightly different full period generator for each instance; this can be done by, *e.g.*, changing the iteration function itself. Cycle splitting takes a full period sequence and splits it into a number of subsequences, each used within an instance. Cycle splitting is often used in linear congruential generators, since it is possible to leap to any arbitrary position in the stream quite easily. Other generators, such as the Mersenne Twister, rely on different initial parameters (*i.e.*, parametrization) to differentiate between threads.

In the case of GPUs and other vector processors, we face additional restrictions, because the amount of fast memory per core is quite limited, thus restricting the internal state we can use. Furthermore, GPUs lack some important instructions, such as native integer multiplication and/or division, leading to a large slowdown for some popular random number generators. Still, linear congruential generators have been studied in GPUs [15].

There have been some initial attempts to adapt cryptographic functions for fast GPU random number generation. Tzeng and Wei [16] used the MD5 hash function and reduced-round versions thereof to generate high-quality random numbers. Zafar and Olano [17] used the smaller and faster 8-round TEA block cipher to generate high-quality random numbers for Perlin noise generation.

3 Tyche

This section will present Tyche. In the following sections, all values are assumed to be 32 bits long, unless otherwise noted. $+$ represents addition modulo 2^{32}; $\oplus$ denotes the exclusive-or (xor) operation; $\lll$ means bitwise rotation towards the most significant bits.

3.1 Initialization

The state of Tyche is composed of 4 32-bit words, which we will call a, b, c and d. Tyche, when initialized, takes a 64-bit integer *seed* and a 32-bit integer *idx*. Algorithm 3.1 describes the operations performed during initialization.

Algorithm 3.1: Tyche Init$(a, b, c, d, seed, idx)$

```
$a \leftarrow \lfloor seed/2^{32} \rfloor$
$b \leftarrow seed \bmod 2^{32}$
$c \leftarrow 2654435769$
$d \leftarrow 1367130551 \oplus idx$
for $i \leftarrow 0$ to $20$
  do $MIX(a, b, c, d)$
return $(a, b, c, d)$
```

The MIX function called in Algorithm 3.1 is used here to derive the initial state; it is described further in Section 3.3. The constants used in the initialization, 2654435769 and 1367130551, were chosen to be $\lfloor 2^{32}/\varphi \rfloor$ and $\lfloor 2^{32}/\pi \rfloor$, where φ is the golden ratio and π is the well-known constant. Their purpose is to prevent a starting internal state of $(0, 0, 0, 0)$.

3.2 The Algorithm

Once its internal state is initialized, Tyche is quite simple. It calls the MIX function once and returns the second word of the internal state, as shown in Algorithm 3.2.

Algorithm 3.2: TYCHE (a, b, c, d)

```
$(a, b, c, d) = MIX(a, b, c, d)$
**return** $(b)$
```

3.3 The MIX Function

The MIX function, used both in initialization and state update, is derived directly from the *quarter-round* function of the ChaCha stream cipher [5]. As described in Algorithm 3.3, it works on 4 32-bit words and uses only addition modulo 2^{32}, XOR and bitwise rotations.

Algorithm 3.3: MIX(a, b, c, d)

```
$a \leftarrow a + b$
$d \leftarrow (d \oplus a) \lll 16$
$c \leftarrow c + d$
$b \leftarrow (b \oplus c) \lll 12$
$a \leftarrow a + b$
$d \leftarrow (d \oplus a) \lll 8$
$c \leftarrow c + d$
$b \leftarrow (b \oplus c) \lll 7$
**return** $(a, b, c, d)$
```

4 Analysis of Tyche

4.1 Design

We can find many different designs for random number generators. The design we propose here attempts to achieve high period, speed, and very low memory consumption. One of the ways in which it achieves this is by using a very simple recursion:

$$x_{i+1} = f(x_i) \tag{1}$$

This requires no extra space other than the state's size and perhaps some overhead to execute f. One could use, *e.g.*, a counter to ensure certain minimum period — this would evidently require more registers per state, which goes against our main objectives. A similar approach to ours has been used in the LEX [18] stream cipher, using the AES block cipher in Output Feedback mode (OFB) and extracting 4 bytes of the state per iteration.

Another crucial design choice concerns function f. Should it be linear? Most current random number generators are indeed linear: LCG, Xorshift, LFSR constructions, etc. These functions have the advantage of being very simple and easily analyzed. However, linear random number generators tend to have highly regular outputs: their outputs lie on simple lattice structures of some dimension. This makes such generators unsuitable for some types of simulations and considerably reduces their useful period [19]. Nonlinear generators generally do not have this problem [2]. Moreover, despite being very simple, linear generators may not be very fast. Linear congruential generators and their derivatives require multiplications and modular reductions. Unfortunately, these operations are not present in every instruction set and can be hard to implement otherwise.

One could then simply search for a good nonlinear random number generator. However, nonlinear generators for simulation purposes are hard to find, and generally much slower than their linear counterparts. Indeed, one could simply use a cryptographic stream cipher as a random number generator. That would, however, be several times slower and would require a much larger state. Indeed, even TEA$_8$ as described in [17] requires 136 instructions per 64 bits of random output, while MIX only requires 12 instructions per 32 bits of random output.

In light of these reasons, we chose our function to be nonlinear and to use exclusively instructions available in almost every chip — addition, xor, bit rotations[1]. The overlap of 32-bit addition and xor creates a high amount of nonlinearity and simultaneously allows for very fast implementations, owing to the simplicity of such instructions.

4.2 Period

The MIX function, used to update the internal state, is trivially reversible. Thus it is a permutation, with only one possible state before each other state. How does this affect the expected period? Were the MIX function irreversible, it would behave like a random mapping—in that case, the period would be about $2^{n/2}$ for an n-bit state [20]. In our case, the expected period is the average cycle length of a random element in a random permutation: $(2^n + 1)/2 \approx 2^{n-1}$ for an n-bit state [21, Section 1.3.3].

It is also known that random permutations do have small-length cycles. In fact, we can trivially find one cycle of length 1 in the MIX function: MIX(0,0,0,0) = (0,0,0,0) — this is in fact its only fixed point [22]. However, if using the initialization described in Section 3.1, this state will never be reached. It is also extremely unlikely to reach a very short cycle—the probability of reaching a cycle of length m is $1/2^n$; the probability of reaching a cycle of length *m or less* is $\sum_i^m 1/2^n = m/2^n$ [23]. In our case, the chance of reaching a state with period less than or equal to 2^{32} is roughly 2^{-96}.

[1] While many chips do not have bit rotations natively, they are trivially achievable with simple logical instructions such as shifts and xor.

4.3 Parallelization

Our algorithm is trivial to use in parallel environments. When initializing a state (using Algorithm 3.1 or 5.1), each computing unit (*e.g.*, thread, vector element, core) uses the same 64-bit seed, but its own index in the computation (the `idx` argument of Algorithm 3.1). We chose a 64-bit seed to avoid collisions; since seeds are often chosen at random, it would only require about 2^{16} initializations for a better than 50% chance to rerun a simulation using the same seed if one used 32-bit seeds. This would be unacceptable.

What about overlaps? Parallel streams will surely overlap eventually, given that the function is cyclic and reversible. This is as bad as a small period in a random number generator. To find out how fast streams overlap, consider a simple case: s streams outputting a single value each. Given that each stream begins at an arbitrary state out of n possible states, the probability of an overlap (*i.e.*, a collision) would be given by the birthday paradox:

$$1 - \frac{n!}{(n-s)!n^s} \tag{2}$$

This is, however, a simplified example; what we want to know is the likelihood that, given s streams and a function f that is a random permutation, no stream will meet the starting point of any other in less than d calls to f. This can be seen as a generalization of the birthday problem, and was first solved by Naus [24]. The probability that at least one out of s streams overlaps in less than d steps in a cycle of length m is given by

$$1 - \frac{(m - sd + s - 1)!}{(m - sd)!m^{s-1}} \tag{3}$$

In our particular case, m is in average 2^{127}; s should be no more than 2^{16}; d should be a large enough distance to make the generator useful — we choose 2^{64} here as an example minimum requirement. Thus, 2^{16} parallel streams producing 2^{64} numbers will overlap with a probability of roughly 2^{-32}. Conversely, when running 2^{16} parallel streams, an overlap is not expected until about $2^{64}/2^{-32} = 2^{96}$ iterations have passed.

5 A Faster Variant

One issue with the construction described in the previous section is that it is completely sequential. Each instruction of the MIX function directly depends on the immediately preceding one. This does not take any advantage of modern superscalar CPU machinery. Thus, we propose a variant of Tyche, which we call Tyche-i, able to take advantage of pipelined processors. Tyche-i is presented in Algorithms 5.1, 5.2, and 5.3.

Algorithm 5.1: TYCHE-I INIT$(a, b, c, d, seed, idx)$

```
$a \leftarrow \lfloor seed/2^{32} \rfloor$
$b \leftarrow seed \bmod 2^{32}$
$c \leftarrow 2654435769$
$d \leftarrow 1367130551 \oplus idx$
**for** $i \leftarrow 0$ **to** $20$
  **do** MIX-i$(a, b, c, d)$
**return** $(a, b, c, d)$
```

Algorithm 5.2: TYCHE-I(a, b, c, d)

```
$(a, b, c, d) = $ MIX-i$(a, b, c, d)$
**return** $(a)$
```

Algorithm 5.3: MIX-I(a, b, c, d)

```
$b \leftarrow (b \ggg 7) \oplus c; c \leftarrow c - d$
$d \leftarrow (d \ggg 8) \oplus a; a \leftarrow a - b$
$b \leftarrow (b \ggg 12) \oplus c; c \leftarrow c - d$
$d \leftarrow (d \ggg 16) \oplus a; a \leftarrow a - b$
**return** $(a, b, c, d)$
```

The main difference between Tyche and Tyche-i is the MIX-i function. The MIX-i function is simply the *inverse function* of Tyche's MIX. Unlike MIX, MIX-i allows for 2 simultaneous executing operations at any given time, which is a better suit to superscalar processors than MIX is. The downside, however, is that MIX-i diffuses bits slower than MIX does: for 1-bit differences in the internal state, 1 MIX call averages 26 bit flipped bits, while MIX-i averages 8.

6 Experimental Evaluation

6.1 Performance

We implemented and compared the performance of Tyche and Tyche-i against the XORWOW generator found in the CURAND library [25]. XORWOW is a combination of a Xorshift [8] and an additive generator, with a combined period of $2^{192} - 2^{32}$. The test setup was the Intel Core 2 E8400 processor for the CPU benchmarks, and the NVIDIA GTX580 GPU for the GPU benchmarks. Table 1 summarizes our performance results in both architectures; note that GPU figures do not take into account kernel and memory transfer overheads, as those are essentially equal for every option.

As expected (cf. Section 5), Tyche-i is roughly twice as fast as Tyche on the Core 2, a processor with high instruction-level parallelism. In the GPU, Tyche-i is still quite faster, but by a lower (roughly 1.5) ratio.

Table 1. Period, state size, results of TestU01's "BigCrush", and performances, in cycles per 32-bit word, of various pseudorandom number generators in the CPU and GPU

Algorithm	Period	State	BigCrush	CPU	GPU
Tyche	$\approx 2^{127}$	128	160/160	12.327321	1.156616
Tyche-i	$\approx 2^{127}$	128	160/160	6.073590	0.763572
XORWOW [25]	$2^{192} - 2^{32}$	192	157/160	7.095927	0.578620
TEA$_8$ [26]	2^{64}	64	160/160	49.119271	5.641948

The XORWOW algorithm only requires bit shifts, not bit rotations. Unfortunately, the Fermi GPU architecture does not support native bit rotations, which are replaced by two shifts and a logical operation. This explains the slight speed advantage of XORWOW. Note that we are able to improve 2 out of the 4 rotations by using the Fermi `PRMT` instruction, which allows one to permute bytes of a word arbitrarily. On the CPU, native rotations are as fast as shifts, and Tyche-i actually beats XORWOW in speed.

We also include a comparison with TEA$_8$, which reveals to be markedly slower than any of the other choices for GPU (and CPU) random generation. As we already pointed out in Section 4.1, TEA$_8$ requires at least 136 instructions per 64-bit word, which is much higher than either Tyche, Tyche-i or XORWOW.

6.2 Statistical Quality Tests

In order to assess the statistical quality of Tyche and Tyche-i, we performed a rather exhaustive battery of tests. We employed the ENT and DIEHARD suites and the TestU01 "BigCrush" battery of tests [27,28,29]. Every test performed in both variants showed no statistical weaknesses (cf. Table 1).

Another aspect of Tyche is that it is based on the ChaCha stream cipher. ChaCha's "quarter-round" function is also employed, albeit slightly modified, in the BLAKE SHA-3 candidate[22]. The "quarter-round" has been extensively analyzed for flaws, but both functions are still regarded as secure [30,31]. This increases our confidence in the quality of Tyche as a generator.

Finally, note that the XORWOW algorithm fails 3 tests in the "BigCrush" battery: CollisionOver ($t = 7$), SimpPoker ($r = 27$), and LinearComp ($r = 29$), the latter being a testament of its linear nature.

7 Conclusion

In this paper we presented and analyzed Tyche and Tyche-i, fast and small nonlinear pseudorandom generators based on the ChaCha stream cipher building blocks.

Tyche and Tyche-i use a very small amount of state that fits entirely into 4 32-bit registers. Our experiments show that Tyche and Tyche-i are much faster than the also nonlinear and cryptographic function-derived TEA$_8$, while exhibiting a large enough period for serious simulations with many parallel threads. On the other hand, when we compare Tyche and Tyche-i to the slightly faster (but linear) XORWOW algorithm, statistical tests (*i.e.*, BigCrush) suggest that both Tyche and Tyche-i have better statistical properties.

Acknowledgments. This work has been supported by the project CMU-PT/RNQ/0015/2009, TRONE — Trustworthy and Resilient Operations in a Network Environment.

References

1. Metropolis, N., Ulam, S.: The Monte Carlo Method. Journal of the American Statistical Association 44(247), 335–341 (1949)
2. Gentle, J.E.: Random Number Generation and Monte Carlo Methods, 2nd edn. Springer (2003)
3. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro. 28(2), 39–55 (2008)
4. Vangal, S.R., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N., Borkar, S.: An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. IEEE Journal of Solid-State Circuits 43(1), 29–41 (2008)
5. Bernstein, D.J.: ChaCha, a variant of Salsa20 (January 2008), `http://cr.yp.to/papers.html#chacha`
6. Knuth, D.E.: Art of Computer Programming, 3rd edn. Seminumerical Algorithms, vol. 2. Addison-Wesley Professional (November 1997)
7. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. 8(1), 3–30 (1998)
8. Marsaglia, G.: Xorshift RNGs. Journal of Statistical Software 8(14) (July 2003)
9. Pawlikowski, K., Jeong, H.D., Lee, J.S.R.: On Credibility of Simulation Studies of Telecommunication Networks. IEEE Communications Magazine, 132–139 (January 2002)
10. Hellekalek, P.: Inversive Pseudorandom Number Generators: Concepts, Results, and Links. In: Alexopoulos, C., Kang, K., Lilegdon, W.R., Goldsman, D. (eds.) Proceedings of the 1995 Winter Simulation Conference, pp. 255–262. IEEE Press (1995)
11. Blum, L., Blum, M., Shub, M.: A Simple Unpredictable Pseudo-Random Number Generator. SIAM J. Comput. 15(2), 364–383 (1986)
12. Eddy, W.F.: Random Number Generators for Parallel Processors. Journal of Computational and Applied Mathematics 31, 63–71 (1990)
13. Brent, R.: Uniform random number generators for supercomputers. In: Proc. Fifth Australian Supercomputer Conference, Melbourne, pp. 95–104 (December 1992)
14. Schoo, M., Pawlikowski, K., McNickle, D.: A Survey and Empirical Comparison of Modern Pseudo-Random Number Generators for Distributed Stochastic Simulations. Technical report, Department of Computer Science and Software Development, University of Canterbury (2005)
15. Langdon, W.B.: A fast high quality pseudo random number generator for nvidia cuda. In: GECCO 2009: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference, pp. 2511–2514. ACM, New York (2009)

16. Tzeng, S., Wei, L.Y.: Parallel white noise generation on a GPU via cryptographic hash. In: Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games, I3D 2008, pp. 79–87. ACM, New York (2008)
17. Zafar, F., Olano, M., Curtis, A.: GPU random numbers via the tiny encryption algorithm. In: Proceedings of the Conference on High Performance Graphics. HPG 2010, pp. 133–141. Eurographics Association, Aire-la-Ville (2010)
18. Biryukov, A.: The Design of a Stream Cipher LEX. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 67–75. Springer, Heidelberg (2007)
19. L'Ecuyer, P., Simard, R.: On the performance of birthday spacings tests with certain families of random number generators. Math. Comput. Simul. 55(1-3), 131–137 (2001)
20. Flajolet, P., Odlyzko, A.M.: Random Mapping Statistics. In: Quisquater, J.-J., Vandewalle, J. (eds.) EUROCRYPT 1989. LNCS, vol. 434, pp. 329–354. Springer, Heidelberg (1990)
21. Knuth, D.E.: Art of Computer Programming. Fundamental Algorithms, vol. 1. Addison-Wesley (July 2002)
22. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE. In: Submission to NIST, Round 3 (2010)
23. Chambers, W.G.: On Random Mappings and Random Permutations. In: Preneel, B. (ed.) FSE 1994. LNCS, vol. 1008, pp. 22–28. Springer, Heidelberg (1995)
24. Naus, J.I.: An extension of the birthday problem. The American Statistician 22(1), 27–29 (1968), http://www.jstor.org/stable/2681879
25. NVIDIA: CUDA Toolkit 4.0 CURAND Guide (January 2011)
26. Zafar, F., Curtis, A., Olano, M.: GPU Random Numbers via the Tiny Encryption Algorithm. In: HPG 2010: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on High Performance Graphics, Saarbrucken, Germany (June 2010)
27. Walker, J.: A Pseudorandom Number Sequence Test Program (January 2008), http://www.fourmilab.ch/random/
28. Marsaglia, G.: The Marsaglia random number CDROM including the DIEHARD battery of tests of randomness (1996), http://stat.fsu.edu/pub/diehard
29. L'Ecuyer, P., Simard, R.: TestU01: A C library for empirical testing of random number generators. ACM Trans. Math. Softw. 33(4), 22 (2007)
30. Aumasson, J.P., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba, 470–488 (2008)
31. Aumasson, J.-P., Guo, J., Knellwolf, S., Matusiewicz, K., Meier, W.: Differential and Invertibility Properties of BLAKE. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, pp. 318–332. Springer, Heidelberg (2010)

Parallel Quantum Algorithm for Finding the Consistency of Saaty's Matrices

Henryk Piech and Olga Siedlecka-Lamch

Institute of Computer and Information Sciences,
Czestochowa University of Technology,
Dabrowskiego 73, Czestochowa 42-201, Poland
{henryk.piech,olga.siedlecka}@icis.pcz.pl

Abstract. Features of the quantum systems enable simple calculation of the eigenvalues in uniquely - parallel way. We propose the use of quantum algorithms for the implementation of an iterative method to search for consistency in the Saaty's matrix of relative judgments, by step by step closing up to the consistent matrix structure. Typically, the matrix of relative judgments is prepared on the basis of the opinion of an expert or group of experts. In practice if is necessary to obtain consistent form of opinions set, but when we want to get the desired level of their consistency we must even in the minimal scope correct them. Criteria of correction are: the minimum number of seats of correcting (the fastest convergence to the consistency) or minimum summary value of alterations. In our proposition we can choose several variants of the iterative corrections as a way of consolidation. The method most often chosen by experts is based on the minimal correction in every iteration. Sometimes we want to make minimal iteration steps. Details of this classical approach are presented in [9].

In this paper we want to support the classical algorithm by the quantum convention and parameters. The measurement realization will be connected with the state transition and reading of the eigenvalue. The superposition procedure will be activated after the transition, what causes the change of the probability of the choice of location of next correction(s). In the aspect of quantum calculations we use quantum vectors, qubits, inner and tensor vectors, linear operators, projectors, gates etc. The resulting effect (for simulation of quantum calculations) concerning the complexity of the calculations is comparable to the classical algorithm.

Keywords: quantum algorithms, consistency, Saaty's matrices.

1 Introduction

Quantum computers perform calculations by using physical systems that are subject to laws and the limitations of quantum mechanics [4][7]. Quantum theory introduces the concept of quantum state of the system. Quantum state is a full description of the quantum system. Often the actual state of the basis

system is presented as a superposition of basis states (achievable) [1][3][8][12]. The basic concept is the measurement.
In the quantum model we should not analyze the intermediate state of the system, because each observation, in generally, destroys the superposition. However, computer simulations are allowed to read some of the parameters specific to a given state for example eigenvector described by actual state [2][6][11]. At the same time there is a danger that the systems which are exactly in the same state before measurement, after the measurement will be in different states. Quantum algorithms work on quantum registers. The most widely used way of describing a quantum algorithm is to present the quantum gates that implement it. Quantum gate is a transformation acting on the registry and changing the state of the system. In the quantum theory it exists the requirement placed on the reversibility of such a transformation. Measurement is not reversible, that means that we cannot return to previous state. Often exists the ability to simultaneously perform calculations, or implement them in parallel for different states of the base, since any measurement in any way changes the states of the same basis system [5][13][15]. Therefore designing a quantum algorithm consists in using quantum strategies and adjust the probabilities to obtain the desired state of the system. In some algorithms, based on a combination of several quantum systems, a entanglement is used, but in our solution, or popular Grover and Shor algorithms, this phenomenon does not occur.

2 Vector Variants for Saaty's Matrices

In the structures of relative judgments, we can extract the estimators of various objects, attributes, experts and criteria. Applications of Saaty's matrix structures are extensive and extract the most important (from the viewpoint of the prospective use) groups (sets), according to representatives of the characteristics of estimation. According toclassical assumption for Saaty's matrix, the size of the judgement structure should be not more than 10. Relative judgment is represented by the relation of absolute estimators: $w(i,j) = v(i)/v(j)$, where i,j are codes of objects. An important assumption is the independence of the representatives (objects, attributes, experts, etc.). In this case we can easily exploit the vector space. However estimators are scalars and have the real (not complex) representation, but their ordered sets can be treated as vector elements.
Let us start with the construction of the relative judgments matrix structure. We can describe this structure with help of the Hermitian transformation that means - with using tensor product: $MS = \mathbf{V} \otimes \mathbf{RV}^{+}$, where $\mathbf{RV}^{+}$ is Hermit's adjoint (generated by the complex adjoint and the transposition) of vector of the absolute judgment invertion:

$$\mathbf{RV}^{+} = \{rv(1) = 1/v(1), rv(2) = 1/v(2), ..., rv(n) = 1/v(n)\}.$$

The judgments matrix structure is created in the following way:

$$\begin{pmatrix} v(1) \\ v(2) \\ \vdots \\ v(n) \end{pmatrix} \otimes \big(1/v(1),\, 1/v(2),\, \cdots, 1/v(n)\big) = \begin{pmatrix} v(1)/v(1) & v(1)/v(2) & \cdots & v(1)/v(n) \\ v(2)/v(1) & v(2)/v(2) & \cdots & v(2)/v(n) \\ \vdots & \vdots & \ddots & \vdots \\ v(n)/v(1) & v(n)/v(2) & \cdots & v(n)/v(n) \end{pmatrix}.$$

The matrix of relative judgments take a form of a linear operator described in the quantum convention:

$$U \equiv \sum_{i=1}^{n} |v(i)\rangle\langle rv(j)|.$$

The nature of the Saaty's matrix let us to confirm the permanent value of the scalar product. It is independ of the relative consistency of judgments:

$$\sum_{i=1}^{n} \langle rv(i)|v(i)\rangle = n \quad \text{or} \quad \sum_{j=1}^{n}\sum_{i=1}^{n} \langle v(j)rv(i)|v(i)rv(j)\rangle = n^2$$

In practice, experts present relative judgments $w(i,j)$ as:

$$MS' = \begin{pmatrix} w(1,1) & w(1,2) & \cdots & w(1,n) \\ w(2,1) & w(2,2) & \cdots & w(2,n) \\ \vdots & \vdots & \ddots & \vdots \\ w(n,1) & w(n,1) & \cdots & w(n,n) \end{pmatrix}. \tag{1}$$

MS' matrix structure may be inconsistent [5]. The scale of the inconsistency can be evaluated by selecting the corrective row (or column) number and calculating the relative correction (relating to amending the structure of the element) [2]:

$$d(i,j) = w(i,j) - w(k,j)/w(k,i) \text{ if we choose correction row k,}$$
$$d(i,j) = w(i,j) - w(i,k)/w(k,j) \text{ if we choose correction column k.}$$

Using these correcting increments we can build consistent Saaty's matrix $MS'' = MS^k = MS' - D$, where MS'' is the matrix of relative judgments after the correction procedure, MS^k is the matrix consistent with respect to the element k, D is the matrix of corrections. If we choose the first row as a correcting element $k = 1$ then we can write:

$$MS'' = \begin{pmatrix} w(k,1) & w(k,2) & \cdots & w(k,n) \\ w(k,1)/w(k,2) & w(k,2)/w(k,2) & \cdots & w(k,n)/w(k,2) \\ \vdots & \vdots & \ddots & \vdots \\ w(k,1)/w(k,n) & w(k,2)/w(k,n) & \cdots & w(k,n)/w(k,n) \end{pmatrix}$$

$$= \begin{pmatrix} w(1,1) & w(1,2) & \cdots & w(1,n) \\ w(2,1) - d(2,1) & w(2,2) - d(2,2) & \cdots & w(2,n) - d(2,n) \\ \vdots & \vdots & \ddots & \vdots \\ w(n,1) - d(n,1) & w(n,2) - d(n,2) & \cdots & w(n,n) - d(n,n) \end{pmatrix}. \tag{2}$$

Of course, $w(i,i) = 1$ and $d(i,i) = 0$. The description of correction column will be only a little different. The matrix is a set of data needed to determine the amplitudes of transitions to the next state [8]. Generally new states are presented as matrices structures $MS^{(k)} = (MS^{(k-1)}, w^{(k)}(i,j) = w^{(k-1)}(i,j) - d^{(k-1)}(i,j), w^{(k)}(j,i) == 1/w^{(k)}(i,j))$, where k is number of the transition. State's transitions are realized with probability: $p(i,j) = d(i,j)^2/||D||^2$, where $||D|| = \sqrt{\sum_{i=1}^{n}\sum_{i=1}^{n} d(i,j)^2}$.

Let us introduce the base vectors and matrices that represent them:

$$|00\cdots 0\rangle = |bin\{0\}\rangle = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \; |00\cdots 1\rangle = |bin\{1\}\rangle = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \cdots$$

$$\cdots |11\cdots 1\rangle = |bin\{n\}\rangle = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}.$$

Let us introduce linear operator $U_e \equiv |i\rangle\langle j|$, where $|i\rangle\langle j|$ are base vectors. Assuming that the element "1" in the matrices are in positions i and j corresponding to the value of $d(i,j)$ selected with probability $p(i,j)$. Then we obtain a matrix with only one element "1":

$$U_e = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} 0\,0, \cdots 1 \cdots 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & \cdots & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1(pos(i,j)) & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & \cdots & 0 \end{pmatrix}. \tag{3}$$

Transition to the new state is realised with help of the linear, unitary operator U_e (in every step the location of element "1" changes). The Saaty's matrix after correction:

$$MS^{(k)} = MS^{(k-1)} - U_e D^{(k-1)} - U_e^T D^{(k-1)}.$$

Action of linear operator U_e requires the use of gate corresponding to the operation $U_e + U_e^T$. This gate is represented by the matrix:

$$|i\rangle\langle j| + |j\rangle\langle i| = \begin{pmatrix} 0 & 0 & \cdots & 0 & \cdot & 0 \\ 0 & 0 & \cdots & 1(pos(i,j)) & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 1(pos(j,i)) & & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & \cdots & 0 \end{pmatrix}.$$

3 Eigenvalues of Relative Judgements Operator and Assumption for Correction of Consistency

A complex number λ is called an eigenvalue of the operator U if there exists a vector $v \neq 0$ such that $Uv = \lambda v$. Vector v is a eigenvector of U associated with the eigenvalue λ. Eigenvalues of self-adjoint operator [6] are real numbers. Operator U in space H_n corresponds to the Hermitian matrix MS'. Eigenvector v has a corresponding description: $v(1), v(2), ..., v(n)$. Eigenvalue equation becomes:

$$\begin{pmatrix} w(1,1) & w(1,2) & \cdots & w(1,n) \\ w(2,1) & w(2,2) & \cdots & w(2,n) \\ \vdots & \vdots & \ddots & \vdots \\ w(n,1) & w(n,1) & \cdots & w(n,n) \end{pmatrix} \begin{pmatrix} v(1) \\ v(2) \\ \vdots \\ v(n) \end{pmatrix} = \lambda \begin{pmatrix} v(1) \\ v(2) \\ \vdots \\ v(n) \end{pmatrix}$$

and after the transformation we obtain

$$\begin{pmatrix} w(1,1)-\lambda & w(1,2) & \cdots & w(1,n) \\ w(2,1) & w(2,2)-\lambda & \cdots & w(2,n) \\ \vdots & \vdots & \ddots & \vdots \\ w(n,1) & w(n,1) & \cdots & w(n,n)-\lambda \end{pmatrix} \begin{pmatrix} v(1) \\ v(2) \\ \vdots \\ v(n) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Necessary and sufficient condition for the existence of solution to the system of equations is a linear relationship $MS' - \lambda\mathbf{I}$, where $\mathbf{I}$ is the unit matrix. Simple sum of bases of subspaces $\mathbf{H}(\lambda_i)$ is the base of the whole Hilbert space $\mathbf{H}$, where $\lambda_i,\ i = 1, 2, ..., k$ are all eigenvalues of the self-adjoint linear operator. Projection operators in the corrected subspace are defined as follows:

$$P_i v = \langle w|v_i\rangle w,\ \ i = 1, 2, ..., n (\text{correction elements - selected row, column}) \quad (4)$$

Vectors that corresponds to projection are described as:

$$w'(j,k) = w(i,j)v(i)w(j,k) \text{ for } i \neq j,\ \ k = 1, 2, ..., n \quad (5)$$

or

$$w'(j,k) = 1/(n-1) \sum_{i=1, i\neq j}^{n} w(i,j)v(i)w(j,k). \quad (6)$$

4 The Physical Interpretation of Quantum Conversion

Hilbert space is a mathematical apparatus, by means of which reality for quantum systems may be described. Quantum circuit is described by quantum states. In our case the code of the state is represented by the matrix (3). Tracking changes in the system is made by a quantum measurement. The result of a measurement is the transition to one of the possible states of the system. The parameters of the state which will be used for measurement are described by the matrix (1). The conversion process is connected with eigenvalue measurements and the transition to the new state. In our example we have n^2, which reflects the random chooice of the correction location. The time evolution of quantum system is realized by a universal operator. Linear Hermitian operator corresponds to each measurement, which takes place in the state space. The eigenvalue in classical computation is calculated with help of characteristics of corrections and states matrices. The new state can be also an equivalent to changes of row and column according to equations (4),(5),(6). The probability of the transition to the new state equals: $p(i,j) = ||P_i v||^2 = \langle P_i v | P_i v\rangle = 1/n$ or

$$p(i) = ||P_i v - w||^2 = \langle P_i v - w | P_i v - w\rangle =$$

$$\sum_{k=1}^{n} \{w(k,i)v(i)w(j,k) - w(j,k)\}^2 / \sum_{j=1}^{n}\sum_{k=1}^{n}\{w(k,i)v(i)w(j,k) - w(j,k)\}^2,$$

according to 4. The linear operator in our variant equals $U_e = |i\rangle\langle 2^n - 1|$, where $\langle 2^n - 1|$ will be represented by the vector $(1,\ 1, ..., 1)$ with length n. The sum of probabilities of transition to all possible states obviously equals 1. Expected value of system's measurement resulted in state v for the Hermitian operator U has the following form:

$$\langle v|Uv\rangle = \langle \sum_{i=1}^{n} P_i v | U \sum_{j=1}^{n} P_j v\rangle = \sum_{i=1}^{n}\sum_{j=1}^{n} \lambda_j \langle v|P_i P_j v\rangle = \sum_{i=1}^{n} \lambda_i \langle v|P_i P_i v\rangle =$$

$$= \sum_{i=1}^{n} \lambda_i \langle P_i v | P_i v\rangle = \sum_{i=1}^{n} \lambda_i p_i, \tag{7}$$

and the variance of measurement result in state v becomes:

$$\Delta^2 U(v) = \sum_{i=1}^{n} p_i(\lambda_i - \sum_{i=1}^{n} \lambda_i p_i v)^2 = \sum_{i=1}^{n} p_i \lambda_i^2 - (\sum_{i=1}^{n} \lambda_i p_i v)^2 =$$

$$= \langle v|U^2|v\rangle - \langle v|U|v\rangle^2. \tag{8}$$

We can assume, that the operator in the space of system states depends on the time in a continuous manner, thus, if the system at time 0 was in state v, so at the moment t will be in the state $U(t)v$. Operators transform the system state, but if we want to get a specific value of the parameter we have to make a measurement. After the measurement the system will be in the new state and information about the previous state cannot be recovered. The measurement is in fact an irreversible operation. To follow the state changes makes sense until we reach zero or a given level of inconsistency $((\lambda = n)$ or $|\lambda - n| < \epsilon)$. But according to the quantum theory the system will realize the infinite number of iterations. In these cases we get either independence from the time (stationary states [10]) or the expected value of the measurement result can stay on constant level: $\frac{d}{dt}\langle v(t)|Uv(t)\rangle = 0$. After every transition also the probabilities of transition will change. Every state is determined by discrete part of the operator U. The characteristics of these states and relative judgements of selected items change. They are components of the operator. We can therefore conclude that the spectrum of the operator may consist of:

- discrete spectrum consisting of a finite number of states,
- continuous spectrum of probabilities of state transitions,
- continuous spectrum of correction values,
- continuous spectrum of relative judgments of objects,
- continuous spectrum corresponding to judgments and operator values of the eigenvector ($Uv = \lambda v$).

Although quantum theory forbids the cloning of state of the system [2][11] but it is possible to create an operator to copy any given basis state [3]. We can also transpose a state of quantum register to another.

5 Parallel Superposition of Quantum Matrix Characteristics in Simulation Variant

One of the important assumptions in quantum approach is the exploitation of superposition algorithm for transition of states and the change of amplitudes. In our problem we have the matrix of states and the matrix of corrections. The second one is used to determine the first: $D \rightarrow P \rightarrow MS \rightarrow U_e$. The correction process consists of independent operations in every iteration and implements the superposition strategy. As a result we obtain the following corrections matrix:

$$D_k = \begin{pmatrix} d_k(1,1) & d_k(1,2) & \cdots & d_k(1,n) \\ d_k(2,1) & d_k(2,1) & \cdots & d_k(1,1) \\ \vdots & \vdots & \ddots & \vdots \\ d_k(n,1) & d_k(n,2) & \cdots & d_k(n,n) \end{pmatrix}, \tag{9}$$

or accumulated corrections matrix:

$$SD_k = \begin{pmatrix} \sum_{l=1}^k d_l(1,1) & \sum_{l=1}^k d_l(1,2) & \cdots & \sum_{l=1}^k d_l(1,n) \\ \sum_{l=1}^k d_l(2,1) & \sum_{l=1}^k d_l(2,1) & \cdots & \sum_{l=1}^k d_l(1,1) \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{l=1}^k d_l(n,1) & \sum_{l=1}^k d_l(n,2) & \cdots & \sum_{l=1}^k d_l(n,n) \end{pmatrix}, \tag{10}$$

where SD_k is the accumulated correction matrix after k iterations, $d_l(i,j)$ is a value of correction on position (i,j) after l iterations, n is a number of judgment objects (n^2 - maximal number of used processors in simulation process). Superposition strategy is implemented on the set of processors because all operations in particular iterations are treated as independent. The corrections matrix D will be used to define the matrix:

$$P_k = \begin{pmatrix} d_k^2(1,1)/mw & d_k^2(1,2)/mw & \cdots & d_k^2(1,n)/mw \\ d_k^2(2,1)/mw & d_k^2(2,1)/mw & \cdots & d_k^2(1,1)/mw \\ \vdots & \vdots & \ddots & \vdots \\ d_k^2(n,1)/mw & d_k^2(n,2)/mw & \cdots & d_k^2(n,n)/mw \end{pmatrix}, \tag{11}$$

where $mw = \max_{i,j=1,\ldots,n}\{w(i,j)\}^2$ is correction rank.

Subsequent iterations come down the system of judgments to a consistent state $|w_0\rangle$. If we rely solely on one correction element, then the amplitude will decrease monotonically. If a different correction element is exploited, then the system will converge monotonically to consistency. The variance of eigenvector components tends to zero. Starting from the premise that a system of judgments is in superposition of base states with equal amplitudes, the initial state is described by formula:

$$|v_1\rangle = \frac{1}{n}\sum_{j=1}^{n}\frac{|w^{(0)}\rangle}{\sum_{i=1}^{n} w^{(0)}(i,j)}$$

and the following iterations are implemented according to convention:

$$MS^{(k)} = MS^{(k-1)} - U_e D^{(k-1)} - U^T D^{(k-1)} \tag{12}$$

$$|v_k\rangle = \frac{1}{n}\sum_{j=1}^{n}\frac{|w^{(k-1)}\rangle}{\sum_{i=1}^{n} w^{(k-1)}(i,j)}$$

Elements of the eigenvector can be calculated in the simulation in another way, for example by the method in [9].
Parallel quantum algorithm was realised with use of C++ language and implemented in the following steps:

Step 1: The measurement of the relative judgments eigenvector. One processor is engaged in simulation [5],[9],
Step 2: Values of corrections are calculated with help of correction element (matrix D or SD (10)). n^2 processors are engaged in simulation,
Step 3: Probability distribution for state transitions is built on the base of corrections matrix (12). n^2 processors are engaged in simulation,
Step 4: Generation of the new state according to the probability distribution. n^2 processors are engaged in simulation [9],
Step 5: Procedure of judgments correction, engaging n^2 processors,
and return to Step1.

In theory, the acceleration of parallel simulation of quantum algorithm should achieve $0.8n^2$ speedup, but in practice, due to processor communication the acceleration achieves value not greater then $0.1n^2$.

We present the results of statistical analysis which takes into account the different levels of inconsistency, the size of the Saaty's matrix while one can observe the convergence of the system consistency for: - classic calculations,
- simulation of quantum calculations,
- quantum calculations.

These results are presented using a normalized parameter of time, for maximum conversion time for a classical variant. We show diagrams comparing the results of statistical analysis (fig.1-3).

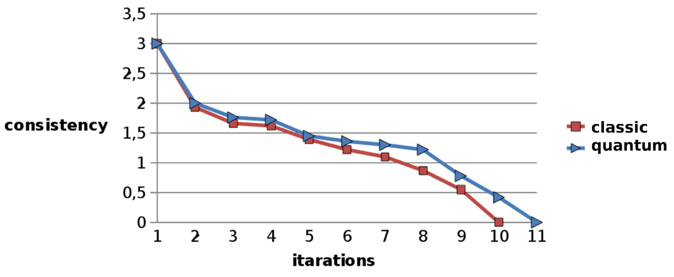


Fig. 1. Diagram of the convergence to consistency for classic calculations and parallel realization of simulation of quantum algorithm

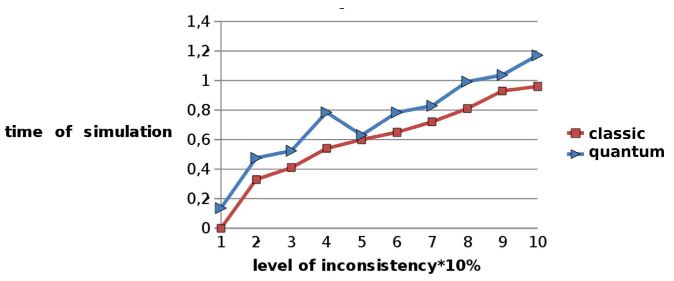


Fig. 2. Time of realization classical calculations and parallel simulation of quantum algorithm

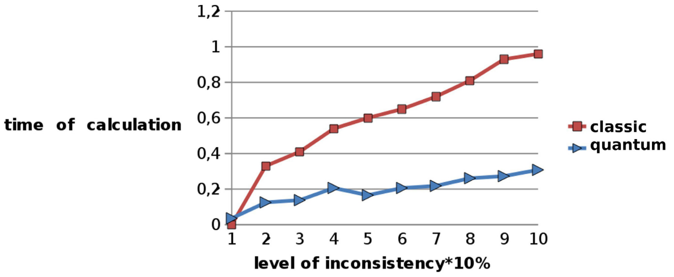


Fig. 3. Time of realization classical and quantum calculations in dependency of level of inconsistence

6 Conclusions

Parallel quantum algorithm, as in the classical version, allow to accelerate the search of consistency, but the complexity remains the same. The limitation, typical for a quantum variant inferring from the superposition convention, do not increases complexity at all. Additional acceleration of the calculations (fig.3) is the result of the characteristics of quantum systems based on the physical nature of quantum machines (use of spin of electron or ions trapping). Results based on statistical analysis can give a little better effect (about 7%) when the corrections matrix (but not accumulated) is used to create a matrix of probabilities (10). Simulation takes into account the limitations and requirements of quantum theory of computation [12]. These requirements can not improve the efficiency of the calculations in terms of algorithm structure.

References

1. Berman, G.D., Doolen, G.D., Mainieri, R., Tsifrinovich, V.I.: Introduction to Quantum Computers. World Scientific, Singapore (1998)
2. Berthiaume, A., Brassard, G.: The quantum challenge to structural complexity theory. In: Proc. 7th IEEE Conf. Structure in Complexity Theory (1992)
3. Broadsky, A., Pippenger, N.: Characterizations of 1-way quantum finite automata. SIAM J.Comput. 31, 1456–1478 (2002)
4. Cohen, D.W.: An Introduction to Hilbert Space and Quantum Logic. Springer, New York (1989)
5. Gruska, J.: Quantum Computing. McGraw-Hill, London (1999)
6. Gudder, S.: Basic properties of quantum automata. Found. Phys. 30, 301–319 (2000)
7. Kozen, D.C.: Automata and Computability. Springer, New York (1997)
8. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge (2000)
9. Piech, H., Bednarska, U.: Iterative Method for Improving Consistency of Multi-attribute Object Judgments Performed by Teams of Decision Makers. In: Jędrzejowicz, P., Nguyen, N.T., Howlet, R.J., Jain, L.C. (eds.) KES-AMSTA 2010. LNCS (LNAI), vol. 6071, pp. 150–159. Springer, Heidelberg (2010)
10. Paz, A.: Introduction to Probabilistic Automata. Academic Press, New York (1971)
11. Shor, P.W.: Polynomial-time algorithm for prime factorization and discrete logarithms on quantum computer. In: Proc. 35th Ann.Symp.on Foundations of Computer Science, Santa Fe. IEEE Computer Society Press, Silver Spring (1994)
12. Siedlecka, O.: A Brief Overview of Quantum Computing Theory. Computing, Multimedia and Intelligent Techniques 2(1), 35–44 (2006)
13. Townsend, J.S.: A Modern Approach to Quantum Mechanics. McGraw-Hill, New York (1992)
14. Williams, P., Clearwater, S.H.: Explorations in Quantum Computing. Springer, New York (1998)
15. Yao, A.: Quantum circuit complexity. In: Proc. 34th IEEE Symp. on Foundations of Computer Science (1993)

A Numerical Approach to the Determination of 3D Stokes Flow in Polygonal Domains Using PIES

Eugeniusz Zieniuk, Krzysztof Szerszen, and Marta Kapturczak

Faculty of Mathematics and Computer Science,
University of Bialystok,
Sosnowa 64, 15-887 Bialystok, Poland
{ezieniuk,kszerszen,mkapturczak}@ii.uwb.edu.pl

Abstract. This paper discusses a generalization of intensively developed parametric integral equation system (PIES) to solve 3D steady Stokes flow. Given the preliminary nature of this research, the effectiveness of the generalized PIES has been verified for the solutions of the Stokes problems defined on polygonal domains. The boundary of such domains has been modeled directly in PIES by joining rectangular Coons parametric patches of the first degree. With them it is possible to model relatively large linear segments of the boundary by small number of corner points of the considered polygonal domain. Two numerical examples were used to validate the solutions of PIES with analytical and numerical results available in the literature.

Keywords: Stokes flow, boundary integral equations (BIE), parametric integral equation system (PIES), Coons parametric surface patches.

1 Introduction

For many years the authors of this paper have been used parametric integral equation system (PIES) to solve boundary value problems. So far, however, PIES has been mainly used to solve 2D potential boundary value problems modeled by partial differential equations such as: Laplace [7], Poisson [8], Helmholtz [9] and Navier-Lame [10]. These equations have been written in the alternative form, with the help of PIES, which take into account in its mathematical formalism the shape of the boundary modeled by curves known from computer graphics. The shape of the boundary could be defined by such curves as: Bezier, B-spline, Hermite, and its definition is practically reduced to giving a small set of control points. The complexity of modeling the shape of the boundary in PIES depends on the complexity of the shape of the concerned boundary problem. However, this eliminates the need for definition of traditional boundary or finite elements. Traditional approaches based on finite or boundary element geometry are ineffective in many cases, for example in synthesis problems, that require an iterative process. This inefficiency, especially even more increases in the case of boundary problems defined in 3D domains.

The results obtained in PIES for problems modeled by mentioned equations were compared with the results obtained by FEM and BEM. High accuracy and effectiveness of PIES for those problems has been encouraging its generalization to the 3D boundary problems. So far, in the case of 3D problems, we have only dealt with the problems for Laplace and Helmholtz [12] equations. In order to illustrate the numerical solution of the Laplace equation in PIES we consider the distribution of temperature fields [11] as well as potential flow of a perfect fluid [13]. Our numerical results confirm the effectiveness of the proposed strategy from the point of view of simplifying the geometry modeling process as well as improving the accuracy of obtained solutions compared with element methods.

The aim of this paper is to explore the potential of using PIES for solving 3D Stokes flows. PIES for the Stokes equation is obtained similarly as for the previously mentioned partial differential equations.

2 PIES for the Stokes Equation on Polygonal Domains

A three dimensional steady-state Stokes differential equation is a linearized approximation of Navier-Stokes equation for low Reynolds numbers and formulated in velocity-pressure form as [6]

$$\begin{aligned} \mu\nabla^2\boldsymbol{u} - \nabla p = 0, \\ \nabla\boldsymbol{u} = 0, \end{aligned} \tag{1}$$

where are $\boldsymbol{u}$ is fluid velocity vector, p is the pressure of viscous incompressible fluid flow and μ is the dynamic viscosity of the fluid.

PIES for Stokes equations in 3D was obtained as a result of the analytical modification of the traditional BIE. Methodology of this modification for potential problems on 2D polygonal domains is presented in [10] and on 2D curved domains in [7]. Generalizing this modification for Stokes equations in 3D, we obtain the following formula of PIES

$$\begin{aligned} 0.5\boldsymbol{u}_l(v_1,w_1) = \sum_{j=1}^{n}\int_{v_{j-1}}^{v_j}\int_{w_{j-1}}^{w_j} \{ & \bar{\boldsymbol{U}}_{lj}{}^*(v_1,w_1,v,w)\boldsymbol{p}_j(v,w) \\ & -\bar{\boldsymbol{P}}_{lj}{}^*(v_1,w_1,v,w)\boldsymbol{u}_j(v,w)\}J_j(v,w)dvdw, \end{aligned} \tag{2}$$

where

$$v_{l-1} < v_1 < v_l, w_{l-1} < w_1 < w_l, v_{j-1} < v < v_j, w_{j-1} < w < w_j, l = 1,2,3,...,n,$$

and n is the number of parametric patches that create the domain boundary in 3D.

The integrands $\bar{\boldsymbol{U}}_{lj}{}^*(v_1,w_1,v,w)$, $\bar{\boldsymbol{P}}_{lj}{}^*(v_1,w_1,v,w)$ in equation (2) are represented in the following matrix form

$$\bar{\boldsymbol{U}}_{lj}{}^*(v_1,w_1,v,w) = -\frac{1}{8\pi\mu\eta}\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ U_{21} & U_{22} & U_{23} \\ U_{31} & U_{32} & U_{33} \end{bmatrix}, \tag{3}$$

$$\bar{\boldsymbol{P}}_{lj}^{\;*}(v_1, w_1, v, w) = \frac{6}{8\pi\eta^5} \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{bmatrix}. \tag{4}$$

The individual elements in matrix (3) can be expressed in an explicit form as

$$\begin{array}{lll} U_{11} = 1 + \eta_1^2/\eta^2, & U_{12} = \eta_1\eta_2/\eta^2, & U_{13} = \eta_1\eta_3/\eta^2, \\ U_{21} = \eta_2\eta_1/\eta^2, & U_{22} = 1 + \eta_2^2/\eta^2, & U_{23} = \eta_2\eta_3/\eta^2, \\ U_{31} = \eta_3\eta_1/\eta^2, & U_{32} = \eta_3\eta_2/\eta^2, & U_{33} = 1 + \eta_3^2/\eta^2, \end{array}$$

while in matrix (4) by

$$\begin{array}{ll} P_{11} = \eta_1^3 n_1 + \eta_1^2\eta_2 n_2 + \eta_1^2\eta_3 n_3, & P_{12} = \eta_1^2\eta_2 n_1 + \eta_1\eta_2^2 n_2 + \eta_1\eta_2\eta_3 n_3, \\ P_{13} = \eta_1^2\eta_3 n_1 + \eta_1\eta_2\eta_3 n_2 + \eta_1\eta_3^2 n_3, & P_{21} = \eta_1^2\eta_2 n_1 + \eta_1\eta_2^2 n_2 + \eta_1\eta_2\eta_3 n_3, \\ P_{22} = \eta_1\eta_2^2 n_1 + \eta_2^3 n_2 + \eta_2^2\eta_3 n_3, & P_{23} = \eta_1\eta_2\eta_3 n_1 + \eta_2^2\eta_3 n_2 + \eta_2\eta_3^2 n_3, \\ P_{31} = \eta_1^2\eta_3 n_1 + \eta_1\eta_2\eta_3 n_2 + \eta_1\eta_3^2 n_3, & P_{32} = \eta_1\eta_2\eta_3 n_1 + \eta_2^2\eta_3 n_2 + \eta_2\eta_3^2 n_3, \\ P_{33} = \eta_1\eta_3^2 n_1 + \eta_2\eta_3^2 n_2 + \eta_3^3 n_3. & \end{array}$$

Function $J_j(v, w)$ is the Jacobian, while n_1, n_2, n_3 are the components of the normal vector to the surface patch designated by index j. Kernels (3) and (4) include in its mathematical formalism the shape of a closed boundary, created by means of appropriate relationships between surfaces $(l, j = 1, 2, 3, ..., n)$, which are defined in Cartesian coordinates using the following relations

$$\begin{gathered} \eta_1 = P_j^{(1)}(v, w) - P_l^{(1)}(v_1, w_1), \eta_2 = P_j^{(2)}(v, w) - P_l^{(2)}(v_1, w_1), \\ \eta_3 = P_j^{(3)}(v, w) - P_l^{(3)}(v_1, w_1), \eta = [\eta_1^2 + \eta_2^2 + \eta_2^2]^{0.5}, \end{gathered} \tag{5}$$

where $\boldsymbol{P}_j(v, w) = [P_j^{(1)}(v, w), P_j^{(2)}(v, w), P_j^{(3)}(v, w)]$ are the scalar components of the surface patch, which depends on the parameters v, w . This notation is also valid for the patch labeled by index l with parameters v_1, w_1, ie. for $j = l$ and for parameters $v = v_1$ and $w = w_1$.

In our previous papers we have considered functions $\boldsymbol{P}_j(v, w)$ in the form of known from computer graphics parametric surface patches. The possibility of analytical description of the boundary directly in the formula of PIES is the main advantage of the presented approach in comparison with traditional BIE. In classical BIE, a description of the boundary is not included in the mathematical formalism of the equation, but very generally defined by the integral boundary. This necessitates the discretization of the domain boundary into elements, as is the case in classical BEM. Advantages of the proposed approach of modelling the boundary, so on directly in the mathematical equations of PIES, was shown in the case 2D [7-10] as well as 3D [11-13] problems. Taking into account the obtained efficiency of the method it seems natural to consider the same way of modelling for Stokes problems.

The preliminary nature of the presented research for the Stokes equation led the authors to focus on only polygonal domains. To model such domains, rectangular Coons surface patches of the first degree was used. Coons surface

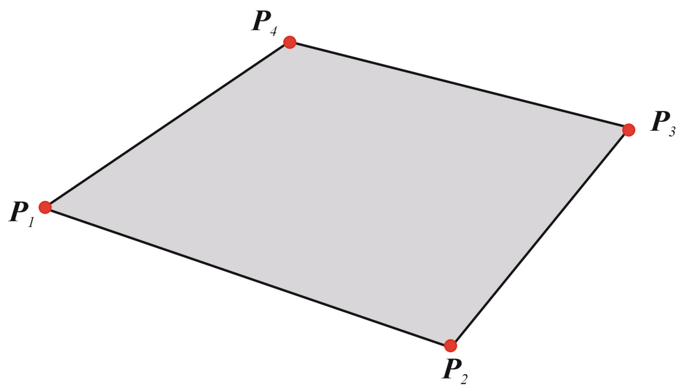


Fig. 1. Coons surface patch with 4 corner points

patch of degree 1 takes the form of a quadrangular plate defined by four points $\boldsymbol{P}_1, \boldsymbol{P}_2, \boldsymbol{P}_3, \boldsymbol{P}_4$ placed at the corners of defined rectangle as shown in Fig. 1. The location of any point of the Coons patch is dependent on two parameter v, w in accordance with the following formula [1]

$$\boldsymbol{P}(v,w) = \begin{bmatrix} 1-v & v \end{bmatrix} \begin{bmatrix} \boldsymbol{P}_1 & \boldsymbol{P}_4 \\ \boldsymbol{P}_2 & \boldsymbol{P}_3 \end{bmatrix} \begin{bmatrix} 1-w \\ w \end{bmatrix}, 0 \leq v, w \leq 1. \tag{6}$$

2.1 Approximation of the Boundary Functions over the Surface Patches

The use of PIES for solving 2D and 3D boundary problems made it possible to eliminate the need for discretization of both the mentioned above boundary geometry, and the boundary functions. In previous research works, the boundary functions both defined as the boundary conditions, as well as obtained after solving PIES are approximated by Chebyshev series. Unlike in 2D problem (modeled by Laplace's equation), boundary conditions for 3D Stokes problems are expressed in vector notation. Therefore, we have generalized previously-used approximation series to vector notation so as to represent the scalar components of the velocity vector $\boldsymbol{u}_j(v,w)$ and the viscous stresses $\boldsymbol{p}_j(v,w)$. These boundary functions $\boldsymbol{u}_j(v,w)$, $\boldsymbol{p}_j(v,w)$, are defined on each Coons patch j by means of the following series

$$\boldsymbol{u}_j(v,w) = \sum_{p=0}^{N} \sum_{r=0}^{M} \boldsymbol{u}_j^{(pr)}\; T_j^{(p)}(v)\; T_j^{(r)}(w), \tag{7}$$

$$\boldsymbol{p}_j(v,w) = \sum_{p=0}^{N} \sum_{r=0}^{M} \boldsymbol{p}_j^{(pr)}\; T_j^{(p)}(v)\; T_j^{(r)}(w),$$

where $\boldsymbol{u}_j^{(pr)}$, $\boldsymbol{p}_j^{(pr)}$ are requested coefficients and $T_j^{(p)}(v)$, $T_j^{(r)}(w)$ are Chebyshev polynomials and $\bar{n} = N * M$ number of terms in the Chebyshev series. One of these functions, either $\boldsymbol{u}_j(v,w)$ or $\boldsymbol{p}_j(v,w)$, depending on the type of the resolved boundary problem, is posed in the form of boundary conditions, whereas the other is the searched function resulting from the solution of PIES. After substituting series (6) to PIES (1) we obtain the following formula

$$0.5\boldsymbol{u}_l(v_1,w_1) = \sum_{j=1}^{n}\sum_{p=0}^{N}\sum_{r=0}^{M}\{\boldsymbol{p}_j^{(pr)} \int_{v_{j-1}}^{v_j}\int_{w_{j-1}}^{w_j} \bar{\boldsymbol{U}}_{lj}{}^*(v_1,w_1,v,w)$$
$$-\boldsymbol{u}_j^{(pr)} \int_{u_{j-1}}^{v_j}\int_{w_{j-1}}^{w_j} \bar{\boldsymbol{P}}_{lj}{}^*(v_1,w_1,v,w)\}\ T_j^{(p)}(v)\ T_j^{(r)}(w)J_j(v,w)dvdw. \qquad (8)$$

Next writing down expression (8) at the collocation points on individual Coons patches we obtain an algebraic equation system with respect to the unknown coefficients $\boldsymbol{u}_j^{(pr)}$ or $\boldsymbol{p}_j^{(pr)}$.

2.2 Solutions in the Domain

After solving PIES we obtain the solution of the boundary problem only on its boundary, represented by series (7). To find solution in the domain we need to obtain an integral identity known for BIE that makes use of the solution on the boundary obtained by PIES. After using similar modifications as in the case of 2D problems [7,10] we have had an integral identity which used solutions (7) at the boundary, previously obtained by PIES. The modified identity takes the following form

$$\boldsymbol{u}(\boldsymbol{x}) = \sum_{j=1}^{n}\int_{v_{j-1}}^{v_j}\int_{w_{j-1}}^{w_j}\left\{\hat{\bar{\boldsymbol{U}}}_j^*(\boldsymbol{x},v,w)\boldsymbol{p}_j(v,w) - \hat{\bar{\boldsymbol{P}}}_j^*(\boldsymbol{x},v,w)\boldsymbol{u}_j(v,w)\right\} J_j(v,w)dvdw. \qquad (9)$$

The integrands appearing in the identity (9) are expressed in the following form

$$\hat{\bar{\boldsymbol{U}}}_{lj}^*(\boldsymbol{x},v,w) = -\frac{1}{8\pi\mu\hat{r}}\begin{bmatrix}\hat{U}_{11} & \hat{U}_{12} & \hat{U}_{13}\\ \hat{U}_{21} & \hat{U}_{22} & \hat{U}_{23}\\ \hat{U}_{31} & \hat{U}_{32} & \hat{U}_{33}\end{bmatrix}, \qquad (10)$$

$$\hat{\bar{\boldsymbol{P}}}_{lj}^*(\boldsymbol{x},v,w) = \frac{6}{8\pi\hat{r}^5}\begin{bmatrix}\hat{P}_{11} & \hat{P}_{12} & \hat{P}_{13}\\ \hat{P}_{21} & \hat{P}_{22} & \hat{P}_{23}\\ \hat{P}_{31} & \hat{P}_{32} & \hat{P}_{33}\end{bmatrix}. \qquad (11)$$

The individual items in matrix (10) can be represented in explicit form as follows

$$\hat{U}_{11} = 1+\hat{r}_1^2/\hat{r}^2,\ \hat{U}_{12} = \hat{r}_1\hat{r}_2/\hat{r}^2,\ \hat{U}_{13} = \hat{r}_1\hat{r}_3/\hat{r}^2,$$
$$\hat{U}_{21} = \hat{r}_2\hat{r}_1/\hat{r}^2,\ \hat{U}_{22} = 1+\hat{r}_2^2/\hat{r}^2,\ \hat{U}_{23} = \hat{r}_2\hat{r}_3/\hat{r}^2,$$
$$\hat{U}_{31} = \hat{r}_3\hat{r}_1/\hat{r}^2,\ \hat{U}_{32} = \hat{r}_3\hat{r}_2/\hat{r}^2,\ \hat{U}_{33} = 1+\hat{r}_3^2/\hat{r}^2,$$

while in matrix (11) are introduced through

$$\hat{P}_{11} = \hat{r}_1^3 n_1 + \hat{r}_1^2 \hat{r}_2 n_2 + \hat{r}_1^2 \hat{r}_3 n_3,\ \hat{P}_{12} = \hat{r}_1^2 \hat{r}_2 n_1 + \hat{r}_1 \hat{r}_2^2 n_2 + \hat{r}_1 \hat{r}_2 \hat{r}_3 n_3,$$
$$\hat{P}_{13} = \hat{r}_1^2 \hat{r}_3 n_1 + \hat{r}_1 \hat{r}_2 \hat{r}_3 n_2 + \hat{r}_1 \hat{r}_3^2 n_3,\ \hat{P}_{21} = \hat{r}_1^2 \hat{r}_2 n_1 + \hat{r}_1 \hat{r}_2^2 n_2 + \hat{r}_1 \hat{r}_3 \hat{r}_3 n_3,$$
$$\hat{P}_{22} = \hat{r}_1 \hat{r}_2^2 n_1 + \hat{r}_2^3 n_2 + \hat{r}_2^2 \hat{r}_3 n_3,\ \hat{P}_{23} = \hat{r}_1 \hat{r}_2 \hat{r}_3 n_1 + \hat{r}_2^2 \hat{r}_3 n_2 + \hat{r}_2 \hat{r}_3^2 n_3,$$
$$\hat{P}_{31} = \hat{r}_1^2 \hat{r}_3 n_1 + \hat{r}_1 \hat{r}_2 \hat{r}_3 n_2 + \hat{r}_1 \hat{r}_3^2 n_3,\ \hat{P}_{32} = \hat{r}_1 \hat{r}_2 \hat{r}_3 n_1 + \hat{r}_2^2 \hat{r}_3 n_2 + \hat{r}_2 \hat{r}_3^2 n_3,$$
$$\hat{P}_{33} = \hat{r}_1 \hat{r}_3^2 n_1 + \hat{r}_2 \hat{r}_3^2 n_2 + \hat{r}_3^3 n_3,$$

where

$$\hat{r}_1 = P_j^{(1)}(v,w) - x_1, \hat{r}_2 = P_j^{(2)}(v,w) - x_2, \hat{r}_3 = P_j^{(3)}(v,w) - x_3,$$
$$\hat{r} = [\hat{r}_1^2 + \hat{r}_2^2 + \hat{r}_2^2]^{0.5}. \qquad (12)$$

Both integrands in the identity (9) are visually very similar to kernels (3,4). The main difference, however, lies in the fact that in kernels (10,11), apart from Coons patches defining boundary geometry, we have the coordinates of the points in domain $\boldsymbol{x} \equiv \{x_1, x_2, x_3\}$. Using these coordinates we can pose any point in the domain in which we look for the solutions.

3 Numerical Examples

In order to validate our approach, we have implemented it as a computer program. The newly developed program have been used to perform some numerical tests which are presented below. In presented examples, the accuracy of results in PIES have been compared with analytical and numerical solutions available in the literature. We consider internal problems with applied Dirichlet boundary conditions without additional source terms.

3.1 Example 1

A first example is illustrated in Fig. 2a. We consider steady state Stokes flow in a cubic cavity induced by the motion on its upper face with a constant velocity u_1 [5]. In the absence of analytical solutions for such problem, the PIES results have been compared with the available literature results. Table 1 contains the numerical results obtained for the above problem using FEM with different mesh densities: published in [3,4] and computed with open-source software package FreeFem++.

The fourth rows contain minimum values for velocity profile u_1 along x_3 at $x_1 = x_2 = 0.5$, while the fifth and sixth rows show minimum and maximum values for u_3 along x_2 at $x_1 = x_3 = 0.5$. These values slightly vary with respect to the number of used finite elements and degrees of freedom presented in the second and third rows.

In Table 2, we summarize the corresponding values of components u_1 and u_3 for the same flow problem obtained in PIES. In the absence of discretization in PIES all computations have been performed for the model of the domain

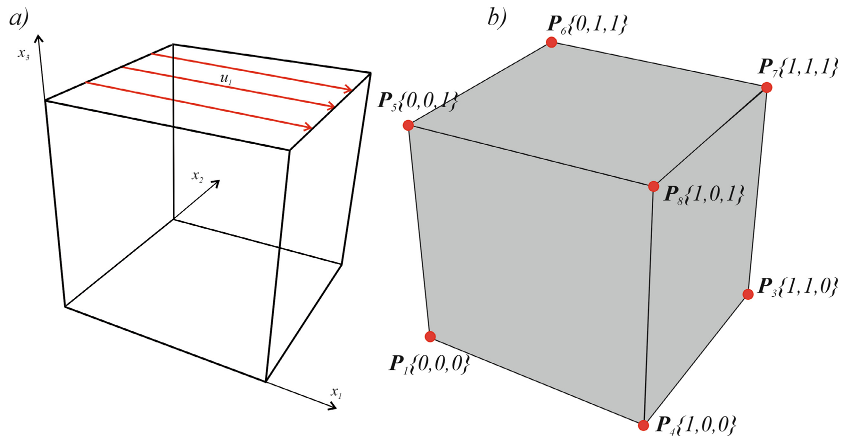


Fig. 2. Stokes flow in a cubic cavity induced (a), modeling of the cubic domain in PIES by 6 Coons patches (b)

Table 1. Comparison of results for cubic cavity obtained in FEM

Method	FEM [3,4]				FreeFem++
Number of elements	1000	27000	8000	1250000	3072
Number of nodes	1331	29791	9261	132651	4713
min u_1	-0.21316	-0.21958	-0.21884	-0.22235	-0.2276
min u_3	-0.17071	-0.18026	-0.18036	-0.18063	-0.18210
max u_3	0.17071	0.18026	0.18036	0.18063	0.18195

described by Coons patches shown in Fig. 2b. This domain has been practically defined by only eight corner points $\boldsymbol{P}_i (i = 1, ..., 8)$. Declared corner points, in turn, define six rectangular Coons surface patches.

Comparing the results from Tables 1,2 it is possible to notice that we obtain comparative values of u_1 and u_3 to FEM results with smaller input data needed to model the computational domain in PIES, which is model by only 8 corner points. Moreover, there is a important difference in the convergence investigation of solutions between PIES and FEM. In FEM, it is necessary to physically increase the number of boundary elements into which the boundary is discretized, whereas in PIES, we only need to increase the number $\bar{n}$ in the Chebyshev series. Table 2 present the results in PIES obtained for 36, 49 and 64 components of Chebyshev series (row 3) posed on each of 6 Coons patches. From the programming point of view the operation simply involves changing this number $\bar{n}$ in the program, which makes it possible to quickly verify the convergence and thus add another advantage to the proposed method. It is a considerable advantage over element methods in which the increase of accuracy involves the increase of the number of elements.

Table 2. Comparison of results for cubic cavity obtained in PIES

Method	PIES		
Number of Coons patches	6	6	6
Number $\bar{n}$ in the Chebyshev series	36	49	64
Number of equations	1944	2646	3456
min u_1	-0.22373	-0.21378	-0.21180
min u_3	-0.17894	-0.17558	-0.17435
max u_3	0.17894	0.17558	0.17435

3.2 Example 2

In the second example the modification of the computational domain modeled by Coons patches is demonstrated. Fig. 3b shows the modification of the initial cube by moving 4 corner points. The effectiveness of this modification consists in changing only the coordinates of corner points to modify the domain.

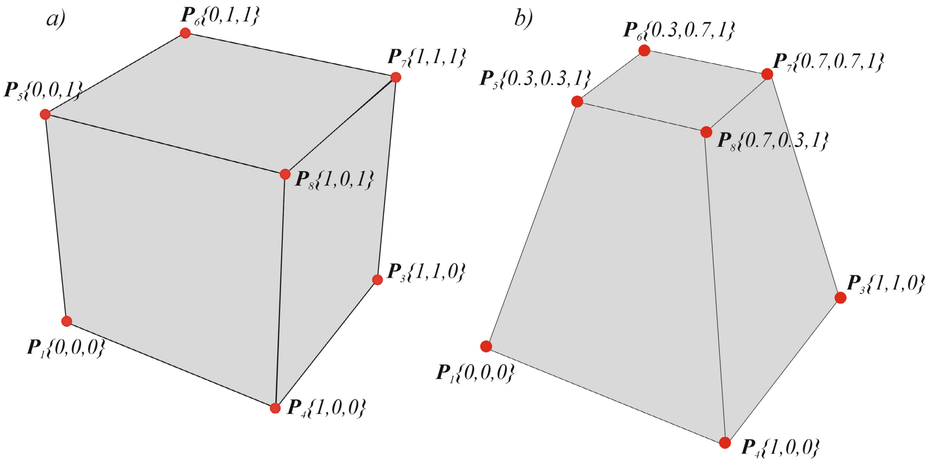


Fig. 3. Modification of the cubic domain after moving corner points

In order to evaluate proposed PIES for such modified domain, a numerical verification was performed with Dirichlet boundary conditions for the Stokes equation obtained from the following exact solutions [2]

$$\begin{aligned} u_1(x_1,x_2,x_3) &= x_1 + x_1^2 + x_1x_2 + x_1^3x_2, \\ u_2(x_1,x_2,x_3) &= x_2 + x_1x_2 + x_2^2 + x_1^2x_2^2, \\ u_3(x_1,x_2,x_3) &= -2x_3 - 3x_1x_3 - 3x_2x_3 - 5x_1^2x_2^2x_3. \end{aligned} \tag{13}$$

The impact of the number of terms $\bar{n} = N * M$ in the Chebyshev series (7) on the accuracy of the results on the boundary, and then in the domain was examined. In Table 3, we summarize the L_2 relative error norms for u_1, u_2, u_3 components

Table 3. L_2 relative error norms of solutions in PIES for two domains from Fig. 3

Number $\bar{n}$ in the Chebyshev series		4	9	16	25
Number of equations		216	486	864	1350
u_1		0.509215	0.056526	0.052355	0.052159
u_2	Fig. 3a	0.686895	0.292041	0.272546	0.271387
u_3		0.285673	0.089854	0.083754	0.083446
u_1		0.327237	0.041312	0.039998	0.038791
u_2	Fig. 3b	0.540369	0.175733	0.170385	0.171071
u_3		0.590327	0.077786	0.075169	0.071174

of solutions obtained inside two domains from Fig. 3. As expected, the accuracy of obtained solutions improves as number $\bar{n}$ of terms in series (7) is increased.

4 Conclusions

This paper generalizes the existing and intensively developed PIES scheme to solve steady-state 3D Stokes flow problems. In order to model boundary shape parametric Coons rectangular patches are used. The patches are applied in analytic modification of traditional BIE and to obtain the PIES formula for 3D Stokes problems. The explicit form of PIES for Stokes equation has been presented. In addition, it has presented the identity for solutions in the domain. Obtained PIES formula has been tested on elementary examples, but with analytical and numerical solutions. The analysis showed the previously existing advantages of PIES also in relation to the Stokes flow. These advantages are related to the simplicity of defining and modifying the shape of the edge of the declaration corner points. Discussed numerical examples show good accuracy of obtained solutions

Acknowledgements. The scientific work is founded by resources for sciences in the years 2010-2013 as a research project NN 519579538.

References

1. Farin, G.: Curves and Surfaces for CAGD: A Practical Guide. Morgan Kaufmann Publishers, San Francisco (2002)
2. Olshanskii, M.A.: Analysis of semi-staggered Finite-difference method with application to Bingham flows. Computer Methods in Applied Mechanics and Engineering 198, 975–985 (2009)
3. Murugesan, K., Lo, D.C., Young, D.L.: An effcient global matrix free finite element algorithm for 3D flow problems. Communications in Numerical Methods in Engineering 21, 107–118 (2005)
4. Shu, C., Wang, L., Chew, Y.T.: Numerical computation of three-dimensional incompressible Navier-Stokes equations in primitive variable form by DQ method. International Journal for Numerical Methods in Fluids 43, 345–368 (2003)

5. Young, D.L., Jane, S.J., Lin, C.Y., Chiu, C.L., Chen, K.C.: Solution of 2D and 3D Stokes Laws using Multiquadrics Method. Engineering Analysis with Boundary Elements 28, 1233–1243 (2004)
6. Youngren, G.K., Acrivos, A.: Stokes flow past a particle of arbitrary shape: a numerical method of solution. Journal of Fluid Mechanics 69(2), 377–403 (1975)
7. Zieniuk, E.: Bezier curves in the modification of boundary integral equations (BIE) for potential boundary-values problems. International Journal of Solids and Structures 9(40), 2301–2320 (2003)
8. Zieniuk, E., Szerszen, K., Boltuc, A.: Globalne obliczanie calek po obszarze w PURC dla dwuwymiarowych zagadnien brzegowych modelowanych ronwnaniem Naviera-Lamego i Poissona. Modelowanie Inzynierskie 33, 181–186 (2007) (in Polish)
9. Zieniuk, E., Boltuc, A.: Bezier curves in the modeling of boundary geometries for 2D boundary problems defined by Helmholtz equation. Journal of Computational Acoustics 3(14), 1–15 (2006)
10. Zieniuk, E., Boltuc, A.: Non-element method of solving 2D boundary problems defined on polygonal domains modeled by Navier equation. International Journal of Solids and Structures 43, 7939–7958 (2006)
11. Zieniuk, E., Szerszen, K.: Liniowe platy powierzchniowe Coonsa w modelowaniu wielokatnych obszarow w trojwymiarowych zagadnieniach brzegowych definiowanych rownaniem Laplace'a. Archiwum Informatyki Teoretycznej i Stosowanej, Instytut Informatyki Teoretycznej i Stosowanej Polskiej Akademii Nauk 17, 127–142 (2005) (in Polish)
12. Zieniuk, E., Szerszen, K.: Triangular Bezier patches in modelling smooth boundary surface in exterior Helmholtz problems solved by PIES. Archives of Acoustics 1(34), 1–11 (2009)
13. Zieniuk, E., Szerszen, K.: Trojkatne platy powierzchniowe w modelowaniu gladkiej powierzchni brzegu w PURC dla zagadnien ustalonego przepywu cieczy doskonalej. Modelowanie Inynierskie 37, 289–296 (2009) (in Polish)

Cache Blocking for Linear Algebra Algorithms

Fred G. Gustavson[1,2]

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

Abstract. We briefly describe Cache Blocking for Dense Linear Algebra Algorithms on computer architectures since about 1985. Before that one had uniform memory architectures. The Cray I machine was the last holdout. We cover the where, when, what, how and why of Cache Blocking. Almost all computer manufacturers have recently (about seven years ago) dramatically changed their computer architectures to produce Multicore (MC) processors. It will be seen that the arrangement in memory of the submatrices A_{ij} of A is a critical factor for obtaining high performance. From a practical point of view, this work is very important as it will allow existing codes using LAPACK and ScaLAPACK to remain usable by new versions of LAPACK and ScaLAPACK.

1 Introduction

During my last 25 years at IBM research I devoted a lot of time to library development; this library is called ESSL standing for the IBM Engineering Scientific Subroutine Library and PESSL for parallel ESSL. In 2011, ESSL celebrated its 25th birthday. A large part of ESSL and PESSL is devoted to Dense Linear Algebra, DLA. Myself and others at IBM chose to make ESSL compatible with LAPACK [6] and ScaLAPACK [7] libraries. DLA researchers, myself included, have contributed heavily to the development and understanding of cache blocking. This paper focuses on cache blocking as it relates to DLA.

The main questions for any broad area are where, when, what, how and why. For DLA the where is everywhere as the data for DLA are matrices. These matrices should be laid out in memory properly as almost all processors use a design that incorporates cache blocking; i.e., their memory hierarchies are designed in a tiered fashion called caches. The processing of matrix data only occurs in the lowest level caches; today these data processing areas are called cores.

In the mid 1980's the when occurred; this was when caches appeared for the first time. Cache blocking was "invented first" by my group at IBM in 1984 [23] and by the Cedar project at the University of Illinois [13]. We are not claiming that DLA researchers did not previously have and use the concept of partitioned matrices; they certainly did. We are claiming that we recognized the partitioning fact and proposed that submatrices be reformatted, if necessary, when they are moved back and forth from memory to caches to take advantage of the principles underlying cache blocking. Long ago, before the mid 1980's, processor design could afford a uniform or single memory design and thus all data could be ready for processing in a single CPU operation. A prime example was the Cray I vector machines. However, later on, the Cray II machines found it necessary to become cache based machines. That is when DLA researchers first introduced the Level-3 BLAS [11] for improving new DLA libraries that were later to be produced. The LAPACK and ScaLAPACK libraries were two examples. At IBM research, my group was fortunate to get a head start into cache blocking research as IBM had earlier just introduced cache based machines. "Moore's law" had just started to apply; this law accurately predicted processor cycle time decreases (machine speed increases) for the next twenty years in a compound manner. Those cycle time decreases ultimately had to end as the law of physics that governed the speed of light and energy started to adversely affect all processor design. This time was around 2005. To increase processing power further chip manufactures introduced **M**ulti-**C**ore, MC, chips as a way to continue "Moore's law".

An application of the Algorithms and Architecture Approach [17] describes the how or what part. From the algorithms side we use a "fundamental principle" of Linear Algebra called the "Principle of Linear Superposition". We use it to describe the factorization algorithms of DLA of a matrix A in terms of its sub matrices A_{ij} instead of its elements a_{ij}. These submatrices were to be laid out optimally on a given platform to ensure automatic cache blocking! The LAPACK and ScaLAPACK libraries are also based on this fundamental principle. However, both of these libraries use standard data layout for matrices and the Level-3 BLAS to gain their performance on all platforms. The decision worked until the introduction of MC.

Last we describe the why of Cache Blocking. The "why" deals with the speed of data processing. Peak performance for DLA factorization only occurs if all matrix operands are used multiple times when they enter an L1 cache or core. This ensures that the initial cost of bringing an operand into cache is then amortized by the ratio of $O(n^3)$ arithmetic to $O(n^2)$ elements or `nb`[1] flops per matrix element a_{ij}. Multiple reuse of all operands *only* occurs if all matrix operands map well into the L1 caches. For MC processors, an "L1 cache" is the data area of a core. For MC it is critical to get submatrices to the cores as fast as possible. The standard programming interface, called API, that hold matrices A that the BLAS and DLA libraries use is the 2-D array of the Fortran and C programming languages. For this API, submatrices A_{ij} of A are *not* stored contiguously. Thus it is *impossible* to move A_{ij} to and from the memory hierarchy from and to the various cores in a fast or optimal manner! This problem is corrected by using New Data Structures, acronym NDS, to hold these submatrices A_{ij}. By using dimension theory [29] we shall "prove" why this is true.

Multicore/Manycore (MC) has been called a revolution in Computing. However, MC is only a radical change in Architectures. Several times in the past we have talked about a "fundamental triangle" that relates Algorithms, Architectures and Compilers [3,17,12]. The fundamental triangle concept simply says

[1] `nb` is the order of a square submatrix A_{ij} of A that enters a core.

that these three research areas are inter-related and it means that Compilers and Algorithms should change, if possible, in significant ways when there is a significant change in computer architectures. Over the last seven years the codes of LAPACK library have been carefully examined. The LAPACK team has made basic structural changes to several of their basic codes to gain better performance on MC including a major change to adopt NDS. Other research teams and vendors have done the same thing and for the most part they are adopting NDS. Time and space does not allow us to cite this large research activity. Their combined findings indicate that "cache blocking" is still very important.

For MC the disproportion between multiple CPU processing and memory speed has become much higher. On a negative side, the API for BLAS-3 hurts performance; it requires repeated matrix data reformatting from its API to NDS. A new "BLAS-3" concept has emerged; it is to use NDS in concert with "BLAS-3" kernels [17,27,26,9]. For MC, the broad idea of "cache blocking" is mandatory as matrix elements must be fed to SPE's or GPU's as fast as possible. Also important is the arrangement in memory of the submatrices A_{ij} of A that are to be processed. This then defines "cache blocking" on MC processors for DLA.

We only use two matrix layouts in this paper. First, we assume that the matrices are stored in **R**ectangular **B**lock (RB) format. RB format stores a M by N matrix A as contiguous rectangular submatrices A_{ij} of size MB by NB. Square Block (SB) format is a special case of RB format when the rectangle is a square. It was first introduced in 1997, see [15], and has been described in five recent mini-symposiums at PARA06, PPAM07, PARA08, PPAM09, and PARA10. It turns out that our results on NDS [15,17,12,5,19] are very relevant to MC: of all 2-D data layouts for common matrix operations that treat rows and columns equally, SB format minimizes L1 and L2 cache misses as well as TLB misses[28]. The essential reason is that a SB of order NB is also a contiguous 1-D array of size $\mathtt{NB}^2$ and for most cache designs a contiguous array whose size fits into the cache is mapped from its place in memory into the cache by the *identity* mapping.

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

Section 2 gives a discussion of Dimension Theory. It shows why Fortran and C arrays *cannot* be truly multi-dimensional. In Section 3, we describe the features of In-Place Transformations between standard full layouts of matrices and the

RB or square block SB formats of NDS. These algorithms demonstrate a novel form of "cache blocking" in that memory is reorganized to be very efficient for DLA algorithms. We briefly describe the "random nature" of the memory layout of the standard API. Some early history of my involvement with Cache Blocking is given in Section 4. A short Summary and Conclusion is given in Section 5.

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

Fortran and C use 1-D layouts for their multi-dimensional arrays. Most library software for DLA use the Fortran and C API for their matrices which are clearly 2-D. The Fundamental Theorem of Dimension Theory states that it is *impossible* to preserve closeness of all points p in a neighborhood $\mathcal{N}$ of a D dimensional object when one uses a $d < D$ dimensional coordinate system to describe the object; see pages 106 to 120 of [29]. We are concerned with a submatrix A_{ij} representing $\mathcal{N}$ of a matrix A representing a 2-D object. This result says that it is impossible to lay out a matrix in 1-D fashion and maintain closeness of all of the elements of A_{ij}. This result warns us about moving A_{ij} to and from cache which represent $\mathcal{N}$. Computer scientists use the phrase "preserve data locality" when data is mapped from memory into a cache. We also note that when data is contiguous in computer memory then its mapping into cache is the identity mapping. Clearly, this is the fastest way to move data and also to preserve its data locality in cache.

2.1 Submatrices A_{ij} of A in Fortran and C

Let A have m rows and n columns with $\mathtt{LDA} \geq m$. In Fortran the columns of A are stored stride one and the row elements are stored LDA elements apart. This is a 1-D layout whereas A is conceptually 2-D. A^T has n rows and m columns with $\mathtt{LDAT} \geq n$. Its rows are stored stride one and its columns are laid out LDAT elements apart. Again, this is a 1-D layout whereas A^T is conceptually 2-D. Actually A and A^T are the same object; this is how we "view" A or A^T or conceive of them. Clearly both A and A^T contain the same information. Now, everything we just said about A and A^T applies equally well to every submatrix A_{ij} of A and its transpose A^T_{ij}. However, copies of submatrices are usually made during the processing of DLA algorithms when A is in standard layout. Here is the reason why: one *cannot* transpose a submatrix $A_{r:s,u:v}$ in-place as its image $A^T_{r:s,u:v}$ does not map onto $A_{r:s,u:v}$. We use colon notation [14]. This is why the transposition of submatrices in Fortran and C has to be an out-of-place operation. However, one can transpose any submatrix of A in-place if it is stored contiguously. NDS possess this property.

2.2 Generalization of Standard Format to RB format

In RB format each scalar $a_{i,j}$ element of standard format becomes a rectangular or square submatrix $A(\mathtt{I}:\mathtt{I}+\mathtt{MB}-1,\mathtt{J}:\mathtt{J}+\mathtt{NB}-1)$ of size MB rows and NB columns. All submatrices are contiguous, meaning LDA = MB. Simple and non-simple layouts of $A(\mathtt{I}:\mathtt{I}+\mathtt{MB}-1,\mathtt{J}:\mathtt{J}+\mathtt{NB}-1)$ are used; see Section 2.1 of [18] and [19] for the meaning of non-simple format. Today, these non-simple formats are called rb formats standing for **r**egister **b**lock formats. The last block rows and columns of RB A are called left-over blocks. These A_{IJ} blocks reside in MB*NB locations of array storage even though they require less storage to store. It is very important to pad these left-over blocks; otherwise the theory behind in-place fast data movement of RB A breaks down. With these changes one can transpose or transform RB A submatrices both in-place and out-of-place.

2.3 Changes to RB Format Have Happened

The advent of Multi-core (MC) in 2005 is forcing this change in DLA library codes. The new codes are using “2-D” layouts which are stored in conventional “4-D” Fortran and C arrays. Other names given to these layouts are tiling [27,26], hierarchical tiling, super matrix [10] or block data layout [28] (There are too many papers to cite here). The codes that access these “2-D” layouts have been proven to allow better scaling; this better scaling is called strong scaling. Some of the benefits that these NDS give are:

– BLAS Coding Changes. There is no data copying. The new codes are now kernel BLAS which are tailored to a MC architecture or a GPU.
– Overlapping communication and computation can be fully exploited.
– The principle of “Lookahead” is used to expose parallelism to a very great extent.
– A programming price is being paid by algorithm designers by “forcing” them to innovate.

2.4 Tutorial on the Essence of Dimension

We now return to our description of Dimension Theory. This is a deep mathematical subject and we think it sheds light on the subject of cache blocking. Before going further let us study 2-D domains; e.g. a matrix or a City Map. The concept to emerge is “closeness of points in a neighborhood” of an arbitrary domain point p. How does one describe an arbitrary neighborhood?, e.g. of a city map? Users of maps need to be able to identify their location on a city map (this is their point p). A rectangle of squares labeled like a Chess board is commonly used to label city maps. Why is this so? Well, for example, tourists living in a hotel in Torun, Poland will locate the square that holds their hotel and then they can easily walk to neighboring squares. The key concept is closeness for all points in any neighborhood of a point. In the example, the point p denotes the hotel and the activity is to walk to nearby places which is the neighborhood $\mathcal{N}$. Here is the key question: when does a labeling of a domain satisfy the neighborhood property of closeness? This notion can be made mathematically precise and correct. So, we can define dimension in a satisfactory manner.

Before answering we back up and try other labelings of domain points. Let us try natural Numbers: $1, 2, 3, \ldots$, Fortran and C use this labeling to lay out matrices A; e.g., in Fortran scalar element a_{ij} is located in computer memory at word location `i + (j - 1)*LDA` past the beginning word of A. Notice that some neighboring a_{ij} elements are widely separated with this single labeling; e.g., in Fortran CM format row elements are widely separated in memory by `LDA` elements. The same thing occurs for city maps. Is this true for all single labelings? The answer is yes. Now we need to measure distance. We give a metric for a neighborhood that uses two coordinates. We use a one norm[2]: let $p = (u, v)$ and $q = (x, y)$ be two points. Then $\texttt{norm}(p, q) = |u - x| + |v - y|$.

We are now ready to give the mathematical essence of dimension. Indexing with single numbers, or simple enumeration is applicable only to those cases where the objects have the character of a sequence. Simple, single indexing must obey the neighborhood property. These objects are labeled to be one dimensional. Now consider two dimensions. Maps, matrices, etc. *cannot* be labeled by a simple sequential ordering. Here is the reason why: "the neighborhood property will be violated" (we have said this was so above). However, two simple sequences suffice. The use of the one norm shows us why visually. Now we generalize this idea and give the dimension number of any domain: dimension is the amount of numbers (symbols) to suitably characterize the elements of the domain; i.e., closeness of all points must hold when this number of coordinates is used to fully describe all the points in any neighborhood of the domain. It is the amount of the numbers (symbols) that give the dimension of the domain; e.g. a line is 1-D, a circle is 2-D and a solid sphere is 3-D. Now we discuss some prior history about dimension and its resolution. There was an "Erroneous Prior Notion" that a rectangle had more points than a line; and that a solid had more points than a rectangle! However, Cantor's theory of infinities asserted that "All domains have the same number of points"! Thus this "erroneous prior notion" needed to be corrected. However, a difficult problem remained: "is it possible to label a domain with two different labelings that both obey the neighborhood principle of a higher to lower labeling"? For cache blocking, the relevance of a yes answer to this question is important as countless papers have been written on "effective ways" to perform cache blocking. However, the Fundamental Theorem of Dimension Theory stated at the beginning of Section 2 says the answer is no! In 1913 L. E. J. Brouwer stated and proved this theorem which we now phrase in a slightly different manner. "It is *not* possible to label a domain with two different labelings that both obey the neighborhood principle".

3 Converting Standard Format to RB Format In-Place via Vector Transposition

In [20] we demonstrated fast in-place transposition for matrices stored in CM format. In terms of speed they improved the existing state-of-the-art slightly.

[2] The familiar Euclidean norm $\sqrt{(u - x)^2 + (v - y)^2}$ is not used as the one norm is more easily understood for the matrix and city map examples.

However, they were very slow compared to out-of-place transpose algorithms. Here is an explanation. Let $\mathtt{M} \times \mathtt{N}$ matrix A be laid out CM format. In Fortran we have `A(0:M-1,0:N-1)`. The element a_{ij} is stored at offset $k = i + j\mathtt{M}$ or $A[k]$. The algorithms of [20] implement an in-place permutation P of the entries of A so that $a_{ij} = A[k]$ ends up at offset $\bar{k} = i\mathtt{N}+j$ or $A[\bar{k}]$. Note that $P(k)$ also equals $k\mathtt{N} \bmod q$ where $q = \mathtt{MN} - 1$. Thus $P(k)$ is governed by modular arithmetic. Now modular arithmetic with parameters N and q also define different pseudo-random number generators; see [25, Section 3.2.1.3]. It follows that in-place transpose algorithms in Fortran and C must exhibit a random memory access pattern and thus have very poor performance: each memory access will likely miss in each level of the cache hierarchy. Cache miss penalties are huge (in the hundreds of cycles) for MC processors.

Today's processors lay out memories in chunks of size LS called lines and when an element is accessed the entire line containing the element is brought into the L1 cache. To obtain high performance it is therefore imperative to utilize or process all elements in a line when the line enters the L1 and L0 caches[3]. Section 2.1 gave the reason why it was *impossible* to transpose a standard format submatrix A_{ij} of A in-place. However, we just showed here that A can be transposed in-place. However, this algorithm was *very slow.* Instead, out-of-place transposition algorithms are almost universally used.

3.1 Dense Linear Algebra Algorithms for MC Use RB or SB Format

We need a fast way to transform A, in a standard CM or RM format, to be in RB format [21,24,22]. The idea is to move contiguous lines of data; see Section 3. Fortunately, standard CM Fortran format consists of M columns stored as contiguous lines. This can be done by using a vector version of the point or scalar algorithms of [20]. The VIPX algorithm of [21] maps in-place a column swath of A to become m RB's. A column swath is an $M = m\mathtt{MB}$ by NB submatrix of A, in standard CM format, where $\mathtt{LDA} = m\mathtt{MB}$. We assume A is laid out in Fortran array as `A(0:M-1,0:N-1)`. Under the vector mapping it will become m size MB by NB RB's. Repeating algorithm VIPX n times on the n concatenated column swaths that make up CM A converts CM A to become A in RB format.

Now we can describe the VIPX algorithm. It assumes CM A has a certain layout space in terms of standard 2-D layout terminology. CM A and RB format A will occupy $M \le m\mathtt{MB}$ by $N \le n\mathtt{NB}$ arrays with $\mathtt{LDA} = m\mathtt{MB}$ where $m = \lceil M/\mathtt{MB} \rceil$; see section 2.2 where this layout was mentioned as crucial. Algorithm VIPX is embarrassingly parallel: it is applied n times on the n column swaths of A to produce RB A. The VIPX algorithm will be *very efficient* compared algorithm MIPT or IPT of [20] applied directly to A stored in standard CM or RM format; see the start of Section 3 where the reason why is given. In [22] we have improved in the VIPX algorithm by discovering the exact nature of the vector P mapping a priori using an algorithm based on number theory. This number theory is conjectured to be related to the dimension theory explanations of Section 2.

[3] The L0 cache is the register file of a core.

3.2 The VIPX Vector Transpose Algorithm

We overview how one gets from standard CM format to RB format. Let A be a CM Fortran array having M $=m$MB rows and N $=n$NB columns with its LDA $=$ M. Thus, A consists of n column swaths that are concatenated together. Denote any such swath as a subarray B of A and note that B consists of NB contiguous columns of CM array A. So, array B has M rows and NB columns. Think of array B as holding an m by NB matrix C whose elements are column vectors of length MB. Now apply algorithm MIPT or IPT of [20] to this m by NB matrix C of vectors of length MB. Now C has been replaced (over-written) by C^T which is a size NB by m matrix of vectors of length MB. To see this, we give a small example. Let $m=4$, MB=2 and NB=3. Then C is a scalar matrix of size 8 by 3 and at the same time is a 4 by 3 matrix of vectors of length 2 residing in array B. Originally, the 12 vectors of C are stored in CM format in array B. After in-place transposition the 12 vectors of C are stored in RM format in array B. This means the original 12 vectors now occupy array B in the permuted order $0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11$. This permuted order of the 12 vectors is identical to $m=4$ RB of size MB by NB concatenated together. For array A holding matrix A we do n parallel $\mathtt{B} \rightarrow \mathtt{B}^T$ operations for each of the n concatenated subarrays B that make up the array A. After completion of these n parallel computation steps we have transformed CM matrix A in array A in-place to become the same matrix A but now A is represented in RB format in array A. Now A consists of m block rows by n block columns where each size MB by NB contiguous block matrix is stored in standard CM block order. Thus, after vector in-place transpose we have "cache blocked" matrix A! The VIPX algorithm is algorithm MIPT or IPT of [20] modified to move contiguous vectors of length MB instead of scalars of length one. A more efficient VIPX parallel algorithm, based in number theory, is given in [22].

3.3 Interpretation of Vector Transposition as a Form of Cache Blocking for MC

We hope the reader now clearly sees at a deeper level why NDS significantly improves MC DLA algorithm performance. The transformation of A in standard format to RB format by in-place *vector* transposition was orders of magnitude faster than ordinary scalar in-place methods. Finally, we conclude this short section by restating that NDS are a form of cache blocking matrices A that are originally stored in standard format for MC.

4 Some Early IBM History on Cache Blocking

I became manager of a small research group and project called Algorithms and Architectures in the early 1980's. IBM decided to introduce a Vector Processor into its new cache based 3080 series of mainframe lines in order to gain more scientific market share. My group initially consisted of researchers Ramesh Argawal, James Cooley and Bryant Tuckerman. Our first project was to supply elementary functions for new compilers that were to support the IBM programming languages that would accompany the new mainframes. We produced novel scalar and vector elementary functions that were nearly perfectly rounded and very fast. This work became state-of-the-art [1]; today this design remains state-of-the-art. Our next project involved research and development. It had to do with the formation of the ESSL IBM product. This effort became a joint venture with IBM Development in Poughkeepsie and Kingston, NY headed by Stanley Schmidt and Joan McComb. ESSL was conceived during 1982. For linear algebra, our team decided to make ESSL subroutines compatible with Linpack and later with the BLAS and LAPACK. In May to June of 1984 my group produced successful designs of matrix multiply, _GEFA and _POFA[4]. Our internal report said "a conceptual design has been identified in which data is brought into cache (and completely used) only once. This approach allows full use of the multiply add instruction". This is when "cache blocking" was born at IBM. The ESSL Programming Product was formally announced and released for the first time in February 1986 [23]. In 1987 ESSL release II added a practical high performing Strassen matrix multiply algorithm. Before that Strassen's algorithm only possessed academic or theoretical value. In 1988, my group showed how "algorithmic lookahead" could be used to obtain perfect parallel speed-up for the Linpack benchmark [2]. This achievement was part of a very early TOP 500 result. This key idea is heavily used today to get high performance on MC processors. In the late 1980's ESSL and my group was presented a new challenge. IBM decided to introduce RISC computers called the POWER (RS6000) line of workstations. ESSL had grown substantially and had put out four mainframe releases. A huge programming effort began and 497,000 lines of Fortran code was produced by my small group of four regular people assisted by a small group of talented young programmers. We called our effort EFL standing for ESSL Fortran Library; the whole library was written in Fortran! After POWER1, there came a remarkable machine called POWER2 [3]. It possessed very high bandwidth. I wrote an internal memo that POWER2 could have been a GFlop machine; POWER2 had a peak MFlop rate of 268 MFlops. Our ESSL Level-2 BLAS ran at nearly 200 MFlops with matrix data in memory. In 1992 my group published a report [4] on how to use overlapped communication to produce peak performing matrix multiplication on distributed memory computers. Today, this algorithm remains the algorithm of choice for MC. Sometime later Jim Demmel and his graduate students at UC Berkeley started a project with a grant from IBM to try to automatically produce DGEMM code which would obtain performance equal to or better than EFL DGEMM code. They produced PHIPAC [8]; later Jack Dongarra's group followed suit and produced ATLAS [30]. Automatic tuning of kernel routines by a computer is now (its new name is Autotuning) a main stay research tool of MC researchers. A lot happened to ESSL during the next eighteen years or so. Space does not allow the reporting of these accomplishments.

[4] At that time the Level-3 BLAS and LAPACK had not been conceived.

5 Conclusions and Summary

We indicated [16,17] that DLAFA (**F**actorization **A**lgorithms) are mainly MM (**M**atrix **M**ultiply) algorithms. The standard API for matrices and Level-3 BLAS use arrays; see page 739 of [15]. All standard array layouts are *one* dimensional. It is *impossible* to maintain locality of reference in a matrix or any higher than 1-D object using a 1-D layout; see [29]. MM requires row and column operations and thus requires MT (**M**atrix **T**ransformation) to NDS. Our results on in-place MT show that performance suffers greatly when one uses a 1-D layout. Using NDS for matrices "approximates" a 2-D layout; thus, one can dramatically improve in-place MT performance as well as DLAFA performance. Our message is that DLAFA are mostly MM. MM requires MT and both require NDS. Thus, DLAFA can and do perform well on multicore but only if one uses NDS.

References

1. Agarwal, R.C., Cooley, J.W., Gustavson, F.G., Shearer, J.B., Slishman, G., Tuckerman, B.: New scalar and vector elementary functions for the IBM System/370. IBM Journal of Research and Development 30(2), 126–144 (1986)
2. Agarwal, R.C., Gustavson, F.G.: A Parallel Implementation of Matrix Multiplication and LU factorization on the IBM 3090. In: Wright, M. (ed.) Proceedings of the IFIP WG 2.5 on Aspects of Computation on Asynchronous Parallel Processors, Stanford CA, pp. 217–221. North Holland (August 1988)
3. Agarwal, R.C., Gustavson, F.G., Zubair, M.: Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. IBM Journal of Research and Development 38(5), 563–576 (1994)
4. Agarwal, R.C., Gustavson, F.G., Zubair, M.: A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. IBM J. R. & D. 38(6), 673–681 (1994); See also IBM RC 18694 with dates 8/5/92 & 8/10/92 & 2/8/93
5. Andersen, B.S., Gunnels, J.A., Gustavson, F.G., Reid, J.K., Waśniewski, J.: A Fully Portable High Performance Minimal Storage Hybrid Cholesky Algorithm. ACM TOMS 31(2), 201–227 (2005)
6. Anderson, E., et al.: LAPACK Users' Guide Release 3.0. SIAM, Philadelphia (1999)
7. Blackford, L.S., et al.: ScaLAPACK Users' Guide. SIAM, Philadelphia (1997)
8. Bilmes, J., Asanovic, K., Whye Chin, C., Demmel, J.: Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In: Proceedings of International Conference on Supercomputing, Vienna, Austria (1997)
9. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algorithms for MC architectures. Parallel Comput. 35(1), 38–53 (2009)
10. Chan, E., Quintana-orti, E.S., Quintana-orti, G., Van De Geijn, R.: Super-Matrix Out-of-Core Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In: SPAA 2007, June 9-11, pp. 116–125 (2007)
11. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: A Set of Level 3 Basic Linear Algebra Subprograms. TOMS 16(1), 1–17 (1990)
12. Elmroth, E., Gustavson, F.G., Jonsson, I., Kågström, B.: Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. SIAM Review 46(1), 3–45 (2004)

13. Gallivan, K., Jalby, W., Meier, U., Sameh, A.: The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design. International Journal of Supercomputer Applications 2(1), 12–48 (1988)
14. Golub, G., VanLoan, C.: Matrix Computations, 3rd edn. John Hopkins Press, Baltimore and London (1996)
15. Gustavson, F.G.: Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. IBM J. R. & D. 41(6), 737–755 (1997)
16. Gustavson, F.G.: New Generalized Data Structures for Matrices Lead to a Variety of High-Performance Algorithms. In: Boisvert, R.F., Tang, P.T.P. (eds.) Proceedings of the IFIP WG 2.5 Working Group on The Architecture of Scientific Software, Ottawa, Canada, October 2-4, pp. 211–234. Kluwer Academic Publishers (2000)
17. Gustavson, F.G.: High Performance Linear Algebra Algs. using New Generalized Data Structures for Matrices. IBM J. R. & D. 47(1), 31–55 (2003)
18. Gustavson, F.G.: New Generalized Data Structures for Matrices Lead to a Variety of High Performance Dense Linear Algebra Algorithms. In: Dongarra, J., Madsen, K., Waśniewski, J. (eds.) PARA 2004. LNCS, vol. 3732, pp. 11–20. Springer, Heidelberg (2006)
19. Gustavson, F.G., Gunnels, J., Sexton, J.: Minimal Data Copy For Dense Linear Algebra Factorization. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 540–549. Springer, Heidelberg (2007)
20. Gustavson, F.G., Swirszcz, T.: In-Place Transposition of Rectangular Matrices. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 560–569. Springer, Heidelberg (2007)
21. Gustavson, F.G.: The Relevance of New Data Structure Approaches for Dense Linear Algebra in the New Multicore/Manycore Environments. IBM Research report RC24599, also, to appear in PARA 2008 Proceeding, 10 pages (2008)
22. Gustavson, F.G., Karlsson, L., Kågström, B.: Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion. ACM TOMS, 34 pages (to appear, 2012)
23. IBM. IBM Engineering and Scientific Subroutine Library. IBM Pub. No. SA22-7272-00 (February 1986); Also, Release II, 1987 & AIX Version 3, Release 3
24. Karlsson, L.: Blocked in-place transposition with application to storage format conversion. Tech. Rep. UMINF 09.01. ISSN 0348-0542, Department of Computing Science, Umeå University, Umeå, Sweden (January 2009)
25. Knuth, D.: The Art of Computer Programming, 3rd edn., vol. 1, 2 & 3. Addison-Wesley (1998)
26. Kurzak, J., Buttari, A., Dongarra, J.: Solving systems of Linear Equations on the Cell Processor using Cholesky Factorization. IEEE Trans. Parallel Distrib. Syst. 19(9), 1175–1186 (2008)
27. Kurzak, J., Dongarra, J.: Implementation of mixed precision in solving mixed precision of linear equations on the Cell processor: Research Articles. Concurr. Comput.: Pract. Exper. 19(10), 1371–1385 (2007)
28. Park, N., Hong, B., Prasanna, V.: Tiling, Block Data Layout, and Memory Hierarchy Performance. IEEE Trans. Parallel and Distributed Systems 14(7), 640–654 (2003)
29. Tietze, H.: Three Dimensions–Higher Dimensions. In: Famous Problems of Mathematics, pp. 106–120. Graylock Press (1965)
30. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimization of Software and the ATLAS Project. Parallel Computing (1-2), 3–35 (2001)

Reducing the Amount of Pivoting in Symmetric Indefinite Systems

Dulceneia Becker[1], Marc Baboulin[4], and Jack Dongarra[1,2,3]

[1] University of Tennessee, USA
{dbecker7,dongarra}@eecs.utk.edu
[2] Oak Ridge National Laboratory, USA
[3] University of Manchester, United Kingdom
[4] INRIA / Université Paris-Sud, France
marc.baboulin@inria.fr

Abstract. This paper illustrates how the communication due to pivoting in the solution of symmetric indefinite linear systems can be reduced by considering innovative approaches that are different from pivoting strategies implemented in current linear algebra libraries. First a tiled algorithm where pivoting is performed within a tile is described and then an alternative to pivoting is proposed. The latter considers a symmetric randomization of the original matrix using the so-called recursive butterfly matrices. In numerical experiments, the accuracy of tile-wise pivoting and of the randomization approach is compared with the accuracy of the Bunch-Kaufman algorithm.

Keywords: dense linear algebra, symmetric indefinite systems, LDL$^{\mathrm{T}}$ factorization, pivoting, tiled algorithms, randomization.

1 Introduction

A symmetric matrix A is called indefinite when the quadratic form x^TAx can take on both positive and negative values. By extension, a linear system $Ax = b$ is called symmetric indefinite when A is symmetric indefinite. These types of linear systems are commonly encountered in optimization problems coming from physics of structures, acoustics, and electromagnetism, among others. Symmetric indefinite systems also result from linear least squares problems when they are solved via the augmented system method [7, p. 77].

To ensure stability in solving such linear systems, the classical method used is called the diagonal pivoting method [9] where a block-LDL$^{\mathrm{T}}$ factorization[1] is obtained such as

$$PAP^T = LDL^T \tag{1}$$

where P is a permutation matrix, A is a symmetric square matrix, L is unit lower triangular and D is block-diagonal, with blocks of size 1×1 or 2×2;

[1] Another factorization method is for example the Aasen's method [13, p.163]: $PAP^T = LTL^T$ where L is unit lower triangular and T is tridiagonal.

all matrices are of size $n \times n$. If no pivoting is applied, *i.e.* $P = I$, D becomes diagonal. The solution x can be computed by successively solving the triangular or block-diagonal systems $Lz = Pb$, $Dw = z$, $L^T y = w$, and ultimately we have $x = P^T y$.

There are several pivoting techniques that can be applied to determine P. These methods involve different numbers of comparisons to find the pivot and have various stability properties. As for the LU factorization, the *complete pivoting* method (also called *Bunch-Parlett* algorithm [9]) is the most stable pivoting strategy. It guarantees a satisfying growth factor bound [14, p. 216] but also requires up to $\mathcal{O}(n^3)$ comparisons. The well-known *partial pivoting* method, based on the *Bunch-Kaufman* algorithm [8], is implemented in LAPACK [1] and requires at each step of the factorization the exploration of two columns, resulting in a total of $\mathcal{O}(n^2)$ comparisons. This algorithm has good stability properties [14, p. 219] but in certain cases $\|L\|$ may be unbounded, which is a cause for possible instability [3], leading to a modified algorithm referred to as *rook pivoting* or *bounded Bunch-Kaufman pivoting*. The latter involves between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ comparisons depending on the number of 2×2 pivots. Another pivoting strategy, called *Fast Bunch-Parlett* strategy (see [3, p. 525] for a description of the algorithm), searches for a local maximum in the current lower triangular part. It is as stable as the rook pivoting but it also requires between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ comparisons.

With the advent of architectures such as multicore processors [19] and Graphics Processing Units (GPU), the growing gap between communication and computation efficiency made the communication overhead due to pivoting more critical. These new architectures prompted the need for developing algorithms that lend themselves to parallel execution. A class of such algorithms for shared memory architectures, called *Tiled Algorithms*, has been developed for one-sided dense factorizations[2] [10,11] and made available as part of the PLASMA library [12].

Tiled algorithms are based on decomposing the computation into small tasks in order to overcome the sequential nature of the algorithms implemented in LAPACK. These tasks can be executed out of order, as long as dependencies are observed, rendering parallelism. Furthermore, tiled algorithms make use of a tile data-layout where data is stored in contiguous blocks, which differs from the column-wise layout used by LAPACK, for instance. The tile data-layout allows the computation to be performed on small blocks of data that fit into cache, and hence exploits cache locality and re-use. However, it does not lend itself straightforwardly for pivoting, as this requires a search for pivots and permutations over full columns/rows. For symmetric matrices, the difficulties are even greater since symmetric pivoting requires interchange of both rows and columns. The search for pivots outside a given tile curtails memory locality and increases data dependence between tiles (or tasks). The former has a direct impact on the performance of serial kernels and the latter on parallel performance [18].

In this paper, the possibility of eliminating the overhead due to pivoting by considering randomization techniques is investigated. These techniques were

[2] $\mathrm{LDL^T}$ is still under development and shall be available in the future [6].

initially proposed in [16] and modified approaches were studied in [4,5] for the LU factorization. In this context, they are applied to the case of symmetric indefinite systems. According to this random transformation, the original matrix A is transformed into a matrix that would be sufficiently "random" so that, with a probability close to 1, pivoting is not needed. This transformation is a multiplicative preconditioning by means of random matrices called *recursive butterfly matrices.* The LDL$^{\mathrm{T}}$ factorization without pivoting is then applied to the preconditioned matrix. One observes that two levels of recursion for butterfly matrices are enough to obtain an accuracy close to that of LDL$^{\mathrm{T}}$ with either partial (Bunch-Kaufman) or rook pivoting on a collection of matrices. The overhead is reduced to $\sim 8n^2$ operations, which is negligible when compared to the cost of pivoting.

2 Tile-Wise Pivoting

Given Equation (1), the tiled algorithm starts by decomposing A in $nt \times nt$ tiles[3] (blocks), where each A_{ij} is a tile of size $nb \times nb$. The same decomposition can be applied to L and D. For instance, for $nt = 3$:

$$\begin{bmatrix} A_{11} & A_{21}^T & A_{31}^T \\ A_{21} & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} D_{11} & & \\ & D_{22} & \\ & & D_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{bmatrix} \tag{2}$$

Upon this decomposition and using the same principle as the Schur complement, a series of tasks can be set to calculate each L_{ij} and D_{ii}:

$$[L_{11}, D_{11}] = \mathrm{LDL}(A_{11}) \tag{3}$$
$$L_{21} = A_{21}^T (D_{11} L_{11}^T)^{-1} \tag{4}$$
$$L_{31} = A_{31}^T (D_{11} L_{11}^T)^{-1} \tag{5}$$
$$\tilde{A}_{22} = A_{22} - L_{21} D_{11} L_{21}^T \tag{6}$$
$$\tilde{A}_{32} = A_{32} - L_{31} D_{11} L_{21}^T \tag{7}$$
$$[L_{22}, D_{22}] = \mathrm{LDL}(\tilde{A}_{22}) \tag{8}$$
$$L_{32} = \tilde{A}_{32} (D_{22} L_{22}^T)^{-1} \tag{9}$$
$$\tilde{A}_{33} = A_{33} - L_{31} D_{11} L_{31}^T - L_{32} D_{22} L_{32}^T \tag{10}$$
$$[L_{33}, D_{33}] = \mathrm{LDL}(\tilde{A}_{33}) \tag{11}$$

LDL(X_{kk}) at Equations (3), (8) and (11) means the actual LDL$^{\mathrm{T}}$ factorization of tile X_{kk}. These tasks can be executed out of order, as long as dependencies are observed, rendering parallelism (see [6] for more details).

Following the same approach, for $PAP^T = LDL^T$, and given

$$P = \begin{bmatrix} P_{11} & & \\ & \ddots & \\ & & P_{nt,nt} \end{bmatrix} \tag{12}$$

[3] For rectangular matrices, A is decomposed into $mt \times nt$ tiles.

the tasks for $nt = 3$ may be described as:

$$[L_{11}, D_{11}, P_{11}] = \mathrm{LDL}(A_{11}) \tag{13}$$
$$L_{21} = P_{22}A_{21}P_{11}^T(D_{11}L_{11}^T)^{-1} \tag{14}$$
$$L_{31} = P_{33}A_{31}P_{11}^T(D_{11}L_{11}^T)^{-1} \tag{15}$$
$$\tilde{A}_{22} = A_{22} - (P_{22}^T L_{21})D_{11}(P_{22}^T L_{21})^T \tag{16}$$
$$\tilde{A}_{32} = A_{32} - (P_{33}^T L_{31})D_{11}(P_{33}^T L_{21})^T \tag{17}$$
$$[L_{22}, D_{22}, P_{22}] = \mathrm{LDL}(\tilde{A}_{22}) \tag{18}$$
$$L_{32} = P_{33}\tilde{A}_{32}P_{22}^T(D_{22}L_{22}^T)^{-1} \tag{19}$$
$$\tilde{A}_{33} = A_{33} - (P_{33}^T L_{31})D_{11}(P_{33}^T L_{31})^T - (P_{33}^T L_{32})D_{22}(P_{33}^T L_{32})^T \tag{20}$$
$$[L_{33}, D_{33}, P_{33}] = \mathrm{LDL}(\tilde{A}_{33}) \tag{21}$$

Equations (13) to (21) are similar to Equations (3) to (11), except that the permutation matrix P_{kk} has been added. This permutation matrix P_{kk} generates a circular dependence between equations, which is not an issue when pivoting is not used. For instance, in order to calculate

$$L_{21} = P_{22}A_{21}P_{11}^T\left(D_{11}L_{11}^T\right)^{-1} \tag{22}$$

P_{22} is required. However, to calculate

$$[L_{22}, D_{22}, P_{22}] = \mathrm{LDL}\left(A_{22} - (P_{22}^T L_{21})D_{11}(P_{22}^T L_{21})^T\right) \tag{23}$$

L_{21} is required. To overcome this circular dependence, instead of actually calculating L_{21}, $P_{22}^T L_{21}$ is calculated, since the equations can be rearranged such as $P_{22}^T L_{21}$ is always used and therefore L_{21} is not needed. Hence, Equations (14), (15) and (19) become, in a general form,

$$P_{ii}^T L_{ij} = A_{ij}P_{jj}^T\left(D_{jj}L_{jj}^T\right)^{-1} \tag{24}$$

After P_{ii} is known, L_{ij}, for $1 \geq j \geq i-1$, can be calculated such as

$$L_{ij} = P_{ii}L_{ij} \tag{25}$$

This procedure may be described as in Algorithm 1, where A is a symmetric matrix of size $n \times n$ split in $nt \times nt$ tiles A_{ij}, each of size $nb \times nb$.

The permutation matrices P_{kk} of Algorithm 1 are computed during the factorization of tile A_{kk}. If pivots were searched only inside tile A_{ii}, the factorization would depend only and exclusively on A_{kk}. However, for most pivoting techniques, pivots are searched throughout columns, which make the design of efficient parallel algorithm very difficult [18].

The tile-wise pivoting restricts the search of pivots to the tile A_{kk} when factorizing it, *i.e.* if LAPACK [1] routine `xSYTRF` was chosen to perform the factorization, it could be used as it is. In other words, the same procedure used to

Algorithm 1. Tiled LDL$^{\mathrm{T}}$ Factorization with Tile-wise Pivoting

```
1: for $k = 1$ to $nt$ do
2:     [ $L_{kk}$ , $D_{kk}$ , $P_{kk}$ ] = LDL( $A_{kk}$ )
3:     for $j = k + 1$ to $nt$ do
4:         $L_{jk} = A_{jk} P_{jj}^T (D_{kk} L_{kk}^T)^{-1}$
5:     end for
6:     for $i = k + 1$ to $nt$ do
7:         $A_{ii} = A_{ii} - L_{ik} D_{kk} L_{ik}^T$
8:         for $j = k + 1$ to $i - 1$ do
9:             $A_{ij} = A_{ij} - L_{ik} D_{kk} L_{jk}^T$
10:        end for
11:    end for
12:    for $i = 1$ to $j - 1$ do
13:        $L_{ki} = P_{kk} L_{ki}$
14:    end for
15: end for
```

factorize an entire matrix A is used to factorize the tile A_{kk}. This approach does not guarantee the accuracy of the solution; it strongly depends on the matrix to be factorized and how the pivots are distributed. However, it guarantees numerical stability of the factorization of each tile A_{kk}, as long as an appropriate pivoting technique is applied. For instance, LDL$^{\mathrm{T}}$ without pivoting fails as soon as a zero is found on the diagonal, while the tile-wise pivoted LDL$^{\mathrm{T}}$ does not, as shown in Section 4. Note that pivoting is applied as part of a sequential kernel, which means that the pivot search and hence the permutations are also serial.

3 An Alternative to Pivoting in Symmetric Systems

A randomization technique that allows pivoting to be avoided in the LDL$^{\mathrm{T}}$ factorization is described. This technique was initially proposed in [16] in the context of general linear systems where the randomization is referred to as Random Butterfly Transformation (RBT). It is shown in [16] that, with probability 1, Gaussian Elimination can operate without pivoting on the randomized matrix. Then a modified approach has been described in [5] for the LU factorization of general dense matrices and we propose here to adapt this technique specifically to symmetric indefinite systems. It consists of a multiplicative preconditioning $U^T AU$ where the matrix U is chosen among a particular class of random matrices called *recursive butterfly matrices*. Then LDL$^{\mathrm{T}}$ factorization without pivoting is performed on the symmetric matrix $U^T AU$ and, to solve $Ax = b$, $(U^T AU)y = U^T b$ is solved instead, followed by $x = Uy$. We study the random transformation with *recursive* butterfly matrices, and minimize the number of recursion steps required to get a satisfying accuracy. The resulting transformation will be called Symmetric Random Butterfly Transformation (SRBT). We define two types of matrices that will be used in the symmetric random transformation. These definitions are inspired from [16] in the particular case of real-valued matrices.

Definition 1. *A butterfly matrix is defined as any $n \times n$ matrix of the form:*

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{pmatrix}$$

where $n \geq 2$ and R_0 and R_1 are random diagonal and nonsingular $\frac{n}{2} \times \frac{n}{2}$ matrices.

Definition 2. *A recursive butterfly matrix of size n and depth d is a product of the form*

$$W^{<n,d>} = \begin{pmatrix} B_1^{<n/2^{d-1}>} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_{2^{d-1}}^{<n/2^{d-1}>} \end{pmatrix} \times \cdots \times \begin{pmatrix} B_1^{<n/2>} & 0 \\ 0 & B_2^{<n/2>} \end{pmatrix} \times B_1^{<n>}$$

where $B_i^{<n/2^{k-1}>}$ are butterfly matrices of size $n/2^{k-1}$ with $1 \leq k \leq d$.

Note that this definition requires n to be a multiple of 2^{d-1} which can always be obtained by "augmenting" the matrix A with additional 1's on the diagonal. Note also that Definition 2 differs from the definition of a recursive butterfly matrix given in [16], which corresponds to the special case where $d = \log_2 n + 1$, *i.e.* the first term of the product expressing $W^{<n,d>}$ is a diagonal matrix of size n. For instance, if $n = 4$ and $d = 2$, the recursive butterfly matrix $W^{<4,2>}$ is defined by

$$\begin{aligned} W^{<4,2>} &= \begin{pmatrix} B_1^{<2>} & 0 \\ 0 & B_2^{<2>} \end{pmatrix} \times B^{<4>} \\ &= \frac{1}{2} \begin{pmatrix} r_1^{<2>} & r_2^{<2>} & 0 & 0 \\ r_1^{<2>} & -r_2^{<2>} & 0 & 0 \\ 0 & 0 & r_3^{<2>} & r_4^{<2>} \\ 0 & 0 & r_3^{<2>} & -r_4^{<2>} \end{pmatrix} \begin{pmatrix} r_1^{<4>} & 0 & r_3^{<4>} & 0 \\ 0 & r_2^{<4>} & 0 & r_4^{<4>} \\ r_1^{<4>} & 0 & -r_3^{<4>} & 0 \\ 0 & r_2^{<4>} & 0 & -r_4^{<4>} \end{pmatrix} \\ &= \frac{1}{2} \begin{pmatrix} r_1^{<2>} r_1^{<4>} & r_2^{<2>} r_2^{<4>} & r_1^{<2>} r_3^{<4>} & r_2^{<2>} r_4^{<4>} \\ r_1^{<2>} r_1^{<4>} & -r_2^{<2>} r_2^{<4>} & r_1^{<2>} r_3^{<4>} & -r_2^{<2>} r_4^{<4>} \\ r_3^{<2>} r_1^{<4>} & r_4^{<2>} r_2^{<4>} & -r_3^{<2>} r_3^{<4>} & -r_4^{<2>} r_4^{<4>} \\ r_3^{<2>} r_1^{<4>} & -r_4^{<2>} r_2^{<4>} & -r_3^{<2>} r_3^{<4>} & r_4^{<2>} r_4^{<4>} \end{pmatrix}, \end{aligned}$$

where $r_i^{<j>}$ are real random entries.

The objective here is to minimize the computational cost of the RBT defined in [16] by considering a number of recursions d such that $d < log_2 n \ll n$, resulting in the transformation defined as follows.

Definition 3. *A symmetric random butterfly transformation (SRBT) of depth d of a square matrix A is the product:*

$$A_r = U^T A U$$

where U is a recursive butterfly matrix of depth d.

Remark 1. Let A be a square matrix of size n, the computational cost of a multiplication B^TAB with B butterfly of size n is $M(n) = 4n^2$. Then the number of operations involved in the computation of A_r by an SRBT of depth d is

$$C(n,d) = \sum_{k=1}^{d} \left((2^{k-1})^2 \times M(n/2^{k-1})\right) = \sum_{k=1}^{d} \left((2^{k-1})^2 \times 4(n/2^{k-1})^2\right)$$
$$= \sum_{k=1}^{d} \left(4n^2\right) = 4dn^2$$

Note that the maximum cost in the case of an RBT as described in [16] is

$$C(n, \log_2 n + 1) \simeq 4n^2 \log_2 n.$$

We can find in [16] details on how RBT might affect the growth factor, and in [5] we can find more information concerning the practical computation of A_r as well as a packed storage description and a condition number analysis. Note that, since we know that we do not pivot when using SRBT, the LDL$^\text{T}$ factorization without pivoting can be performed with a very efficient tiled algorithm [6].

4 Numerical Experiments

Experiments to measure the accuracy of each procedure described in the previous sections were carried out using Matlab version 7.12 (R2011a) on a machine with a precision of $2.22 \cdot 10^{-16}$. Table 1 presents accuracy comparisons of linear systems solved using the factors of A calculated by LDL$^\text{T}$ with: no pivoting (NP), partial pivoting (PP), tile-wise pivoting (TP), and the Symmetric Random Butterfly Transformation followed by no pivoting (SRBT NP) and tile-wise pivoting (SRBT TP). For tile-wise pivoting (TP), the matrices have 64 tiles (8×8). The partial pivoting corresponds to the Bunch-Kaufman algorithm as it is implemented in LAPACK. Note that for all experiments the rook pivoting achieves the same accuracy as the partial pivoting and therefore is not listed.

All matrices are of size 1024×1024, either belonging to the Matlab gallery or the Higham's Matrix Computation Toolbox [14] or generated using Matlab function `rand`. Matrices $|i-j|$, $max(i,j)$ and `Hadamard` are defined in the experiments performed in [16]. Matrices `rand1` and `rand2` correspond to random matrices with entries uniformly distributed in $[0,1]$ with all and $1/4$ of the diagonal elements set to 0, respectively. Matrices `rand0` and `rand3` are also random matrices, where the latter has its diagonal elements scaled by $1/1000$.

For all test matrices, we suppose that the exact solution is $x = [1\ 1 \ldots 1]^T$ and we set the right-hand side $b = Ax$. In Table 1, the 2-norm condition number of each matrix is listed. Note that we also computed the condition number of the randomized matrix which, similarly to [5], is of same order of magnitude as cond A and therefore is not listed. For each LDL$^\text{T}$ solver, the component-wise backward error is reported. The latter is defined in [15] and expressed as

$$\omega = \max_i \frac{|A\hat{x} - b|_i}{(|A| \cdot |\hat{x}| + |b|)_i},$$

Table 1. Component-wise backward error for LDL^T solvers on a set of test matrices of size 1024×1024 and 64 tiles (8×8) when applicable

Matrix	Cond A	NP	PP	TP	SRBT	
					NP (IR)	TP (IR)
`condex`	$1 \cdot 10^2$	$5 \cdot 10^{-15}$	$6 \cdot 10^{-15}$	$7 \cdot 10^{-15}$	$6 \cdot 10^{-15}$ (0)	$4 \cdot 10^{-15}$ (0)
`fiedler`	$7 \cdot 10^5$	Fail	$2 \cdot 10^{-15}$	$7 \cdot 10^{-15}$	$9 \cdot 10^{-15}$ (0)	$1 \cdot 10^{-15}$ (0)
`orthog`	$1 \cdot 10^0$	$8 \cdot 10^{-1}$	$1 \cdot 10^{-14}$	$5 \cdot 10^{-1}$	$3 \cdot 10^{-16}$ (1)	$4 \cdot 10^{-16}$ (1)
`randcorr`	$3 \cdot 10^3$	$4 \cdot 10^{-16}$	$3 \cdot 10^{-16}$	$4 \cdot 10^{-16}$	$5 \cdot 10^{-16}$ (0)	$3 \cdot 10^{-16}$ (0)
`augment`	$5 \cdot 10^4$	$7 \cdot 10^{-15}$	$4 \cdot 10^{-15}$	$8 \cdot 10^{-15}$	$2 \cdot 10^{-16}$ (1)	$5 \cdot 10^{-15}$ (0)
`prolate`	$6 \cdot 10^{18}$	$8 \cdot 10^{-15}$	$8 \cdot 10^{-16}$	$2 \cdot 10^{-15}$	$2 \cdot 10^{-15}$ (0)	$1 \cdot 10^{-15}$ (0)
`toeppd`	$1 \cdot 10^7$	$5 \cdot 10^{-16}$	$7 \cdot 10^{-16}$	$6 \cdot 10^{-16}$	$2 \cdot 10^{-16}$ (0)	$1 \cdot 10^{-16}$ (0)
`ris`	$4 \cdot 10^0$	Fail	$3 \cdot 10^{-15}$	$8 \cdot 10^{-1}$	$6 \cdot 10^{-1}$ (10)	$6 \cdot 10^{-1}$ (10)
$\lvert i-j \rvert$	$7 \cdot 10^5$	$2 \cdot 10^{-15}$	$2 \cdot 10^{-15}$	$7 \cdot 10^{-15}$	$1 \cdot 10^{-14}$ (0)	$1 \cdot 10^{-15}$ (0)
`max(i,j)`	$3 \cdot 10^6$	$2 \cdot 10^{-14}$	$2 \cdot 10^{-15}$	$5 \cdot 10^{-15}$	$1 \cdot 10^{-14}$ (0)	$1 \cdot 10^{-15}$ (0)
`Hadamard`	$1 \cdot 10^0$	0	0	0	$7 \cdot 10^{-15}$ (0)	$4 \cdot 10^{-15}$ (0)
`rand0`	$2 \cdot 10^5$	$1 \cdot 10^{-12}$	$7 \cdot 10^{-14}$	$1 \cdot 10^{-13}$	$1 \cdot 10^{-15}$ (1)	$1 \cdot 10^{-15}$ (0)
`rand1`	$2 \cdot 10^5$	Fail	$1 \cdot 10^{-13}$	$2 \cdot 10^{-11}$	$1 \cdot 10^{-15}$ (1)	$1 \cdot 10^{-15}$ (0)
`rand2`	$1 \cdot 10^5$	Fail	$5 \cdot 10^{-14}$	$6 \cdot 10^{-13}$	$1 \cdot 10^{-15}$ (1)	$2 \cdot 10^{-15}$ (0)
`rand3`	$8 \cdot 10^4$	$4 \cdot 10^{-13}$	$7 \cdot 10^{-14}$	$4 \cdot 10^{-13}$	$1 \cdot 10^{-15}$ (1)	$1 \cdot 10^{-15}$ (0)

NP: LDL^T with No Pivoting
PP: LDL^T with Partial Pivoting
TP: LDL^T with Tile-wise Pivoting
SRBT: Symmetric Random Butterfly Transformation followed by LDL^T without pivoting
IR: Iterative refinement number of iterations

where $\hat{x}$ is the computed solution.

Similarly to [16], the random diagonal matrices used to generate the butterfly matrices described in Definition 1 have diagonal values $exp(\frac{r}{10})$ where r is randomly chosen in $[-\frac{1}{2}, \frac{1}{2}]$ (matlab instruction `rand`). The number of recursions used in the SRBT algorithm (parameter d in Definition 3) has been set to 2. Hence, the resulting cost of SRBT is $\sim 8n^2$ operations (see Remark 1). To improve the stability, iterative refinement (in the working precision) is added when SRBT is used. Similarly to [2,17], the iterative refinement algorithm is called while $\omega > (n+1)u$, where u is the machine precision. The number of iterations (IR) in the iterative refinement process is also reported in Table 1.

For all matrices, except `orthog` and `ris` with TP and `ris` with SRBT, the factorization with both tile-wise pivoting and randomization achieves satisfactory results. Iterative refinement turns out to be necessary in a few cases when using SRBT but with never more than one iteration (except for `ris` for which neither TP nor SRBT have achieved accurate results). SBRT TP shows slightly better results than SRBT NP. The former only requires iterative refinement for one of the test cases while the latter for a few. For matrix `prolate`, all methods result in a small backward error. However, the solution cannot be accurate at all due to the large condition number. Note that when matrices are orthogonal (`orthog`) or proportional to an orthogonal matrix (`Hadamard`), LDL^T must not be used. Also, `toeppd` is positive definite and would normally be solved by Cholesky and not LDL^T. These three test cases have been used only for testing purposes. In the case of the integer-valued matrix `Hadamard` , SRBT destroys the integer structure and transforms the initial matrix into a real-valued one. For the four random matrices, TP achieves results slightly less accurate than SRBT. However, in these cases iterative refinement added to TP would enable us to achieve an accuracy similar to SRBT.

TP and SRBT are always more accurate than NP but they both failed to produce results as accurate as PP for at least one of the test matrices. Nevertheless, despite the reduced number of test cases, they cover a reasonable range of matrices, including those with zeros on the diagonal. Test case `rand1` has only zeros on the diagonal and was accurately solved by both techniques. This case fails at the very first step of the LDL^T method without pivoting. Test case `orthog` has been solved accurately with SRBT but not with TP. For this particular case, when the pivot search is applied on the full matrix, rows/columns 1 and n are permuted, then rows/columns 2 and $n-1$ are permuted, and so forth. In others, the pivots are spread far apart and the tile-wise pivoting cannot reach them, *i.e.* there are not *good enough* pivots within each tile.

5 Conclusion and Future Work

A tiled LDL^T factorization with tile-wise pivoting and a randomization technique to avoid pivoting in the LDL^T factorization have been presented. The tile-wise pivoting consists of choosing a pivoting strategy and restraining the pivot search to the tile being factored. The randomization technique, called Symmetric Random Butterfly Transformation (SRBT), involves a multiplicative preconditioning which is computationally very affordable and negligible compared to the communication overhead due to classical pivoting algorithms.

Both techniques give accurate results on most test cases considered in this paper, including pathological ones. However, further development of the tile-wise pivoting is required in order to increase its robustness. In particular, techniques such as search by pairs of tiles, also called incremental pivoting, have to be investigated for symmetric indefinite factorizations. Also, to improve stability, the solution obtained after randomization should be systematically followed by iterative refinement in fixed precision (one iteration is sufficient in general). The algorithms presented in this paper shall be integrated into PLASMA, which will allow performance comparisons of the LDL^T solvers and more extensive testing using the matrices available as part of LAPACK.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK User's Guide, 3rd edn. SIAM (1999)

2. Arioli, M., Demmel, J.W., Duff, I.S.: Solving sparse linear systems with sparse backward error. SIAM J. Matrix Anal. and Appl. 10(2), 165–190 (1989)
3. Ashcraft, C., Grimes, R.G., Lewis, J.G.: Accurate symmetric indefinite linear equation solvers. SIAM J. Matrix Anal. and Appl. 20(2), 513–561 (1998)
4. Baboulin, M., Dongarra, J., Tomov, S.: Some issues in dense linear algebra for multicore and special purpose architectures. In: Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, PARA 2008 (2008)
5. Baboulin, M., Dongarra, J., Herrmann, J., Tomov, S.: Accelerating linear system solutions using randomization techniques. Lapack Working Note 246 and INRIA Research Report 7616 (May 2011)
6. Becker, D., Faverge, M., Dongarra, J.: Towards a Parallel Tile LDL Factorization for Multicore Architectures. Technical Report ICL-UT-11-03, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA (April 2011)
7. Björck, Å.: Numerical Methods for Least Squares Problems. Society for Industrial and Applied Mathematics (1996)
8. Bunch, J.R., Kaufman, L.: Some stable methods for calculating inertia and solving symmetric linear systems. Math. Comput. 31, 163–179 (1977)
9. Bunch, J.R., Parlett, B.N.: Direct methods for solving symmetric indefinite systems of linear equations. SIAM J. Numerical Analysis 8, 639–655 (1971)
10. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: Parallel tiled QR factorization for multicore architectures. Concurrency Computat.: Pract. Exper. 20(13), 1573–1590 (2008)
11. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. Syst. Appl. 35, 38–53 (2009)
12. Dongarra, J., Kurzak, J., Langou, J., Langou, J., Ltaief, H., Luszczek, P., YarKhan, A., Alvaro, W., Faverge, M., Haidar, A., Hoffman, J., Agullo, E., Buttari, A., Hadri, B.: PLASMA Users' Guide, Version 2.3. Technical Report, Electrical Engineering and Computer Science Department, Univesity of Tennessee, Knoxville, TN (September 2010)
13. Golub, G.H., van Loan, C.F.: Matrix Computations, 3rd edn. The Johns Hopkins University Press (1996)
14. Higham, N.J.: Accuracy and Stability of Numerical Algorithms, 2nd edn. SIAM (2002)
15. Oettli, W., Prager, W.: Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides. Numerische Mathematik 6, 405–409 (1964)
16. Parker, D.S.: Random butterfly transformations with applications in computational linear algebra. Technical Report CSD-950023, Computer Science Department, UCLA (1995)
17. Skeel, R.D.: Iterative refinement implies numerical stability for Gaussian elimination. Math. Comput. 35, 817–832 (1980)
18. Strazdins, P.E.: Issues in the Design of Scalable Out-of-Core Dense Symmetric Indefinite Factorization Algorithms. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J., Zomaya, A.Y. (eds.) ICCS 2003, Part III. LNCS, vol. 2659, pp. 715–724. Springer, Heidelberg (2003)
19. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 30(3) (2005)

A High Performance Dual Revised Simplex Solver

Julian Hall and Qi Huangfu

School of Mathematics and Maxwell Institute for Mathematical Sciences, University of Edinburgh, JCMB, King's Buildings, Edinburgh, EH9 3JZ, United Kingdom
J.A.J.Hall@ed.ac.uk

Abstract. When solving families of related linear programming (LP) problems and many classes of single LP problems, the simplex method is the preferred computational technique. Hitherto there has been no efficient parallel implementation of the simplex method that gives good speed-up on general, large sparse LP problems. This paper presents a variant of the dual simplex method and a prototype parallelisation scheme. The resulting implementation, ParISS, is efficient when run in serial and offers modest speed-up for a range of LP test problems.

Keywords: Linear programming, Dual revised simplex method, Parallel algorithms.

1 Introduction

When solving families of related linear programming (LP) problems and many classes of single LP problems, the simplex method is the preferred computational technique in the academic and commercial worlds. There is, therefore, considerable motivation for exploring how the simplex method may exploit modern high performance computing (HPC) desktop architectures. This paper describes a variant of the dual simplex method that offers scope for exploiting such architectures. The relevant background is set out in Section 2 and a dual simplex variant, prototype parallelisation scheme and implementation as ParISS are described in Section 3. Computational results using ParISS are given in Section 4 and conclusions in Section 5.

2 Background

A general bounded linear programming (LP) problem in standard form is

$$\begin{array}{rl} \text{minimize} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0}, \end{array} \qquad (1)$$

where $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{b} \in \mathbb{R}^m$. It may be assumed that the matrix A is of full rank.

In the simplex method, the indices of variables are partitioned into sets $\mathcal{B}$ corresponding to m basic variables $\boldsymbol{x}_{B}$, and $\mathcal{N}$ corresponding to $n-m$ nonbasic variables $\boldsymbol{x}_{N}$, such that the basis matrix B formed from the columns of A corresponding to $\mathcal{B}$ is nonsingular. The set $\mathcal{B}$ itself is conventionally referred to as the basis. The columns of A corresponding to $\mathcal{N}$ form the matrix N. The components of $\boldsymbol{c}$ corresponding to $\mathcal{B}$ and $\mathcal{N}$ are referred to as, respectively, the basic costs $\boldsymbol{c}_{B}$ and non-basic costs $\boldsymbol{c}_{N}$.

When each nonbasic variable is set zero, the basic variables take the values $\hat{\boldsymbol{b}} = B^{-1}\boldsymbol{b}$ and if they are non-negative then the partition $\{\mathcal{B},\mathcal{N}\}$ is primal feasible. The *reduced costs* are $\hat{\boldsymbol{c}}_{N}^{T} = \boldsymbol{c}_{N}^{T} - \boldsymbol{c}_{B}^{T}B^{-1}N$ and if they are non-negative then the partition is dual feasible. Primal and dual feasibility is a necessary and sufficient condition for $\{\mathcal{B},\mathcal{N}\}$ to be optimal. The primal simplex method modifies primal feasible partitions until dual feasibility is achieved. Conversely, the dual simplex method modifies dual feasible partitions until primal feasibility is achieved.

A full discussion of the dual simplex method and modern techniques for its efficient implementation are given by Koberstein [11]. At the start of any iteration it is assumed that there is an invertible representation of B, that $\boldsymbol{x}_{N} = \boldsymbol{0}$, that values of $\hat{\boldsymbol{b}} = B^{-1}\boldsymbol{b}$ are known and not feasible, and that the values of $\hat{\boldsymbol{c}}_{N}^{T}$ are feasible. An outline of the computational components of a dual simplex iteration is given in Figure 1, where $\boldsymbol{e}_p$ is column p of the identity matrix. Although dual feasibility is assumed, the algorithm is readily adapted to state-of-the-art techniques for achieving dual feasibility, as presented by Koberstein and Suhl [12].

In CHUZR, the immediate quality of a candidate to leave the basis is the amount by which it is negative. Efficient dual simplex implementations weight this quality by (a measure of) the magnitude of the corresponding row of B^{-1}, the most popular being the steepest edge weight of Forrest and Goldfarb [4] which, for row p, is $s_p = \|B^{-1}\boldsymbol{e}_p\|_2$. To update these weights requires the pivotal column $\hat{\boldsymbol{a}}_q$ and $\boldsymbol{\tau} = B^{-1}\hat{\boldsymbol{a}}_q$. Although calculating $\boldsymbol{\tau}$ adds significantly to the cost of an iteration, the investment usually more than pays off due to the steepest edge strategy reducing the number of iterations.

For the efficient implementation of the revised simplex method, the means by which B^{-1} is represented is of fundamental importance. It is based on an

```
CHUZR: Scan $\hat{\boldsymbol{b}}$ for the row $p$ of a good candidate to leave the basis.
BTRAN: Form $\boldsymbol{\pi}_p^T = \boldsymbol{e}_p^T B^{-1}$.
PRICE: Form the pivotal row $\hat{\boldsymbol{a}}_p^T = \boldsymbol{\pi}_p^T N$.
CHUZC: Scan the ratios $\hat{c}_j/\hat{a}_{pj}$ for a good candidate $q$ to enter the basis.
       Update $\hat{\boldsymbol{c}}_N^T := \hat{\boldsymbol{c}}_N^T - \beta\hat{\boldsymbol{a}}_p^T$, where $\beta = \hat{c}_q/\hat{a}_{pq}$.
FTRAN: Form the pivotal column $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$, where $\boldsymbol{a}_q$ is column $q$ of $A$.
       Update $\hat{\boldsymbol{b}} := \hat{\boldsymbol{b}} - \alpha\hat{\boldsymbol{a}}_q$, where $\alpha = \hat{b}_p/\hat{a}_{pq}$.
If {growth in representation of $B$} then
    INVERT: Form a new representation of $B^{-1}$.
else
    UPDATE: Update the representation of $B^{-1}$ corresponding to the basis change.
end if
```

Fig. 1. Operations in an iteration of the dual revised simplex method

LU decomposition of some basis matrix B_0, determined by a sparsity-exploiting variant of Gaussian elimination as discussed, for example, by Tomlin [18] and Suhl and Suhl [17]. Invertible representations of subsequent basis matrices are obtained by updates until it is preferable on grounds of efficiency or numerical stability to form a new representation via Gaussian elimination. There are many approaches to updating the invertible representation of the basis matrix, the simplest of which is the product form update of Dantzig and Orchard-Hays [3].

2.1 Suboptimization

When in-core memory was severely restricted, a popular variant of the primal revised simplex method incorporated minor iterations of the standard (tableau) primal simplex method restricted to a small subset of the variables. This is described by Orchard-Hays [14] and is referred to as *multiple pricing*. Rosander [15] applied the concept to the dual simplex method, using the term *suboptimization*, performing minor iterations of the standard dual simplex method restricted to a small subset of the constraints. An outline of the computational components of a dual simplex iteration with suboptimization and steepest edge pricing is given in Figure 2.

The disadvantages of using suboptimization for the dual simplex method are that, after the first minor iteration, $\mathcal{P}$ may not contain the row of the best

```
CHUZR: Scan $\hat{b}_p/s_p$ for $p = 1,\dots,m$ to identify a set $\mathcal{P}$ of rows of good candidates to
       leave the basis.
BTRAN: Form $\boldsymbol{\pi}_p^T = \boldsymbol{e}_p^T B^{-1}$, $\forall p \in \mathcal{P}$.
PRICE: Form the pivotal row $\hat{\boldsymbol{a}}_p^T = \boldsymbol{\pi}_p^T N$, $\forall p \in \mathcal{P}$.
Loop {minor iterations}
    CHUZR_MI: Scan $\hat{\boldsymbol{b}}$ for the row $p \in \mathcal{P}$ of a good candidate to leave the basis.
              If $p$ is not defined End loop {minor iterations}.
    CHUZC: Scan the ratios $\hat{c}_j/\hat{a}_{pj}$ for a good candidate $q$ to enter the basis.
           Update $\hat{\boldsymbol{c}}_N^T := \hat{\boldsymbol{c}}_N^T - \beta\hat{\boldsymbol{a}}_p^T$, where $\beta = \hat{c}_q/\hat{a}_{pq}$.
    UPDATE_MI: Update $\mathcal{P} := \mathcal{P}\backslash\{p\}$ and $\hat{\boldsymbol{c}}_N^T := \hat{\boldsymbol{c}}_N^T - \beta\hat{\boldsymbol{a}}_p^T$, where $\beta = \hat{c}_q/\hat{a}_{pq}$.
               Update the rows $\hat{\boldsymbol{a}}_P^T$ and $\hat{\boldsymbol{b}}_P$.
End loop {minor iterations}
For {each basis change} do
    FTRAN1: Form $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$, where $\boldsymbol{a}_q$ is column $q$ of $A$.
            Update $\hat{\boldsymbol{b}} := \hat{\boldsymbol{b}} - \alpha\hat{\boldsymbol{a}}_q$, where $\alpha = \hat{b}_p/\hat{a}_{pq}$.
    FTRAN2: Form $\boldsymbol{\tau} = B^{-1}\hat{\boldsymbol{a}}_q$.
            Update $s_p$ for $p = 1,\dots,m$.
    If {growth in representation of $B$} then
        INVERT: Form a new representation of $B^{-1}$.
    else
        UPDATE: Update the representation of $B^{-1}$ corresponding to the basis change.
    end if
End do
```

Fig. 2. Operations in an iteration of the dual revised simplex method with suboptimization and steepest edge pricing

(global) candidate to leave the basis, and for some of the rows in $\mathcal{P}$ the candidate may no longer be attractive. Thus the number of iterations required to solve the LP problem may increase, and the work of computing some of the pivotal rows may be wasted.

To the knowledge of the authors of this paper, there has been no serial implementation of the dual simplex method with suboptimization beyond the very limited experiments of Rosander. Indeed, there appears to be no meaningful reference to his paper [15]. Thus, in Figure 2, this paper introduces the incorporation of algorithmically efficient techniques into Rosander's framework of the dual simplex method with suboptimization.

2.2 Parallelising the Simplex Method

Previous attempts to develop simplex implementations with the aim of exploiting HPC architectures are reviewed by Hall [7]. Work on exploiting parallelism in the revised simplex method has mainly been restricted to the primal simplex method, notably Forrest and Tomlin [5], Shu [16], Hall and McKinnon [8,9] and Wunderling [19,20]. Only Bixby and Martin [1] considered the dual simplex method, parallelising little more than the matrix-vector product $\boldsymbol{\pi}_p^T N$. This focus on the primal simplex method is understandable since it is the more natural variant to study and only since most of the work on parallel simplex was done has the dual simplex method become the preferred variant.

No parallel implementation of the simplex method has yet offered significantly better performance relative to an efficient serial simplex solver for general large scale sparse LP problems [7]. In two of the more successful implementations [8,9], the limited speed-up came at the cost of unreliability due to numerical instability. Thus any performance improvement in the revised simplex method resulting from an efficient and reliable parallel implementation would be a significant achievement.

Several parallel implementations [8,9,19,20] exploited multiple pricing within the primal revised simplex method. This had the attractive properties of independent computational components that could be overlapped on multiple processors, and minor iterations of the standard simplex method that offered immediate data parallelism. This paper considers the scope for parallelising the dual simplex method with suboptimization.

Historically, revised simplex schemes for HPC architectures have been implemented on relatively small numbers of single-core processors with shared or distributed memory. The widespread availability of desktop HPC systems containing small numbers of multi-core CPUs and a single GPU represents, for optimization practitioners, an environment whose exploitation by existing efficient simplex solvers is limited in the extreme. Thus the development of efficient simplex schemes for such systems is a largely open field of research in which any developments can be expected to have considerable practical value. Consideration of issues relating to massive HPC systems and scalability are of secondary importance to the authors.

3 A Parallel Scheme and Its Implementation

This section introduces a prototype parallelisation scheme for the dual revised simplex method with suboptimization and steepest edge pricing, together with the computational techniques underlying its implementation as the solver ParISS. The inherent strengths and weaknesses of both the parallel scheme and its implementation are discussed.

A Parallelisation Scheme. The following scheme exploits some, but by no means all, of the available task and data parallelism in the dual revised simplex method with suboptimization. It is discussed in detail below, taking the operations in Figure 2 in order. Its implementation, ParISS, assigns one tableau row to each available core and Figure 3 illustrates an example of the distribution of operations in the case of four cores.

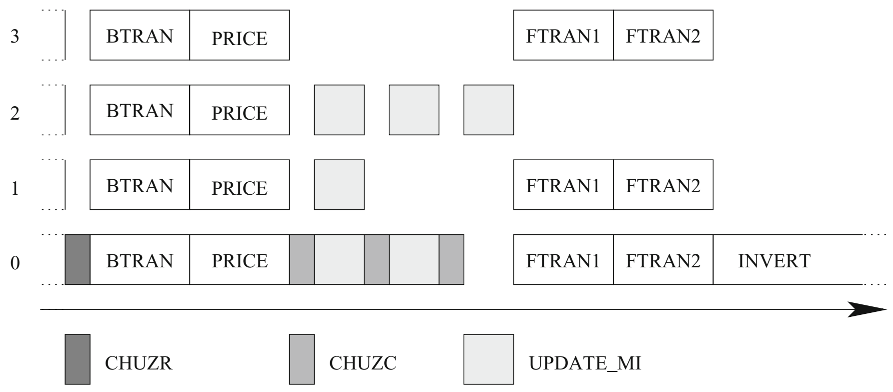


Fig. 3. ParISS: a prototype parallel implementation of the dual revised simplex with suboptimization

Firstly, the relatively cheap CHUZR operation of selecting a set $\mathcal{P}$ of good candidates to leave the basis is executed on just one core. Following this, the multiple BTRAN ($\boldsymbol{\pi}_p^T = \boldsymbol{e}_p^T B^{-1}$), and PRICE ($\hat{\boldsymbol{a}}_p^T = \boldsymbol{\pi}_p^T N$) operations for $p \in \mathcal{P}$ are distributed over all cores. Since minor iterations are performed with only a very small subset of rows, the CHUZR_MI operation is trivial so is performed by one core and not illustrated. The selection of the entering column performed by CHUZC is relatively cheap so is also performed on one core. The minor iteration is completed by UPDATE_MI, in which the data parallel update of the dual simplex tableau rows (that remain candidates) and the reduced costs $\hat{\boldsymbol{c}}_N^T$ is distributed over all cores. The trivial update of $\hat{\boldsymbol{b}}_P$ is performed by the core that will perform CHUZR_MI. Following the minor iterations, FTRAN operations $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$ and $\boldsymbol{\tau} = B^{-1}\hat{\boldsymbol{a}}_q$ for each basis change are distributed over all cores. If INVERT is performed, it is executed serially, without overlapping any other computation.

Serial Efficiency. In many earlier parallel implementations of the revised simplex method [7], the serial inefficiency of the computational components yielded greater scope for parallel efficiency but the resulting implementation was wholly inferior to a good serial solver. For a parallel simplex solver to be of greatest practical value, it is important that its components are computationally efficient. Not only must they exploit the sparsity of the constraint matrix, but they should consider the more advanced techniques for exploiting hyper-sparsity identified by Hall and McKinnon [10] that have led to huge performance improvements for important classes of LP problems.

Implementation of the Scheme and Its Parallel Efficiency. The aim of the prototype implementation, ParISS, is to identify the immediate scope for parallelism when using efficient serial components. Like the underlying scheme, ParISS is not fully parallel efficient. In particular, for convenience, it identifies a "master" core that is used for computation when an alternative core might be preferable on grounds of data locality.

Figure 3 clearly illustrates the major parallel inefficiency in the scheme, that of INVERT being performed serially with no other overlapping computation. There is a similar, but less serious, source of parallel inefficiency when performing the very much cheaper operations CHUZR and CHUZC. Both of these perform comparison operations on each component of a full-length vector, so it may be significantly more efficient to distribute the data over multiple cores and accumulate the results from each. In ParISS, CHUZC is performed on the master core rather than the core where the pivotal row is likely to be available in cache.

Another source of parallel inefficiency in the scheme occurs if not all the candidates in $\mathcal{P}$ yield a basis change, in which case the number of FTRAN operations may not be a multiple of the number of available cores. In ParISS, a core performs both FTRAN operations if and only if its (single) tableau row is pivotal. Thus, if the candidate variable in this row does not leave the basis, the core performs no FTRAN operations. Figure 3 illustrates this in the case where the last candidate in $\mathcal{P}$ does not leave the basis. With hindsight, this tableau row is computed and updated unnecessarily but this is a serial consequence of suboptimization.

The idealised Gantt chart assumes that all BTRAN, PRICE and FTRAN operations have the same computational cost but this is not so in practice when they are performed efficiently. Immediate variance results from the fact that for FTRAN1 ($\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$) vector $\boldsymbol{a}_q$ is a column of the (sparse) constraint matrix whereas, for FTRAN2 ($\boldsymbol{\tau} = B^{-1}\hat{\boldsymbol{a}}_q$), $\hat{\boldsymbol{a}}_q$ may be a full vector. When exploiting hyper-sparsity, the variance increases further. In ParISS, hyper-sparsity is exploited partially during BTRAN and FTRAN, and fully during PRICE, with A being held row-wise and combined according to the nonzeros in $\boldsymbol{\pi}_p$. Thus ParISS can also be expected to suffer from parallel inefficiency due to load imbalance.

4 Results

ParISS, a prototype implementation of the dual revised simplex with suboptimization has been developed in C++ with OpenMP using the techniques

Table 1. Speed-up of ParISS on up to 8 cores and performance relative to `Clp`

				Speed-up				
				Relative to 1 core			Relative to `Clp`	
Problem	Rows	Columns	Entries	2 cores	4 cores	8 cores	1 cores	8 cores
25fv47	822	1571	11127	1.18	1.07	0.53	0.51	0.27
80bau3b	2263	9799	29063	0.94	1.03	0.85	0.21	0.18
cre-b	9649	72447	328542	0.86	1.27	1.07	1.13	1.21
cre-d	8927	69980	312626	1.16	1.33	1.68	0.90	1.51
degen3	1504	1818	26230	1.28	1.19	1.12	0.31	0.35
fit2p	3001	13525	60784	0.96	0.92	0.94	0.44	0.41
osa-14	2338	52460	367220	0.93	0.84	1.01	0.17	0.17
osa-30	4351	100024	700160	1.06	1.00	0.90	0.20	0.18
pds-06	9882	28655	82269	1.62	2.19	3.06	0.47	1.43
pds-10	16559	48763	140063	1.27	1.83	1.97	0.51	1.00
qap8	913	1632	8304	1.37	1.24	1.72	0.31	0.54
stocfor3	16676	15695	74004	1.80	2.57	3.39	0.13	0.46
truss	1001	8806	36642	0.98	1.08	1.10	0.46	0.50
Mean				1.16	1.28	1.30	0.37	0.48

discussed in Section 3. Using the Intel C++ compiler, the code was tested on a dual quad-core AMD Opteron 2378 system.

Experiments were performed using problems from the standard Netlib [6] and Kennington [2] test sets of LP problems. Results are given in Table 1 for a subset of problems chosen as follows. Of the original 114 problems, many are unrealistically small so the 84 that ParISS solved in less than one second were discounted. Of the remaining 30, ParISS failed to solve 14 in one or more of the runs using 1, 2, 4 and 8 cores. To ensure that results are given for problems that ParISS solves efficiently, observe that when a single core is used only one tableau row is computed so ParISS executes the standard serial revised dual simplex algorithm. Thus its efficiency was assessed by comparing ParISS on one core with version 1.06 of the COIN-OR [13] (serial) dual simplex solver `Clp`. For the remaining 16 problems ParISS was more than ten times slower than CLP on three, so these problems were also discounted. Thus the results in Table 1 are for the 13 LP test problems for which ParISS required at least one second of CPU on one core but were solved efficiently on one core and successfully on 2, 4 and 8 cores.

As the results in columns 5–7 of Table 1 show, some speed-up was obtained for all but two of the problems and the (geometric) mean speed-up was about 30% on 4 and 8 cores. Column 8 indicates the speed of ParISS on one core relative to `Clp`: ParISS was faster for only one problem and was slower by a factor of 2.7 on average. However, using 8 cores, ParISS was at least as fast as `Clp` for four problems, and 2.1 times slower on average.

5 Conclusions

For a prototype solver with the limited parallel efficiency recognised in Section 3, the results in Table 1 are very encouraging, both in terms of speed-up and performance relative to `Clp`. This is particularly so when considering the limited success of previous attempts to exploit HPC architectures when solving general sparse LP problems with efficient implementations of the revised simplex method. The results demonstrate that it is possible to get a worthwhile performance improvement in an efficient implementation of the dual revised simplex method by exploiting the scope for parallelism offered by suboptimization when solving a range of sparse LP problems. Enhancements to the parallel efficiency of the scheme and its implementation, together with improved serial performance of its components are all the subject of current research.

References

1. Bixby, R.E., Martin, A.: Parallelizing the dual simplex method. INFORMS Journal on Computing 12, 45–56 (2000)
2. Carolan, W.J., Hill, J.E., Kennington, J.L., Niemi, S., Wichmann, S.J.: An empirical evaluation of the KORBX algorithms for military airlift applications. Operations Research 38, 240–248 (1990)
3. Dantzig, G.B., Orchard-Hays, W.: The product form for the inverse in the simplex method. Math. Comp. 8, 64–67 (1954)
4. Forrest, J.J.H., Goldfarb, D.: Steepest-edge simplex algorithms for linear programming. Mathematical Programming 57, 341–374 (1992)
5. Forrest, J.J.H., Tomlin, J.A.: Vector processing in the simplex and interior methods for linear programming. Annals of Operations Research 22, 71–100 (1990)
6. Gay, D.M.: Electronic mail distribution of linear programming test problems. Mathematical Programming Society COAL Newsletter 13, 10–12 (1985)
7. Hall, J.A.J.: Towards a practical parallelisation of the simplex method. Computational Management Science 7, 139–170 (2010)
8. Hall, J.A.J., McKinnon, K.I.M.: PARSMI, a Parallel Revised Simplex Algorithm Incorporating Minor Iterations and Devex Pricing, in Applied Parallel Computing. In: Madsen, K., Olesen, D., Waśniewski, J., Dongarra, J. (eds.) PARA 1996. LNCS, vol. 1184, pp. 67–76. Springer, Heidelberg (1996)
9. Hall, J.A.J., McKinnon, K.I.M.: ASYNPLEX, an asynchronous parallel revised simplex method algorithm. Annals of Operations Research 81, 27–49 (1998)
10. Hall, J.A.J., McKinnon, K.I.M.: Hyper-sparsity in the revised simplex method and how to exploit it. Computational Optimization and Applications 32, 259–283 (2005)
11. Koberstein, A.: Progress in the dual simplex algorithm for solving large scale LP problems: techniques for a fast and stable implementation. Computational Optimization and Applications 41, 185–204 (2008)
12. Koberstein, A., Suhl, U.H.: Progress in the dual simplex method for large scale LP problems: practical dual phase 1 algorithms. Computational Optimization and Applications 37, 49–65 (2007)
13. Lougee-Heimer, R., et al.: The COIN-OR initiative: Open source accelerates operations research progress. ORMS Today 28, 20–22 (2001)
14. Orchard-Hays, W.: Advanced Linear programming computing techniques. McGraw-Hill, New York (1968)
15. Rosander, R.R.: Multiple pricing and suboptimization in dual linear programming algorithms. Mathematical Programming Study 4, 108–117 (1975)

16. Shu, W.: Parallel implementation of a sparse simplex algorithm on MIMD distributed memory computers. Journal of Parallel and Distributed Computing 31, 25–40 (1995)
17. Suhl, U.H., Suhl, L.M.: Computing sparse LU factorizations for large-scale linear programming bases. ORSA Journal on Computing 2, 325–335 (1990)
18. Tomlin, J.A.: Pivoting for size and sparsity in linear programming inversion routines. J. Inst. Maths. Applics. 10, 289–295 (1972)
19. Wunderling, R.: Paralleler und objektorientierter simplex, Tech. Report TR-96-09, Konrad-Zuse-Zentrum für Informationstechnik Berlin (1996)
20. Wunderling, R.: Parallelizing the simplex algorithm. ILAY Workshop on Linear Algebra in Optimzation, Albi (April 1996)

TFETI Coarse Space Projectors Parallelization Strategies

Vaclav Hapla and David Horak

Department of Applied Mathematics,
VSB-Technical University of Ostrava,
17. listopadu 15,
CZ 708 33 Ostrava-Poruba,
Czech Republic
vaclav.hapla@vsb.cz

Abstract. This paper deals with an analysis of various parallelization strategies for the TFETI algorithm. The data distributions and the most relevant actions are discussed, especially those concerning coarse problem. Being a part of the coarse space projector, it couples all the subdomains computations and accelerates convergence. Its direct solution is more robust but complicates the massively parallel implementation. Presented numerical results confirm high parallel scalability of the coarse problem solution if the dual constraint matrix is distributed and then orthonormalized in parallel. Benchmarks were implemented using PETSc parallelization library and run on HECToR service at EPCC in Edinburgh.

Keywords: domain decomposition, FETI, Total FETI, TFETI, PETSc, natural coarse space, coarse problem.

1 Introduction

The spread of parallel computers became the main impulse for development of new numerical methods to respond to brand new criterion of efficiency - parallel scalability. It means that the time of solution should be nearly inversely proportional to the number of processors used. It is often impaired with decreasing discretization parameter resulting in rapid increase of number of iterations. This is motivation for developing algorithms which are also numerically scalable, number of iterations is nearly independent of the number of nodal variables.

Domain decomposition methods form a class of methods for parallel solution of elliptic partial differential equations arising from many technical problems. These methods are based on the "divide and conquer" strategy. The FETI (Finite Element Tearing and Interconnecting) method proposed by Farhat and Roux [5] turned out to be one of the most successful. The FETI-1 method is based on the decomposition of the spatial domain into the non-overlapping subdomains that are "glued" by Lagrange multipliers.

Efficiency of the FETI-1 method was further improved by introducing special orthogonal projectors and preconditioners. By projecting the Lagrange multipliers in each iteration onto an auxiliary space to enforce continuity of the primal solutions at the crosspoints, Farhat, Mandel and Tezaur obtained a faster converging FETI method for plate and shell problems - FETI-2.

Total-FETI (TFETI) by Dostal et al. [1] simplified the implementation of a stiffness matrices pseudoinverse using Lagrange multipliers not only for gluing the subdomains along the auxiliary interfaces, but also to enforce the Dirichlet boundary conditions.

FETI methods are also very suitable for the solution of variational inequalities [5,2]. The reason is that the application of the duality principle not only reduces the problem size and condition number; it also transforms the affine inequality constraints into the bound constraints so that efficient algorithms that exploit cheap projections and other tools may be exploited.

The stiffness matrix has a perfect block-diagonal layout with subblocks corresponding to subdomains. So the most difficult part – the application of the stiffness matrix – may be carried out in parallel without any communication so that high parallel scalability is enjoyed. The increasing number of subdomains decreases the block size resulting in shorter time for subdomain stiffness matrices factorization and subsequently forward and backward substitution during the pseudoinverse application.

On the other hand, assuming fixed discretization parameter, the dual dimension and the coarse problem size increase with increasing number of subdomains resulting in longer time for all dual vector operations and orthogonal projectors application.

The coarse problem couples all the subdomains computations and accelerates convergence. Direct solution of the coarse problem improves the robustness but complicates the massively parallel implementation. In this paper, coarse problem parallelization strategies and their impact on the parallel scalability will be discussed.

2 FETI-1 and TFETI

Let us consider linear elasticity problem in 2D or 3D. It is assumed that the boundary Γ is decomposed into disjoint parts Γ_u, Γ_f and that there are imposed Dirichlet boundary conditions on Γ_u and Neumann boundary conditions on Γ_f. The Dirichlet boundary conditions represent the prescribed displacements and the Neumann conditions represent the surface tractions.

To apply the FETI-1 based domain decomposition let us partition the domain Ω into N_s subdomains Ω^s and denote by K^s, f^s, and u^s the subdomain stiffness matrix, the subdomain load vector and the subdomain displacement vector. The matrix B enforces the continuity of the displacements across the subdomain interfaces (gluing conditions). The continuity between i-th and j-th DOF means that $u_i = u_j$ and can be enforced by appending the row b with entries $b_i = 1, b_j = -1, b_k = 0, k \neq i, j$, to B.

The finite element discretization with appropriate nodes numbering results in the quadratic programming problem

$$\min \frac{1}{2}u^T K u - u^T f \quad \text{s.t.} \quad Bu = 0 \tag{1}$$

with

$$K = \begin{bmatrix} K^1 & & \\ & \ddots & \\ & & K^{N_s} \end{bmatrix}, \quad f = \begin{bmatrix} f^1 \\ \vdots \\ f^{N_s} \end{bmatrix}, \quad u = \begin{bmatrix} u^1 \\ \vdots \\ u^{N_s} \end{bmatrix}, \quad B = [B^1, \ldots, B^{N_s}]. \tag{2}$$

The original FETI-1 method assumes that the boundary subdomains inherit the Dirichlet conditions from the original problem, so that the defect of the stiffness matrices K^s may vary from zero corresponding to the boundary subdomains with sufficient Dirichlet data to fix rigid body modes to the maximum corresponding to the floating subdomains.

The basic idea of TFETI [1] is to keep all the subdomains floating and to incorporate the imposed displacements into the matrix of constraints B. To implement the boundary conditions like $u_i = 0$, the row b with $b_i = 1, b_k = 0, k \neq i$, is appended to B. The prescribed displacements will be then enforced by the Lagrange multipliers which may be interpreted as reaction forces. The remaining procedure is exactly the same as described for FETI-1. The key point is that the kernels of local stiffness matrices K^s are known a priori, have the same dimension and can be formed directly. Matrix R whose columns form a basis of the kernel of K has a block-diagonal structure

$$R = \begin{bmatrix} R^1 & & \\ & \ddots & \\ & & R^{N_s} \end{bmatrix}. \tag{3}$$

Let us apply the duality theory to (1) 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 $KK^\dagger K = K$. Our minimization problem reads

$$\min \frac{1}{2}\lambda^T F \lambda - \lambda^T d \quad \text{s.t.} \quad G\lambda = e. \tag{4}$$

Equality constraints $G\lambda = e$ can be homogenized to $G\lambda = 0$ by choosing arbitrary $\widetilde{\lambda}$ which satisfies $G\widetilde{\lambda} = e$ and replacing d by $d - F\widetilde{\lambda}$. Then the solution of original problem can be obtained as $\lambda + \widetilde{\lambda}$.

Further significant improvement is based on the observation, that the Lagrangian for problem (4) can be decomposed by orthogonal projectors

$$Q = G^T(GG^T)^{-1}G \qquad \text{and} \qquad P = I - Q$$

onto the image space of G^T and onto the kernel of G, respectively ($\mathrm{Im}Q = \mathrm{Im}G^T$ and $\mathrm{Im}P = \mathrm{Ker}G$), so that the final problem reads

$$\min \frac{1}{2}\lambda^T PFP\lambda - \lambda^T Pd \quad \text{s.t.} \quad G\lambda = 0. \tag{5}$$

Alternatively, we can reformulate problems (4)-(5) using $\bar{G} = TG$ and $\bar{e} = Te$ instead of G and e, respectively, where T denotes a nonsingular matrix that defines the orthonormalization of the rows of G so that $\bar{G}\bar{G}^T = I$. In this case Q can be simplified

$$Q = G^T(GG^T)^{-1}G = \bar{G}^T\bar{G}$$

and $\widetilde{\lambda}$ can be chosen as the least square solution $\widetilde{\lambda} = G^T(GG^T)^{-1}e = \bar{G}^T\bar{e}$.

Problem (5) may be solved effectively by a scalable algorithm PCGP (Preconditioned Conjugate Gradient method with Projectors). The proof of the classical estimate $\kappa(PFP|\mathrm{Im}P) \leq C\frac{H}{h}$ of the spectral condition number κ of PFP on the range of P by the ratio of the decomposition parameter H and the discretization parameter h which was published by Farhat, Mandel and Roux remains valid for TFETI.

3 Coarse Problem Parallelization Strategies

3.1 Data Distribution

The parallelization of the TFETI algorithm can be achieved mostly by SPMD technique – distributed matrices. This allows algorithms to be almost the same for both sequential and parallel cases; only implementations of data structures (mainly matrix and vector) differ. Most computations (subdomains problems) are purely local and therefore parallelizable without any data transfers. However, if we want to accelerate also dual actions, some communication is needed due to primal-dual transition – TFETI solver is then not "embarassingly parallel".

Let us firstly describe distributions of primal data K, B, R, f. Let N_p denote the global primal dimension (number of DOFs); N_d the dual dimension (number of constraints); N_n the null space dimension (defect) of K; $N_c, N_c \leq N_s$, the number of computational cores. For sufficiently finely discretized problems, it holds that $N_n \ll N_d \ll N_p$. For the special case of 2D elasticity it holds that $N_n = N_s \times 3$. Sizes of the primal objects are as follows: $size(K) = [N_p, N_p]$, $size(B) = [N_d, N_p]$, $size(R) = [N_p, N_n]$, $size(f) = [N_p, 1]$.

Distribution of primal matrices K, B, R is quite straightforward as every subblock corrresponds to a subdomain; see Figure 1. They can be stored using general distributed column-block or row-block matrix type (e.g. `MPIAIJ` in PETSc, codistributed arrays in MATLAB). However, K and R possess nice

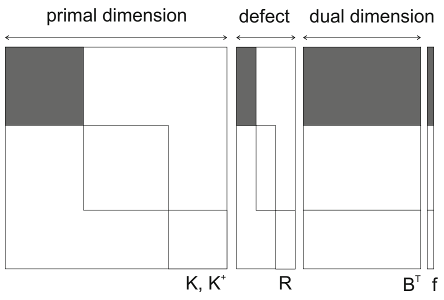


Fig. 1. Natural distributions of primal data. (Filled part means local portion owned by single core).

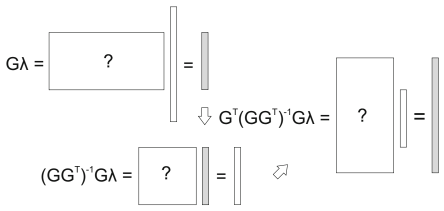


Fig. 2. Application of Q projector

block-diagonal layout with nonzeros only in the diagonal subblocks and can be implemented more sophisticatedly using block-diagonal composite type where subblocks are ordinary sequential matrices and every process holds one or array of them.

There is no doubt about perfect parallel scalability of factorization of the stiffness matrix K and its pseudoinverse $K^\dagger$ action because of their block-diagonal layout; for the fixed decomposition and discretization parameters the time is reduced proportionally to N_c. Natural effort is to maximize N_c so that sizes of subblocks are reduced which accelerates their factorization and pseudoinverse application, which belong to the most time consuming actions. On the other hand, the negative effect of that is increasing N_d and N_n. It is a kind of "seesaw effect".

The critical point and potential bottleneck of the TFETI method is the application of projector Q (and therefore P) as depicted in Figure 2. We should consider following aspects affecting action time and the level of communication:

- whether and how G should be distributed,
- how the action of $(GG^T)^{-1}$ and hence Q and P applications should be implemented,
- whether G should be orthonormalized obtaining $\bar{G}$ so that $(\bar{G}\bar{G}^T)^{-1} = I$.

3.2 Implementation of the Dual Actions

We suggest seven main ways (A1-A3, B1-B4) of handling the matrix G and subsequent realizations of coarse problem $GG^Tx = b$ solution. These strategies of the projector application can be viewed from two points:

I. how G is *distributed* and its *action carried out* (see figures 3a, 3c):
   A) horizonatal blocks,
   B) vertical blocks.
II. how *the coarse problem is solved* which implies *the level of preprocessing of G and GG^T* (see figures 3b, 3d):

1) iteratively using CG by the master process,
2) directly using Cholesky factorization by the master process,
3) applying explicit inverse of GG^T in parallel,
4) the coarse problem is eliminated, provided that the rows of G are orthonormalized.

Let us analyze necessary actions in detail. Matrices R and B are naturally divided into sparse sequential blocks $R_{[rank]}$ and $B_{[rank]}$ as told in previous section and depicted in Figure 3a.

Firstly, focus on the viewpoint I. **In both A and B cases**, local blocks of rows of the natural coarse space matrix G are computed in parallel without any communication: $G_{[rank]} = R^T_{[rank]}B^T_{[rank]}$.

In B case, an additional redistribution is performed so that G is divided horizontally into column blocks. In fact, due to the row-major storage of parallel sparse matrices in PETSc, G is tranposed and horizontal blocks $G^T_{[rank]}$ are stored. This approach has two big advantages:

1. smaller amount of communication because collective operations (global sums) are performed on N_n-sized vectors instead of N_d-sized (see Figure 3c),
2. all N_d-sized vectors and operations on them like AXPY and DOT can be parallelized.

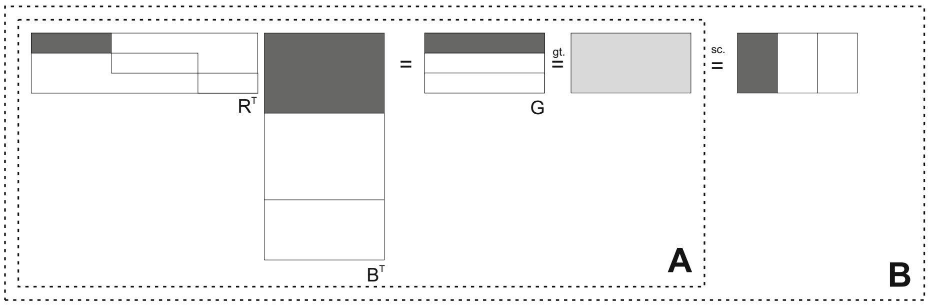
Taking into account decomposition of huge problems into large number of subdomains resulting in large dual dimension and dimension of kernel R, this can significantly reduce the computational time.

Now let us consider the viewpoint II. In the pictures 3d, 3b, we can see the four ways of coarse problem solution and corresponding levels of preprocessing. In **cases 1, 2** it is necessary to transfer whole G matrix to the zeroth core or in **case 3** to all cores. Master core or each of the cores then computes the sequential product GG^T using matrix-matrix multiplication.

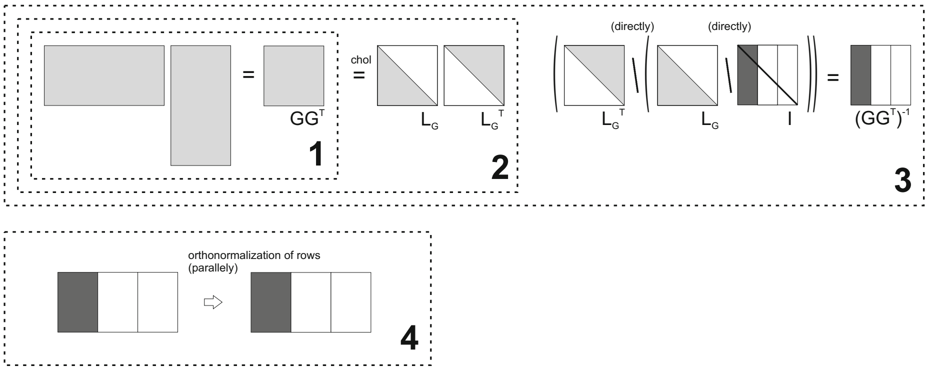
In **case 1** we employ iterative (KSP) solver for the coarse problem solution.

In **cases 2, 3** the Cholesky factorization of GG^T is performed. In case 2, a finite solver is employed for the coarse problem solution on the zeroth core. In case 3, an explicit inverse $(GG^T)^{-1}$ is carried out, which can be efficiently computed and applied in parallel. Coarse problem is then solved by means of ordinary matrix-vector multiplication by the distributed $(GG^T)^{-1}$ matrix.

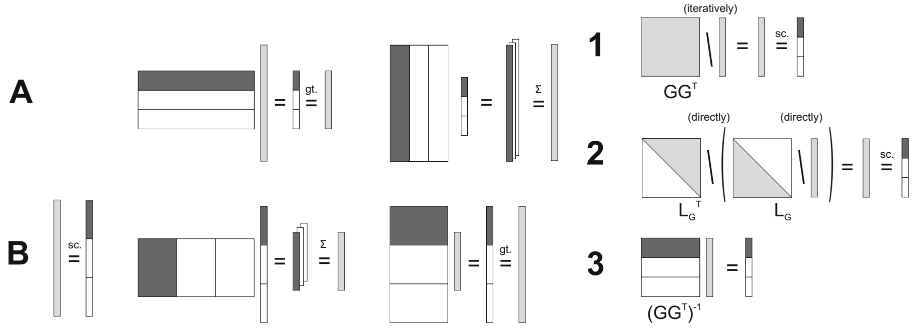
In **case B4** rows of G are orthonormalized and matrix $\bar{G}$ is obtained. This way has a big advantage - we eliminate the coarse problem ($\bar{G}\bar{G}^T = (\bar{G}\bar{G}^T)^{-1} = I$) completely. Like for every B case, we have to redistribute G into the parallel dense matrix G^T. Then we perform orthonormalization of its columns, i.e. rows of G. This action can be virtually described by the nonsingular matrix T and the result is $\bar{G} = TG$. For this purpose the classical Gram-Schmidt algorithm was chosen, that appears more suitable for parallelization of this problem than the modified or iterated classical Gram-Schmidt (the classical Gram-Schmidt requires half the number of floating-point operations, on parallel computers it can be much faster than the modified Gram-Schmidt, and its parallel efficiency equals that of the iterated classical Gram-Schmidt [4]). The columns of matrix



(a) parallelization of G



(b) preprocessing of G and GG^T



(c) action of G depending on (a) **(d)** action of $(GG^T)^{-1}$ depending on (b)

Fig. 3. Various ways of realization of the Q action

G^T are copied into the array of parallel vectors $g[\,]$ (local size N_d/N_c, global size N_d) and process of orthonormalization is performed in parallel according to

$$g[i] = g[i] - \sum_{j=0}^{i-1} (g[i]^T g[j]) g[j], \;\; g[i] = \frac{g[i]}{\|g[i]\|}, i = 0, ..., N_n - 1,$$

using standard PETSc functions. The obtained vectors form columns of required parallel dense matrix $\bar{G}^T$. Considering this type of distribution, this process requires only transfers of dot products and can be very efficient [4]. Alternatively, $\bar{G}$ can be carried out by the forward substitution of the factorized GG^T applied to N_d/N_c columns of original G matrix as RHS.

4 Numerical Experiments

Described strategies were tested on matrices B and R obtained from decomposition and discretization of 2D elastostatic problem of the steel traverse represented by a domain $\Omega = (0, 600) \times (0, 200)$ with the sizes given in [mm] (see Fig. 4). The material constants are defined by the Young modulus $E = 2.1 \cdot 10^5$ [MPa], the Poisson ratio $\nu = 0.3$, and the density $\rho = 7.85 \cdot 10^{-9}$ [ton/mm^3]. The body is fixed in all directions along the left side $\overline{\Gamma}_U = \{0\} \times [0, 200]$ and loaded by gravitational forces with the gravity acceleration $g = 9800$mm/s^2.

To illustrate both the efficiency of the different strategies of the coarse problem solution and the scalability of the TFETI we decomposed the body Ω into identical square subdomains with the side length H (see Fig. 5). We gradually chose decompositions into 8×24, 16×48, 24×72, 32×96, and 40×120 boxes. All subdomains were further discretized by the same uniform triangular meshes characterized by the discretization parameter h and the ratio $H/h = 180$. An example of the deformed body together with the traces of decomposition for the choice of parameters $h = 16.7$mm and $H = 66.7$mm is depicted in Fig. 5.

The implementation was built on PETSc 3.0.0.10. Regarding parallel machine, *HECToR* service at EPCC in Edinburgh [8] was used. It consists of the *phase 2a* (Cray XT5h) machine, the *phase 2b* (Cray XT6) machine, and an archiving facility. The current main HECToR facility is the phase 2b. This was used for our experiments. Phase 2b is a Cray XE6 system offering a total of 1856 XE6 computing nodes. Each compute node contains two AMD 2.1 GHz 12-core processors giving a total of 44,544 cores. Theoretical peak performance is around 373 Tflops. There is presently 32 GB of main memory available per node, which is shared between its twenty-four cores, the total memory is 58 TB. The processors are connected with a high-bandwidth interconnect using Cray Gemini communication chips. The Gemini chips are arranged on a 3 dimensional torus.

Distribution into horizontal blocks in A cases leads to enormous increase of communication because sequential dual vectors have to be scattered to zeroth core and added together. On the other hand, the reduction of computation times is not so significant. So the only way to run huge jobs on massively parallel computers is to distribute the G matrix into vertical blocks and distribute all dual vectors as well – B cases. A big advantage is that the AXPY action and the dot product is faster in parallel compared to sequential case – see Table 1.

Regarding communication, the B variants behave almost identically. Therefore, the best strategy should be chosen according to the computational time. While for the finitely and iteratively solved coarse problem, these times stay constant with increasing number of processors, for the variants with parallelized coarse problem the speedup is nearly optimal up to thousands of cores.

Additional advantage of orthonormalized approach is less than 50% cost of preprocessing.

In some cases the experiments done in parallel are not faster than sequential ones. This can be caused by the fact, that the example was not large enough - for larger dual dimensions and dimension of null space the situation should change and expected profit from the parallelization will be more significant. We should also take into account the reduction of memory requirements if many computational nodes are used. The importance of this parallelization will be magnified with more complicated problems as 3D elasticity or shell problems. This is a topic of our current work.

In Table 1, we report the computational times for preprocessing and Q action for the coarse problem solution. Obviously the best strategy is $B4$, where G is distributed in vertical blocks and orthonormalized.

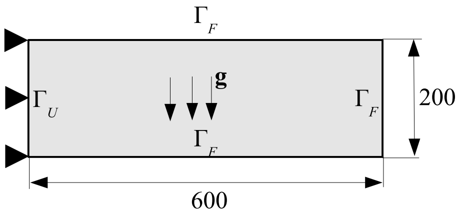


Fig. 4. Benchmark geometry

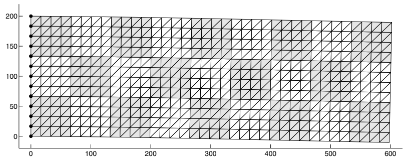


Fig. 5. Displacements with traces of decomposition (scaled 6000x)

Table 1. Performance of the coarse problem solution for varying strategy, decomposition and discretization

Primal variables	12,580,224	50,320,896	114,476,544	201,283,584
Dual variables	129,984	537,216	1,228,464	2,183,424
Kernel dimension	576	2,304	5,184	9,216
Number of subdom.	192	768	1,728	3,072
Number of cores	192	768	1,728	3,072
$\mathbf{G}_{[rank]}$	1.001e-02	1.152e-02	1.489e-02	1.527e-02
broadcast of **G** to all cores	9.102e-02	3.710e-01	8.353e-01	1.389e+00
B1,2,3: $\mathbf{GG}^T$ assembling	6.710e-02	2.469e-01	7.155e-01	1.203e+00
B2,3: $\mathbf{GG}^T$ Chol. fact.	8.090e-03	1.042e-01	8.108e-01	2.004e+00
$B3$: inverse	1.767e-01	1.149e+00	6.401e+00	9.264e+00
$B4$: orthonormalization	9.669e-02	5.983e-01	3.262e+00	4.629e+00
$B1$: **Q** action	1.070e-02	6.934e-02	3.204e-01	6.424e-01
$B2$: **Q** action	8.046e-03	5.404e-02	2.321e-01	4.576e-01
$B3$: **Q** action	5.822e-03	3.742e-02	1.760e-01	3.621e-01
$B4$: **Q** action	6.096e-03	2.694e-02	6.424e-02	9.874e-02
AXPY seq. dual.	1.452e-03	6.622e-03	1.503e-02	3.018e-02
AXPY par. dual.	2.766e-06	3.195e-06	3.386e-06	3.833e-06
DOTS seq. dual.	1.073e-03	4.549e-03	1.031e-02	2.482e-02
DOTS par. dual.	2.384e-04	5.014e-04	5.044e-03	1.081e-02

5 Conclusion and Further Work

This paper analyzes seven parallelization strategies for coarse problem of FETI-1 and TFETI algorithm and projector application. The data distributions and the most relevant actions are discussed with the conclusion that all B variants behave much better in parallel up to thousands cores reducing significantly times of all dual actions (AXPY, DOT).

Further work should involve fine tuning and optimization of presented actions, theoretical performance analysis, more holistic analyses of FETI-based algorithms extending this paper, comparison of times presented in this paper (pure MPI approach) with hybrid parallelization approaches, testing coarse problem solved by parallel direct solver such as MUMPS, experiments with parallelization frameworks other than PETSc.

Acknowledgments. This research has been supported by the grants: GA CR 101/09/P601, Ministry of Education of the CR (No. MSM 6198910027), HPC-Europa programme funded under the European Commission Research Infrastructures activity of the Structuring the European Research Area programme (contract No. RII3-CT-2003-506079) and PRACE 1IP project receiving funding from the EU's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. RI-261557.

References

1. Dostal, Z., Horak, D., Kucera, 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)
2. Dostal, Z., Horak, D.: Theoretically supported scalable FETI for numerical solution of variational inequalities. SIAM Journal on Num. Anal. 45, 500–513 (2007)
3. Balay, S., Gropp, W., McInnes, L.C., Smith, B.: PETSc 3.0.0 Users Manual, Argonne National Laboratory, http://www.mcs.anl.gov/petsc/
4. Lingen, F.J.: Efficient Gram-Schmidt orthonormalization on parallel computers. Research report, Department of Aerospace Engineering, Delft University of Technology (1999)
5. Farhat, C., Roux, F.-X.: An unconventional domain decomposition method for an efficient parallel solution of large-scale finite element systems. SIAM Journal on Scientific Computing 13, 379–396 (1992)
6. Farhat, C., Mandel, J., Roux, F.-X.: Optimal convergence properties of the FETI domain decomposition method. Comput. Methods Appl. Mech. Eng. 115, 365–385 (1994)
7. Roux, F.-X., Farhat, C.: Parallel implementation of direct solution strategies for the coarse grid solvers in 2-level FETI method. Contemporary Math. 218, 158–173 (1998)

8. HECToR Home Page, http://www.hector.ac.uk/
9. Vondrak, V., Kozubek, T., Markopoulos, A., Dostal, Z.: Parallel solution of contact shape optimization problems based on Total FETI domain decomposition method. Structural and Multidisciplinary Optimization 42(6), 955–964 (2010)
10. Kozubek, T., Dostal, Z., Vondrak, V., Markopoulos, A., Brzobohaty, T.: Scalable TFETI algorithm for the solution of multibody contact problems of elasticity. International Journal for Numerical Methods in Engineering 82(11), 1384–1405 (2010)

FFTs and Multiple Collective Communication on Multiprocessor-Node Architectures

Andreas Jocksch

CSCS, Swiss National Supercomputing Centre,
Galleria 2, Via Cantonale, 6928 Manno, Switzerland
jocksch@cscs.ch

Abstract. We consider FFTs for networks with multiprocessor nodes using 2D data decomposition. In this application, processors perform collective all-to-all communication in different groups independently at the same time. Thus the individual processors of the nodes might be involved in independent collective communication. The underlying communication algorithm should account for that fact. For short messages, we propose a sparse version of Bruck's algorithm which handles such multiple collectives. The distribution of the FFT data to the nodes is discussed for the local and global application of Bruck's original algorithm, as well as the suggested sparse version. The performance of the different approaches is compared.

Keywords: FFT, all-to-all personalized communication, multiprocessor nodes.

1 Introduction

The parallelization of one and multi-dimensional Fast Fourier Transforms (FFTs) [12,7] is a key aspect in scientific computing, and methods for different architectures have been developed [15,2,5] and analyzed [9,4,16]. We follow the row-column approach for parallel 3D FFTs here using a 2D decomposition. In this method 1D FFT algorithms are applied separately in each dimension. Between those, the method performs all-to-all personalized communication (index operation) within independent groups of processors, see Sec. 4.

An extensive amount of effort has been put into optimizing collective all-to-all communication and other particular types of collectives by developing various algorithms for different specific topologies and communication models of the network [11,6,17]. We focus here on Bruck's algorithm [3] for all-to-all personalized communication. The more general communication problem — where every node sends messages of different size, including zero, to every node and receives the corresponding messages — is hard to solve, a fact which is apparent from its underlying communication graph. In general heuristic approaches are applied [8,10,1].

Recently computer architectures with processors sharing nodes of a network have become prevalent, and work has been done on the extension of the collective communication algorithms to these systems. The hierarchy of fast connection between processors sharing nodes, and slow connection between the nodes using the network has been considered, e. g., for the all-to-all communication algorithm [14] and for the broadcast and reduce [18].

In this contribution, we consider a communication model in which the nodes are assumed to be fully connected, meaning all the nodes communicate directly and with the same cost. The nodes have k ports, which is the number of partners a node can communicate with independently at the same time. Furthermore, we assume linearity for the communication costs between the nodes. Thus the communication time t is determined by the latency t_{lat} plus the packet size s divided by the bandwidth b, $t = t_{lat} + s/b$. Bruck's algorithm [3] for all-to-all communication provides a minimum of communication cost as a tradeoff between latency-optimal and bandwidth-optimal communication for this model. As part of our communication model, we neglect any communication costs within a single node.

If, for multiprocessor nodes, every single collective operation incorporates a certain number of nodes and all processors of those nodes, the extension of a single processor node communication algorithm is straightforward. For such all-to-all communication on the source nodes, the messages from the individual processors are collected on single master processors. Then, between the master processors, the messages are communicated with the algorithm for single processor nodes. On the target nodes the messages are distributed from the master processors to the individual processors. However, here messages are exchanged for the FFTs within independent groups of processors. It is called multiple collective communication here. Those all-to-all collectives in general share nodes. Thus, between the nodes a special message exchange problem is present. An efficient way of handling this communication by an adaptation of Bruck's algorithm will be discussed here.

In the following we will present Bruck's algorithm and its adaptation, show how to integrate it into FFTs, and provide an evaluation of the performance.

2 Bruck's Algorithm

In this contribution, we focus on Bruck's algorithm [3] for all-to-all communication, since it is optimal with respect to communication steps, for the single communication with equal message size. The algorithm is illustrated in Fig. 1, where for the nodes A-D the messages sent are ordered in columns. The data are arranged such that every line displayed needs to be shifted a certain number of times in order to transfer the data from the source node to the target node. Starting from zero, the line number is the number of shifts for the data to be communicated to the right. The algorithm transfers the data not necessarily directly but in steps according to the following scheme. For every line, the number of shifts is described on a radix of r_0. Every digit of the number represents a single step, where different values of the digit are different substeps. Thus, in every step x, $1 \leq x$, substeps z, $1 \leq z < r_0$, with step size $z \cdot r_0^{x-1}$ are performed, lines are jointly communicated. In Fig. 1, $r_0 = 2$ is chosen where every step has only one substep. A first step with one shift and a second step with two shifts are performed. Multiple substeps within a step are performed by the k ports in parallel. Complexity measures — the number of communications C_1 and the amount of data C_2 — are estimated to be

$$C_1 \leq (r_0 - 1) \log_{r_0} n \quad \text{and} \tag{1a}$$

$$C_2 \leq b(r_0 - 1) \frac{n}{r_0} \log_{r_0} n \ , \tag{1b}$$

respectively [3], where n is the number of nodes and b the size of the data. The overall communication time then becomes $t = t_{lat} C_1 + C_2/b$. Thus, for pure latency-dominated communication, $r_0 = 2$ is chosen, and for bandwidth-dominated communication $r_0 = n$. For the special case $r_0 = 2$ the number of communications becomes

$$C_1 = \log_2 n \tag{2}$$

[3] (for the special case $r_0 = n$ see [3]). For brevity, the arrangement of the data on the nodes before and after the communication — considered as separate steps of so called local rotation in the original description [3] — is not discussed here.

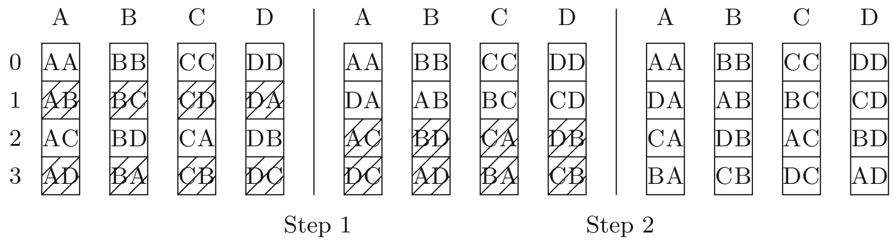


Fig. 1. All-to-all communication using Bruck's algorithm; Nodes A-D contain columns of messages, which are labeled with two letters: the first indicates the sending node and the second one the receiving node. The line number on the left is the number of shifts to the right. Shaded lines participate in the following step.

For multiple collectives the participating processes are mapped on the nodes such that as many nodes as possible are used in only one particular collective; furthermore, no more than two collectives should share one node (plus collectives involving only one node, which according to our communication model have zero communication cost). As an example, the top of Fig. 2 illustrates nodes in different groups communicating, where adjacent empty circles are nodes involved in one particular collective, and filled circles are nodes involved in two collectives, one on the left and one on the right. We assume the case $r_0 = 2$ of Bruck's algorithm. The individual (local) application of Bruck's algorithm to the collectives with three or four nodes requires two steps with one and two shifts. Taking into account that nodes are shared between two collectives, the total number of steps required is four. An alternative is the application of a global all-to-all operation with message sizes of zero between the different groups. The arrangement of data for Bruck's algorithm is also shown in Fig. 2. The number of lines in which the data are communicated is smaller than the number of nodes. The global application of the algorithm for the 21 nodes requires 5 steps with 1, 2, 4, 8 and 16 shifts. Due to the communication pattern, not all of the steps or substeps of Bruck's algorithm are necessary. Only the steps with 1, 2, 4 and 16 shifts are required (8 can be omitted). This is the same number of steps as for the local application of the algorithm.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
	○	○	○	●	○	○	●	○	○	●	○	○	●	○	○	●	○	○	●	○	○
0	AA	BB	CC	DD	EE	FF	GG	HH	I I	J J	KK	L L	MM	NN	OO	PP	QQ	RR	S S	TT	UU
1	AB	BC	CD	DE	EF	FG	GH	H I	I J	JK	KL	LM	MN	NO	OP	PQ	QR	RS	S T	TU	
2	AC	BD		DF	EG		G I	H J		J L	KM		MO	NP		PR	QS		S U		
3	AD			DG			G J			JM			MP			P S					
18				DA			GD			J G			M J			PM			S P		
19			CA	DB		FD	GE		I G	J H		L J	MK		OM	PN		RP	SQ		US
20		BA	CB	DC	ED	FE	GF	HG	I H	J I	K J	LK	ML	NM	ON	PO	QP	RQ	SR	TS	UT

Fig. 2. Multiple all-to-all communication; The top letters and circles indicate the nodes A-U in groups of collectives. Adjacent empty circles are part of one collective, filled circles are shared between neighbors. The bottom shows the setup for Bruck's algorithm. In the columns the first and second letter indicate the sending and receiving nodes, respectively. The number of shifts to the right is indicated on the left.

3 Sparse Bruck Algorithm

The concept of omitting particular communication steps in order to realize only the numbers of shifts necessary can be generalized as follows. At the different steps the same number of substeps are still made, but the radix r_x might be larger than r_0 (the number of substeps plus one). In those steps, the substeps do not cover all numbers of shifts required for a communication between all nodes but certain numbers of shifts are omitted. Thus the number of shifts at substep z of step x is $z \cdot \prod_{1 \leq i \leq x} r_i$ shifts to the right. We call this procedure the sparse version of Bruck's algorithm.

It should be noted that as soon as messages with different sizes, including zero, are sent, the order of the steps, which is essentially arbitrary, influences the communication costs. We do not attempt to find the best solution for the general communication problem using the sparse Bruck algorithm, but we exploit the characteristic communication pattern present in a heuristic approach.

Pairs of nodes that do exchange data are indicated in the so-called communication matrix. Figure 3 shows the matrix for the example of all-to-all collectives executed simultaneously. It has a banded structure. While in the original algorithm the data of the n nodes are shifted for all steps within the interval from zero to the matrix size minus one $[0..n-1]$ (in the example $[0..20]$), the necessary numbers of shifts for our problem are two sub-intervals visible in the matrix. One interval starts from zero, representing the diagonal, to the number of upper subdiagonals u $[0..u]$ ($[0..3]$) and the other interval from size of the matrix minus the number of lower side diagonals l to the size of the matrix minus one $[n-l..n-1]$ ($[18..20]$). The two subintervals also allow entries in the lower left and upper right corner of the communication matrix. In our case, the first node would also communicate with the last three ones. Rotations for two intervals are realized by a larger radix in the last step of the communication without increasing the number of substeps r_0-1. The step sizes of the substeps is such that they match the subintervals. One part of the substeps points to the right interval and the other one to the left interval. Thus for the example (Fig. 2) the communication can be realized with 1, 2 and 18 shifts.

In the communication procedure several messages of single collectives are delivered by incorporating nodes for message forwarding which are not involved in the particular collective, for nodes used for forwarding only see also [13].

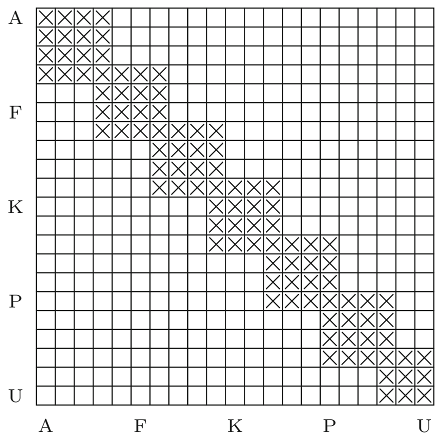


Fig. 3. Communication matrix of multiple collectives, pairs of nodes A-U which exchange data are ticked

For the complexity measure of the sparse Bruck algorithm, the estimate Eq. (1) still holds for all r_0, n, b and becomes more conservative. Considering a banded communication matrix, a more precise bound is estimated as follows. The original estimate is valid for one interval of shifts; the absolute distance does not play any role. For the adaptation to two intervals and r_0 even — r_0 odd is not discussed for brevity here — we assume that in the last step $r_0/2$ substeps are made to the left interval and $r_0/2-1$ to the right one. If the intervals $[0..u]$ and $[n-l..n-1]$ had equal size, the original measure would apply, for the number of nodes being twice the interval size. Considering the larger interval size, the number of communications and the amount of data can be estimated to be

$$C_1 \le (r_0 - 1)\log_{r_0}[2\max(l, u+1)] \quad \text{and} \tag{3a}$$

$$C_2 \le b(r_0 - 1)\frac{2\max(l, u+1)}{r_0}\log_{r_0}[2\max(l, u+1)] \ , \tag{3b}$$

respectively. For $r_0 = 2$ the number of substeps becomes

$$C_1 = \log_2[2\max(l, u+1)] \ . \tag{4}$$

The choice of $r_0 = 2$ in the example (Fig. 2) leads to a communication within three steps with one substep each.

4 FFTs Using 2D Decomposition

In the so-called split and transposition approach the 1D FFT is shared between processors such that between the computations, communication phases for data transposition are introduced. For optimal communication on the specific fully connected network (bandwidth, latency, see Sec. 1) the number of communication phases or r_0 of Bruck's algorithm can be adjusted [9,3]. If the resolution permits, one communication phase is preferred, otherwise multiple collective communications are present.

For two communication phases, this approach is equivalent to a 3D FFT which uses the row-column algorithm with a 2D decomposition. We focus on such 3D FFTs. Figure 4 illustrates the data, a (N_1, N_2, N_3) array which is split to $P_1 \times P_2$ processors. We restrict ourselves to the case $N = N_1 = N_2 = N_3$ and $P = P_1 = P_2$. The FFT consists of five phases, three of them are one-dimensional computations performed sequentially in the coordinate directions x, y and z labeled a, b and c for any of the dimensions, respectively, while between them the data is rearranged from a to b and also from b to c in two communication phases. A communication back to arrangement a is not considered as part of the FFT here.

For the two communication phases, P groups of collectives are applied independently in the different planes. Considering the $y - z$ plane of arrangement a as nomenclature, these are rows and columns of data exchange.

Since the processors are involved in two different collectives, the assignment of the data to the nodes is not straightforward and is done in the following

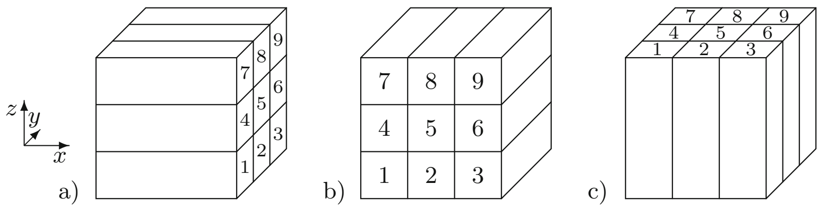


Fig. 4. Data distribution to the processors (1-9)

heuristic way. Rectangles with the dimensions $p = a_x \times a_z$ — representing the processors of the nodes — are placed into the plane. If the rectangles are exclusively filling the plane, all collectives are still independent. If the dimensions of the rectangle are not a fraction of the size of the cube surface, the situation is the following (see Fig. 5): the rectangles are placed in a mesh pattern from the left to the right, where a_z is the greatest common divisor (gcd) of p and P, $a_z = gcd(p, P)$ ($a_x = p/a_z$). In the example these are rectangles of size 4×3 (Fig. 5a). The communication is decoupled for the different columns of rectangles and the band on the right (Fig. 5a), i. e., the original Bruck algorithm can be applied independently for groups of processors. A further decoupling applies for packets of rows of rectangles where the size of the rows is $p/gcd(a_x, P)$. In Fig. 5a these are rows of width 12, 12 and 3 processors. Alternatively, a_z can be chosen smaller $a_z = gcd(p, gcd(P, p/gcd(p, P)))$, thus the vertical decoupling is provided for a size of rows $p/gcd(p, P)$. The differences in the results are rather small, so we do not consider this alternative.

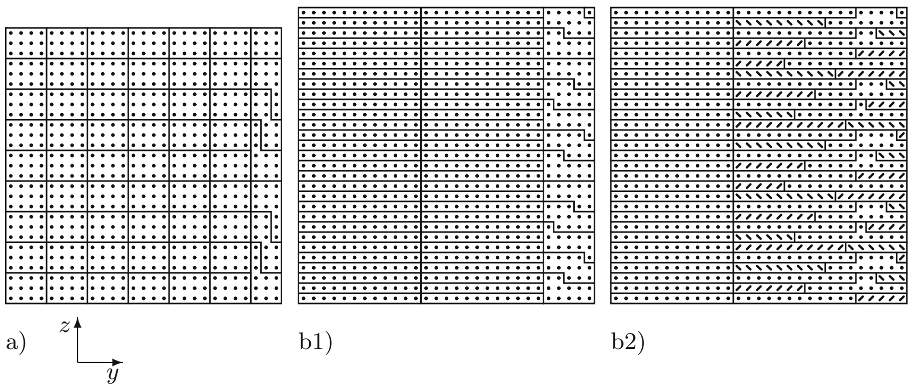


Fig. 5. Processor arrangement a) $P = 27$ b1) $P = 29$ b2) $P = 29$, processors are represented by bullets for nodes within a single frame or by identical stripes for nodes distributed to two frames

In order to ensure that nodes do not belong to more than two rows of rectangles (four rows in Fig. 5b1), the width of the band on the right is chosen larger or equal to the rectangle size minus one (Fig. 5b2). This is advantageous if Bruck's algorithm is applied locally to single rows of rectangles: not more than two collectives share one node. For the a-to-b rearrangement, the nodes can then be ordered such that the communication matrix is of the type discussed in the previous section and the sparse Bruck algorithm can be applied.

Figure 6 shows the cost for the FFT in terms of the number of steps required for the communication a to b plus b to c (Fig. 4) for $r_0 = 2$, $k = 1$ and $p = 12$. The local application of Bruck's algorithm to the all-to-all collectives, the application of the original algorithm to the complete problem, and the sparse version for banded communication matrices have been compared. If nodes are shared by multiple collectives, their steps for the local application are summed up.

The global application of Bruck's algorithm without counting zero-size messages we refer to as global Bruck, and with counting those messages as full Bruck. It is apparent that for groups comprising complete nodes only, there is no difference between the algorithms. Which algorithm is otherwise the most advantageous one depends on the configuration. In many cases the sparse version of Bruck's algorithm is the most efficient one.

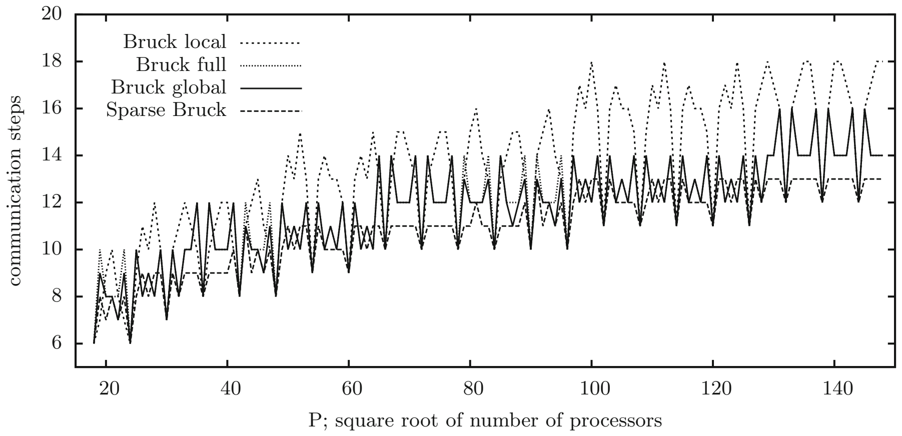


Fig. 6. Number of communication steps for the different algorithms, $r_0 = 2$, $k = 1$ and $p = 12$

5 Experimental Results

The various versions of the algorithm discussed have been tested on a 12 processor (core) per node Cray XT5 system. The network of the machine is a 3D torus. For the typical operation of the machine, programs use an unstructured and not necessarily coherent partition of the torus. Thus the algorithm which does not consider the specific network topology comes into play. We consider an average of 20 samples using a random node placement.

For the 1D FFTs the FFTW3 implementation [7] is used. The communication between the nodes is implemented using point-to-point communication MPI calls. Within the node, data is exchanged by a shared memory segment.

Table 1 indicates the total number of steps and substeps, and the actual time required, for one complex to complex FFT of single precision where $N = P$, furthermore $r_0 = 6$, (for step count Bruck local $k = 5$ is assumed) and $p = 12$. The sparse version performs generally more efficiently than the local and global application of Bruck's algorithm. There are differences between the real network and the modeling assumptions: the bandwidth of the network and node internal communication become apparent. The computational effort for the FFT is always less than 1%. Processor arrangements less structured than the suggested ones increase the communication costs significantly.

Table 1. Communication costs for the different algorithms, $r_0 = 6$, $(k = 5)$ and $p = 12$

P; square root of number of processors	number of steps, substeps; execution time $\cdot 10^4 s$		
	Bruck local	Bruck global	sparse Bruck
19	6, 24; 1.9	4, 16; 1.8	3, 15; 1.9
23	6, 20; 1.8	4, 16; 2.1	3, 13; 1.8
25	6, 22; 1.9	4, 18; 2.1	3, 14; 1.9
39	8, 26; 2.7	5, 18; 2.7	4, 15; 2.4
40	8, 28; 2.9	5, 17; 2.7	4, 17; 2.4
44	8, 26; 2.9	5, 17; 2.8	4, 16; 2.5
45	8, 28; 3.0	5, 18; 2.9	4, 16; 2.6
51	8, 28; 3.2	5, 18; 3.2	4, 17; 2.8
52	8, 28; 3.1	5, 18; 3.1	4, 17; 2.7

6 Conclusions

We have investigated 3D FFTs based on the row-column approach and 2D data decomposition. These FFTs perform simultaneously multiple all-to-all collective communications which may share nodes. In this case, a communication pattern with a banded communication matrix is present. A sparse version of Bruck's algorithm has been introduced, which exploits this banded structure. On a fully connected network (Sec. 1) the original algorithm provides the optimal number of steps for all-to-all communication, the sparse version reduces the number of steps for our specific communication pattern.

The assignment of the processors to the decomposed data for the local and global application of Bruck's algorithm and for the sparse version has been suggested. The sparse version requires fewer communication steps compared to the local and global application of the original algorithm for many configurations. Experimental results confirm the advantage of the processor assignment and the sparse version of Bruck's algorithm.

The modification to Bruck's all-to-all personalized communication algorithm can be analogously applied to Bruck's all-to-all broadcast. An application is matrix multiplication, where all-to-all broadcasts are performed independently for all rows and all columns.

Acknowledgments. The authors wish to thank T. Hoefler (Univ. of Illinois UC), W. Sawyer and T.W. Robinson for helpful discussions.

References

1. Adler, M., Byers, J.W., Karp, R.M.: Scheduling Parallel Communication: The h-Relation Problem. In: Wiedermann, J., Hájek, P. (eds.) MFCS 1995. LNCS, vol. 969, pp. 1–20. Springer, Heidelberg (1995)
2. Brass, A., Pawley, G.S.: Two and three dimensional FFTs on highly parallel computers. Parallel Comput. 3, 167–184 (1986)
3. Bruck, J., Ho, C.T., Kipnis, S., Upfal, E., Weathersby, D.: Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEE T. Parall. Distr. 8(11), 1143–1156 (1997)

4. Chan, A., Balaji, P., Gropp, W., Thakur, R.: Communication Analysis of Parallel 3D FFT for Flat Cartesian Meshes on Large Blue Gene Systems. In: Sadayappan, P., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2008. LNCS, vol. 5374, pp. 350–364. Springer, Heidelberg (2008)
5. Fang, B., Deng, Y., Martyna, G.: Performance of the 3D FFT on the 6D network torus QCDOC parallel supercomputer. Comput. Phys. Commun. 176, 531–538 (2007)
6. Fraigniaud, P., Lazard, E.: Methods and problems of communication in usual networks. Discrete Appl. Math. 53, 79–133 (1994)
7. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. P IEEE 93(2), 216–231 (2005)
8. Goldman, A., Peters, J.G., Trystram, D.: Exchanging messages of different size. J. Parallel Distr. Com. 66, 1–18 (2006)
9. Gupta, A., Kumar, V.: The scalability of FFT on parallel computers. IEEE T. Parall. Distr. 4(8), 922–932 (1993)
10. Helman, D.R., Bader, D.A., JáJá, J.: Parallel algorithms for personalized communication and sorting with an experimental study (extended abstract). In: SPAA 1996: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 211–222. ACM, New York (1996)
11. Johnsson, S.L., Ho, C.T.: Optimum broadcasting and personalized communication in hypercubes. IEEE T. Comput. 38(9), 1249–1268 (1989)
12. van Loan, C.: Computational Frameworks for the Fast Fourier Transfrom. SIAM, Philadelphia (1992)
13. Sanders, P., Solis-Oba, R.: How helpers hasten h-relations. J. Algorithm. 41, 86–98 (2001)
14. Sanders, P., Träff, J.L.: The Hierarchical Factor Algorithm for All-to-All Communication. In: Monien, B., Feldmann, R. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 799–803. Springer, Heidelberg (2002)
15. Swarztrauber, P.N.: Multiprocessor FFTs. Parallel Comput. 5, 197–210 (1987)
16. Takahashi, D.: An Implementation of Parallel 3-D FFT with 2-D Decomposition on a Massively Parallel Cluster of Multi-core Processors. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 606–614. Springer, Heidelberg (2010)
17. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. Int. J. High Perform. C. 19(1), 49–66 (2005)
18. Tipparaju, V., Nieplocha, J., Panda, D.: Fast collective operations using shared and remote memory access protocols on clusters. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003), Nice, France (April 2003)

Performance Analysis of Parallel Alternating Directions Algorithm for Time Dependent Problems

Ivan Lirkov[1], Marcin Paprzycki[2], and Maria Ganzha[2]

[1] Institute of Information and Communication Technologies, Bulgarian Academy of Sciences, Acad. G. Bonchev, Bl. 25A, 1113 Sofia, Bulgaria
ivan@parallel.bas.bg
http://parallel.bas.bg/~ivan/

[2] Systems Research Institute, Polish Academy of Sciences, ul. Newelska 6, 01-447 Warsaw, Poland
{paprzyck,maria.ganzha}@ibspan.waw.pl
http://www.ibspan.waw.pl/~paprzyck/
http://inf.ug.edu.pl/~mganzha/

Abstract. We consider the time dependent Stokes equation on a finite time interval and on a uniform rectangular mesh, written in terms of velocity and pressure. In a parallel algorithm, based on a new direction splitting approach, the pressure equation is derived from a perturbed form of the continuity equation, in which the incompressibility constraint is penalized in a negative norm induced by the direction splitting. The scheme used in the algorithm is composed of: pressure prediction, velocity update, penalty step, and pressure correction. In order to achieve good parallel performance, the solution of the Poison problem for the pressure correction is replaced by solving a sequence of one-dimensional second order elliptic boundary value problems in each spatial direction. The parallel code was developed using MPI and tested on modern computer systems. The performed numerical tests illustrate the parallel efficiency, and the scalability, of the direction-splitting based algorithm.

1 Introduction

The objective of this paper is to analyze the parallel performance of a novel fractional time stepping technique, based on a direction splitting strategy, developed to solve the incompressible Navier-Stokes equations.

Projection schemes were introduced in [2,9] and they have been used in Computational Fluid Dynamics (CFD) since. During these years, such techniques went through some evolution, but the main paradigm, consisting of decomposing vector fields into a divergence-free part and a gradient, has been preserved; see [4] for a review. In terms of computational efficiency, projection algorithms are far superior to the methods that solve the coupled velocity-pressure system, making them the most popular techniques for solving unsteady Navier-Stokes equations.

The alternating directions algorithm proposed in [3] reduces the computational complexity of the enforcement of the incompressibility constraint. The key idea consists of abandoning the projection paradigm in which vector fields are decomposed into a divergence-free part plus a gradient part. Departure from the projection paradigm has been proved to be very efficient for solving variable density flows [5,6]. In the new method, the pressure equation is derived from a perturbed form of the continuity equation, in which the incompressibility constraint is penalized in a negative norm induced by the direction splitting. The standard Poisson problem for the pressure correction is replaced by the series of one-dimensional second-order boundary value problems. This technique is proved to be stable and convergent (see [3]). Furthermore, a very brief initial assessment, found in [3], indicates that the new approach should be efficiently parallelizable. The aim of this paper is to experimentally investigate this claim on three distinct parallel systems, for two dimensional problems.

2 Stokes Equation

We consider the time-dependent Navier-Stokes equations on a finite time interval $[0, T]$, and in a rectangular domain Ω. Since the nonlinear term in the Navier-Stokes equations does not interfere with the incompressibility constraint, we henceforth mainly focus our attention on the time-dependent Stokes equations written in terms of velocity with components (u, v) and pressure p:

$$\begin{cases} u_t - \nu\left(u_{xx} + u_{yy}\right) + p_x = f & \\ v_t - \nu\left(v_{xx} + v_{yy}\right) + p_y = g & \text{in } \Omega \times (0, T) \\ u_x + v_y = 0 & \\ u|_{\partial\Omega} = v|_{\partial\Omega} = 0, \quad \partial_n p|_{\partial\Omega} = 0 & \text{in } (0, T) \\ u|_{t=0} = u_0, \quad v|_{t=0} = v_0, \quad p|_{t=0} = p_0 & \text{in } \Omega \end{cases}, \tag{1}$$

where a smooth source term has components (f, g), ν is the kinematic viscosity, and (u_0, v_0) is a solenoidal initial velocity field with a zero normal trace. The time interval $[0, T]$ was discretized on a uniform mesh and τ was the time step.

3 Parallel Alternating Directions Algorithm

Guermond and Minev introduced (in [3]) a novel fractional time stepping technique for solving the incompressible Navier-Stokes equations, based on a direction splitting strategy. They used a singular perturbation of Stokes equation with a perturbation parameter τ. The standard Poisson problem was replaced by series of one-dimensional second-order boundary value problems.

3.1 Formulation of the Scheme

The scheme used in the algorithm is composed of the following parts: pressure prediction, velocity update, penalty step, and pressure correction. We now describe an algorithm that uses the direction splitting operator

$$A := \left(1 - \frac{\partial^2}{\partial x^2}\right)\left(1 - \frac{\partial^2}{\partial y^2}\right).$$

– *Pressure predictor.*
Denoting p_0 the pressure field at $t = 0$, the algorithm is initialized by setting $p^{-\frac{1}{2}} = p^{-\frac{3}{2}} = p_0$. Then for all $n \geq 0$ a pressure predictor is computed:

$$p^{*,n+\frac{1}{2}} = 2p^{n-\frac{1}{2}} - p^{n-\frac{3}{2}}. \tag{2}$$

– *Velocity update.*
The velocity field is initialized by setting $\mathbf{u}^0 = \begin{pmatrix} u_0 \\ v_0 \end{pmatrix}$, and for all $n \geq 0$ the velocity update is computed by solving the following series of one-dimensional problems

$$\frac{\boldsymbol{\xi}^{n+1} - \mathbf{u}^n}{\tau} - \nu \Delta \mathbf{u}^n + \nabla p^{*,n+\frac{1}{2}} = \mathbf{f}^{n+\frac{1}{2}},\ \boldsymbol{\xi}^{n+1}|_{\partial\Omega} = 0, \tag{3}$$

$$\frac{\boldsymbol{\eta}^{n+1} - \boldsymbol{\xi}^{n+1}}{\tau} - \frac{\nu}{2}\frac{\partial^2(\boldsymbol{\eta}^{n+1} - \mathbf{u}^n)}{\partial x^2} = 0,\ \boldsymbol{\eta}^{n+1}|_{\partial\Omega} = 0, \tag{4}$$

$$\frac{\mathbf{u}^{n+1} - \boldsymbol{\eta}^{n+1}}{\tau} - \frac{\nu}{2}\frac{\partial^2(\mathbf{u}^{n+1} - \mathbf{u}^n)}{\partial y^2} = 0,\ \mathbf{u}^{n+1}|_{\partial\Omega} = 0, \tag{5}$$

where $\mathbf{f}^{n+\frac{1}{2}} = \begin{pmatrix} f|_{t=\left(n+\frac{1}{2}\right)\tau} \\ g|_{t=\left(n+\frac{1}{2}\right)\tau} \end{pmatrix}$.

– Penalty step
The intermediate parameter ϕ is approximated by solving $A\phi = -\frac{1}{\tau}\nabla \cdot \mathbf{u}^{n+1}$. Owing to the definition of the direction splitting operator A, this is done by solving the following series of one-dimensional problems:

$$\begin{array}{ll} \psi - \psi_{xx} = -\frac{1}{\tau}\nabla \cdot \mathbf{u}^{n+1}, & \psi_x|_{\partial\Omega} = 0, \\ \phi - \phi_{yy} = \psi, & \phi_y|_{\partial\Omega} = 0, \end{array} \tag{6}$$

– *Pressure update*
The last sub-step of the algorithm consists of updating the pressure:

$$p^{n+\frac{1}{2}} = p^{n-\frac{1}{2}} + \phi - \chi\nu\nabla \cdot \frac{\mathbf{u}^{n+1} + \mathbf{u}^n}{2} \tag{7}$$

The algorithm is in a standard incremental form when the parameter $\chi = 0$; while the algorithm is in a rotational incremental form when $\chi \in (0, \frac{1}{2}]$.

3.2 Parallel Algorithm

We use a rectangular uniform mesh combined with a central difference scheme for the second derivatives for solving equations (4–5), and (6). Thus the algorithm requires only the solution of tridiagonal linear systems. The parallelization is based on a decomposition of the domain into rectangular sub-domains. Let us associate with each such sub-domain a set of coordinates (i_x, i_y), and identify it with a given processor. The linear systems, generated by one-dimensional problems that need to be solved in each direction, are divided into systems for sets of unknowns corresponding to the internal nodes for each block that can be solved independently by a direct method. The corresponding Schur complement for the interface unknowns between the blocks that have an equal coordinate i_x or i_y is also tridiagonal and can be inverted directly. The overall algorithm requires only exchange of the interface data, which allows for a very efficient parallelization with an efficiency comparable to that of explicit schemes.

4 Experimental Results

The problem (1) is solved in $\Omega = (0,1)^2$, for $t \in [0,2]$ with Dirichlet boundary conditions. The discretization in time is done with time step 10^{-2}, the parameter $\chi = \frac{1}{2}$, the kinematic viscosity $\nu = 10^{-3}$. The discretization in space uses mesh sizes $h_x = \frac{1}{n_x - 1}$ and $h_y = \frac{1}{n_y - 1}$. Thus, (4) leads to linear systems of size n_x and (5) leads to linear systems of size n_y. The total number of unknowns in the discrete problem is $600\, n_x\, n_y$.

To solve the problem, a portable parallel code was designed and implemented in C, while the parallelization has been facilitated using the MPI library [8,10]. We use the LAPACK subroutines DPTTRF and DPTTS2 (see [1]) for solving tridiagonal systems in equations (4), (5), and (6) for the unknowns corresponding to the internal nodes of each sub-domain. The same subroutines are used to solve the tridiagonal systems with the Schur complement.

The parallel code has been tested on three computer systems: Galera, located in the Centrum Informatyczne TASK; Sooner, located in the Oklahoma Supercomputing Center (OSCER); and the IBM Blue Gene/P machine at the Bulgarian Supercomputing Center. In our experiments, times have been collected using the MPI provided timer and we report the best results from multiple runs. We report the elapsed time T_c in seconds using c cores, the parallel speed-up $S_c = T_1/T_c$, and the parallel efficiency $E_c = S_c/c$.

Table 1 shows the results collected on the Galera. It is a Linux cluster with 336 nodes, and two Intel Xeon quad core processors per node. Each processor runs at 2.33 GHz. Processors within each node share 8, 16, or 32 GB of memory, while nodes are interconnected with a high-speed InfiniBand network (see also `http://www.task.gda.pl/kdm/sprzet/Galera`). Here, we used an Intel C compiler, and compiled the code with the option "-O3".

Table 2 shows the results collected on the Sooner, a quad core Linux cluster (see `http://www.oscer.ou.edu/resources.php`). It has 486 Dell PowerEdge 1950 III nodes, and two quad core processors (Dell Pentium4 Xeon E5405; running at 2 GHz, sharing 16 GB of memory) per node. Nodes are interconnected with a high-speed InfiniBand network. We have used an Intel C compiler, and compiled the code with the following options: "-O3 -march=core2 -mtune=core2".

The results in each column of Tables 1 and 2 are obtained for an equal number of unknowns per core. For large discrete problems the execution time is much

Table 1. Execution time on Galera

c	$n_x n_y/c$										
	10^4	$2\ 10^4$	$4\ 10^4$	$8\ 10^4$	$1.6\ 10^5$	$3.2\ 10^5$	$6.4\ 10^5$	$1.28\ 10^6$	$2.56\ 10^6$	$5.12\ 10^6$	$1.024\ 10^7$
1	0.38	0.67	1.55	3.59	9.16	24.06	53.32	109.73	228.37	508.85	1204.13
2	0.38	0.67	1.51	3.97	10.73	26.67	57.74	120.03	245.60	572.72	1399.29
4	0.35	0.73	1.77	5.17	14.22	34.93	77.91	160.31	331.14	772.65	2171.64
8	0.36	0.84	3.29	9.56	24.05	54.78	113.65	232.26	517.02	1416.69	3606.02
16	0.40	0.93	3.44	9.61	24.18	54.13	114.15	237.16	526.47	1408.33	3680.95
32	0.48	1.03	3.48	9.90	24.48	55.41	114.57	233.91	530.87	1452.24	3664.30
64	0.74	1.22	3.91	10.19	25.26	55.88	117.21	243.30	550.54	1454.70	3826.07

Table 2. Execution time on Sooner

c	$n_x n_y/c$										
	10^4	$2\ 10^4$	$4\ 10^4$	$8\ 10^4$	$1.6\ 10^5$	$3.2\ 10^5$	$6.4\ 10^5$	$1.28\ 10^6$	$2.56\ 10^6$	$5.12\ 10^6$	$1.024\ 10^7$
1	0.58	1.38	2.78	5.72	12.52	27.56	59.60	120.33	248.72	511.40	1196.21
2	0.59	1.37	2.76	5.83	12.55	28.73	61.20	126.34	255.50	543.37	1268.61
4	0.58	1.38	2.77	5.88	13.57	32.42	72.43	147.28	301.04	628.10	1636.03
8	0.61	1.42	3.32	9.02	23.02	53.37	109.13	220.54	455.72	1226.75	3308.30
16	0.63	1.44	3.35	9.02	23.01	52.26	109.56	219.52	456.44	1213.96	3352.22
32	0.67	1.51	3.45	9.21	23.21	54.49	110.22	222.13	457.92	1235.84	3454.01
64	0.73	1.57	3.61	9.34	23.55	53.28	111.74	222.56	463.14	1256.47	3499.66
128	0.85	1.85	3.96	10.21	24.52	56.61	114.53	235.95	471.35	1283.83	3507.78
256	0.98	1.88	4.13	10.01	25.07	55.00	116.02	227.57	476.61	1288.46	3580.68
512	1.36	2.39	5.15	12.99	27.61	63.71	126.36	250.11			

larger on two processors (8 cores) than on one processor, but on more processors the time is approximately constant. The obtained execution times confirm that the communication time between processors is larger than the communication time between cores of one processor. Furthermore, the execution time for solving one and the same discrete problem decrease when increasing the number of cores, which shows that the communication in our parallel algorithm is mainly local.

The somehow slower performance using 8 cores is clearly visible. The same effect was observed during our previous work, see [7]. There are some factors which could play role for the slower performance using all processors of a single node. Generally, they are a consequence of limitations of memory subsystems and their hierarchical organization in modern computers. One such factor might be the limited bandwidth of the main memory bus. This causes the processors literally to "starve" for data, thus decreasing the overall performance. Since the L2 cache memory is shared among each pair of cores within the processors, this boost the performance of programs utilizing only a single core within such pair (this core can monopolize use of the L2 cache). Conversely, this leads to a somehow decreased speedups when all cores are used. For the memory intensive programs, these factors can play a crucial role for the performance.

Comparing the performance of the Galera and the Sooner we can observe that times on Sooner are shorter across the board. This is somewhat surprising, as

Table 3. Execution time on IBM Blue Gene/P

c	$n_x n_y / c$										
	10^4	$2\ 10^4$	$4\ 10^4$	$8\ 10^4$	$1.6\ 10^5$	$3.2\ 10^5$	$6.4\ 10^5$	$1.28\ 10^6$	$2.56\ 10^6$	$5.12\ 10^6$	$1.024\ 10^7$
1	5.79	12.33	24.51	49.02	103.58	210.81	431.43	877.01	1764.68	3586.12	7416.03
2	5.96	11.84	24.93	49.89	105.23	214.55	437.71	880.14	1793.94	3604.56	7526.88
4	6.16	13.01	25.68	51.34	108.00	219.95	450.15	913.40	1839.21	3742.63	7706.22
8	6.34	12.48	26.21	52.59	109.85	223.77	455.95	917.29	1865.41	3764.16	7813.90
16	6.61	13.80	27.31	54.38	113.83	230.67	471.25	959.19	1930.85	3908.36	8049.69
32	6.71	13.24	27.45	54.92	114.41	232.50	473.96	952.89	1931.31	3882.43	8059.08
64	6.84	14.19	27.71	55.15	115.56	233.56	476.84	964.14	1935.43	3925.14	8070.35
128	7.04	13.72	28.15	56.20	116.68	236.12	478.39	962.12	1944.61	3915.91	8117.98
256	7.17	14.69	28.34	56.44	117.87	237.57	482.76	978.74	1959.50	3980.72	8183.79
512	7.55	14.59	29.10	58.08	119.14	241.37	486.01	972.81	1971.07	3958.97	8205.10
1024	7.91	15.70	29.78	58.66	120.68	242.69	488.99	987.21	1980.31	4003.01	8216.88
2048	8.81	16.71	31.35	62.83	124.67	251.24	501.47	1027.31	2028.63	4200.09	
4096	9.86	18.31	33.81	65.22	130.91	268.78	559.89	1163.18	2443.63		

Galera is much newer and has more powerful processors. We plan to investigate this peculiarity in the near future.

Table 3 presents execution time on the IBM Blue Gene/P machine at the Bulgarian Supercomputing Center (see also `http://www.scc.acad.bg/`). It consists of 2048 compute nodes with quad core PowerPC 450 processors (running at 850 MHz). Each node has 2 GB of RAM. For the point-to-point communications a 3.4 Gb 3D mesh network is used. Reduction operations are performed on a 6.8 Gb tree network. We have used the IBM XL C compiler and compiled the code with the following options: "-O5 -qstrict -qarch=450d -qtune=450".

We observed that using 2 or 4 cores per processor leads to slower execution time, e.g. the execution time for $n_x = n_y = 6400$, $c = 512$ is 58.08 seconds using 512 nodes, 58.83 seconds using 256 nodes, and 60.34 seconds using 128 nodes. This fact shows that using the MPI communication functions, the communication between processors is faster than the communication between cores of one processor. In order to get better parallel performance we plan to develop a mixed MPI/OpenMP code and to use the nodes of the Blue Gene supercomputer in the SMP mode with 4 OpenMP processes per node. This code will also allow us to run efficiently on the upcoming machines with 16-core AMD processors (and future computers with ever increasing number of cores per processor). Note that, for the time being, in our work we omit all issues concerning GPU-based parallelization.

To round up the performance analysis, the speed-up obtained on Galera is reported in Table 4 and the parallel efficiency is shown in Table 5, while the speed-up on Sooner — in Table 6 and the parallel efficiency — in Table 7. Finally, the speed-up on the IBM Blue Gene/P — in Table 8, and the parallel efficiency — in Table 9. In each case, when increasing the number of cores of the two clusters, the parallel efficiency decreases on 8 cores and after that it increases to 100%. Moreover, a super-linear speed-up is observed in multiple cases. The main reasons

Table 4. Speed-up on Galera

n_x	n_y	c					
		2	4	8	16	32	64
800	800	2.00	3.75	5.58	15.49	51.64	71.86
800	1600	1.90	3.14	4.56	11.42	31.49	90.07
1600	1600	1.90	2.93	4.17	9.45	23.08	58.40
1600	3200	2.07	3.17	4.48	9.40	20.78	49.95
3200	3200	2.10	3.64	5.18	10.55	21.73	47.66
3200	6400	2.05	3.72	5.56	12.12	25.09	51.44
6400	6400	1.90	3.84	5.88	15.82	35.61	71.06
6400	12800	1.60	2.49	3.76	9.63	25.55	55.74
12800	12800	2.12	2.90	3.98	10.08	25.55	67.39

Table 5. Parallel efficiency on Galera

n_x	n_y	c					
		2	4	8	16	32	64
800	800	1.000	0.938	0.697	0.968	1.614	1.123
800	1600	0.950	0.785	0.570	0.714	0.984	1.407
1600	1600	0.951	0.733	0.521	0.590	0.721	0.913
1600	3200	1.036	0.794	0.560	0.588	0.649	0.780
3200	3200	1.051	0.909	0.648	0.659	0.679	0.745
3200	6400	1.027	0.930	0.695	0.758	0.784	0.804
6400	6400	0.949	0.959	0.735	0.989	1.113	1.110
6400	12800	0.800	0.622	0.470	0.602	0.798	0.871
12800	12800	1.060	0.726	0.498	0.630	0.798	1.053

for this fact can be related to splitting the entire problem into subproblems which helps the memory management. In particular, it allows for better usage of cache memories of individual parallel processors. As expected, the parallel efficiency on the IBM Blue Gene/P improves with the size of the discrete problems. The efficiency on 1024 cores increases from 57% for the smallest problems to 94% for the largest problems.

Execution time on the Blue Gene/P is substantially larger than that on the Sooner and the Galera, but in some cases the parallel efficiency obtained on the supercomputer is better. For example, the execution time on single core on Sooner is seven times faster than on the Blue Gene/P, in comparison with four times faster performance on 256 cores.

The decomposition of the computational domain in sub-domains is important for the parallel performance of the studied algorithm. Table 10 shows the execution time for the problem with $n_x = n_y = 3200$ on 128 cores using different number of sub-domains in each space direction.

Finally, computing time on both parallel systems is shown in Fig. 1 and the obtained speed-up is shown in Fig. 2.

Table 6. Speed-up on Sooner

n_x	n_y	c								
		2	4	8	16	32	64	128	256	512
800	800	2.07	4.39	6.61	17.79	39.57	81.17	110.21	188.19	134.30
800	1600	1.97	3.71	5.23	13.34	34.93	76.64	142.35	228.36	237.10
1600	1600	1.97	3.43	4.66	10.81	27.00	68.89	134.56	254.61	328.64
1600	3200	2.00	3.47	4.69	9.79	22.04	54.74	129.16	272.71	375.11
3200	3200	2.20	3.97	5.42	10.92	21.95	50.79	117.21	289.73	500.82
3200	6400	2.15	4.35	6.00	12.45	24.80	51.31	111.49	273.13	530.73
6400	6400	1.98	4.01	5.35	14.38	29.55	58.93	115.96	261.83	505.35
6400	12800	1.88	3.21	4.30	11.71	31.03	63.84	124.07	258.34	514.72

Table 7. Parallel efficiency on Sooner

n_x	n_y	c								
		2	4	8	16	32	64	128	256	512
800	800	1.037	1.098	0.826	1.112	1.237	1.268	0.861	0.735	0.262
800	1600	0.983	0.928	0.653	0.834	1.091	1.198	1.112	0.892	0.463
1600	1600	0.984	0.858	0.583	0.676	0.844	1.076	1.051	0.995	0.642
1600	3200	1.001	0.868	0.586	0.612	0.689	0.855	1.009	1.065	0.733
3200	3200	1.101	0.993	0.678	0.682	0.686	0.794	0.916	1.132	0.978
3200	6400	1.077	1.088	0.750	0.778	0.775	0.802	0.871	1.067	1.037
6400	6400	0.988	1.003	0.669	0.899	0.924	0.921	0.906	1.023	0.987
6400	12800	0.938	0.802	0.537	0.732	0.970	0.998	0.969	1.009	1.005

Table 8. Speed-up on IBM Blue Gene/P

n_x	n_y	c											
		2	4	8	16	32	64	128	256	512	1024	2048	4096
800	800	2.01	3.99	8.20	15.80	32.59	63.08	114.74	216.58	330.83	585.7	631.5	782.5
800	1600	2.00	3.99	7.98	16.13	31.95	61.81	124.52	226.79	401.70	655.0	944.8	1177.6
1600	1600	2.00	3.92	7.89	15.50	32.13	63.68	128.66	246.03	424.30	745.3	1002.3	1541.7
1600	3200	2.00	3.93	7.87	15.55	31.34	65.02	127.38	244.20	474.86	812.0	1290.0	1902.5
3200	3200	2.06	4.03	8.08	15.74	31.90	64.17	131.95	261.66	508.46	937.1	1411.7	2274.2

Table 9. Parallel efficiency on IBM Blue Gene/P

n_x	n_y	c											
		2	4	8	16	32	64	128	256	512	1024	2048	4096
800	800	1.005	0.999	1.025	0.987	1.018	0.986	0.896	0.846	0.646	0.572	0.308	0.191
800	1600	1.002	0.997	0.998	1.008	0.998	0.966	0.973	0.886	0.785	0.640	0.461	0.287
1600	1600	1.002	0.980	0.986	0.969	1.004	0.995	1.005	0.961	0.829	0.728	0.489	0.376
1600	3200	1.000	0.982	0.983	0.972	0.979	1.016	0.995	0.954	0.927	0.793	0.630	0.464
3200	3200	1.029	1.008	1.011	0.984	0.997	1.003	1.031	1.022	0.993	0.915	0.689	0.555

Table 10. Execution time on 128 cores

machine	sub-domains			
	8×16	4×32	2×64	1×128
Sooner	10.30	13.14	16.80	84.90
IBM Blue Gene/P	56.20	60.17	74.12	170.46

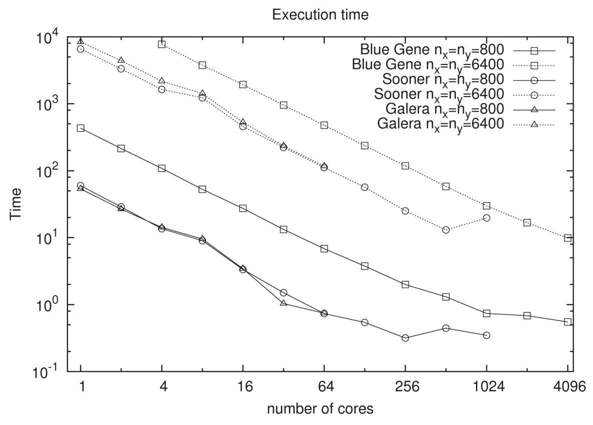


Fig. 1. Execution time for $n_x = n_y = 800, 6400$

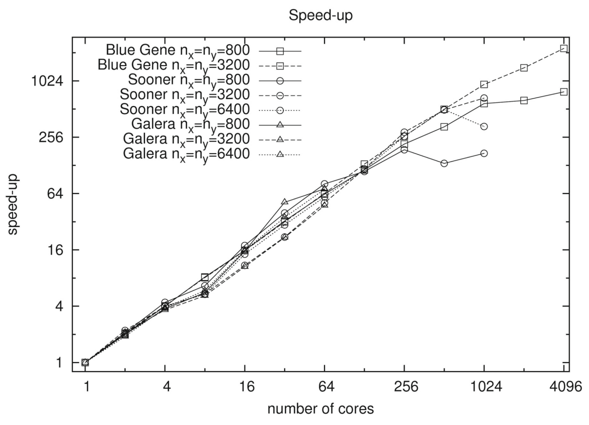


Fig. 2. Speed-up for $n_x = n_y = 800, 3200, 6400$

5 Conclusions and Future Work

We have studied the parallel performance of the recently developed parallel algorithm based on a new direction splitting approach for solving of the time dependent Stokes equation on a finite time interval and on a uniform rectangular mesh. The performance was evaluated on three different parallel architectures. Satisfactory parallel efficiency is obtained on all three parallel systems, on up to 1024 processors. The faster CPUs on Sooner lead to shorter runtime, on the same number of processors.

In order to get better parallel performance using four cores per processor on the IBM Blue Gene/P (and future multi-core computers) we plan to develop mixed MPI/OpenMP code.

Acknowledgments. Computer time grants from the Oklahoma Supercomputing Center (OSCER) and the Bulgarian Supercomputing Center (BGSC) are kindly acknowledged. This research was partially supported by grants DCVP 02/1 and DO02-147 from the Bulgarian NSF. Work presented here is a part of the Poland-Bulgaria collaborative grant "Parallel and distributed computing practices".

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. SIAM, Philadelphia (1999)
2. Chorin, A.J.: Numerical solution of the Navier-Stokes equations. Math. Comp. 22, 745–762 (1968)
3. Guermond, J.L., Minev, P.: A new class of fractional step techniques for the incompressible Navier-Stokes equations using direction splitting. Comptes Rendus Mathematique 348(9-10), 581–585 (2010)
4. Guermond, J.L., Minev, P., Shen, J.: An overview of projection methods for incompressible flows. Comput. Methods Appl. Mech. Engrg. 195, 6011–6054 (2006)
5. Guermond, J.L., Salgado, A.: A fractional step method based on a pressure poisson equation for incompressible flows with variable density. Comptes Rendus Mathematique 346(15-16), 913–918 (2008)
6. Guermond, J.L., Salgado, A.: A splitting method for incompressible flows with variable density based on a pressure Poisson equation. Journal of Computational Physics 228(8), 2834–2846 (2009)
7. Lirkov, I., Vutov, Y., Paprzycki, M., Ganzha, M.: Parallel Performance Evaluation of MIC(0) Preconditioning Algorithm for Voxel μFE Simulation. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2009, Part II. LNCS, vol. 6068, pp. 135–144. Springer, Heidelberg (2010)
8. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. Scientific and engineering computation series. The MIT Press, Cambridge (1997); second printing
9. Temam, R.: Sur l'approximation de la solution des équations de Navier-Stokes par la méthode des pas fractionnaires. Arch. Rat. Mech. Anal. 33, 377–385 (1969)
10. Walker, D., Dongarra, J.: MPI: a standard Message Passing Interface. Supercomputer 63, 56–68 (1996)

A Novel Parallel Algorithm for Gaussian Elimination of Sparse Unsymmetric Matrices

Riccardo Murri

Grid Computing Competence Centre,
Organisch-Chemisch Institut, University of Zürich,
Winterthurerstrasse 190, CH-8006 Zürich,
Switzerland
riccardo.murri@gmail.com

Abstract. We describe a new algorithm for Gaussian Elimination suitable for general (unsymmetric and possibly singular) sparse matrices of any entry type, which has a natural parallel and distributed-memory formulation but degrades gracefully to sequential execution.

We present a sample MPI implementation of a program computing the rank of a sparse integer matrix using the proposed algorithm. Some preliminary performance measurements are presented and discussed, and the performance of the algorithm is compared to corresponding state-of-the-art algorithms for floating-point and integer matrices.

Keywords: Gaussian Elimination, unsymmetric sparse matrices, exact arithmetic.

1 Introduction

This paper presents a new algorithm for Gaussian Elimination, initially developed for computing the rank of some homology matrices with entries in the integer ring $\mathbb{Z}$. It has a natural parallel formulation in the message-passing paradigm and does not make use of collective and blocking communication, but degrades gracefully to sequential execution when run on a single compute node.

Gaussian Elimination algorithms with exact computations have been analyzed in [3]; the authors however concluded that there was —to that date— no practical parallel algorithm for computing the rank of sparse matrices, when exact computations are wanted (e.g., over finite fields or integer arithmetic): well-known Gaussian Elimination algorithms fail to be effective, since, during elimination, entries in pivot position may become zero.

The "Rheinfall" algorithm presented here is based on the observation that a sparse matrix can be put in a "block echelon form" with minimal computational effort. One can then run elimination on each block of rows of the same length independently (i.e., in parallel); the modified rows are then sent to other processors, which keeps the matrix arranged in block echelon form at all times.

The procedure terminates when all blocks have been reduced to a single row, i.e., the matrix has been put in row echelon form.

The "Rheinfall" algorithm is independent of matrix entry type, and can be used for integer and floating-point matrices alike: numerical stability is comparable with Gaussian Elimination with partial pivoting (GEPP). However, some issues related to the computations with inexact arithmetic have been identified in the experimental evaluation, which suggest that "Rheinfall" is not a convenient alternative to existing algorithms for floating-point matrices; see Section 4.2 for details.

Any Gaussian Elimination algorithm can be applied equally well to a matrix column- or row-wise; here we take the row-oriented approach.

2 Description of the "Rheinfall" Algorithm

We shall first discuss the Gaussian Elimination algorithm for reducing a matrix to row echelon form; practical applications like computing matrix rank or linear system solving follow by simple modifications.

Let $A = (a_{ij} | i = 0, \dots, n-1; j = 0, \dots, m-1)$ be a $n \times m$ matrix with entries in a "sufficiently good" ring $\Bbbk$ (a field, a unique factorization domain or a principal ideal domain).

Definition 1. *Given a matrix A, let $z_i := \min\{j | a_{ij} \neq 0\}$ be the column index of the first non-zero entry in the i-th row of A; for a null row, define $z_i := m$. We say that the i-th row of A* starts *at column z_i.*

The matrix A is in block echelon form *iff $z_i \geq z_{i-1}$ for all $i = 1, \dots, n-1$.*

The matrix A is in row echelon form *iff either $z_i > z_{i-1}$ or $z_i = m$.*

Every matrix can be put into *block* echelon form by a permutation of the rows. For reducing the $n \times m$ matrix A to row echelon form, a "master" process starts m Processing Units $P[0], \dots, P[m-1]$, one for each matrix column: $P[c]$ handles rows starting at column c. Each Processing Unit (PU for short) runs the code in procedure ProcessingUnit from Algorithm 1 concurrently with other PUs; upon reaching the done state, it returns its final output to the "master" process, which assembles the global result.

A Processing Unit can send messages to every other PU. Messages can be of two sorts: Row messages and End messages. The payload of a Row message received by $P[c]$ is a matrix row r, extending from column c to $m-1$; End messages carry no payload and just signal the PU to finalize computations and then stop execution. In order to guarantee that the End message is the last message that a running PU can receive, we make two assumptions on the message exchange system: *(1)* that messages sent from one PU to another arrive in the same order they were sent, and *(2)* that it is possible for a PU to wait until all the messages it has sent have been delivered. Both conditions are satisfied by MPI-compliant message passing systems.

The eliminate function at line 16 in Algorithm 1 returns a row $r' = \alpha r + \beta u$ choosing $\alpha, \beta \in \Bbbk$ so that r' has a 0 entry in all columns $j \leq c$. The actual

Algorithm 1. Reduce a matrix to row echelon form by Gaussian Elimination. *Left and top right:* Algorithm run by processing unit $P[c]$. *Bottom right:* Sketch of the "master" procedure. Input to the algorithm is an $n \times m$ matrix A, represented as a list of rows r_i. Row and column indices are 0-based.

```
 1  def ProcessingUnit($c$):
 2    $u \leftarrow$ NIL
 3    $Q \leftarrow$ empty list
 4    $output \leftarrow$ NIL
 5    $state \leftarrow$ RUNNING
 6    while $state$ is RUNNING:
 7      wait for messages to arrive
 8      append Row messages to $Q$
 9      select best pivot row $t$ from $Q$
10      if $u$ is NIL:
11        $u \leftarrow t$
12      else:
13        if $t$ has a better pivot than $u$:
14          exchange $u$ and $t$
15      for each row $r$ in $Q$:
16          $r' \leftarrow$ ELIMINATE($r$,$u$)
17          $c' \leftarrow$ first nonzero col. of $r'$
18          send $r'$ to $P[c']$
19          delete $r$ from $Q$
20      if received message End:
21        wait for all sent messages to arrive
22        $output \leftarrow u$
23        send End to $P[c+1]$
24        $state \leftarrow$ DONE
25    return $output$
```

```
1  def Master($A$):
2    start a PU $P[c]$ for each column $c$ of $A$
3    for $i$ in $\{0, \ldots, n-1\}$:
4      $c \leftarrow$ first nonzero column of $r_i$
5      send $r_i$ to $P[c]$
6    send End message to $P[0]$
7    wait until $P[m-1]$ recv. a End message
8    $result \leftarrow$ collect $output$ from all PUs
9    return $result$
```

definition of ELIMINATE depends on the coefficient ring of A. Note that $u[c] \neq 0$ by construction.

The "master" process runs the Master procedure in Algorithm 1. It is responsible for starting the m independent Processing Units $P[0]$, ..., $P[m-1]$; feeding the matrix data to the processing units at the beginning of the computation; and sending the initial End message to PU $P[0]$. When the End message reaches the last Processing Unit, the computation is done and the master collects the results.

Lines 3–5 in Master are responsible for initially putting the input matrix A into block echelon form; there is no separate reordering step. This is an invariant of the algorithm: by exchanging rows among PUs after every round of elimination is done, the working matrix is kept in block echelon form at all times.

Theorem 1. *Algorithm 1 reduces any given input matrix A to row echelon form in finite time.*

The simple proof is omitted for brevity.

2.1 Variant: Computation of Matrix Rank

The Gaussian Elimination algorithm can be easily adapted to compute the rank of a general (unsymmetric and possibly rectangular) sparse matrix: one just needs to count the number of non-null rows of the row echelon form.

Function ProcessingUnit in Algorithm 1 is modified to return an integer number: the result shall be 1 if at least one row has been assigned to this PU ($u \neq$ NIL) and 0 otherwise.

Procedure Master performs a sum-reduce when collecting results: replace line 8 with *result* $\leftarrow$ sum-reduce of *output* from $P[c]$, for $c = 1, \dots, m$.

2.2 Variant: LUP Factorization

We shall outline how Algorithm 1 can be modified to produce a variant of the familiar LUP factorization. For the rest of this section we assume that A has coefficients in a field and is square and full-rank.

It is useful to recast the Rheinfall algorithm in matrix multiplication language, to highlight the small differences with the usual LU factorization by Gaussian Elimination. Let Π_0 be the permutation matrix that reorders rows of A so that $\Pi_0 A$ is in block echelon form; this is where Rheinfall's PUs start their work. If we further assume that PUs perform elimination and only *after that* they all perform communication at the same time,[1] then we can write the k-th elimination step as multiplication by a matrix E_k (which is itself a product of elementary row operations matrices), and the ensuing communication step as multiplication by a permutation matrix Π_{k+1} which rearranges the rows again into block echelon form (with the proviso that the u row to be used for elimination of other rows in the block comes first). In other words, after step k the matrix A has been transformed to $E_k \Pi_{k-1} \cdots E_0 \Pi_0 A$.

Theorem 2. *Given a square full-rank matrix A, the Rheinfall algorithm outputs a factorization $\Pi A = LU$, where:*

- $U = E_{n-1} \Pi_{n-1} \cdots E_0 \Pi_0 A$ *is upper triangular;*
- $\Pi = \Pi_{n-1} \cdots \Pi_0$ *is a permutation matrix;*
- $L = \Pi_{n-1} \cdots \Pi_1 \cdot E_0^{-1} \Pi_1^{-1} E_1^{-1} \cdots \Pi_{n-1}^{-1} E_{n-1}^{-1}$ *is lower unitriangular.*

The proof is omitted for brevity.

The modified algorithm works by exchanging triplets (r, h, s) among PUs; every PU stores one such triple (u, i, l), and uses u as pivot row. Each processing unit $P[c]$ receives a triple (r, h, s) and sends out (r', h, s'), where:

- The r rows are initially the rows of $\Pi_0 A$; they are modified by successive elimination steps as in Algorithm 1: $r' = r - \alpha u$ with $\alpha = r[c]/u[c]$.
- h is the row index at which r originally appeared in $\Pi_0 A$; it is never modified.
- The s rows start out as rows of the identity matrix: $s = e_h$ initially. Each time an elimination step is performed on r, the corresponding operation is performed on the s row: $s' = s + \alpha l$.

When the End message reaches the last PU, the Master procedure collects triplets (u_c, i_c, l_c) from PUs and constructs:

- the upper triangular matrix $U = (u_c)_{c=1,\dots,n}$;

[1] As if using a BSP model [7] for computation/communication. This assumption is not needed by "Rheinfall" (and is actually not the way it has been implemented) but does not affect correctness.

– a permutation π of the indices, mapping the initial row index i_c into the final index c (this corresponds to the Π permutation matrix);
– the lower triangular matrix L by assembling the rows l_c after having permuted columns according to π.

2.3 Pivoting

A key observation in Rheinfall is that all rows assigned to a PU start at the same column. This implies that pivoting is restricted to the rows in a block, but also that each PU may independently choose the row it shall use for elimination.

A form of threshold pivoting can easily be implemented within these constraints: assume that A has floating-point entries and let $Q^+ = Q \cup \{u\}$ be the block of rows worked on by Processing Unit $P[c]$ at a certain point in time (including the current pivot row u). Let $b = \max\{|r[c]| : r \in Q^+\}$; choose as pivot the sparsest row r in Q^+ such that $|r[c]| \geq \gamma \cdot b$, where $\gamma \in [0, 1]$ is the chosen threshold. This guarantees that elements of L are bounded by γ^{-1}.

When $\gamma = 1$, threshold pivoting reduces to partial pivoting (albeit restricted to block-scope), and one can repeat the error analysis done in [4, Section 3.4.6] almost verbatim. The main difference with the usual column-scope partial pivoting is that different pivot rows may be used at different times: when a new row with a better pivoting entry arrives, it replaces the old one. This could result in the matrix growth factor being larger than with GEPP; only numerical experiments can tell how much larger and whether this is an issue in actual practice. However, no such numerical experiments have been carried out in this preliminary exploration of the Rheinfall algorithm.

Still, the major source of instability when using the Rheinfall algorithm on matrices with floating-point entries is its sensitivity to "compare to zero": after elimination has been performed on a row, the eliminating PU must determine the new starting column. This requires scanning the initial segment of the (modified) row to determine the column where the first nonzero lies. Changes in the threshold $\epsilon > 0$ under which a floating-point number is considered zero can significantly alter the final outcome of Rheinfall processing.

Stability is not a concern with exact arithmetic (e.g., integer coefficients or finite fields): in this cases, leeway in choosing the pivoting strategy is better exploited to reduce fill-in or avoid entries growing too fast. Experiments on which pivoting strategy yields generally better results with exact arithmetic are still underway.

3 Sample Implementation

A sample program has been written that implements matrix rank computation and LU factorization with the variants of Algorithm 1 described before. Source code is publicly available from `http://code.google.com/p/rheinfall`.

Since there is only a limited degree of parallelism available on a single computing node, processing units are not implemented as separate continuously-running threads; rather, the `ProcessingUnit` class provides a `step()` method, which implements a single pass of the main loop in procedure ProcessingUnit (cf. lines 15–24 in Algorithm 1). The main computation function consists of an inner loop that calls each PU's `step()` in turn, until all PUs have performed one round of elimination. Incoming messages from other MPI processes are then received and dispatched to the destination PU. This outer loop repeats until there are no more PUs in Running state.

When a PU starts its `step()` procedure, it performs elimination on all rows in its "inbox" Q and immediately sends the modified rows to other PUs for processing. Incoming messages are only received at the end of the main inner loop, and dispatched to the appropriate PU. Communication among PUs residing in the same MPI process has virtually no cost: it is implemented by simply adding a row to another PU's "inbox". When PUs reside in different execution units, `MPI_Issend` is used: each PU maintains a list of sent messages and checks at the end of an elimination cycle which ones have been delivered and can be removed.

4 Sequential Performance

The "Rheinfall" algorithm can of course be run on just one processor: processing units execute a single `step()` pass (corresponding to lines 15–24 in Algorithm 1), one after another; this continues until the last PU has switched to Done state.

4.1 Integer Performance

In order to get a broad picture of "Rheinfall" sequential performance, the rank-computation program is being tested an all the integer matrices in the SIMC collection [2]. A selection of the results are shown in Table 1, comparing the performance of the sample Rheinfall implementation to the integer GEPP implementation provided by the free software library LinBox [1,6].

Results in Table 1 show great variability: the speed of "Rheinfall" relative to LinBox changes by orders of magnitude in one or the other direction. The performance of both algorithms varies significantly depending on the actual arrangement of nonzeroes in the matrix being processed, with no apparent correlation to simple matrix features like size, number of nonzeroes or fill percentage.

Table 2 shows the running time on the transposes of the test matrices. Both in LinBox's GEPP and in "Rheinfall", the computation times for a matrix and its transpose could be as different as a few seconds versus several hours! However, the variability in Rheinfall is greater, and looks like it cannot be explained by additional arithmetic work alone. More investigation is needed to better understand how "Rheinfall" workload is determined by the matrix nonzero pattern.

Table 1. CPU times (in seconds) for computing the matrix rank of selected integer matrices; boldface font marks the best result in each row. The "Rheinfall" column reports times for the sample C++ implementation. The "LinBox" column reports times for the GEPP implementation in LinBox version 1.1.7. The programs were run on the UZH "Schroedinger" cluster, equipped with Intel Xeon X5560 CPUs @ 2.8GHz and running 64-bit SLES 11.1 Linux; codes were compiled with GCC 4.5.0 with options `-O3 -march=native`.

Matrix	rows	columns	nonzero	fill%	Rheinfall	LinBox
M0,6-D8	862290	1395840	8498160	0.0007	**23.81**	36180.55
M0,6-D10	616320	1274688	5201280	0.0007	23378.86	**13879.62**
olivermatrix.2	78661	737004	1494559	0.0026	**2.68**	115.76
Trec14	3159	15905	2872265	5.7166	**116.86**	136.56
GL7d24	21074	105054	593892	0.0268	95.42	**61.14**
IG5-18	47894	41550	1790490	0.0900	1322.63	**45.95**

Table 2. CPU times (in seconds) for computing the matrix rank of selected integer matrices and their transposes; boldface font marks the best result in each row. The table compares running times of the Rheinfall/C++ and GEPP LinBox 1.1.7 codes. The columns marked with *(T)* report CPU times used for the transposed matrix. Computation of the transposes of matrices "M0,6-D8" and "Trec14" exceeded the available 24 GB of RAM. Hardware, compilation flags and running conditions are as in Table 1, which see also for matrix size and other characteristics.

Matrix	Rheinfall *(T)*	Rheinfall	LinBox *(T)*	LinBox
M0,6-D8	*No mem.*	**23.81**	50479.54	36180.55
M0,6-D10	**37.61**	23378.86	26191.36	13879.62
olivermatrix.2	**0.72**	2.68	833.49	115.76
Trec14	*No mem.*	116.86	**43.85**	136.56
GL7d24	**4.81**	95.42	108.63	61.14
IG5-18	12303.41	1322.63	787.05	**45.95**

Table 3. Average Mflop/s attained in running LU factorization of square $N \times N$ matrices; boldface font marks the best result in each row. The table compares the performance of the sample Rheinfall/C++ LU factorization with SuperLU 4.2. The test matrices are a subset of those used in [5]. See Table 1 for hardware characteristics.

Matrix	N	nonzero	fill%	Rheinfall	SuperLU
bbmat	38744	1771722	0.118	83.37	**1756.84**
g7jac200sc	59310	837936	0.023	87.69	**1722.28**
lhr71c	70304	1528092	0.030	*No mem.*	**926.34**
mark3jac140sc	64089	399735	0.009	92.67	**1459.39**
torso1	116158	8516500	0.063	97.01	**1894.19**
twotone	120750	1224224	0.008	91.62	**1155.53**

4.2 Floating-Point Performance

In order to assess the "Rheinfall" performance in floating-point uses cases, the LU factorization program has been tested on a subset of the test matrices used in [5]. Results are shown in Table 3, comparing the Mflop/s attained by the "Rheinfall" sample implementation with the performance of SuperLU 4.2 on the same platform.

The most likely cause for the huge gap in performance between "Rheinfall" and SuperLU lies in the strict row-orientation of "Rheinfall": SuperLU uses block-level operations, whereas Rheinfall only operates on rows one by one. However, row orientation is a defining characteristics of the "Rheinfall" algorithm (as opposed to a feature of its implementation) and cannot be circumvented. Counting also the "compare to zero" issue outlined in Section 2.3, one must conclude that "Rheinfall" is generally not suited for inexact computation.

5 Parallel Performance and Scalability

The "Rheinfall" algorithm does not impose any particular scheme for mapping PUs to execution units. A column-cyclic scheme has been currently implemented.

Let p be the number of MPI processes (ranks) available, and m be the total number of columns in the input matrix A. The input matrix is divided into vertical stripes, each comprised of w adjacent columns. Stripes are assigned to MPI ranks in a cyclic fashion: MPI process k (with $0 \leq k < p$) hosts the k-th, $(k+p)$-th, $(k+2p)$-th, etc. stripe; in other words, it owns processing units $P[w \cdot (k + a \cdot p) + b]$ where $a = 0, 1, \dots$ and $0 \leq b < w$.

5.1 Experimental Results

In order to assess the parallel performance and scalability of the sample "Rheinfall" implementation, the rank-computation program has been run on the matrix `M0,6-D10` (from the Mgn group of SIMC [2]; see Table 1 for details). The program has been run with a varying number of MPI ranks and different values of the stripe width parameter w: see Figure 1.

The plots in Figure 1 show that running time generally decreases with higher w and larger number p of MPI ranks allocated to the computation, albeit not regularly. This is particularly evident in the plot of running time versus stripe width (Figure 1, right), which shows an alternation of performance increases and decreases. A more detailed investigation is needed to explain this behavior; we can only present here some working hypotheses.

The w parameter influences communication in two different ways. On the one hand, there is a lower bound $O(m/w)$ on the time required to pass the End message from $P[0]$ to $P[m]$. Indeed, since the End message is always sent from one PU to the next one, then we only need to send one End message per stripe over the network. This could explain why running time is almost the same for $p = 128$ and $p = 256$ when $w = 1$: it is dominated by the time taken to pass the End message along.

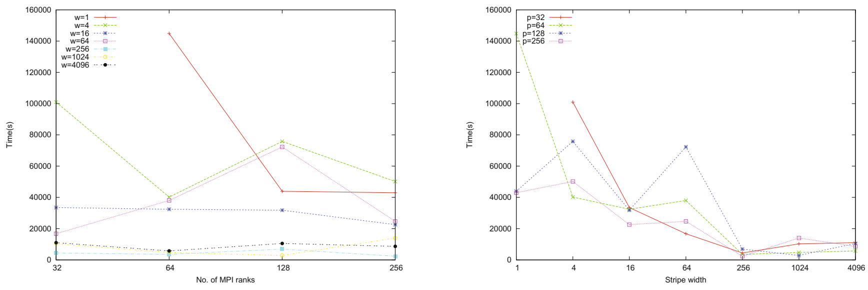


Fig. 1. *Left:* Plot of the running time (in seconds, y-axis) of the sample Rheinfall implementation on the matrix M0,6-D10, versus the number p of MPI ranks (x-axis). *Right:* Plot of the running time (in seconds, y-axis) of the sample Rheinfall implementation on the matrix M0,6-D10, versus stripe width w (x-axis).

Table 4. Percentage of running time spent in MPI communication for the sample Rheinfall/C implementation on the matrix M0,6-D10, with varying number of MPI ranks and stripe width parameter w. Columns `MPI_Recv`, `MPI_Iprobe` and `MPI_Barrier` report on the percentage of MPI time spent spent servicing these calls; in these cases, the minimum is always very close to zero hence it is omitted from the table. Tests were executed on the UZH cluster "Schroedinger"; see Table 1 for hardware details. The MPI layer was provided by OpenMPI version 1.4.3, using the TCP/IP transport.

p	w	MPI Total%			MPI_Recv%		MPI_Iprobe%		MPI_Barrier%	
		Avg.$\pm\sigma$	Max.	Min.	Avg.	Max.	Avg.	Max.	Avg.	Max.
16	16	18.79 ± 0.32	19.61	18.37	79.96	81.51	3.13	3.29	0.00	0.00
32	16	11.54 ± 0.53	12.87	10.93	4.41	71.97	14.19	14.97	0.00	0.00
64	16	12.25 ± 0.20	12.78	11.77	1.12	55.94	34.57	36.13	0.00	0.00
128	16	20.51 ± 0.55	23.87	19.33	31.62	35.06	61.08	64.17	0.00	0.00
16	256	26.77 ± 1.79	29.50	23.29	89.69	92.31	1.27	1.50	0.00	0.20
32	256	10.17 ± 1.34	13.57	7.83	78.51	85.33	13.10	18.08	0.00	0.01
64	256	16.55 ± 2.16	22.15	11.74	61.46	73.13	27.20	43.73	0.00	0.67
128	256	15.43 ± 0.64	18.65	14.35	6.15	10.97	90.98	95.27	0.00	0.02
256	256	38.92 ± 1.94	43.24	32.99	6.12	14.20	89.38	97.41	0.00	0.60
16	4096	9.08 ± 1.62	13.81	7.22	50.57	66.97	7.52	10.35	0.00	0.19
32	4096	6.53 ± 1.97	12.33	3.71	36.80	58.48	28.30	51.23	1.51	3.71
64	4096	6.81 ± 1.52	12.01	4.53	8.73	30.15	72.13	93.92	10.88	26.93
128	4096	16.73 ± 7.80	44.72	8.59	5.12	21.82	46.82	92.24	43.69	86.65
256	4096	45.78 ± 28.32	88.22	9.91	0.00	9.18	11.94	96.68	86.09	98.63

On the other hand, MPI messages are collected after each processing unit residing on a MPI rank has performed a round of elimination; this means that a single PU can slow down the entire MPI rank if it gets many elimination operations to perform. The percentage of running time spent executing MPI calls has been collected using the `mpiP` tool [8]; a selection of relevant data is available in Table 4. The three call sites for which data is presented measure three different aspects of communication and workload balance:

– The `MPI_Recv` figures measure the time spent in actual row data communication (the sending part uses `MPI_Issend` which returns immediately).
– The `MPI_Iprobe` calls are all done after all PUs have performed one round of elimination: thus they measure the time a given MPI rank has to wait for data to arrive.
– The `MPI_Barrier` is only entered after all PUs residing on a given MPI rank have finished their job; it is thus a measure of workload imbalance.

Now, processing units corresponding to higher column indices naturally have more work to do, since they get the rows at the end of the elimination chain, which have accumulated fill-in. Because of the way PUs are distributed to MPI ranks, a larger w means that the last MPI rank gets more PUs of the final segment: the elimination work is thus more imbalanced. This is indeed reflected in the profile data of Table 4: one can see that the maximum time spent in the final `MPI_Barrier` increases with w and the number p of MPI ranks, and can even become 99% of the time for some ranks when $p = 256$ and $w = 4096$.

However, a larger w speeds up delivery of Row messages from $P[c]$ to $P[c']$ iff $(c' - c)/w \equiv 0 (\text{mod } p)$. Whether this is beneficial is highly dependent on the structure of the input matrix: internal regularities of the input data may result on elimination work being concentrated on the same MPI rank, thus slowing down the whole program. Indeed, the large percentages of time spent in `MPI_Iprobe` for some values of p and w show that the matrix nonzero pattern plays a big role in determining computation and communication in Rheinfall. Static analysis of the entry distribution could help determine an assignment of PUs to MPI ranks that keeps the work more balanced.

6 Conclusions and Future Work

The “Rheinfall” algorithm is basically a different way of arranging the operations of classical Gaussian Elimination, with a naturally parallel and distributed-memory formulation. It retains some important features from the sequential Gaussian Elimination; namely, it can be applied to general sparse matrices, and is independent of matrix entry type. Pivoting can be done in Rheinfall with strategies similar to those used for GEPP; however, Rheinfall is not equally suited for exact and inexact arithmetic.

Poor performance when compared to state-of-the-art algorithms and some inherent instability due to the dependency on detection of nonzero entries suggest that “Rheinfall” is not a convenient alternative for floating-point computations.

For exact arithmetic (e.g., integers), the situation is quite the opposite: up to our knowledge, “Rheinfall” is the first practical distributed-memory Gaussian Elimination algorithm capable of exact computations. In addition, it is competitive with existing implementations also when running sequentially.

The distributed-memory formulation of “Rheinfall” can easily be mapped on the MPI model for parallel computations. An issue arises on how to map Rheinfall’s Processing Units to actual MPI execution units; the simple column-cyclic distribution discussed in this paper was found experimentally to have poor workload balance. Since the workload distribution and the communication graph are both determined by the matrix nonzero pattern, a promising future direction could be to investigate the use of graph-based partitioning to determine the distribution of PUs to MPI ranks.

Acknowledgments. The author gratefully acknowledges support from the University of Zurich and especially from Prof. K. K. Baldridge, for the use of the computational resources that were used in developing and testing the implementation of the Rheinfall algorithm. Special thanks go to Professors J. Hall and O. Schenk, and Dr. D. Fiorenza for fruitful discussions and suggestions. Corrections and remarks from the PPAM reviewers greatly helped the enhancement of this paper from its initial state.

References

1. Dumas, J.G., Gautier, T., Giesbrecht, M., Giorgi, P., Hovinen, B., Kaltofen, E., Saunders, B.D., Turner, W.J., Villard, G.: LinBox: A Generic Library for Exact Linear Algebra. In: Cohen, A., Gao, X.S., Takayama, N. (eds.) Mathematical Software: ICMS 2002, Proceedings of the First International Congress of Mathematical Software, pp. 40–50. World Scientific (2002)
2. Dumas, J.G.: The Sparse Integer Matrices Collection, http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/simc.html
3. Dumas, J.G., Villard, G.: Computing the rank of large sparse matrices over finite fields. In: CASC 2002 Computer Algebra in Scientific Computing, pp. 22–27. Springer, Heidelberg (2002)
4. Golub, G., Van Loan, C.: Matrix Computation, 2nd edn. Johns Hopkins University Press (1989)
5. Grigori, L., Demmel, J.W., Li, X.S.: Parallel symbolic factorization for sparse LU with static pivoting. SIAM J. Scientific Computing 29(3), 1289–1314 (2007)
6. LinBox website, http://linalg.org/
7. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM 33, 103–111 (1990), http://doi.acm.org/10.1145/79173.79181
8. Vetter, J., Chambreau, C.: mpiP: Lightweight, Scalable MPI profiling (2004), http://www.llnl.gov/CASC/mpiP

Parallel FEM Adaptation on Hierarchical Architectures

Tomasz Olas[1], Roman Wyrzykowski[1], and Pawel Gepner[2]

[1] Czestochowa University of Technology,
Dabrowskiego 69, 42-201 Czestochowa, Poland
{olas,roman}@icis.pcz.pl
[2] Intel Corporation
pawel.gepner@intel.com

Abstract. The parallel FEM package NuscaS allows us to solve adaptive FEM problems with 3D unstructured meshes on distributed-memory parallel computers such as PC-clusters. In our previous works, a new method for parallelizing the FEM adaptation was presented, based on using the 8-tetrahedra longest-edge partition. This method relies on a decentralized approach, and is more scalable in comparison to previous implementations requiring a centralized synchronizing node.

At present nodes of clusters contain more and more processing cores. Their efficient utilization is crucial for providing high performance of numerical codes. In this paper, different schemes of mapping the mesh adaptation algorithm on such hierchical architectures are presented and compared. These schemes use either the pure message-passing model, or the hybrid approach which combines shared-memory and message-passing models. Also, we investigate an approach for adapting the pure MPI model to hierarchical topology of clusters with multi-core nodes.

1 Introduction

The finite element method (FEM) is a powerful tool for studying different phenomena in various areas. Parallel computing allows FEM users to overcome computational and/or memory bottlenecks of sequential applications [8]. In particular, an object-oriented environment for the parallel FEM modeling, called *NuscaS*, was developed at the Czestochowa University of Technology [16]. This package allows for solving adaptive FEM problems with 3D unstructured meshes on distributed-memory parallel computers such as PC-clusters [10].

Adaptive techniques [12] are powerful tools to perform efficiently the FEM analysis of complex and time-consuming 3D problems described by time-dependent PDEs. However, during the adaptation computational workloads change unpredictably at the runtime, and a dynamic load balancing is required. In our previous work [10], we focused on the problem how to implement the dynamic load balancing efficiently on distributed-memory parallel computers such as PC clusters. At the same time, a serious performance bottleneck of this numerical code was the sequential implementation of the mesh adaptation step. So in the next paper [11], a method which eliminates this drawback by implementing the mesh adaptation in parallel was proposed. The new solution relies on a decentralized approach, and is more scalable in comparison to previous implementations [2,13,14], requiring a centralized synchronizing node. The method was developed for the message-passing model, and implemented using the MPI standard.

Multiple computing cores become ubiquitous in commodity microprocessor chips used to build modern HPC systems [6]. The basic building block of such systems consists of multiple multi-core chips sharing memory in a single node. As a result, currently the dominant HPC architectures comprise clusters of such nodes interconnected via a high-speed network supporting a heterogeneous memory model – shared memory with non-uniform memory access within a single node and distributed memory across the nodes.

From the user's perspective, the easiest way of programming these systems is to ignore the hybrid memory architecture and use a pure message-passing programming model. This is a reasonable approach since most MPI libraries take advantage of the shared memory within a node and optimize intra-node communications. At the same time, MPI based implementations of many algorithms that are scalable to a large number of uni-core processors suffer efficiency loses when implemented on such architectures [15]. This loss of scalability is in part because of the limited memory bandwidth available to multiple cores within the processor, and in part because traditional message-passing parallelization strategies do not account for the memory and communication hierarchy of clusters. One way of overcoming these limitations is to use a hybrid message-passing/shared-memory parallelization approach, which exploit the hierarchy of multi-core clusters via nested parallelization schemes that use MPI for inter-node communications and OpenMP for inter-node data exchange. However, even though hybrid parallelization methods offer various advantages in theory [1], pure MPI implementations frequently outperform hybrid ones in practice as overheads in shared memory threading can outweigh performance benefits [15,3].

This paper is devoted to investigating different schemes of mapping the FEM adaptation algorithm on hierarchical architectures of PC-clusters with multi-core nodes. Our previous works [8,9], related to mapping FEM computations on hierarchical architectures of clusters with SMP nodes, confirmed the usability of the hybrid approach for increasing performance of FEM computations. For implementing this approach, we used the combination of MPI for interprocessor communications, and POSIX threads for multithreading inside SMP nodes. For the hybrid approach, two methods for mapping FEM algorithms were considered: (i) method of global data inside the node, and (ii) two-level domain decomposition. In the first method, a single, global subsystem of equations is build and solved by multiple threads inside each node. For the second method, each subdomain obtained after coarse-grain partitioning is further decomposed into p_n fine-grain subdomains, where p_n is the number of processors in a node. Then each thread is responsible for building and solving a separate subsystem of equations.

The material of this paper is organized as follows. In Section 2, the parallel FEM adaptation algorithm is recalled concisely. Section 3 is devoted to investigation of schemes for mapping this algorithm using the hybrid approach, while Section 4 deals with adapting the pure MPI approach to hardware topology of clusters. Conclusions and feature work are presented in Section 5.

2 Parallel FEM Adaptation

2.1 Mesh Decomposition in NuscaS

The parallel version of the library is based on the geometric decomposition applied for nodes of a parallel system. In this case, a FEM mesh is divided into p submeshes (domains), which are assigned to separate processors (cores) of a parallel architecture. Every processor (or process) keeps the assigned part of the mesh. As a result, the j-th domain will own a set of N_j nodes selected from all the N nodes of the mesh. For an arbitrary domain with index j, we have three types of nodes [11]: (i) N_j^i of internal nodes; (ii) N_j^b of boundary nodes; (iii) N_j^e of external nodes. Internal and boundary nodes are called *local* ones, so the number of local nodes is $N_j^l = N_j^i + N_j^b$;

When implementing computations in parallel, information about domains are exchanged between them. Hence at the preprocessing stage, it is necessary to generate data structures for each domain, to provide an efficient implementation of communication. Furthermore, values computed in boundary nodes must be agreed between neighbor domains. For this aim, in each domain (process) j for every neighbor process k the following sets are stored:

– S_j^k - set of those indexes of boundary nodes in process j that are external nodes for process k;
– R_j^k - set of those indexes of external nodes in process j that are assigned to process k.

2.2 Parallel Algorithm for Mesh Adaptation

The first step in a mesh adaptation procedure using the h-method [13] is selection of elements which should be partitioned, based on estimating a discretization error [10]. The next step divides elements using the longest-edge bisection method. To implement the mesh adaptation in parallel, we utilized [11] the Longest-Edge Propagation Path method (LEPP in short) [13]. The partitioning of tetrahedral elements is then performed based on the iterative algorithm of 8-tetrahedra longest edge partition.

When implementing the mesh adaptation in parallel, we base on the available mesh decomposition and coupling between neighbor domains (processes). Every process stores information about the domain assigned to it, sequence and number of internal, boundary and external nodes, as well as information necessary to organize communication with the neighbor processes. Additionally, the global enumeration of nodes is used, including the external nodes. Each process holds

```
1. for each edge $e$ of every element belonging to the set $E_{selected}$ , perform the
   procedure *SelectEdge(e)*
2. perform locally the algorithm *LEPP*: for each selected edge $e$ belonging to the set
   $e_{selected}$
     – for each element $E$ which uses the edge $e$
         • add the element $E$ to the set $E_{selected}$ of already selected elements (unless
           this element was added before)
         • for the longest edge $e_l$ belonging to faces of the element which uses the
           selected edge $e$, perform *SelectEdge($e_l$)*
3. provide coherency of local propagation paths between neighbor processes
4. derive the global enumeration of nodes taking into account newly created nodes,
   together with assigning newly created nodes to separate processes
5. perform partitioning of elements which belong to $E_{selected}$
6. modify enumeration of nodes in such a way that internal nodes are located in the
   first place, then boundary, and finally external; then upgrade data structures used
   for communication between neighbor processes
```

Fig. 1. Parallel algorithm for mesh adaptation

```
– add the edge $e$ to the list $e_{selected}$ of already selected edges
– check if the edge $e$ is located on the boundary with a neighbor domain (process);
  if so, add this edge to the list $e^i_{send}$ of edges which are sent to this neighbor
– divide the edge $e$
```

Fig. 2. Procedure *SelectEdge(e)*

data structures which make possible transition from the global enumeration to the local one, and vice versa.

The proposed parallel algorithm of mesh adaptation is presented in Fig. 1. Communication between neighbor processes takes place in steps 3 and 4, based on information about edges which are exchanged between processes. This information is stored in sets e^i_{send} $(i = 0, \ldots, p-1)$, after performing the procedure *SelectEdge* (Fig. 2). As a result, in step 3 which is responsible for providing coherency of local propagation paths between neighbor processes, pairs of global indexes (n^1_g, n^2_g) describing edges are sent, using the non-blocking MPI routine `MPI_Isend`. After receiving this information, the mapping from the global enumeration to local one is performed, to allow for placing edges in local data structures.

When performing FEM computation in parallel, elements which are located on boundaries of domains are duplicated in neighbor processes. This solution allows for avoiding communications at the cost of additional computations. We follow this concept of patches when performing the mesh adaptation. In this case, the partitioning of elements must be realized in the same way in neighbor

Table 1. Parameters of FEM meshes before and after adaptation

mesh	before adaptation		after adaptation	
	number of nodes	number of elements	number of nodes	number of elements
14K	14313	77824	109009	622592
100K	109009	622592	850849	4980736
270K	272697	1540096	2116977	12320768
400K	402017	2293760	3136705	18350080
850K	850849	4980736	6723393	39845888

processes. The only difficulty emerges when during the element partitioning there are two or more selected edges with the same length. To avoid this difficulty, we propose a solution based on the global indexes of nodes. So, when determining the longest edge in case of edges with the equal length, we compare additionally the global indexes of edges to choose an edge with the highest value of global indexes. This solution allows us to avoid communication between processes. It is sufficient to derive the global enumeration of nodes.

The implementation of step 4 starts with the parallel procedure of deriving a new global enumeration of nodes, taking into account newly created nodes. This procedure includes the following three stages:

4.1 *Determine interval of global indexes for each process, as well as assign global indexes to the local nodes of processes:*
For this aim, the information about the number n_l^i of local nodes in each process is spread among all the processes, using the `MPI_Allgather` routine. The global index n_g^i of a node i in process j is determined by adding the local index n^i of this node to the sum of numbers of nodes in all the processes from 0 do $j-1$:

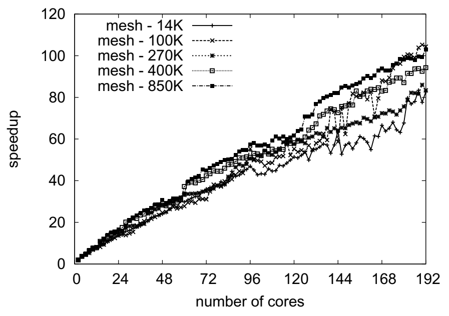
$$n_g^i = n^i + \sum_{k=0}^{j-1} N_k^l. \tag{1}$$

4.2 *Exchange global indexes of nodes located on boundaries of domains*
4.3 *Exchange global indexes of newly created nodes*

2.3 Performance Results

Parameters of FEM meshes which were used in our experiments are presented in Table 1. These meshes were generated for the geometry shown in Fig. 4 from paper [11]. Since the main purpose of these experiments was to investigate the scalability of the proposed algorithm and its implementation, the adaptation was performed for all the elements of FEM meshes. The experiments were executed on two clusters:

– **Westmere** cluster with 16 nodes connected by the Infiniband interconnect; each node contains two Westmere EP 2.93GHz processors (totally 12 cores in each node) with 48GB RAM;

a)


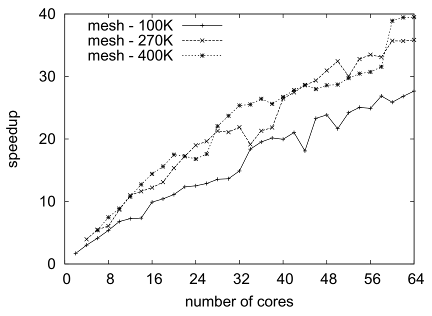
b)


Fig. 3. Speedups versus number of cores, for different FEM meshes: (a) Westmere and (b) Xeon clusters

– **Xeon** cluster which contains nodes with two Dual Core Xeon 64-bit processors operating at a core frequency of 2.66 GHz, with 4 GB RAM and the Gigabit Ethernet network.

All test were performed using the g++ compiler and OpenMPI as an implementation of MPI standard.

Figure 3 presents speedups achieved in our experiments for different numbers of cores. The presented results take into account only the process of mesh adaptation, without auxiliary operations on data structures, as well as without adjustment to object geometry and load balancing. The achieved performance results show quite good scalability of the proposed parallel algorithm of mesh adaptation both for the Infiniband and Gigabit Ethernet interconnects. The achieved speedup increases with increase in the problem size. The maximum speedup value of $\tilde{1}03$ was obtained for the 850K mesh processed on 192 cores.

3 Hybrid Approach

One of advantages of applying the hybrid approach to parallelizing the mesh adaptation algorithm is reducing overheads being results of using patches when processing neighboring domains. In these patches, the partitioning of FEM elements is performed in all the neighboring domains, which increases the algorithm execution time. As a result, the less number of domains, the less overheads are introduced.

In the hybrid approach, the load distribution across cores of a node is performed in parallel sections, according to the fork-and-join model of OpenMP standard. In the developed version of the mesh adaptation algorithm, the communication across nodes is implemented by the main thread, outside parallel sections. Figure 4 presents a snippet of the mesh adaptation algorithm, responsible for the local execution of the LEPP algorithm (step 2 in Fig. 1), parallelized using OpenMP. In this snippet, we use `omp for` pragma to parallelize a loop

```
int nEdges = static_cast< int >(E_selected.size());
#pragma omp parallel for
for (int i = 0; i < nEdges; ++i) {
      Edge* edge = E_selected[i];
      list<Element*>::iterator ib = edge->elements.begin();
      list<Element*>::iterator ie = edge->elements.end();
      for (; ib != ie; ++ib) {
            if (!(*ib)->isSelected()) {
#pragma omp critical
                  E_selected.push_back(*ib);
                  (*ib)->status = 1;
            }
            (*ib)->SelectEdge(edge, E_selected, edges, nodes, adaptationComm);
      }
}
```

Fig. 4. Snippet of the mesh adaptation algorithm: parallelization inside processes is performed using OpenMP

which corresponds to processing all the selected elements. Adding a new element to already selected ones ($E_{selected}$) requires to apply a suitable synchronization (`omp critical` pragma). The synchronization mechanisms are also applied inside `SelectEdge` method.

Figure 5 presents the comparison of speedups achieved for the pure MPI and hybrid approaches. It can be seen that the usage of the hybrid model does not increase the performance. On the contrary, it leads to the performance degradation. The more threads are executed on a node, the worse speedup is achieved. In particular, in case of the Westmere cluster with 12 threads executed on each node, the performance is significantly lower than in case of the Xeon cluster with 4 threads per node.

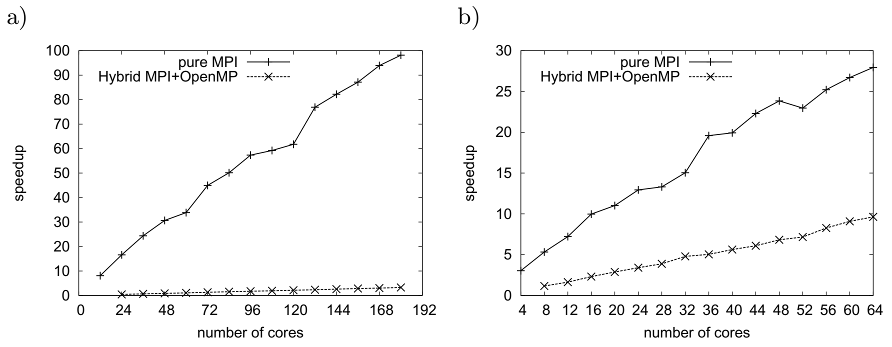


Fig. 5. Comparison of speedups for pure MPI and hybrid models in case of 850K mesh processed on (a) Westmere cluster (12 threads per node), and (b) Xeon cluster (4 threads per node)

The main reason for the above-observed performance degradation are frequent references made by all the threads, executed inside a node, to common data structures storing mesh elements and edges, as well as the necessity to synchronize these references. In fact, in case of the standard implementation of STL containers, there is necessary to synchronize access of all the threads operating on a common container when a certain thread is modifying this container. In addition, the utilization by threads such a common data structures as lists leads to a frequent usage of cache coherence mechanisms, which also reduces the overall performance.

4 Adapting Pure MPI Approach to Hardware Topology of Clusters

For the pure MPI approach, the number of processes executed on each node is equal the number of cores per node. The idea of adapting this parallel programming model to the hierarchical (multi-level) topology (architecture) of a cluster with multi-core nodes is based on such distribution of data among particular MPI processes that minimizes inter-node communications. Because of the lack of tools that perform the graph partitioning with taking into account the hierarchical cluster architecture, we decide to use such standard graph partitioners as METIS [4]. However, this solution requires to develop methods which allow for the graph partitioning which takes into consideration the hierarchical architecture with the use of a tool which performs partitioning only on one level. For this aim, we develop and investigate two methods of this kind:

– method of two-level decomposition (Fig. 6),
– method of grouping (Fig. 7).

In the first stage of the two-level decomposition method, the FEM problem is divided into domains according to the number of nodes in a cluster. Then each domain is further split into smaller parts assigned directly to separate cores of a corresponding node. However, in practice the disadvantage of this method are difficulties in achieving the even load balancing among cores. These difficulties are mainly linked to external nodes that appear after performing the first stage. These nodes of FEM mesh belong to other processes, assigned to other cluster nodes, but are connected (through FEM elements) with mesh nodes assigned to a given process. When assigning to a core mesh nodes connected with such external nodes, it can be seen that some additional FEM elements are assigned to this core, which are connected with these external nodes and local mesh nodes of the core. Appearing these additional nodes increases the process load, which was not taken into account in the first stage.

Because of load balancing problems in this method, we propose an alternative method for the minimization of communication with preserving a good load balancing. In the method of grouping (Fig. 7), the FEM problem is divided

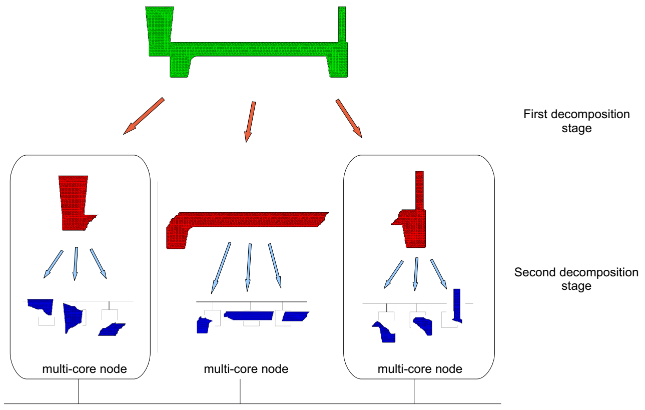


Fig. 6. Using two-level decomposition to adapt pure MPI model to topology of clusters with multi-core nodes

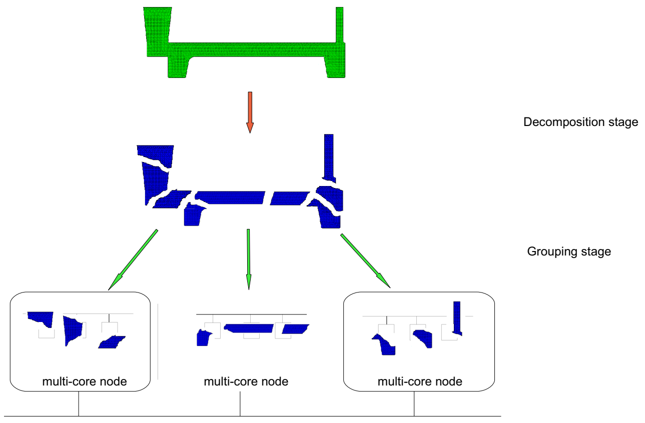


Fig. 7. Using grouping domains to adapt pure MPI model to topology of clusters with multi-core nodes

into domains according to the number of cores in a cluster. These domains are then distributed (grouped) among cluster nodes in such a way that allow for minimizing inter-node communications.

The grouping algorithm, responsible for assigning domains to cluster nodes, is implemented using the METIS package [4]. For the graph partitioning, we utilize the internal routine `MlevelKWayPartitioning`. Unlike standard METIS routines, this routine provides the possibility to control unbalance across graph partitions. By setting the value of unbalance factor on 1.0, we can guarantee that partitioning is performed on equal parts, and their number is equal the number of processor cores in cluster nodes. When implementing the graph partitioning, the graph nodes are previously determined domains, which have to be processed in cores. The graph edges correspond to inter-domain connections, which are results of communication between the domains. In order to guarantee that some domains are assigned to cores of the $i-th$ cluster node, it is enough to renumber them in such a way that they are enumerated using the consecutive integers beginning with $i \cdot C$, where C is the number of processor cores in cluster nodes.

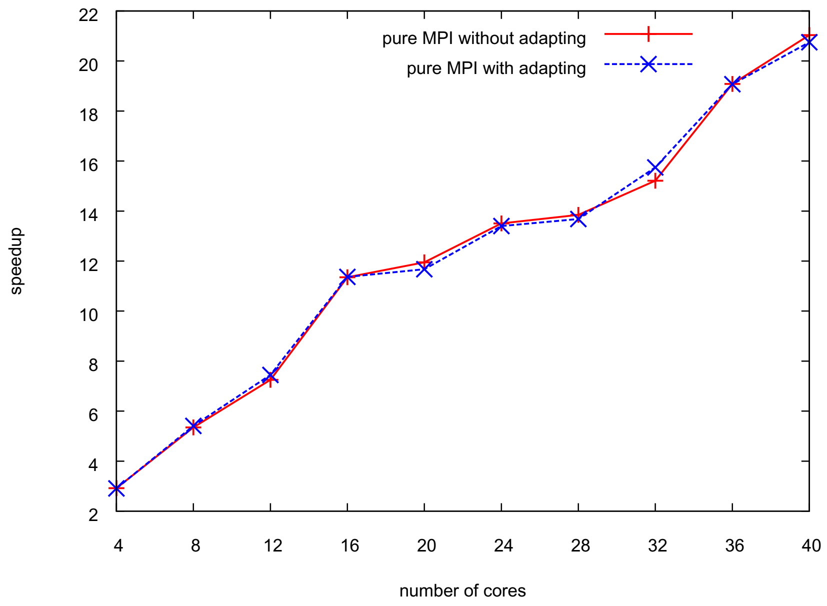


Fig. 8. Comparison of speedups achieved for the pure MPI model and approach with adapting MPI to hardware topology of clusters based on grouping domains: mesh 850K is processed on Xeon cluster (4 threads per node)

Figure 8 shows speedups achieved when adapting MPI to hardware topology of clusters based on grouping domains, as compared with the pure MPI model; the Xeon cluster and 850K mesh are assumed. Unfortunately, it can be seen that the approach discussed in this section does not give any noticeable performance advantages. The same conclusion is true for other FEM meshes and the Westmere cluster.

5 Conclusion and Future Work

This work is devoted to investigation of different approaches for mapping the FEM mesh adaptation algorithm on hierarchical architectures of clusters with multi-core nodes. The performance results of numerical experiments are presented as well.

The basic approach assumes the usage of pure MPI model with one MPI process per processor core. The utilization of the hybrid model (MPI + OpenMP) does not increase the performance. On the contrary, it leads to the performance degradation caused by frequent references made by all the threads, executed inside a node, to common data structures. Also, the presented approach for adapting the pure MPI model to hierarchical topology of clusters does no allow us to improve the performance. The reasons for this effect are difficulties in providing suitable partitionings of FEM problems.

Therefore, it becomes necessary to develop a new tool (or adapt an existing one) which will allow for graph partitioning taking into account the hierarchical architecture of clusters [7,5]. An alternative, promising approach to increasing the performance seems to be the development of mechanisms in the mesh adaptation algorithm allowing for partitioning of a element located on boundaries of domains only by a single process. At present, this partitioning is performed in all the processes which possess mesh modes linked to this element. Such a modification in the algorithm will reduce the amount of inter-process communications. At the same time, this modification will increase the time of computations because of introducing some redundant operations. Consequently, it will be indispensable to perform exhaustive tests on different cluster platforms using different MPI implementations.

Acknowledgments. This work was supported in part by the Polish Ministry of Science and Higher Education under Grant Nos. 648/N-COST/2010/0 COST IC0805 and BS/PB-1-112-3030/2011/S.

We gratefully acknowledge the help and support provided by Jamie Wilcox from Intel EMEA Technical Marketing HPC Lab.

References

1. Avera, R., Martino, B., Rak, M., Venticinque, S., Vilano, U.: Performance prediction through simulation of a hybrid MPI/OpenMP application. Parallel Computing 31, 1013–1033 (2005)
2. Balman, M.: Tetrahedral Mesh Refinement in Distributed Environments. In: 2006 Int. Conf. Parallel Processing Workshops (ICPPW 2006), pp. 497–504. IEEE Computer Soc. (2006)
3. Chorley, M.J., Walker, D.W.: Performance analysis of a hybrod MPI/OpenMP application on multi-core clusters. J. Comput. Sci. 1, 168–174 (2010)
4. Family of Graph and Hypergraph Partitioning Software, http://glaros.dtc.umn.edu/gkhome/views/metis

5. Jeannot, E., Mercier, G.: Improving MPI Applications Performance on Multicore Clusters with Optimized Process Placement. In: 2nd Workshop of COST 0805 Open Network for High-Performance Computing on Complex Environments, Timisoara, January 25-27 (2012)
6. Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., Chapman, B.: High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing 37, 562–575 (2011)
7. Mercier, G., Clet-Ortega, J.: Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI 2009. LNCS, vol. 5759, pp. 104–115. Springer, Heidelberg (2009)
8. Olas, T., Karczewski, K., Tomas, A., Wyrzykowski, R.: FEM Computations on Clusters Using Different Models of Parallel Programming. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 170–182. Springer, Heidelberg (2002)
9. Olas, T., Lacinski, L., Karczewski, K., Tomas, A., Wyrzykowski, R.: Performance of different communication mechanisms for FEM computations on PC-based cluster with SMP nodes. In: Proc. Int. Conf. Parallel Computing in Electrical Engineering, PARELEC 2002, pp. 305–311 (2002)
10. Olas, T., Leśniak, R., Wyrzykowski, R., Gepner, P.: Parallel Adaptive Finite Element Package with Dynamic Load Balancing for 3D Thermo-Mechanical Problems. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 299–311. Springer, Heidelberg (2010)
11. Olas, T., Wyrzykowski, R.: Adaptive FEM Package with Decentralized Parallel Adaptation of Tetrahedral Meshes. In: Lirkov, I. (ed.) LSSC 2011. LNCS, vol. 7116, pp. 622–629. Springer, Heidelberg (in press, 2012)
12. Patzak, B., Rypl, D.: A Framework for Parallel Adaptive Finite Element Computations with Dynamic Load Balancing. In: Proc. First Int. Conf. Parallel, Distributed and Grid Computing for Engineering, Paper 31. Civil-Comp Press (2009)
13. Plaza, A., Rivara, M.: Mesh Refinement Based on the 8-Tetrahedra Longest-Edge Partition. In: Proc. 12th Int. Meshing Roundtable, Sandia National Laboratories, pp. 67–78 (2003)
14. Rivara, M., Pizarro, D., Chrisochoides, N.: Parallel Refinement of Tetrahedral Meshes using Terminal-Edge Bisection Algorithm. In: Proc. 13th Int. Meshing Roundtable, Sandia National Labs, pp. 427–436 (2004)
15. Wei, F., Yilmaz, A.E.: A hybrid message passing/shared memory parallelization of the adaptive integral method for multi-core clusters. Parallel Computing 37, 279–301 (2011)
16. Wyrzykowski, R., Olas, T., Sczygiol, N.: Object-Oriented Approach to Finite Element Modeling on Clusters. In: Sørevik, T., Manne, F., Moe, R., Gebremedhin, A.H. (eds.) PARA 2000. LNCS, vol. 1947, pp. 250–257. Springer, Heidelberg (2001)

Solving Systems of Interval Linear Equations in Parallel Using Multithreaded Model and "Interval Extended Zero" Method

Mariusz Pilarek and Roman Wyrzykowski

Institute of Computer and Information Sciences,
Czestochowa University of Technology,
Dabrowskiego 73, 42-200 Czestochowa, Poland
{mariusz.pilarek,roman}@icis.pcz.pl

Abstract. In this paper, an approach to the solution of systems of interval linear equations with the use of the parallel machines is presented, based on parallel multithreaded model and "interval extended zero" method. This approach not only allows us to decrease the undesirable excess width effect, but makes it possible to avoid the inverted interval solutions too. The efficiency of this method has been already proved for non-parallel systems. Here it is shown that it can be also used to perform efficient calculations on parallel machines using the multithreaded model.

1 Introduction

Most of todays problems concerned with the solution of systems of linear equations are based on the real-number coefficients. However, when dealing with real-world problems we often meet different kinds of uncertainty and imprecision that should be taken into account in the problem formulation [8,7,10]. The known Leontief's input-output model [18] can be considered as an example of such a situation [21] since it is difficult to build the technical coefficient matrix with real numbers as the parameters of this economic model are usually charged by sufficient uncertainty. In many cases, this problem can be solved using methods of applied interval analysis. Since the publication of seminal Moore's work [19] a rapid development of interval arithmetic is observed.

The system of linear interval equations can be presented as follows:

$$[\mathbf{A}][\mathbf{x}] = [\mathbf{b}], \tag{1}$$

where $[\mathbf{A}]$ is an interval matrix, $[\mathbf{b}]$ is an interval vector and $[\mathbf{x}]$ is an interval solution vector. Generally such a system has no exact solutions. However, there are some methods for approximate solution of Eq. (1) presented in the literature. The undesirable feature of known approaches to the solution of Eq. (1) is the so-called excess width effect, i.e., the fast increasing the width of resulting intervals in calculations with the use of conventional interval arithmetic. The so-called "interval extended zero" method developed in [13,14] makes it possible to reduce this effect. In [12], it was proved that this method can be treated as the modified interval division (MID) and used to solve interval linear systems.

In the current paper, it is shown that this method can be used to provide effective computations on parallel machines using multithreaded model. The libraries OpenMP [9] and PThreads [4] are often used to perform parallel computations on parallel machines with shared memory. For example, the PLASMA package [1] uses the PThreads library to perform linear numerical algebra computations (including the solution of systems of linear equations) in parallel on real-valued vectors and matrices. However, we did not find implementations of solving interval linear systems using parallel libraries presented in the literature.

The aim of this work is to present the solution of interval linear systems implemented with the use of "interval extended zero" method and parallel multithreaded model on parallel architectures with shared memory.

The rest of paper is set out as follows. Section 2 presents how to increase the efficiency of interval calculations. In section 3, the parallel implementation of interval block Gaussian elimination algorithm is presented. Section 4 concludes with some remarks.

2 Efficient Implementation of Interval Arithmetic

One of the important issues concerning with the implementation of interval arithmetic operations is the need of instantly switching the rounding mode to perform computations on interval bounds. Normally, when calculating left bounds we have to use rounding to minus infinity and when calculating the right bounds - rounding to plus infinity:

$$\begin{aligned}[x] \circ [y] = \big[&\triangledown \left(\min\left(\underline{x} \circ \underline{y}, \overline{x} \circ \underline{y}, \underline{x} \circ \overline{y}, \overline{x} \circ \overline{y}\right)\right), \\ &\triangle \left(\max\left(\underline{x} \circ \underline{y}, \overline{x} \circ \underline{y}, \underline{x} \circ \overline{y}, \overline{x} \circ \overline{y}\right)\right)\big],\end{aligned} \tag{2}$$

where $\triangledown$ corresponds to the rounding down operation (to minus infinity), $\triangle$ - to the rounding up operation (to plus infinity) and $\circ \in \{+, -, \cdot, /\}$. Switching the rounding mode is a very costly operation as it flushes the pipeline and is effectively slowing down the calculations.

Goualard [15] proposed to keep intervals in the memory with the left bound having negative sign: $\langle \overline{a} : -\underline{a} \rangle$, and use only one rounding mode (to plus infinity) to perform all calculations on intervals. For example, the addition operation is defined as follows:

$$[x] + [y] = \langle \overline{x} : -\underline{x} \rangle + \langle \overline{y} : -\underline{y} \rangle = \langle - \triangle \left((-\underline{x}) + \left(-\underline{y}\right)\right), \triangle \left(\overline{x} + \overline{y}\right) \rangle. \tag{3}$$

The results obtained by Goualard [15] proved the efficiency of this approach. Therefore it is used in the current work to implement all operations on intervals.

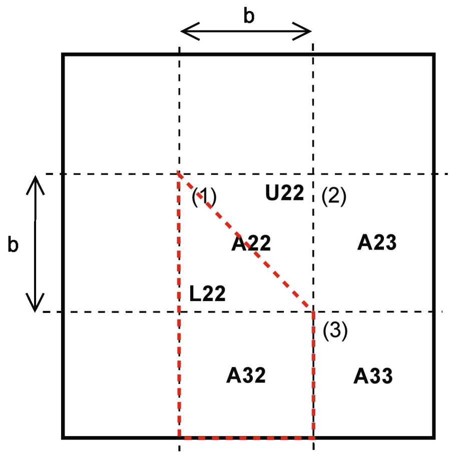


Fig. 1. Block Gaussian elimination algorithm scheme

3 Solving Interval Linear Systems Using "Interval Extended Zero" Method

The Block Gaussian elimination algorithm [11] was chosen to solve the system of interval linear equations presented on Eq. (1). This algorithm contains two stages - forward elimination and backward substitution. Let us assume (Fig. 1) that we have a square interval matrix divided into 9 blocks (3×3). The forward elimination stage is splitted out into three steps:

1. factorization
$$\begin{bmatrix} \mathbf{L}_{22} \\ \mathbf{A}_{32} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{22} \\ \mathbf{A}_{32} \end{bmatrix} \cdot \mathbf{U}_{22}, \tag{4}$$
using:
$$[s] = \frac{[a_{ik}]^{(k)}}{[a_{kk}]^{(k)}}, \tag{5}$$
$$[a_{ij}]^{(k+1)} = [a_{ij}]^{(k)} - [s] \cdot [a_{kj}]^{(k)}, \tag{6}$$
2. solving a triangular linear system of equations with many right-hand sides:
$$\mathbf{A}_{23} = \mathbf{L}_{22}^{-1} \cdot \mathbf{A}_{23}, \tag{7}$$
3. calculating a matrix-matrix multiplication:
$$\mathbf{A}_{33} = \mathbf{A}_{33} - \mathbf{A}_{32} \cdot \mathbf{A}_{23}. \tag{8}$$

Here $\mathbf{L}_{22}$ and $\mathbf{U}_{22}$ are respectively the lower and the upper triangular submatrices of block $\mathbf{A}_{22}$, $[a_{kk}]^{(k)} \neq 0, (k = 1, 2, \ldots, n-1; i, j = k+1, k+2, \ldots, n)$;

$[a_{ij}]$ are the elements of the interval matrix $[\mathbf{A}]$. This algorithm forms the upper triangular matrix. The system with such a matrix can be solved using the backward substitution:

$$[s] = \sum_{j=i}^{j=n} [a_{ij}] \cdot [x_j], \tag{9}$$

$$[x_i] = \frac{([b_i] - [s])}{[a_{ii}]}, \tag{10}$$

where $i = n-1, n-2, \ldots, 0$, $j = i, i+1, \ldots, n$, $[a_{ii}] \neq 0$, b_i are the elements of the vector $[\mathbf{b}]$, and $[x_i]$ are the elements of interval result vector $[\mathbf{x}]$.

The block Gaussian elimination algorithm is used to solve linear systems of equations in numerical algebra packages like LAPACK [2], ScaLAPACK [5] or PLASMA [1]. Its efficiency is very high due to the effective use of the processors cache memory. Blocks of the matrix can be placed in the cache which reduces the data movement between processor and the memory.

In work [13], the interval division operations in Eqs. (5) and (10) were replaced with the modified interval division (MID) based on the "interval extension zero" method. This method allows us to obtain much narrower interval results than those obtained using classic interval division [19]. It was proved in [12] that this method may be used to solve the systems of interval linear equations.

Two implementations of the interval block Gaussian elimination algorithm were developed and tested - usual block Gaussian elimination algorithm (*UBGEA*), using classic interval division in Eqs. (5) and (10), and modified one (*MBGEA*) using MID in Eqs. (5) and (10).

The developed implementations were analyzed with the use of three examples of interval linear systems. As an illustrative example, the Leontief's input-output model of economy [18] was used. This model can be presented as follows:

$$(\mathbf{I} - \mathbf{A}')\mathbf{x} = \mathbf{b}, \tag{11}$$

where $\mathbf{I}$ is the identity matrix, $\mathbf{A}'$ is the technical coefficient matrix, $\mathbf{x}$ is the global production vector, and $\mathbf{b}$ is the final product vector. This model is used mostly in economics for the production prognosis. Denoting $(\mathbf{I} - \mathbf{A}')$ as $\mathbf{A}$, the linear system (11) can be solved using the block Gaussian elimination algorithm. The problem is that usually the elements of $\mathbf{A}'$ and $\mathbf{b}$ may be presented only approximately as intervals or fuzzy values.

Three examples of economic system having 2000, 4000 and 6000 sectors, respectively, were build using randomly generated interval data. The sum of the elements in columns of the coefficient matrix could not be greater than 1, therefore they were generated in the range $0 \div 1/n$, where n is the number of rows in the matrix. The elements of final production vector $\mathbf{b}$ were randomly generated in the range $0 \div 2500000$. Finally, the interval bounds were obtained as follows: $\underline{a}'_{ij} = a'_{ij} - 0.1 \cdot a'_{ij}$, $\overline{a}'_{ij} = a'_{ij} + 0.1 \cdot a'_{ij}$.

To estimate the quality of obtained results, we propose the special relative index of uncertainty, RIU_{out} . It may serve as a quantitative measure of the excess width effect. It was calculated on resulting vectors as a maximal value from all elements: $RIU_{out} = \max((x_m - \underline{x}), (\overline{x} - x_m))/x_m \cdot 100\%$, where $x_m = \frac{\underline{x}+\overline{x}}{2}$. The RIU_{in} index calculated for the input data (coefficient matrix and final production vector) is equal to 10%.

The tests were carried out for three block sizes: 32×32 ($bl = 32$), 64×64 ($bl = 64$) and 128×128 ($bl = 128$) on a machine with 4 AMD Dual-Core Opteron 2.2GHz processors. Three interval libraries were used for comparison - RealLib [17], Boost [6] and Profil/BIAS [16]. Both the Boost and Profil/BIAS libraries switches the rounding mode during computations of interval bounds (to minus infinity for the left bounds and plus infinity for the right bounds). The RealLib library uses the so-called multiplicative rounding, where interval bounds are multiplied by $pred(1) = min\{y|y \in R, y < 1\}$ or $succ(1) = min\{y|y \in R, y > 1\}$ accordingly to their signs, and the rounding mode is set to the nearest for all calculations. All these libraries use the classic interval division method. The block Guassian elimination algorithm was implemented using all of these libraries.

The $UBGEA$ implementation uses the classic interval division, as well as the tested libraries (RealLib, Profil/BIAS and Boost - which are presented in Table 1 in columns RL, $BIAS$ and $Boost$ respectively). The values of RIU_{out} for these implementations (calculated on resulting interval vectors) are the same for respective systems, and are greater than the values obtained for the $MBGEA$ implementation using MID (Table 1). In all cases, the average width of the resulting intervals calculated with the use of classic interval division is practically two times greater than that obtained using MID. The size of block does not affect the width of the resulting intervals.

Table 1. RIU_{out} values [%] and calculation times [s] for interval block Gaussian elimination algorithm

n	bl	$UBGEA$		$MBGEA$		RL		$BIAS$		$Boost$	
		RIU_{out}	$time$	RIU_{out}	$time$	RIU_{out}	$time$	RIU_{out}	$time$	RIU_{out}	$time$
2000	32	29.234	20.948	14.475	20.938	29.234	77.367	29.234	335.228	29.234	342.997
	64	29.234	20.790	14.475	20.529	29.234	75.342	29.234	327.979	29.234	342.300
	128	29.234	28.089	14.475	26.783	29.234	85.344	29.234	318.886	29.234	331.889
4000	32	29.237	169.237	14.218	167.905	29.237	620.735	29.237	2637.313	29.237	2741.115
	64	29.237	164.872	14.218	163.648	29.237	600.465	29.237	2690.532	29.237	2719.020
	128	29.237	220.659	14.218	213.484	29.237	679.521	29.237	2593.070	29.237	2693.917
6000	32	29.248	570.022	14.287	567.828	29.248	2097.512	29.248	9564.101	29.248	9449.121
	64	29.248	554.119	14.287	550.384	29.248	871.954	29.248	2023.964	29.248	9065.829
	128	29.248	734.700	14.287	714.476	29.248	2281.642	29.248	8771.265	29.248	8990.122

Among the tested implementations the $UBGEA$ and $MBGEA$ variants were fastest. The RealLib library was 3 times slower while BIAS and Boost libraries were more than 10 times slower for the biggest matrix. The low efficiency of the Profil/BIAS and Boost libraries is caused mostly by the need for instantly switching the rounding mode during calculations of interval bounds. It is worth noting that both $UBGEA$ and $MBGEA$ implementations are practically evenly effective. Hence, the use of MID method does not affect the speed of the calculations in the block Gaussian elimination algorithm. The best results were obtained for the block size $bl = 64$, although the Profil/BIAS and Boost libraries performed fastest computations for the block size $bl = 128$ in all cases.

4 Parallel Implementation of the Interval Block Gaussian Elimination Algorithm

The block Gaussian elimination algorithm can be effectively parallelized using the parallel multithreaded model, with the assignment of blocks of matrix to different cores or processors. Operations performed inside blocks may be represented as tasks executed by threads. The tasks that have no dependencies between them can be executed in parallel, which effectively speeds up the computations. An example of tasks assignment on a four-core machine is presented in Fig. 2.

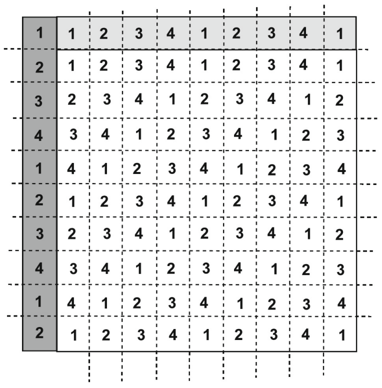


Fig. 2. Exemplary tasks assignment for the interval block Gaussian elimination algorithm on a four-core machine (for the first iteration of the algorithm)

Tasks are performed on different cores. Dark grey boxes correspond to the first step of the algorithm, light grey boxes - to the second step, and white boxes - to the third step. Steps need to be executed sequentially, however, operations performed within the steps of the algorithm can be parallelized as there are no data dependencies between the blocks. We can note that the third step (matrix-matrix multiplication) takes most of the time, more than 90% of all calculations.

For the implementation of the presented algorithm two parallel environments have been used - OpenMP [9] and SMP Superscalar [20].

4.1 OpenMP and SMP Superscalar Parallel Environments

OpenMP [9] is currently the most popular environment for parallel programming on machines with shared memory. It uses the fork-join scheme [9]. When a sequential program reaches its parallel section (specified by the programmer), an appropriate number of threads are created, and this section is executed by each thread. At the end of the parallel section threads are joined. The programmer must take care of all the dependencies between data that threads work on, so that they will not block each other, or use the same resource at the same time. The most common use of the OpenMP is the loops parallelization. Loops are splitted out into chunks that are assigned alternatingly to threads. The programmer can specify the size of chunks and the way they are assigned to threads.

SMP Superscalar [20] is a fairly new environment, based on the function parallelism. It uses tasks to perform calculations in parallel. The biggest difference between OpenMP and SMP Superscalar is that here, the runtime environment takes care of all data dependencies and synchronizations.

At runtime, a dependency graph is created, where tasks are represented by nodes and arcs corresponding to data dependencies between them [3]. If a task needs some data from another task, a thread that executes it waits for that data to be ready. If another task is ready to be executed at the same time (all dependencies are calculated), the scheduler will assign this task to a free thread. The main advantage of this model in comparison to OpenMP is that in OpenMP threads are synchronized at the end of every parallel section, which slows down the computations. In the SMP Superscalar, synchronizations are done only when needed (data dependencies). Moreover, programmers can synchronize the threads that operate only on a particular data.

4.2 Interval Block Gaussian Elimination Algorithm Implementation Using OpenMP and SMP Superscalar

The interval block Gaussian elimination algorithm based on MID method (denoted as $MBGEA$) was implemented and tested using both the OpenMP and SMP Superscalar environments on the same machine as in the previous examples. The input matrix was splitted out into blocks as shown in Fig. 2. Three steps of the algorithm were calculated in different loops. These loops have been parallelized in OpenMP using `#pragma omp parallel for` construction. Threads execute their chunks of the loops in parallel. For the SMP Superscalar environment, appropriate tasks performing block operations have been created using `#pragma smp task` construction. These tasks are assigned by the scheduler to threads, with the consideration of all data dependencies between the blocks.

The tests have been performed using data from previous example on the same machine containing 4 AMD Dual-Core Opteron 2.2GHz processors. The obtained results are presented in Table 2 for different numbers p of cores/processors (equal to the number of created threads). Since our machine have four double-core processors, maximum eight threads could be created. In the table, the speedup S_p is calculated as T_1/T_p, where T_1 - execution time of the algorithm on a single processor, T_p - execution time of the algorithm on p processors, whereas the efficiency E_p is calculated as S_p/p. The ideal value of $E_p = 1$ means that an algorithm has been perfectly parallelized.

Table 2. Results obtained using OpenMP and SMP Superscalar environments

		OpenMP				*SMP Superscalar*			
n	p	RIU_{out} [%]	*time [s]*	S_p	E_p	RIU_{out} [%]	*time [s]*	S_p	E_p
2000	1	14.475	20.348	-	-	14.475	20.131	-	-
	2	14.475	10.279	1.980	0.990	14.475	10.103	1.993	0.996
	4	14.475	5.181	3.928	0.982	14.475	5.100	3.947	0.987
	8	14.475	2.706	7.519	0.940	14.475	2.582	7.796	0.975
4000	1	14.218	162.557	-	-	14.218	160.757	-	-
	2	14.218	81.773	1.988	0.994	14.218	80.573	1.995	0.998
	4	14.218	41.033	3.962	0.990	14.218	40.566	3.963	0.991
	8	14.218	20.767	7.828	0.978	14.218	20.380	7.888	0.986
6000	1	14.287	549.060	-	-	14.287	542.287	-	-
	2	14.287	275.707	1.991	0.996	14.287	272.076	1.993	0.997
	4	14.287	138.189	3.973	0.993	14.287	136.705	3.967	0.992
	8	14.287	69.575	7.892	0.986	14.287	68.758	7.887	0.986

The tests were performed for 1, 2, 4 and 8 threads. Since the best sequential results were obtained for the block size $bl = 64$, it have been used here. The obtained values of RIU_{out} are exactly the same as in the case of sequential algorithms. We can note, that the SMP Superscalar environment was slightly faster than the OpenMP. As mentioned before, the SMP Superscalar scheduler manages the threads better, and this enhances the efficiency of calculations.

5 Conclusions

A method to solve systems of interval linear equations based on the concepts of "interval extended zero" method on parallel machines was developed, using the parallel multithreaded model. It was shown that this method not only allows us to reduce the undesirable excess width effect, but can also be used to perform efficient calculations in parallel. The modifications of interval arithmetic proposed by Goualard were used here to improve the efficiency of calculations on interval bounds. The block Gaussian elimination algorithm was utilized to implement the method for solving the Leontief's input-output model of economy. Three libraries were applied for the comparison - RealLib, Profil/BIAS and Boost. It was proved that the developed implementations give not only better (narrower) interval results, but also perform all calculations faster. Moreover, the "interval extended zero" method did not affect the speed of calculations in the block Gaussian elimination algorithm in comparison with the classic interval division method. The interval block Gaussian elimination algorithm was parallelized with the use of two parallel environments - OpenMP and SMP Superscalar, and tested on a machine with four dual-core processors. The obtained values of efficiency E_p, no less than 0.94, prove that the developed method can be efficiently parallelized using the multithreaded programming model.

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. Journal of Physics: Conference Series 180 (2009)
2. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J.W., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. SIAM (1992)
3. Badia, R.M., Labarta, J., Perez, J.M.: A Flexible and Portable Programming Model for SMP and Multi-cores. Tech. Report, Barcelona Supercomputing Center (2007), http://www.bsc.es/media/994.pdf
4. Berg, D., Lewis, B.: PThreads Primer A Guide to Multithreaded Programming. Prentice Hall PTR (1996)
5. Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, J., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. SIAM (1997)
6. Boost Interval Arithmetic Library, http://www.boost.org/doc/libs/1_43_0/libs/numeric/interval/doc/interval.htm
7. Buckley, J.J.: The fuzzy mathematics of finance. Fuzzy Sets & Systems 21, 257–273 (1987)
8. Buckley, J.J.: Solving fuzzy equations in economics and finance. Fuzzy Sets & Systems 48 (1992)
9. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press (2007)
10. Chen, S.H., Yang, X.W.: Interval finite element method for beam structures. Finite Elements in Analysis and Design 34, 75–88 (2000)
11. Demmel, J.W.: Applied Numerical Linear Algebra. SIAM (1997)
12. Dymova, L., Pilarek, M., Wyrzykowski, R.: Solving Systems of Interval Linear Equations with Use of Modified Interval Division Procedure. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6068, pp. 427–435. Springer, Heidelberg (2010)
13. Sevastjanov, P.V., Dymova, L.: Fuzzy Solution of Interval Linear Equations. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 1392–1399. Springer, Heidelberg (2008)
14. Dymova, L., Sevastjanov, P.: A new method for solving interval and fuzzy equations: linear case. Information Sciences 17, 925–937 (2009)
15. Goualard, F.: Fast and Correct SIMD Algorithms for Interval Arithmetic. INRIA (2008), http://hal.archives-ouvertes.fr/docs/00/28/84/56/PDF/intervals-sse2-long-paper.pdf
16. Knuppel, O.: Profil/BIAS v2.0, http://www.ti3.tu-harburg.de/keil/profil/Profil2.ps.gz
17. Lambov, B.: RealLib 3 Manual, http://www.brics.dk/~barnie/RealLib/RealLib.pdf
18. Leontief, W.: Quantitative input-output relations in the economic system of the United States. Review of Economics and Statistics 18, 100–125 (1936)
19. Moore, R.E.: Interval Arithmetic and Automatic Error Analysis in Digital Computing. PhD thesis, Stanford University (1962)
20. SMP Superscalar (SMPSs) User's Manual. Barcelona Supercomputing Center (2009), http://www.bsc.es/media/3576.pdf
21. Wu, C.C., Chang, N.B.: Grey input-output analysis and its application for environmental cost allocation. European Journal of Operational Research 145, 175–201 (2003)

GPU-Based Parallel Algorithms for Transformations of Quantum States Expressed as Vectors and Density Matrices

Marek Sawerwain

Institute of Control & Computation Engineering,
University of Zielona Góra, ul. Podgórna 50, Zielona Góra 65-246, Poland
M.Sawerwain@issi.uz.zgora.pl

Abstract. In this paper the parallel algorithms to simulate measurements and unitary operations in the quantum computing model are presented. The proposed solutions are implemented using the CUDA technology. Moreover, a more effective routine for processing density matrices is presented. The main advantages of this approach are the reduced memory requirement and a good use of the computing power of the GPU hardware. Additionally, the proposed solution can be used to simulate circuits build from qubits or qudits.

Keywords: GPGPU, CUDA, parallel computations, quantum computations.

1 Introduction

In spite of a ceaseless development of the quantum information science (fundamental information about the quantum computations can be found in books [7], and in the paper [2]) and a significant effort of researchers, the hardware to perform a quantum computations still exists only in a form of laboratory experiments. Unfortunately, the size of quantum data is exponential and generally the computational complexity to perform simulations is denoted as $O(2^n)$.

Currently, in most cases simulations of a quantum computation model are done by direct application of mathematical definitions e.g. [11]. The main advantage of such an approach is easier access to wide set of mathematical tools which are necessary to examine the simulated quantum system (and many parallel programming techniques can be directly applied to simulations of the quantum computation models e.g. [8], [10]). The most used ones are fidelity measure or complete positive maps [1] which are used to model the quantum noise. The direct approach allows also to use e.g. advanced methods of detection of entanglement (this notion does not appear in the classical theory of computer science). The main drawback is exponential computational complexity. Therefore, any additional solution (especially parallel, and for a wide review of numerical computations, see e.g. [6], [4]) which reduces the computational complexity is very useful.

This paper shows the methods which can be regarded as a special case of the approaches presented in works [9], [3] and [5]. This contribution concentrates on the simulation of quantum systems with and without the noise presence. This article is organised as follows. In section 2 the basic information about the quantum computation model directly connected with discussed methods is given. In section 3 the proposed parallel algorithms of processing of state vector and density matrix are discussed. The performance results are given in section 4. Finally, section 5 summarises this contribution and points out some ideas for future research.

2 Basic Information about Quantum Computing Model

The definitions of a qubit (and it generalisation called qudit) as an unit of quantum information and a notion of quantum register (a sequence of such units) are key elements of the quantum computations model. The state of any unknown pure qubit (using the Dirac's notation) can be depicted in the following way:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \quad \alpha, \beta \in \mathbb{C} \quad \text{and} \quad |\alpha|^2 + |\beta|^2 = 1, \tag{1}$$

where $\mathbb{C}$ is a set of a complex numbers and vectors $|0\rangle$, $|1\rangle$ are defined as:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \tag{2}$$

It must be stressed that vectors given in the equation (2) forms the standard computational base for the qubit.

A quantum register is a system formed form n qubits/qudits. Such a system is called a state vector:

$$|\Psi\rangle = |\psi_0\rangle \otimes |\psi_1\rangle \otimes \ldots \otimes |\psi_{n-1}\rangle. \tag{3}$$

Individual elements of a state vector $|\Psi\rangle$ are called the probability amplitudes.

Remark 1. It must be emphasised that there exist cases where the state vector can not be expressed by a tensor product of sequence of qubits or qudits. In such situations a state is called an entangled state.

The dimension of a state vector for a qubit case ($d = 2$) is equal to 2^n, whereas for a more general qudit case ($d > 2$) it equals d^n.

The representation of a density matrix of an unknown pure state of qubit $|\psi\rangle$ is denoted by formula (A) in the equation given below. The representation of a quantum register composed from density matrices is given by formula (B):

$$(A): \quad \rho = |\psi\rangle\langle\psi| = \begin{bmatrix} \alpha^2 & \alpha\beta \\ \alpha\beta & \beta^2 \end{bmatrix}, \quad (B): \quad \rho = \sum_i \lambda_i |\psi_i\rangle\langle\psi_i|. \tag{4}$$

Additionally, the vector $\langle\psi|$ represents the transposed vector $|\psi\rangle$ and the probability for state $|\psi_i\rangle$ is denoted by λ_i (obviously, $\sum_i \lambda_i = 1$). In quantum mechanics such states are called mixed states.

The unitary operators (denoted by U) are one of the basic operations which can act on the quantum registers. Each unitary operation U always has a reverse operation $U^\dagger$, what is a direct consequence of unitarity of U. The application of U for state expressed as the state vector (equation (A) below) and density matrix (equation (B)) are governed by simple formulas:

$$(A): \quad U|\psi_0\rangle = |\psi_1\rangle, \qquad (B): \quad U\rho_0 U^\dagger = \rho_1. \tag{5}$$

These operations are very computationally expensive. The density matrices require two additional multiplications of matrices with size $2^n \times 2^n$.

The essential problem is the process of building the operation matrix U from the set of smaller unitary matrices acting only on a given qubits/qudits of the quantum register. If we want to apply two different unitary operations u_1 and u_2 to the first and third qudit of the quantum register $|\psi\rangle$, one approach is to act on $|\psi\rangle$ with the following tensor product:

$$U = u_1 \otimes I \otimes u_2 \tag{6}$$

where I represents the identity matrix. The operation (6) is however very computationally expensive. A much better option is to utilise the algorithm given in section 3, which avoids the necessity of calculating U altogether.

The second type of operation which can be applied to the quantum register is the measurement procedure. Here we only concentrate on the most commonly used one, termed the von Neumann measurement.

The realisation of von Neumann measurement begins with the preparation of an observable M, calculated using formula (A) of equation (7):

$$(A): \quad M = \sum_i \lambda_i P_i, \qquad (B): \quad p(\lambda_i) = \langle\psi|P_i|\psi\rangle, \quad (C): \quad \frac{P_i|\psi\rangle}{\sqrt{\lambda_i}} \tag{7}$$

where P_i is the projector of eigenspace of an operator M. The results of the measurement are represented by eigenvalues λ_i. The results are given only with some probability function P — see formula (B) of equation (7). The obtained result λ_i means that the register $|\psi\rangle$ is collapsed to the state given by the formula (C) of the before mentioned equation.

3 Algorithms for Parallel Processing of Quantum States

The algorithms for parallel processing of quantum states outlined in this work are based on executing a special algorithm (which are discussed in detail in section 3.2) on a number of sub-marices, cut out of a unitary operation matrix U. First, the following invariants for an algorithm of selection of the before mentioned sub-matrices must be given:

Proposition 1. *Let* n *denote the size of a quantum register,* t *— the number of modifying qudits and* d *— the level of a qudit, then:*

(I) *the number of sub matrices is given by* $\mathrm{m} = \mathrm{d}^{\mathrm{n}-1}$*,*
(II) *the value of step between elements of small matrices is as follows* $\mathrm{step} = \mathrm{d}^{\mathrm{n}-\mathrm{t}-1}$*,*
(III) *the number of blocks equals to* $\mathrm{p} = \mathrm{d}^{\mathrm{t}-1}$*,*
(IV) *the distance or the size of a single block is denoted as* $\mathrm{vstep} = \frac{\mathrm{d}^{\mathrm{n}}}{\mathrm{p}}$*.*

3.1 The Measurement of Quantum States as Parallel Reduction Primitive

The von Neumann measurement can be implemented by reduction operation if the measurement is performed in the standard base (in many currently known quantum algorithms the measurements are performed in the standard base). Figure (1) shows the situation where information about probability is collected using the reduction operation. The fast realisation of this operation is possible, as each element of the state vector can be processed independently.

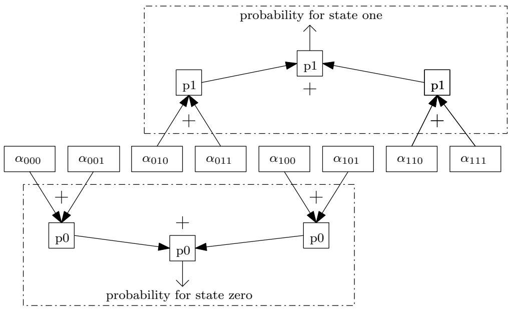


Fig. 1. Gathering information about probability for a three qubits quantum register. It is assumed that the second qubit is measured.

Algorithm 1 shows the pseudo-code for realisation of the quantum measurement. The implementation of this algorithm on the GPU can be realised directly using a reduction parallel primitive. This can be accomplished using several well-known approaches.

In section 4 the timing for the computational kernel where the sequential addressing is used is presented. Additionally, the results when the kernel loops are unrolled and threads process multiple elements (using the coalesced access to the memory) to improve the final performance are also shown. It must be stressed that the improvement of performance is achieved (at least partially) due to the proper use of the shared memory.

Algorithm 1. The computation routine for von Neumann measurement, parameter $qReg$ represents the quantum register, fd denotes the level of qudit, n denotes number of qudits, m_i represents mask of qudits which will be measured, the expression $i \models m_j$ means that index i fulfils the measurement mask m_j

```
procedure QMI( $qReg$, fd, $n$, $m_0$, ..., $m_{\mathrm{fd}}$ )
    $size \leftarrow fd^n$
    for $i \leftarrow 0; i < size; i{+}{+}$ do in parallel
        $f \leftarrow |qReg[i]|^2$
        if $(i \models m_0) = \mathrm{true}$ then $p_0^i \leftarrow p_0^i + f$
        if $(i \models m_1) = \mathrm{true}$ then $p_1^i \leftarrow p_1^i + f$
        $\ldots$
        if $(i \models m_{\mathrm{fd}}) = \mathrm{true}$ then $p_{\mathrm{fd}}^i \leftarrow p_{\mathrm{fd}}^i + f$
    end for
    $\mathrm{r}, \mathrm{m} \leftarrow$ randomly select from $(p_0^i, m_0), (p_1^i, m_1), \ldots, (p_{\mathrm{fd}}^i, m_{\mathrm{fd}})$
    for $i \leftarrow 0; i < size; i{+}{+}$ do in parallel
        zero $qReg[i]$ if not $i \models m$
        if $(|qReg[i]|! = 0)$ $qReg[i] \leftarrow \frac{1}{r} qReg[i]$
    end for
end procedure
```

3.2 Unitary Transformations for States Vectors and Density Matrices

The algorithm 2 shows how the values obtained using proposition 1 are used for addressing the elements in the space of a state vector. The two for-like loops are a key ingredients of the presented solution. Indices generated by these loops are used to select the independently-processed elements of a state vector. For the implementation using the CUDA technology, the before mentioned for-like loops are substituted by the computation grid. Obviously, the configuration of the grid (the number of blocks and threads in the blocks) is critical for obtaining a good performance results.

Algorithm 2. The processing of the state vector for a one-qudit gate case. Parameter qReg denoted the quantum register, fd represents the level of qudit, n denotes the number of qudits, t is an index of the modified qudit and u represents an unitary matrix

```
procedure VST( qReg, fd, $n$, $t$, $u$ )
    $m \leftarrow \mathrm{fd}^{n-1}$ ; $step \leftarrow \mathrm{fd}^{n-t} - 1$ ; $p \leftarrow \mathrm{fd}^{t-1}$ ; $vstep \leftarrow (\mathrm{fd}^n)/p$ ; $irow \leftarrow 0$
    for $ip \leftarrow 0; ip < p; ip{+}{+}$ do
        for $i \leftarrow 0; i < \mathrm{step} + 1; i{+}{+}$ do in parallel
            $r_1 \leftarrow irow + 1 * (step + 1) + i$
            $r_2 \leftarrow irow + 2 * (step + 1) + i$
            $\ldots$
            $r_k \leftarrow irow + \mathrm{fd} * (step + 1) + i$
            OperOnRows(qReg, [$r_1$, $r_2$, ..., $r_k$], u)
        end for
        $irow \leftarrow irow + vstep$
    end for
end procedure
```

Nevertheless, after computing the indices, the transformation of probabilities is performed using a user defined function called OperOnRows. The idea of OperOnRows is easily depicted by the following equation:

$$\begin{bmatrix} \mathrm{qReg}[r_1] \\ \mathrm{qReg}[r_2] \\ \vdots \\ \mathrm{qReg}[r_d] \end{bmatrix} \leftarrow \begin{bmatrix} u_{11} & \cdots\cdots & u_{1d} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ u_{d1} & \cdots\cdots & u_{dd} \end{bmatrix} \begin{bmatrix} \mathrm{qReg}[r_1] \\ \mathrm{qReg}[r_2] \\ \vdots \\ \mathrm{qReg}[r_d] \end{bmatrix} \tag{8}$$

where parameter *qReg* represents the quantum register, r is a set of indices generated by the algorithm 2 and u is a suitable unitary matrix with the dimensions $d \times d$, where d represents the level of qudit. The OperOnRows routine is executed in parallel, as the elements from the state vector can be processed independently. This is shown on figure 2. Similar to the case of a state vector

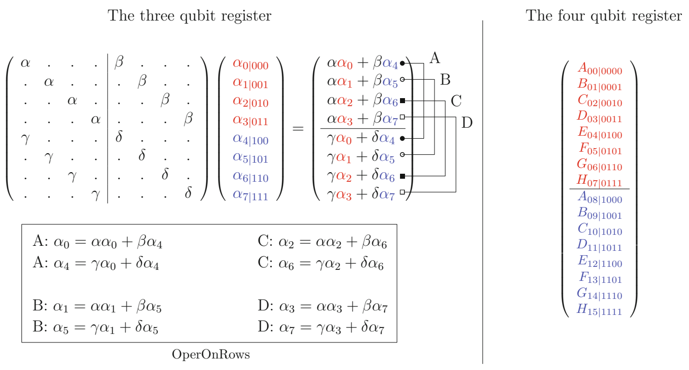

The three qubit register
The four qubit register
A
B
C
D
A: $\alpha_0 = \alpha\alpha_0 + \beta\alpha_4$
A: $\alpha_4 = \gamma\alpha_0 + \delta\alpha_4$
B: $\alpha_1 = \alpha\alpha_1 + \beta\alpha_5$
B: $\alpha_5 = \gamma\alpha_1 + \delta\alpha_5$
C: $\alpha_2 = \alpha\alpha_2 + \beta\alpha_6$
C: $\alpha_6 = \gamma\alpha_2 + \delta\alpha_6$
D: $\alpha_3 = \alpha\alpha_3 + \beta\alpha_7$
D: $\alpha_7 = \gamma\alpha_3 + \delta\alpha_7$
OperOnRows

Fig. 2. The example of independent operations (A, B, C, D – for three qubit register and A, B, ..., H – for four qubit register) for the task of the state vector processing

using the values from proposition 1, the algorithm 3 for processing density matrices can be formulated. The transformation is performed in-place, without using any additional matrices. The processing of density matrices is also divided into independent operations which are depicted on figure 3.

The idea of OperOnMatrix routine is very similar to the routine OperOnRows:

$$\begin{bmatrix} dm_{x,y} & dm_{x+s+1,y} \\ dm_{x,y+s+1} & dm_{x+s+1,y+s+1} \end{bmatrix} \leftarrow \begin{bmatrix} u_1 & u_2 \\ u_3 & u_4 \end{bmatrix} \begin{bmatrix} dm_{x,y} & dm_{x+s+1,y} \\ dm_{x,y+s+1} & dm_{x+s+1,y+s+1} \end{bmatrix} \begin{bmatrix} u_1^\dagger & u_2^\dagger \\ u_3^\dagger & u_4^\dagger \end{bmatrix} \tag{9}$$

where the x, y values determines the row and column of the first element and s represents the step (distance) between elements (see figure 3).

4 Performance of Proposed Methods

The methods proposed in the previous sections belong to a family of methods which perform simulation of a quantum circuit on a gate-by-gate basis (similar to the ones described in [3], [9] and [5]). In every computational kernel the complex number is represented by the user-defined type. The implementation of

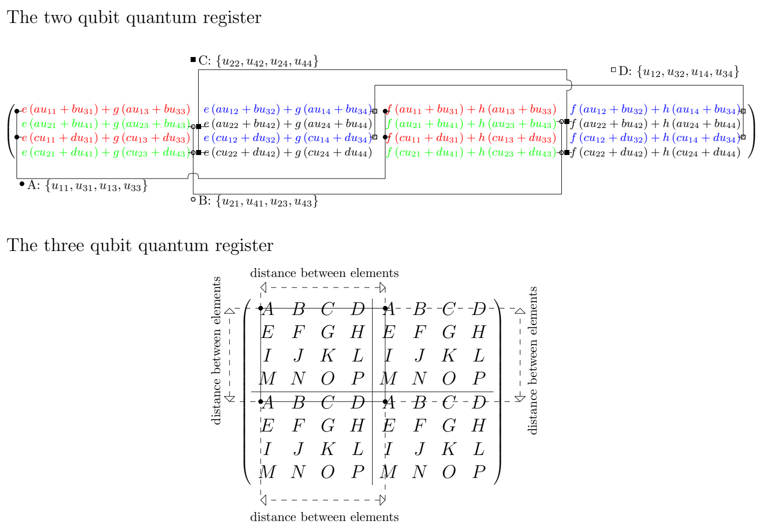

The two qubit quantum register
C: {u22, u42, u24, u44}
D: {u12, u32, u14, u34}
A: {u11, u31, u13, u33}
B: {u21, u41, u23, u43}
The three qubit quantum register
distance between elements
distance between elements
distance between elements
distance between elements

Fig. 3. The example of independent operations (four operations for two qubit denisty matrix, and sixteen operations for three qubit density matrix) in the processing task of the quantum state represented by the density matrix

complex numbers available in the CUDA SDK (types *cuFloatComplex* and *cuDoubleComplex*) reduces the performance due to a more complex implementation to preserve the numerical stability.

The performance results shown in Tables 1 and 2 refer to the simulation of the measurement of a single qubit. Two kernels are used in these benchmarks. The first kernel (columns name ending with *V1*) uses only a sequential addressing whereas the second kernel (columns name ending with *V2*) uses both the unroll loops optimisation technique and sequential addressing to processes several elements with each thread which increasing the final performance. The mentioned computational routines were also implemented and executed on a system with two Intel Xeon E5420 2.5 Ghz processor using eight computational threads (column *CPU MT*).

Algorithm 3. The processing of a density matrix for a one-qubit gate. The first parameter *dm* represents the density matrix and remaining parameters have the same meaning as in the algorithm 2

```
procedure DMT( $dm, n, t, u$ )
    $step = 2^{n-t} - 1$ ; $p = 2^{t-1}$
    $vstep = (2^n)/p$ ; $sizeReg = 2^n$
    for $i = 0; i < sizeReg; i+ = vstep$ do
        for $j = 0; j < sizeReg; j+ = vstep$ do
            for $x = 0; x < step + 1; x + +$ do
                for $y = 0; y < step + 1; y + +$ do
                    OperOnMatrix($(dm, t, i + x, j + y, step, u, u^\dagger)$);
                end for
            end for
        end for
    end for
end procedure
```

Table 1. The time and throughput for the measurement operation of the first qubit of a quantum register

No.	CPU MT		G280 V1		G470 V1		G280 V2		G470 V2	
qubits	Time in s	GB/s	Time in s	GB/s	Time in s	GB/s	Time in s	GB/s	Time in s	GB/s
17	0,005	0,21	0,0001	7,65	0,0002	5,09	0,0001	21,97	0,0001	11,00
18	0,005	0,44	0,0002	9,40	0,0002	8,86	0,0001	31,87	0,0001	20,30
19	0,005	0,83	0,0004	10,59	0,0003	11,73	0,0001	39,83	0,0001	33,66
20	0,008	1,04	0,0007	11,51	0,0005	16,27	0,0002	48,00	0,0002	48,32
21	0,012	1,40	0,0014	12,02	0,0009	19,77	0,0003	53,61	0,0003	64,97
22	0,022	1,42	0,0027	12,27	0,0015	22,36	0,0006	58,02	0,0004	80,63
23	0,046	1,45	0,0054	12,42	0,0028	23,63	0,0011	59,98	0,0007	90,49
24	0,085	1,57	0,0108	13,83	0,0055	24,54	0.0022	61,11	0,0014	96,51

The best results were obtained for the kernel V2, but the size of the computational grid and the number of threads per block are very important. In case of Geforce 280GT the number of threads per block is equal to 256 and for the Geforce 470GT it equals 512. However, the higher performance was obtained only for the quantum register containing 22, 23, 24 qubits which is a simple consequence of the configuration of computational grid optimised for a larger quantum register. The larger number of threads reduces the performance in case of newer GPUs, because these threads do not have enough data to process and the whole computational grid is always kept fully synchronised. A good performance was also obtained in case of density matrices. Table (3) shows the results of a test where the Hadamard operation was applied on each of the 13 qubits of a given quantum register. This test also shows the advantage of a newer GPUs called Fermi, with its improvement memory transfer. A slower sequential addressing algorithm run on the Fermi is faster than the optimised version of kernel V2 running on the Geforce 280GT card. The figure (4) shows the performance of a more practical example where the simulation of the Grover's algorithm (described in [7] and many others) is performed. The best speedup was obtained for the system containing twenty qubits. In this case the circuit contains exactly 485 gates, mainly single qubit gates and a few control gates. The configuration of the computational grid for the GPU computation kernels was the same as in previously discussed benchmarks.

Table 2. The time and throughput for a benchmark where the Hadamard gate is applied on each qubit of a quantum register

No.	CPU MT		G280 v1		G470 v1		G280 v2		G470 v2	
qubits	Time	GB/s	Time	GB/s	Time	GB/s	Time	GB/s	Time in s	GB/s
17	0,022	0,047	0,0012	16,03	0,0014	12,66	0,0006	33,90	0,0012	14,07
18	0,047	0,044	0,0021	17,80	0,0020	19,27	0,0010	37,69	0,0015	24,63
19	0,084	0,049	0,0040	19,57	0,0029	27,94	0,0021	39,87	0,0021	39,05
20	0,158	0,053	0,0081	21,72	0,0046	37,81	0,0041	42,98	0,0031	54,41
21	0,302	0,055	0,0165	22,42	0,0082	44,23	0,0096	44,56	0,0053	68,19
22	0,516	0,065	0,0339	22,79	0,0158	48,06	0,0169	45,12	0,0099	76,69
23	0,911	0,073	0,0763	22,91	0,0314	50,60	0,0382	45,78	0,0188	84,34
24	1,659	0,080	0,1734	23,34	0,0639	51,99	0,0955	46,16	0,0375	88,26

Table 3. The processing time of a density matrix (for a quantum register containing 13 qubits) on the Intel 86x64 architecture processors compared to the computational kernels V1 and V2 on a GPU. The speedup is calculated taking into account the best CPU result

	CPU 1th		CPU 2th		CPU 4th		CPU 8th	
Device	Time	GB/s	Time	GB/s	Time	GB/s	Time	GB/s
Intel Core 2 Duo E8400 3.0 Ghz	125.67	0,0043	74.68	0,0072	—	—	—	—
2 x Xeon E5420 2.5 Ghz	87,14	0,0062	46,83	0,011	14,57	0,0368	7,17	0,0748

	Kernel V1		Kernel V2		Speedup	
	Time	GB/s	Time	GB/s	V1	V2
G280	36,3021	0,20	2,90	2,159	0.19	2,47
G470	0,4978	14,42	0,14	49,064	14,41	51,21

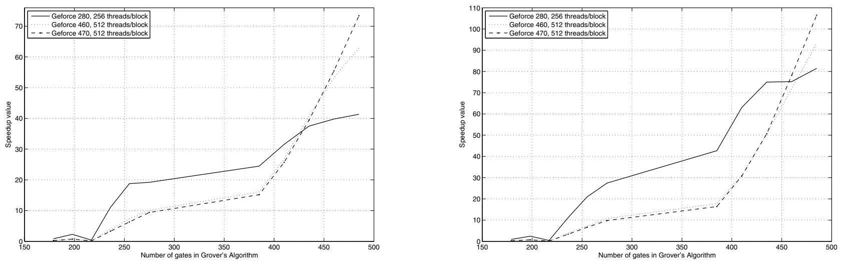


Fig. 4. The values of a speedup obtained for the simulation of Grover's algorithm. The speedup was calculated for the serial implementation executed on a single computational core of Intel Core i7 950 3.0 Ghz processor.

5 Conclusions and Further Work

The proposed methods show the significant speedup of the quantum data processing. The operation on the quantum register can be simulated on the GPU up to one hundred times faster than on a traditional single-core processor or fifty times faster than on a modern eight core CPU.

The presented methods are very flexible, as they can be used not only for systems built from qubits but also from qudits. Moreover, they can be used for transforming the density matrices.

Currently, development concentrates on adaptation of the binary diagrams (BD) techniques for simulating quantum systems (see [12]) on the GPU.

Acknowledgments. I would like to take this opportunity to thank Łukasz Hładowski and Roman Gielerak for useful comments and endless discussions. I would like to thank to anonymous referees for useful comments on the preliminary version of this paper. The numerical results and benchmarks were done using the hardware and software available at the "GPU Mini-Lab" located at the Institute of Control and Computation Engineering (ISSI) of the University of Zielona Góra, Poland.

References

1. Bengtsson, I., Życzkowski, K.: Geometry of Quantum States: An Introduction to Quantum Entanglement. Cambridge University Press (2006)
2. Feynman, R.P.: Simulating physics with computers. Int. J. Theoretical Physics 21(6/7), 467–488 (1982)
3. Glendinning, I., Ömer, B.: Parallelization of the QC-Lib Quantum Computer Simulator Library. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2004. LNCS, vol. 3019, pp. 461–468. Springer, Heidelberg (2004)
4. Guarracino, M.R., Perla, F., Zanetti, P.: A parallel block Lanczos algorithm and its implementation for the evaluation of some eigenvalues of large sparse symmetric matrices on multicomputers. Int. J. App. Math. and Comp. Sci. 16(2), 241–249 (2006)
5. Gutierrez, E., Romero, S., Trenas, M.A., Zapata, E.L.: Quantum computer simulation using the CUDA programming model. Computer Physics Communications 181(2), 283–300 (2010)
6. Hwu, W.W. (ed.): GPU Computing Gems Emerald Edition. Morgan Kaufmann Publishers (2011)
7. Nielsen, M.A., Chuang, I.L.: Quantum computation and quantum information. Cambridge University Press, New York (2000)
8. Niwa, J., Matsumoto, K., Imai, H.: General-purpose parallel simulator for quantum computing. Phys. Rev. A 66, 062317 (2002)
9. Sawerwain, M.: Parallel Algorithm for Simulation of Circuit and One-Way Quantum Computation Models. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 530–539. Springer, Heidelberg (2008)
10. Tabakin, F., Juliá-Díaz, B.: QCMPI: A Parallel Enviroment for Quantum Computing. Computer Physics Communications 180(6), 948–964 (2009)
11. Tóth, G.: QUBIT4MATLAB V3.0: A program package for quantum information science and quantum optics for MATLAB. Computer Physics Communications 179(6), 430–437 (2008)
12. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Quantum Circuit Simulation. Springer, Heidelberg (2009)

Generalizing Matrix Multiplication for Efficient Computations on Modern Computers

Stanislav G. Sedukhin[1] and Marcin Paprzycki[2]

[1] The University of Aizu, Aizuwakamatsu City, Fukushima 965-8580, Japan
sedukhin@u-aizu.ac.jp

[2] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland
marcin.paprzycki@ibspan.waw.pl

Abstract. Recent advances in computing allow taking new look at matrix multiplication, where the key ideas are: decreasing interest in recursion, development of processors with thousands (potentially millions) of processing units, and influences from the Algebraic Path Problems. In this context, we propose a generalized matrix-matrix multiply-add (MMA) operation and illustrate its usability. Furthermore, we elaborate the interrelation between this generalization and the BLAS standard.

Keywords: matrix matrix multiplication, algebraic semiring, path problem, fused multiply add, BLAS, matrix data manipulation.

1 Introduction

Dense matrix multiplication is widely used in solution of computational problems. Despite its simplicity, the arithmetic complexity and data dependencies make it difficult to reduce its *run-time complexity.* The two basic approaches to decrease the run-time of matrix multiplication are: (1) *reducing the number of scalar multiplications*, while increasing the number of scalar additions/subtractions (and introducing irregularity of data access, as well as need for extra memory; see, discussion in [1]), and (2) *parallel implementation* of matrix multiplication (see, for example [2] and references found there). Of course, a combination of these two approaches is also possible (see discussion and references in [3–5]).

The (recursive) matrix multiplication "worked well" in theoretical analysis of arithmetical complexity, and when implemented on early computers. However, its implementation started to became a problem on computers with hierarchical memory (e.g. to reach optimal performance of a Strassen-type algorithm, recursion had to be stopped when the size of divided submatrices approximated the size of cache memory—differing between machines; see, [6, 7]), which contradicts the very idea of recursion. Furthermore, practical implementation of Strassen-type algorithms requires extra memory (e.g. Cray's implementation Strassen's algorithm required extra space of order $2.34 \cdot n^2$). The situation became even more complex when parallel Strassen-type algorithms have been implemented [5]. Interestingly, the research in recursive matrix multiplication seems to be diminishing, with the last paper known to the authors' published in 2006.

In addition to the standard (linear-algebraic) matrix multiplication, a more *general* matrix multiplication appears as a kernel of algorithms solving the Algebraic Path Problem (APP). Here, the examples are: finding the all-pairs shortest paths, finding the most reliable paths, etc. In most of them, a generalized form of a $C \leftarrow C \oplus A \otimes B$ matrix operation plays a crucial role in finding the solution. This, generalized, matrix multiplication is based on the algebraic theory of semirings (for an overview of computational issues in the APP, and their relation to the theory of semirings, see [8], and references collected there). Note that standard linear algebra (with its matrix multiplication) is just one of the examples of algebraic (matrix) semirings.

While algebraic semirings can be seen as a simple "unification through generalization" of a large class of computational problems, they should be viewed in the context of ongoing changes in computer (processor) architectures. Specifically, the success of fused multiply-and-add (FMA) units, which take three scalar (in) operands and produce a single (out) result within a single clock cycle. Furthermore, GPU processors from Nvidia and AMD combine multiple FMA units (e.g. the Nvidia's Fermi chip allows 512 single-precision FMA operations completed in a single cycle; here, we omit issues concerning the data-feed bottleneck).

Our aim is to combine: (a) fast matrix multiplication, (b) mathematics of semirings, and (c) trends in computer hardware, to propose a generalized matrix multiplication, which can be used to develop efficient APP solvers.

2 Algebraic Semirings in Scientific Calculations

Since 1970's, a large number of problems has been combined under a single umbrella, named the *Algebraic Path Problem* (*APP*; see [9]). Furthermore, it was established that the matrix multiply-and-add (MMA) operations, in different algebraic semirings, are used as a centerpiece of various APP solvers.

A closed semiring $(S, \oplus, \otimes, *, \bar{0}, \bar{1})$ is an algebraic structure defined for a set S, with two binary operations: addition $\oplus : S \times S \rightarrow S$ and multiplication $\otimes : S \times S \rightarrow S$, a unary operation called *closure* $\circledast : S \rightarrow S$, and two constants $\bar{0}$ and $\bar{1}$ in S. Here, we are particularly interested in the set S consisting of matrices. Thus, following [9], we introduce a matrix semiring $(S^{n\times n}, \bigoplus, \bigotimes, \bigstar, \bar{O}, \bar{I})$ as a set of $n \times n$ matrices $S^{n\times n}$ over a closed scalar semiring $(S, \oplus, \otimes, *, \bar{0}, \bar{1})$ with two binary operations, matrix addition $\bigoplus : S^{n\times n} \times S^{n\times n} \rightarrow S^{n\times n}$ and matrix multiplication $\bigotimes : S^{n\times n} \times S^{n\times n} \rightarrow S^{n\times n}$, a unary operation called *closure of a matrix* $\bigstar : S^{n\times n} \rightarrow S^{n\times n}$, the zero $n \times n$ matrix $\bar{O}$ whose all elements equal to $\bar{0}$, and the $n \times n$ identity matrix $\bar{I}$ whose all main diagonal elements equal to $\bar{1}$ and $\bar{0}$ otherwise. Here, matrix addition and multiplication are defined as usually in linear algebra. Note that the case of rectangular matrices can be dealt with satisfactorily, and is omitted for clarity of presentation.

As stated, large number of matrix semirings appear in well-studied APP's. We summarize some of them in a table (similar to that presented in [10]). For simplicity of notation, in the Table 1, we represent them in the scalar form. Note that the *Minimum reliability path* problem has not been encountered by

Table 1. Semirings for various APP problems

S	$\oplus$	$\otimes$	$\circledast$	$\bar{0}$	$\bar{1}$	Application
$(0, 1)$	$\vee$	$\wedge$	1	0	1	Transitive and reflexive closure of binary relations
$\mathbb{R} \cup +\infty$	$+$	$\times$	$1/(1-r)$	0	1	Matrix inversion
$\mathbb{R}_+ \cup +\infty$	min	$+$	0	∞	0	All-pairs shortest paths problem
$\mathbb{R}_+ \cup -\infty$	max	$+$	0	-∞	0	Maximum cost (critical path)
$[0, 1]$	max	$\times$	1	0	1	Maximum reliability paths
$[0, 1]$	min	$\times$	1	0	1	Minimum reliability paths
$\mathbb{R}_+ \cup +\infty$	min	max	0	∞	0	Minimum spanning tree
$\mathbb{R}_+ \cup -\infty$	max	min	0	-∞	0	Maximum capacity paths

the authors before. It was defined on the basis of systematically representing possible semirings—as a natural counterpart to the *Maximum reliability problem* (the only difference is the $\oplus$ operation: *min* instead of *max*). Since the maximum reliability path defines the *best* way to travel between two vertices of a graph; the *Minimum reliability problem* could be interpreted as: finding the *worst* pathway, one that should not be "stepped into").

While Table 1 summarizes the *scalar* semirings, and *scalar* operations, kernels of blocked algorithms for solving the APP, are based on (block) MMA operations [11]. Therefore, let us present the relation between the scalar multiply-and-add operation (ω), and the corresponding MMA kernel (α), for semirings in Table 1 (here, Nb is the size of a matrix block; see, also [12]):

– Matrix Inversion Problem:
  (α) $a(i,j) = a(i,j) + \sum_{k=0}^{Nb-1} a(i,k) \times a(k,j)$;
  (ω) $c = a \times b + c$;
– All-Pairs Shortest Paths Problem:
  (α) $a(i,j) = \min\{a(i,j), \min_{k=0}^{Nb-1}[a(i,k) + a(k,j)]\}$;
  (ω) $c = \min(c, a + b)$;
– All-Pairs Longest (Critical) Paths Problem:
  (α) $a(i,j) = \max\{a(i,j), \max_{k=0}^{Nb-1}[a(i,k) + a(k,j)]\}$;
  (ω) $c = \max(c, a + b)$;
– Maximum Capacity Paths Problem:
  (α) $a(i,j) = \max\{a(i,j), \max_{k=0}^{Nb-1} \min[a(i,k), a(k,j)]\}$;
  (ω) $c = \max[c, \min(a, b)]$;
– Maximum Reliability Paths Problem:
  (α) $a(i,j) = \max\{a(i,j), \max_{k=0}^{Nb-1}[a(i,k) \times a(k,j)]\}$;
  (ω) $c = \max(c, a \times b)$;
– Minimum Reliability Paths Problem:
  (α) $a(i,j) = \min\{a(i,j), \min_{k=0}^{Nb-1}[a(i,k) \times a(k,j)]\}$;
  (ω) $c = \min(c, a \times b)$;
– Minimum Spanning Tree Problem:
  (α) $a(i,j) = \min\{a(i,j), \min_{k=0}^{Nb-1}[\max(a(i,k), a(k,j)]\}$;
  (ω) $c = \min[c, \max(a, b)]$.

Summarizing, the *generalized* MMA is one of the key operations for solving APP problems, and a large class of standard MMA-based numerical linear algebraic algorithms. Note that, this latter class includes most of block-oriented formulations of standard problems involving the level 3 BLAS operations [13].

3 Matrix Operations and Computer Hardware: Today and in the Near Future

In the late 1970's it was realized that many algorithms for matrix computations consist of similar building blocks (e.g. a *vector update*, or a *dot-product*). As a result, first, during the design of the Cray-1 supercomputer, vector operations of type $y \leftarrow y + \alpha \cdot x$ (where x and y are n-element vectors, while α is a scalar) have been efficiently implemented in the hardware. Specifically, vector processors have been designed with *chaining* of multiply-and-add operations [14]. Here, results of the multiply operations have been forwarded directly from the multiplication unit to the addition unit. Second, in 1979, the (level 1) BLAS standard was proposed [15], which defined, a set of vector operations. Here, it was assumed that computer vendors would provide their efficient hardware (and software) realizations. In this spirit, Cray Inc. built computers with efficient *vector updates*, and developed the *scilib* library; while the IBM build the ES/9000 vector computers with efficient *dot-product* operation, and developed the *ESSL* library. This library was later ported to the IBM RS/6000 workstations; the first commercial computer to implement the fused multiply-add (FMA) operation [16]). Following this path, most of today advanced processors from IBM, Intel, AMD, Nvidia, and others include scalar floating-point fused multiply-add (FMA) instruction.

The FMA operation combines two basic floating-point operations (flops) into one (three-read-one-write) operation with only one rounding error, throughput of two flops per cycle, and a few cycles latency – depending on the depth of the FMA pipeline. Besides the increased accuracy, the FMA minimizes operation latency, reduces hardware cost, and chip busing [16]. The standard floating-point add, or multiply, are performed by taking $a = 1.0$ (or $b = 1.0$) for addition, or $c = 0.0$ for multiplication. Therefore, the two floating-point constants, 0.0 and 1.0, are needed (and made available within the processor).

FMA-based kernels speed-up ($\sim 2\times$) solution of many scientific, engineering, and multimedia problems, which are based on the linear algebra (matrix) transforms [17]. However, other APP problems suffer from lack of hardware support for the needed scalar FMA operations (see, Table 1). The need for min or/and max operations in the generalized MMA operations introduces one or two conditional branches or comparison/selection instructions, which are highly undesirable for deeply pipelined processors. Recall that each of these operations is repeated Nb-times in the corresponding kernel (see, Section 2), while the kernel itself is called multiple times in the blocked APP algorithm. Here, note the recent results (see, [12]), concerning evaluation of the MMA operation in different semirings, on the Cell/B.E. processor. They showed that the "penalty" for lack of the *generalized* FMA unit may be up to 400%. This can be also viewed as: having an FMA unit, capable of supporting operations and special elements from Table 1 could speed-up solution of APP problems by up to 4 times.

Interestingly, we have just found that the AMD Cypress GPU processor supports the $(\min, \max)$-operation through a single call with 2 clock cycles per result. In this case, the Minimum Spanning Tree (MSP) problem (see, Table 1) could be solved more efficiently than previously realized. Furthermore, this could mean that the AMD hardware has $-\infty$ and ∞ constants already build-in. This, in turn, could constitute an important step towards hardware support of generalized FMA operations, needed to realize all kernels listed in Table 1.

Let us now look into the recent trends in *parallel hardware*. In the 1990's three main designs for parallel computers were: (1) array processors, (2) shared memory parallel computers, and (3) distributed memory parallel computers. After a period of dormancy, currently we observe a resurgence of array-processor-like hardware, placed within a single processor. In particular, the Nvidia promises processors consisting of thousands of computational (FMA) units. This perspective has already started to influence the way we write codes. For instance, one of the key issues is likely to become (again) the *simplicity and uniformity of data manipulation*, while accepting the price of performing seemingly unnecessary operations. As an example, for sparse matrix operations, the guideline can be the statement made recently by John Gustafson, who said: "Go Ahead, Multiply by Zero!" [18]. His assumption, like ours, is that in the hardware of the future, cost of an FMA will be so low, in comparison with data movement (and indexing), that computational sparse linear algebra will have to be re-evaluated.

4 Proposed Generalized Multiply-Add Operation

Let us now summarize the main points made thus far. First, the future of efficient parallel MMA is not likely to involve recursion, focused on reduction the total number of scalar multiplications, while not paying attention to the cost of data movement (and extra memory). Second, without realizing this, scientists solving large number of computational problems, have been working with algebraic semirings. Algebra of semirings involves not only standard linear algebra, but also a large class of APP's. Solutions to these problems involve generalized MMA (which, in turn, calls for hardware-supported generalized FMA operations). Third, benefits of development of *generalized FMA units*, capable of dealing with operations listed in Table 1 have been illustrated. Finally, we have recalled that current trends of development of computer hardware point to existence of processors with thousands of FMA units (possibly generalized), similar to SIMD computers on the chip. Based on these considerations, we can define the needed generic matrix multiply-and-add (MMA) operation

$$\mathtt{C} \leftarrow \mathtt{MMA}[\otimes, \oplus](\mathtt{A}, \mathtt{B}, \mathtt{C}) : \mathtt{C} \leftarrow \mathtt{A} \otimes \mathtt{B} \oplus \mathtt{C},$$

where the $[\otimes, \oplus]$ operations originate from different matrix semirings. Note that, like in the scalar FMA operations, generalized matrix *addition* or *multiplication*, can be implemented by making an $n \times n$ matrix $\mathtt{A}$ (or $\mathtt{B}$) $= \bar{O}$ for addition, or a matrix $\mathtt{C} = \bar{I}$ for multiplication (where the appropriate zero and identity matrices have been defined in Section 2).

Observe that the proposed generalization allows a new approach to the development of efficient codes solving a number of problems. On the one hand, it places in the right context (and subsumes) the level 3 BLAS matrix multiplication (more in Section 5). On the other, the same concepts and representations of operations can be used in solvers for the APP problems.

Let us stress that the idea is not only to generalize matrix multiplication via application of algebraic semirings, but also to "step-up" use of the MMA as the main operation for matrix algorithms. In this way, we should be able (among others) to support processors with thousands of FMA cores. To illustrate the point, let us show how generalized matrix multiplication can be used to perform selected auxiliary matrix operations.

4.1 Data Manipulation by Matrix Multiplication

Observe that the MMA operation, which represents a linear transformation of a vector space, can be used not only for computing but also for data manipulations, such as: reordering of matrix rows/columns, matrix rotation, transposition, etc. This technique is well established and widely used in the algebraic theory (see, for example, [19]). Obviously, data manipulation is an integral part of a large class of matrix algorithms, regardless of parallelism. Due to the space limitation, let us show only how the MMA operation can be used for classical matrix data manipulations, like the row/column interchange, and how it can be extended for more complex data transforms, like th global reduction and replication (broadcast).

Row/Column Interchange. If `zeros(n,n)` is an $n \times n$ zero matrix and `P(n,n)` is an $n \times n$ permutation matrix obtained by permuting the i-th and j-th rows of the identity matrix $I_{n\times n} = \mathtt{eye}(\mathtt{n},\mathtt{n})$ with $i < j$, then multiplication

$$\mathtt{D}(\mathtt{n},\mathtt{n}) = \mathtt{MMA}[\times,+]\big(\mathtt{P}(\mathtt{n},\mathtt{n}),\mathtt{A}(\mathtt{n},\mathtt{n}),\mathtt{zeros}(\mathtt{n},\mathtt{n})\big)$$

gives a matrix `D(n,n)` with the i-th and j-th rows of `A(n,n)` interchanged, and

$$\mathtt{D}(\mathtt{n},\mathtt{n}) = \mathtt{MMA}[\times,+]\big(\mathtt{A}(\mathtt{n},\mathtt{n}),\mathtt{P}(\mathtt{n},\mathtt{n}),\mathtt{zeros}(\mathtt{n},\mathtt{n})\big)$$

gives `D(n,n)` with the i-th and j-th columns of `A(n,n)` interchanged.

Global Reduction and Broadcast. A combination of the global reduction and broadcast, also known in the MPI library as the `MPI_ALLREDUCE` [20], is a very important operation in parallel processing. It is so important that, for example, in the IBM BlueGene/L, a special *collective* network is used, in addition to the other available communication networks, including a 3D toroidal network [21]. However, this operation can be also represented as three matrix multiplications in different semirings,

$$\mathtt{B}(\mathtt{n},\mathtt{n}) = \mathtt{ones}(\mathtt{n},\mathtt{n}) \otimes \mathtt{A}(\mathtt{n},\mathtt{n}) \otimes \mathtt{ones}(\mathtt{n},\mathtt{n}),$$

where $\mathtt{ones}(\mathtt{n},\mathtt{n})$ is the $n \times n$ matrix of ones. For example, the reduction (summation) of all elements of a matrix $\mathtt{A}$ to the single scalar element and its replication (broadcast) to the matrix $\mathtt{B}$ can be completed by two consecutive MMA operations in the $(\times,+)$-semiring:

$$\mathtt{C}(\mathtt{n},\mathtt{n}) = \mathtt{MMA}[\times,+]\big(\mathtt{ones}(\mathtt{n},\mathtt{n}), \mathtt{A}(\mathtt{n},\mathtt{n}), \mathtt{zeros}(\mathtt{n},\mathtt{n})\big),$$

for summation of elements along columns of a matrix $\mathtt{A}(\mathtt{n},\mathtt{n})$, and then,

$$\mathtt{B}(\mathtt{n},\mathtt{n}) = \mathtt{MMA}[\times,+]\big(\mathtt{C}(\mathtt{n},\mathtt{n}), \mathtt{ones}(\mathtt{n},\mathtt{n}), \mathtt{zeros}(\mathtt{n},\mathtt{n})\big),$$

for summation of elements along the rows of an intermediate matrix $\mathtt{C}$. For a sample 4×4 matrix $\mathtt{A}$, it will be completed as follows

$$\begin{pmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix} \times \begin{pmatrix}1&2&3&4\\5&6&7&8\\4&3&2&1\\8&7&6&5\end{pmatrix} \times \begin{pmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix} = \begin{pmatrix}72&72&72&72\\72&72&72&72\\72&72&72&72\\72&72&72&72\end{pmatrix}.$$

If the $(\times,\max)$-semiring is used in these operations, i.e. the scalar multiply-and-add operation would be $c = \max(a \times b, c)$, then the maximal element selection and its broadcast can be implemented as

$$\mathtt{C}(\mathtt{n},\mathtt{n}) = \mathtt{MMA}[\times,\max]\big(\mathtt{ones}(\mathtt{n},\mathtt{n}), \mathtt{A}(\mathtt{n},\mathtt{n}), -\mathtt{inf}(\mathtt{n},\mathtt{n})\big),$$

$$\mathtt{B}(\mathtt{n},\mathtt{n}) = \mathtt{MMA}[\times,\max]\big(\mathtt{C}(\mathtt{n},\mathtt{n}), \mathtt{ones}(\mathtt{n},\mathtt{n}), -\mathtt{inf}(\mathtt{n},\mathtt{n})\big),$$

where $-\mathtt{inf}(\mathtt{n},\mathtt{n})$ is an $n \times n$ matrix of negative infinity. Thus, we have

$$\begin{pmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix} \times \begin{pmatrix}1&2&3&4\\5&6&7&8\\4&3&2&1\\8&7&6&5\end{pmatrix} \times \begin{pmatrix}1&1&1&1\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix} = \begin{pmatrix}8&8&8&8\\8&8&8&8\\8&8&8&8\\8&8&8&8\end{pmatrix}.$$

It is clear that the implementation of this operation in the $(\times,\min)$-semiring with a matrix $\mathtt{inf}(\mathtt{n},\mathtt{n})$ will select the minimal element in the matrix $\mathtt{A}(\mathtt{n},\mathtt{n})$ and its replication, or broadcast.

Interestingly, all these (and other possible) operations for a matrix data manipulation can be realized on an $n \times n$ torus array processor by using a single, or multiple, MMA operation(s), *without any* additional interconnection networks (see, [22]). For implementation of each $n \times n$ MMA operation on the torus array processor, n time-steps are needed and, therefore, only $2n$ steps are required to realize the global reduction and broadcast.

Note that basic elementary matrices, like $\mathtt{eye}(\mathtt{n},\mathtt{n})$, $\mathtt{zeros}(\mathtt{n},\mathtt{n})$, $\mathtt{ones}(\mathtt{n},\mathtt{n})$, and $\pm\mathtt{inf}(\mathtt{n},\mathtt{n})$, can be hardwired within an array processor, where each of their elements is stored in the corresponding processing element. The other required transform matrices can be formed within an array processor by using these elementary matrices, and the corresponding MMA operations. In fact, as stated above, todays' advanced microprocessors already contain special registers for storing the fundamental constants like floating-point 0.0, 1.0, or the $\pm\infty$. An array of such microprocessors will, therefore, automatically include some of the needed elementary matrices.

5 Relating BLAS to the Generalized MMA

In the final part of this paper, let us briefly look into the interrelation between the proposed generalized MMA and the BLAS standard [15, 23, 13]. Due to the lack of space we focus our attention on the level 3 BLAS [13] (the remaining two levels require addressing some subtle points, and will be discussed separately). For similar reasons, we also restrict our attention to square non-symmetric matrices. However, these simplifications do not diminish generality of our conclusions.

In our work we are concerned with the generalized MMA in the form: $C \leftarrow C \oplus A \otimes B$. This matches directly the definition of the level 3 BLAS _GEMM routine, which performs operation: $C \leftarrow \beta C + \alpha op(A) \times op(B)$, while its arguments are:

$$\texttt{\_GEMM}(\texttt{transa}, \texttt{transb}, \texttt{m}, \texttt{n}, \texttt{k}, \alpha, \texttt{A}, \texttt{lda}, \texttt{B}, \texttt{ldb}, \beta, \texttt{C}, \texttt{ldc}).$$

When using _GEMM in computations, the "_" is replaced by one of the letters S, D, C, Z defining the type of matrices (real, double real, complex, and double complex, respectively); while the transa and the transb specify if matrices A and B are in the standard or in the transposed form.

Let us now relate the semiring-based MMA and the _GEMM. To use the semiring constructs in computations, we have to distinguish two aspects. First, the information that is needed to specify the unique semiring and, second, specification of operands of operations within that semiring. As discussed in Section 2, to define a semiring we need to specify: (1) elements (S), (2) operations ($\oplus, \otimes, \circledast$), and (3) special elements ($\bar{0}$ and $\bar{1}$), Obviously, the semiring defined by the BLAS operations is the linear algebra semiring. Here, the only aspect of the semiring that is *explicitly* defined, is the set of elements of the matrices. This definition comes through the "naming convention" and is realized by the "_" part of the subroutine definition; selecting the type of the numbers (real, or complex) and their computer representation (single, or double precision). The remaining parts of the definition are implicit. It is assumed that the objects of the semiring are matrices, while the elements $\bar{0}$ and $\bar{1}$ are matrices consisting of all zeros, and the standard identity matrix, respectively. Here, operations are the basic matrix multiplication, addition and closure. At the same time, the elements m, n, k, α, A, lda, B, ldb, β, C, ldc define specific operands for the MMA.

An interesting issue concerns the transa and transb parameters. They specify if matrices A and B (respectively), used in the _GEMM operation are in their standard or transposed forms. From the point of view of the theory of semirings, it does not matter if a matrix is transposed or not; what matters is if it belongs to the set S. The difference occurs when the actual MMA is to be performed. Therefore, the standard BLAS operation set allowed to combine matrix multiplication, scaling, and transpose operations into a single, relatively simple, code. However, as illustrated in Section 4.1, in the case of the approach advocated in this paper, where "everything is a matrix multiplication," matrix scaling and transpose also become matrix multiplications.

It should be noted that all, but three, remaining level 3 BLAS routines also perform operations of the type $C = \beta C + \alpha op(A) op(B)$, for a variety of specific operands. The only exceptions are the _HER2K, _SYR2K pair, which perform "double updates" of a matrix. Obviously they can be replaced by two operations of the type $C = \beta C + \alpha op(A)op(B)$ performed in a row (two MMA's).

The only operation that is not easily conceptualized, within the scope of the proposal outlined thus far, is the _TRSM which performs a triangular solve. Here the matrix inversion operation (defined in Table 1), could be utilized; but the requires further careful considerations.

6 Concluding Remarks

In this paper we have reflected on the effect that the recent developments in computer hardware and computational sciences can have on the way that dense matrix multiplication is approached. First, we have indicated that the arithmetical operation count (theoretical floating point complexity) looses importance; becoming overshadowed by the complexity of data manipulations performed by processors with hundreds of FMA processing units. Second, we have recalled that the "standard" matrix multiplication is just one of possible operations within an appropriately defined algebraic semiring. This allowed us to illustrate how the semiring-based approach covers large class of APP problems. Finally, on the basis of these considerations, we have proposed a generalized matrix multiply-and-add operation, which allows to further induce efficient matrix multiplication as the key operation driving solution methods not only in linear algebra, but also across a variety of APP problems. Finally, we have outlined the relation of the proposed matrix generalization to the level 3 BLAS standard.

In the near future we plan to (1) propose an object oriented model for the generalized MMA (see, [24] for description of initial work in this direction), (2) proceed with a prototype implementation, and (3) conduce experiments with the, newly-defined, fused multiply-add operations involving *min* and *max* operations.

Acknowledgment. Work of Marcin Paprzycki was concluded while visiting the University of Aizu as a Research Scientist.

References

1. Robinson, S.: Towards an optimal algorithm for matrix multiplication. SIAM News 38 (2005)
2. Li, J., Skjellum, A., Falgout, R.D.: A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. Concurrency - Practice and Experience 9, 345–389 (1997)
3. Hunold, S., Rauber, T., Rünger, G.: Combining building blocks for parallel multi-level matrix multiplication. Parallel Comput. 34, 411–426 (2008)
4. Grayson, B., Van De Geijn, R.: A high performance parallel Strassen implementation. Parallel Processing Letters 6, 3–12 (1996)
5. Song, F., Moore, S., Dongarra, J.: Experiments with Strassen's Algorithm: from Sequential to Parallel. In: International Conference on Parallel and Distributed Computing and Systems (PDCS 2006). ACTA Press (November 2006)

6. Bailey, D.H., Lee, K., Simon, H.D.: Using Strassen's algorithm to accelerate the solution of linear systems. J. Supercomputer 4, 357–371 (1991)
7. Paprzycki, M., Cyphers, C.: Multiplying matrices on the Cray – practical considerations. CHPC Newsletter 6, 77–82 (1991)
8. Sedukhin, S.G., Miyazaki, T., Kuroda, K.: Orbital systolic algorithms and array processors for solution of the algebraic path problem. IEICE Trans. on Information and Systems E93.D, 534–541 (2010)
9. Lehmann, D.J.: Algebraic structures for transitive closure. Theoretical Computer Science 4, 59–76 (1977)
10. Abdali, S.K., Saunders, B.D.: Transitive closure and related semiring properties via eliminants. Theoretical Computer Science 40, 257–274 (1985)
11. Matsumoto, K., Sedukhin, S.G.: A solution of the all-pairs shortest paths problem on the Cell broadband engine processor. IEICE Trans. on Information and Systems 92-D, 1225–1231 (2009)
12. Sedukhin, S.G., Miyazaki, T.: Rapid*Closure: Algebraic extensions of a scalar multiply-add operation. In: Philips, T. (ed.) CATA, ISCA, pp. 19–24 (2010)
13. Dongarra, J.J., Croz, J.D., Duff, I., Hammarling, S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Software 16, 1–17 (1990)
14. Russell, R.M.: The CRAY-1 computer system. Commun. ACM 21, 63–72 (1978)
15. Lawson, C.L., Hanson, R.J., Kincaid, R.J., Krogh, F.T.: Basic linear algebra subprograms for FORTRAN usage. ACM Trans. Math. Software 5, 308–323 (1979)
16. Montoye, R.K., Hokenek, E., Runyon, S.L.: Design of the IBM RISC System/6000 floating-point execution unit. IBM J. Res. Dev. 34, 59–70 (1990)
17. Gustavson, F.G., Moreira, J.E., Enenkel, R.F.: The fused multiply-add instruction leads to algorithms for extended-precision floating point: applications to Java and high-performance computing. In: CASCON 1999: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collab. Research, p. 4. IBM Press (1999)
18. Gustafson, J.L.: Algorithm leadership. HPCwire, April 06 (2007)
19. Birkhoff, G., McLane, S.: A Survey of Modern Algebra. AKP Classics. A K Peters, Massachusetts (1997)
20. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. The MIT Press, Cambridge (1996)
21. Gara, A., Blumrich, M.A., Chen, D., Chiu, G.L.T., Coteus, P., Giampapa, M., Haring, R.A., Heidelberger, P., Hoenicke, D., Kopcsay, G.V., Liebsch, T.A., Ohmacht, M., Steinmacher-Burow, B.D., Takken, T., Vranas, P.: Overview of the Blue Gene/L system architecture. IBM J. Res. and Dev. 49, 195–212 (2005)
22. Sedukhin, S.G., Zekri, A.S., Myiazaki, T.: Orbital algorithms and unified array processor for computing 2D separable transforms. In: International Conference on Parallel Processing Workshops, pp. 127–134 (2010)
23. Dongarra, J.J., Croz, J.D., Hammarling, S., Hanson, R.J.: An extended set of FORTRAN basic linear algebra subprograms. ACM Trans. Math. Software 14, 1–17 (1988)
24. Ganzha, M., Sedukhin, S., Paprzycki, M.: Object oriented model of generalized matrix multipication. In: Proceedings of the Federated Conference on Computer Science and Information Systems, pp. 439–442. IEEE Press, Los Alamitos (2011)

Distributed QR Factorization Based on Randomized Algorithms

Hana Straková[1], Wilfried N. Gansterer[1,⋆], and Thomas Zemen[2]

[1] University of Vienna, Austria
Research Group Theory and Applications of Algorithms
{Hana.Strakova,Wilfried.Gansterer}@univie.ac.at
[2] Forschungszentrum Telekommunication Wien, Austria
Thomas.Zemen@ftw.at

Abstract. Most parallel algorithms for matrix computations assume a static network with reliable communication and thus use fixed communication schedules. However, in situations where computer systems may change dynamically, in particular, when they have unreliable components, algorithms with randomized communication schedule may be an interesting alternative.

We investigate randomized algorithms based on gossiping for the distributed computation of the QR factorization. The analyses of numerical accuracy showed that known numerical properties of classical sequential and parallel QR decomposition algorithms are preserved. Moreover, we illustrate that the randomized approaches are well suited for distributed systems with arbitrary topology and potentially unreliable communication, where approaches with fixed communication schedules have major drawbacks. The communication overhead compared to the optimal parallel QR decomposition algorithm (CAQR) is analyzed. The randomized algorithms have a much higher potential for trading off numerical accuracy against performance because their accuracy is proportional to the amount of communication invested.

Keywords: distributed vs. parallel QR factorization, decentralized QR factorization, evaluation of distributed algorithms, gossip algorithms, push -sum algorithm, randomized communication schedule, fault-tolerance.

1 Introduction

We consider the distributed computation of the QR factorization over a loosely coupled distributed system where existing approaches with fixed communication schedule have major drawbacks. We develop and analyze a randomized distributed QR decomposition based on the push-sum algorithm. Since the nodes proceed in an asynchronous and decentralized way, our approach is more flexible than existing parallel QR decomposition algorithms and capable of handling unreliable links as well as potential dynamic changes of the network. The main

⋆ Corresponding author.

goal of this paper is a comprehensive comparison of this new randomized QR decomposition approach with the existing parallel algorithms.

In this paper, we use the term *parallel* for algorithms developed primarily for tightly coupled parallel computers (comprising shared-memory systems, multicore architectures, tightly coupled distributed memory systems, etc.) which are characterized by a *static* topology and *reliable*, relatively fast, (usually) wired communication. For such target platforms algorithms with fixed communication schedules are suitable. In contrast, we use the term *distributed* for algorithms designed for loosely coupled decentralized distributed systems. These target platforms are characterized by potentially dynamically changing topology, relatively slow and costly communication, and/or by unreliable components (e.g., communication links). This includes, but is not limited to wireless networks, peer to peer (P2P) networks, mobile ad hoc networks, etc. Most existing parallel algorithms have major drawbacks on such systems and thus are not applicable.

Motivation and Approach. Motivating problems arise, for example, in decentralized network analysis (cf. [12]) or in cooperative transceiver design in telecommunications (cf. [8]). In the latter case the objective of a method called *distributed sphere decoding* is to exploit cooperation of receiving node with other receivers in the decoding of a signal sent by a group of transmitters [14]. A distributed QR factorization of a distributed channel matrix, which describes the quality of communication channels between nodes, is needed for solving a maximum likelihood problem.

A theoretical alternative to existing parallel algorithms for the context considered would be a "fusion center" approach. The fusion center first collects the entire matrix, computes the QR factorization locally, and then returns the result to the other nodes. However, this leads to an unfair distribution of energy consumption among the cooperating nodes and may be infeasible due to memory constraints (if no single node can store the entire matrix). It has also been shown that in many situations in-network computation leads to considerable advantages over centralized approach [17,15] (e.g., in terms of energy savings).

Consequently, we pursue a randomized decentralized QR factorization algorithm based on *gossiping*. *Gossip-based* (or *epidemic*) algorithms are characterized by asynchronous randomized information exchange, usually only within the local neighborhood of each node. They do not assume static or reliable networks, do not require any specialized routing and do not have a single point of failure. Thus, they can cope with link failures or more general dynamic network changes, and consequently achieve high fault tolerance. Due to their ideally completely decentralized structure, gossip-based algorithms tend to scale well with the number of nodes. Moreover, gossip-based algorithms allow for trading off time-to-solution against communication cost (and thus energy consumption) or fault tolerance by gradually adapting the intensity and regularity of communication with other nodes. Especially the latter property is very attractive when parallel algorithms are not applicable and fusion center approaches are too expensive, or where not only the highest achievable accuracy, but also intermediate approximate results are of interest. Existing parallel algorithms produce the full result at the full accuracy achievable for the input data only when they terminate regularly.

Related Work. Parallel algorithms for computing QR factorizations on shared memory, distributed memory or on multicore architectures have been studied extensively. State-of-the-art software libraries such as ScaLapack [3] use blocked algorithms for implementing parallel QR factorization. The latest developments are *tiled QR factorization* [5], *communication-avoiding QR factorization* (*CAQR*) [6], and the combination of the two approaches [16].

Much less work exists in the literature on distributed algorithms. Although wireless networks are mentioned as target environment in [1], the assumptions used are those typical for parallel algorithms: all data initially centralized at a single node, specific roles assigned to individual nodes (distributing node, annihilating node, etc.) and the communication schedule is fixed. Relevant for our approach are randomized algorithms, such as the push-sum algorithm [11], utilized for matrix computations. Simple gossip-based algorithms are used due to their attractive properties in many in-network computations (e.g, in distributed signal processing [7]), but to the best of our knowledge, only a few gossip-based matrix algorithms have been investigated. A gossip-based decentralized algorithm for spectral analysis was discussed in [12], and a distributed QR factorization based on push-sum has been used in distributed sphere decoding in [8]. So far, no analysis of numerical behavior or communication cost of a randomized distributed QR factorization is available. Moreover, no comparison to existing parallel approaches is available.

Contributions. We extend existing work in the following algorithmic aspects: We use *modified* Gram-Schmidt orthogonalization (mGS) as the basis for our new algorithm, which we call *dmGS*, whereas the algorithm used in [8] implements the less stable *classical* Gram-Schmidt orthogonalization (cGS). Moreover, the algorithm used in [8] assumes that after each push-sum algorithm one of the local estimates is broadcasted to all the other nodes, whereas our approach proceeds with the local estimate in each node (which is more realistic in practice).

We for the first time provide a full quantitative evaluation of a distributed QR decomposition algorithm in terms of numerical accuracy, reliability, operation count, memory requirements, and communication cost (partly based on theoretical analyses and partly based on simulations) as well as a quantitative comparison with state-of-the-art parallel QR decomposition algorithms. We also consider more general data distributions over random topologies of the underlying distributed system ([8] considered only square matrices over fully connected networks), investigate the effects of communication link failures, and we adapted termination criteria from the literature in order to be able to investigate the trade-off between numerical accuracy and performance or communication cost.

Synopsis. In Section 2, the QR factorization and the push-sum algorithm are reviewed. In Section 3, our new algorithm (dmGS) is presented. In Section 4, numerical simulations and theoretical analyses are summarized. Section 5 concludes the paper.

2 Methodology

Algorithms for QR Factorization. We are interested in computing the *thin* (reduced) QR factorization $A = QR$ (where $A \in \mathbb{C}^{n\times m}$, $n \geq m$, $Q \in \mathbb{C}^{n\times m}$ has orthogonal columns, $R \in \mathbb{C}^{m\times m}$ is upper triangular) by classical (cGS) and modified (mGS) Gram-Schmidt orthogonalization (for mGS see Algorithm 1).

Both Gram-Schmidt orthogonalization processes require $2nm^2$ flops for computing the QR factorization of an $n \times m$ matrix [10]. Denoting the machine precision with $\varepsilon_{\text{mach}}$ and the condition number of the matrix A with $\kappa_2(A)$, the matrix Q produced by mGS satisfies $\|I - Q^H Q\|_2 \approx \varepsilon_{\text{mach}}\kappa_2(A)$ [10], whereas for a square matrix A of size $n \times n$ cGS produces Q for which only $\|I - Q^H Q\|_2 \leq c(n)\varepsilon_{\text{mach}}[\kappa_2(A)]^{n-1}$ with a modest constant $c(n)$ holds [13].

The Push-Sum Algorithm. As mentioned in Section 1, the information exchange in gossiping protocols is randomized and asynchronous, and nodes iteratively update their local information according to information received from other nodes. The network topology can be arbitrary, dynamically changing and does not need to be known in individual nodes. The *push-sum algorithm* [11] is a gossip algorithm for approximating the sum or the average of values distributed over a network with N nodes. If $x(k)$ denotes a value stored in node k, the goal of the algorithm is to compute in each node k an estimate $s_k = \sum_j x(j)$ (or $a_k = \sum_j x(j)/N$). The accuracy of the estimates depends on the number of asynchronous iterations of the push-sum algorithm.

An important question is when to terminate the push-sum algorithm. In [11], an asymptotic number of iterations needed in the push-sum algorithm for reaching a certain error τ with some probability is derived (the notation is adapted to our context): *For a network with N nodes, there is a number of iterations $i_0 = O(\log N + \log \frac{1}{\tau} + \log \frac{1}{\delta})$, such that for all numbers of iterations $i \geq i_0$ with probability of at least $1-\delta$ the estimate of the average satisfies in every node k*

$$\frac{1}{|\sum_{j=1}^{N} x(j)|}\left|\frac{s_k^{(i)}}{w^{(i)}(k)} - \frac{1}{N}\sum_{j=1}^{N} x(j)\right| \leq \tau \frac{\sum_{j=1}^{N}|x(j)|}{|\sum_{j=1}^{N} x(j)|}. \tag{1}$$

where $x(k)$ is a value stored in node k, $w^{(i)}(k)$ is a weight used in node k in iteration i, and $s_k^{(i)}$ is an estimate of the sum $\sum_{j=1}^{N} x(j)$ in node k in iteration i. In [8] the push-sum algorithm is terminated after a fixed number of iterations determined by the authors based on simulations. In [12] a blockwise extension of the push-sum algorithm is proposed and a stopping criterion with an accuracy parameter τ is suggested. We adapt this termination criterion to the scalar push-sum and use it in our simulations.

3 Distributed QR Factorization (dmGS)

In Algorithm 1 the distributed modified Gram-Schmidt orthogonalization dmGS is presented and compared to mGS. The notation used in the algorithm is explained in Section 2. We can see, that mGS and dmGS are very similar.

Algorithm 1. Modified Gram-Schmidt orthogonalization (mGS) and distributed QR factorization (dmGS)

Input: $A \in \mathbb{C}^{n\times m}$, $n \geq m$ (matrix A distributed row-wise over N nodes; if $n > N$, each node k stores r_k consecutive rows of A)
Output: $Q \in \mathbb{C}^{n\times m}$, $R \in \mathbb{C}^{m\times m}$ (matrix Q distributed row-wise, matrix R distributed column-wise over nodes)

```
1:  for $i = 1$ to $m$ do
2:      $x(k) = A(k,i)^2$
3:
4:      $s = \sum_{k=1}^{n} x(k)$
5:      $R(i,i) = \sqrt{s}$
6:      $Q(:,i) = A(:,i)/R(i,i)$
7:
8:      for $j = i+1$ to $m$ do
9:          $x(k) = Q(k,i)A(k,j)$
10:
11:         $R(i,j) = \sum_{k=1}^{n} x(k)$
12:         $A(:,j) = A(:,j) - Q(:,i)R(i,j)$
13:
14:     end for
15: end for
```

```
1:  for $i = 1$ to $m$ do (in node $k$)
2:      $x(k) = A(k,i)^2$
3:      [ $x(k) = \sum_{t=1}^{r_k} A(k_t,i)^2$ ]
4:      $s_k = \textit{push-sum}(x)$
5:      $R_k(i,i) = \sqrt{s_k}$
6:      $Q(k,i) = A(k,i)/R_k(i,i)$
7:      if $k \neq i$ delete $R_k(i,i)$
8:      for $j = i+1$ to $m$ do
9:          $x(k) = Q(k,i)A(k,j)$
10:         [ $x(k) = \sum_{t=1}^{r_k} Q(k_t,i)A(k_t,j)$ ]
11:         $R_k(i,j) = \textit{push-sum}(x)$
12:         $A(k,j) = A(k,j) - Q(k,i)R_k(i,j)$
13:         if $k \neq j$ delete $R_k(i,j)$
14:     end for
15: end for
```

The only difference is that the sums in the mGS needed for computing a 2-norm (Line 4) and a dot product (Line 11) are replaced by the push-sum algorithm. dmGS assumes as input a matrix $A \in \mathbb{C}^{n\times m}, n \geq m$ distributed row-wise over N nodes. In the special case $n = N$, each node contains one row of the input matrix $A \in \mathbb{C}^{n\times m}, n \geq m$ and the push-sum computes the sum of corresponding column elements stored in nodes. In the most common case $n > N$, each node k contains $r_k = \mathcal{O}(n/N)$ rows of the input matrix A. In this case, before each push-sum the corresponding column elements stored in one node are locally preprocessed (Lines 3 and 10) and then summed up by the push-sum algorithm. Except of the push-sum algorithms utilized for computing in all nodes k the local estimates R_k of the elements of the matrix R (Lines 4 and 11), the nodes do not communicate and can execute the computations locally. dmGS, as described in Algorithm 1, stores one column of matrix R in corresponding nodes. However, if the lines 7 and 13 are skipped, the nodes keep the computed estimates of the elements of the matrix R and thus store the full matrix R. This of course results in higher local memory requirements.

4 Evaluation of dmGS

In this section, we first investigate the numerical accuracy of dmGS and analyze the influence of the link failures and of the diameter of the network on its

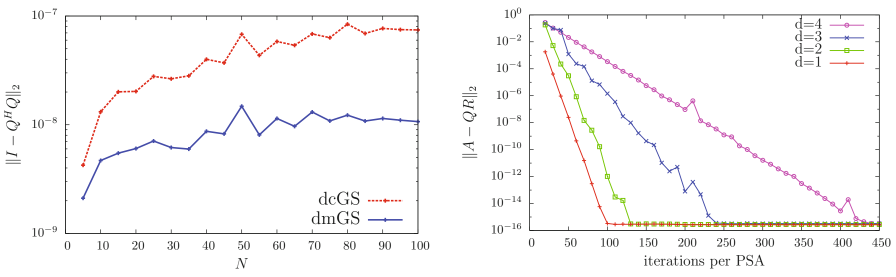


Fig. 1. Comparison of the distributed QR factorization (left) based on the classical (dcGS) resp. modified (dmGS) Gram-Schmidt method in terms of orthogonality measure and influence of random network topologies (right) with different diameter d on convergence speed of dmGS

convergence. Then we compare dmGS to state-of-the-art parallel algorithms in terms of operation count, local memory requirements, and communication cost.

We present simulation results for dmGS applied to a random matrix $A \in \mathbb{C}^{n \times m}$ distributed over N nodes. Simulations are partly based on Matlab and partly on ns-3. In the Matlab simulations, the nodes choose their communication partners uniformly from the entire network, whereas the ns-3 simulations are based on a predetermined random network topology with a given diameter.

In Fig. 1 (left) distributed QR factorization based on cGS (dcGS) is compared to our algorithm dmGS in terms of orthogonality of Q measured as $\|I - Q^H Q\|_2$ for varying number of nodes N, when applied to $A \in \mathbb{C}^{n \times n}, n = N$. We can observe the expected difference in terms of orthogonality of Q (the factorization errors were basically identical), which illustrates that the distributed algorithm preserves the numerical properties of the sequential and parallel cases. In the experiments the overall factorization error and orthogonality was of the same order of magnitude as the accuracy parameter $\tau = 10^{-9}$ used for terminating the algorithm (see Section 2).

4.1 Reliability and Scalability

In this section we investigate the influence of link failures and random topologies on the behavior of dmGS. In Fig. 1 (right) we can see the behavior of dmGS for arbitrary networks with different diameters d. Although the increasing diameter causes slower convergence, all simulations converge to the full accuracy.

Simulations showed that loss of messages due to link failures does not influence the factorization error. However, it causes loss of orthogonality of the computed matrix Q. To achieve satisfactory orthogonality of Q we have to either avoid losing messages or recover from the loss. The simplest strategy is to resend each message several times to ensure a high probability that it will be delivered. The exact number how many times was each message sent in our experiments was chosen such that the combined probability of not delivering the message for all repeated sending attempts is smaller than 10^{-12}. Fig. 2 (left) shows the influence

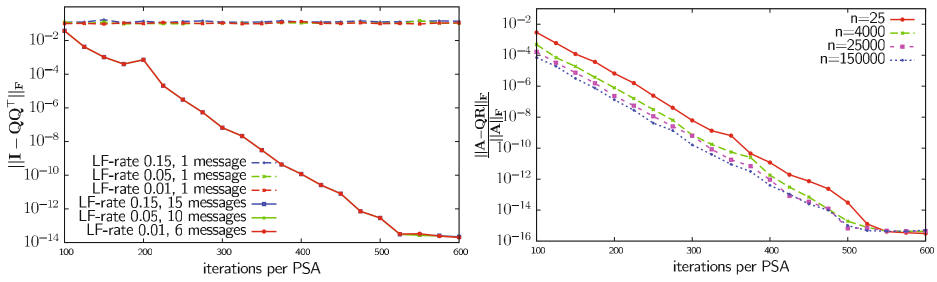


Fig. 2. Comparison of the influence of various failure rates on the orthogonality of Q for sending each message once and several times (left) and scalability of dmGS with increasing number of rows n for matrix $A \in \mathbb{R}^{n \times 25}$ (right)

of the link failures on the orthogonality of the matrix Q for various failure rates with and without resending each message. In [9] we discuss approaches for recovering from message and data loss in more detail.

If $n > N$, nodes have to store more than one row. Consequently, for fixed N, dmGS scales very well with increasing n. As shown in Fig. 2 (right), storing several rows in one node even increases the accuracy, since more computation is done locally in nodes. Note that increasing number of rows per node does not affect the communication cost. Increasing the number of columns m is more difficult, because it causes quadratical increase of communication cost. However, a new version of dmGS which improves the scalability for growing m is currently under development.

4.2 Theoretical Analyses and Comparison with Parallel Algorithms

In the following, dmGS is compared with state-of-the-art parallel algorithms for QR factorization when applied to a rectangular matrix $n \times m$ distributed over N nodes (or P processors). In particular, we compare dmGS with parallel mGS and with parallel CAQR (which is optimal in terms of communication cost up to polylogarithmic factors) [6] along the critical path. For dmGS, the terms along the critical path are equal to the cost per node. We have to emphasize that this comparison has to be interpreted carefully, since these different types of algorithms are designed for very different target systems and are based on very different assumptions on properties of the target hardware and on the initial data distribution (see Section 1).

To simplify the analysis, we assume that push-sum algorithm proceeds in synchronous iterations and that in each iteration every node sends exactly one message. Note that this synchronization is *not* required in practice and that our simulations do not rely on this assumption. Push-sum needs $\mathcal{O}(\log N)$ iterations to converge to a fixed accuracy τ for networks where each node can communicate with any other node [11], which we use in our analyses. Similar convergence rate was also shown for more general topologies [4].

Operation Count. dmGS calls $m(m+1)/2$ push-sum algorithms and before each push-sum, a local summation of $r_k = \mathcal{O}(n/N)$ elements must be performed in node k. In summary, dmGS performs $\mathcal{O}(m^2 \log N + m^2 n/N)$ flops. According to [6], parallel mGS performs $2nm^2/P$ flops and parallel CAQR performs $2m^3/3 + 2m^2 n/P$ flops along the critical path. Consequently, asymptotically parallel mGS has the lowest flop count of the three methods and the count of dmGS is a little lower than the one of parallel CAQR (for fixed $P = N$).

Local Memory Requirements. In dmGS each node k needs to store $r_k = \mathcal{O}(n/N)$ rows of matrix A, r_k rows of matrix Q and on m of the nodes one column of matrix R, each of them of length m. This leads to local memory requirements of $\Theta(r_k m) = \Theta(nm/N)$. During the push-sum algorithm each node needs a constant number of words of local memory. The local memory requirements for parallel CAQR are $\Theta(nm/P)$ [6]. This shows that asymptotically CAQR and dmGS have the same local memory requirements.

Communication Cost. We investigate three communication cost metrics: the number of messages and the number of words sent along the critical path as well as the average message size. The total number C_t of messages sent during dmGS is given as $C_t = S \cdot I \cdot M$, where S is the number of push-sum algorithms, I is the number of iterations per push-sum and M is the number of messages sent during one iteration of the push-sum algorithm. dmGS calls $S = m(m+1)/2$ push-sum algorithms (cf. Algorithm 1) and as described above, $I = \mathcal{O}(\log N)$ for a fixed final accuracy. As conclusion, the number C of messages sent per node in dmGS is $C = \mathcal{O}(m^2 \log(N))$ and that the total number C_t of messages sent is $C_t = \mathcal{O}(Nm^2 \log(N))$. The total number of words sent during dmGS equals the total number of messages C_t multiplied by their size, which in dmGS is constant (two words). Thus the number of words sent is asymptotically the same as the number of messages sent. For the special case of a square $n \times n$ matrix distributed over $n = N$ nodes, we get $C = \mathcal{O}(N^2 \log(N))$ and $C_t = \mathcal{O}(N^3 \log(N))$.

Comparison with State-of-the-Art Parallel Algorithms. According to [6], parallel mGS sends $2m \log(P)$ messages and $m^2 \log(P)/2$ words (leading terms) when factorizing a rectangular $n \times m$ matrix over P processors. Consequently, the average message size $m/4$ increases linearly with increasing m. Parallel CAQR sends $1/4 \cdot \sqrt{mP/n} \cdot \log^2(nP/m) \cdot \log(P\sqrt{nP/m})$ messages and $\sqrt{nm^3/P} \cdot \log P - 1/4 \cdot \sqrt{m^5/nP} \cdot \log(mP/n)$ words along the critical path [6]. For the special case of a square $n \times n$ matrix, CAQR needs $3\sqrt{P}\log^3(P)/8$ messages and $3n^2 \log(P)/(4\sqrt{P})$ words. Consequently, the average message size in parallel CAQR for a square matrix is $2n^2/(P \log^2(P))$.

For the special case $n = m = N = P$, which is relevant in the application problems mentioned in Section 1, we find that dmGS asymptotically sends the same number of *words* along the critical path as parallel mGS and a factor $\sqrt{N}$ more than parallel CAQR. When comparing the total number of *messages* sent along the critical path, there is a bigger difference. This is due to the fact that in dmGS the message size is constant, whereas in the parallel algorithms it grows with N, which reduces the number of messages sent. Asymptotically, parallel mGS sends a factor N fewer messages along the critical path than dmGS. The communication-optimal CAQR algorithm sends even a factor $N\sqrt{N}/\log^2(N)$ fewer messages than dmGS. This shows that dmGS would not be competitive in terms of communication cost when executed on standard target systems for parallel algorithms, because the communication overhead to be paid for reliable execution becomes quite large for large N.

5 Conclusions

We introduced dmGS, a gossip-based algorithm for distributed computation of the QR factorization of a general rectangular matrix distributed over a loosely coupled decentralized distributed system with unreliable communication. Problems of this type arise, for example, in distributed sphere decoding in telecommunications or in decentralized network analysis.

We evaluated dmGS in terms of the factorization error and orthogonality of the computed matrix Q. In terms of numerical accuracy, it preserves known properties of existing sequential and parallel QR factorization algorithms. However, dmGS is designed for completely different target platforms and its decentralized structure makes it more flexible and robust. In particular, we illustrated that although unreliable communication introduces overhead and an increase of the network diameter slows down convergence, dmGS still produces correct results.

In order to complete the picture, a quantitative comparison of dmGS with the state-of-the-art parallel algorithms in terms of operation count, local memory requirements and communication cost has been given. It shows that there is a communication overhead to be paid in dmGS over existing parallel algorithms for robust execution on dynamic and decentralized target platforms with unreliable communication. However, we currently work on an algorithmic improvement of dmGS which reduces the communication overhead. Moreover, dmGS has the additional benefit that its computational and communication cost are proportional to the target accuracy, i. e., it is possible to compute an approximative QR factorization at proportionally reduced cost.

Ongoing and Future Work. More detailed analyses and simulations of the influence of the network topology on the behavior of dmGS will be performed. We will also investigate energy efficiency aspects and the influence of dynamic networks (for example, mobile nodes) on the behavior of the algorithm. The investigation of the utilization of broadcast gossiping methods [2] is another important next task. We are also developing code suitable for runtime performance model with ScaLAPACK and PLASMA routines. Beyond that, other ideas for further reducing the communication cost in distributed QR factorization algorithms will be investigated.

Acknowledgments. This work has been supported by the Austrian Science Fund (FWF) under contracts S10608 and S10607 (NFN SISE). The Telecommunications Research Center Vienna (FTW) is supported by the Austrian Government and the City of Vienna within the competence center program COMET.

References

1. Abdelhak, S., Chaudhuri, R.S., Gurram, C.S., Ghosh, S., Bayoumi, M.: Energy-aware distributed QR decomposition on wireless sensor nodes. The Computer Journal 54(3), 373–391 (2011)
2. Aysal, T., Yildiz, M., Sarwate, A., Scaglione, A.: Broadcast gossip algorithms for consensus. IEEE Trans. Signal Processing 57(7), 2748–2761 (2009)
3. Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. SIAM, Philadelphia (1997)
4. Boyd, S., Ghosh, A., Prabhakar, B., Shah, D.: Randomized gossip algorithms. IEEE Trans. Information Theory 52(6), 2508–2530 (2006)
5. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: Parallel Tiled QR Factorization for Multicore Architectures. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 639–648. Springer, Heidelberg (2008)
6. Demmel, J., Grigori, L., Hoemmen, M.F., Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations. Tech. rep. no. UCB/EECS-2008-89, EECS Department, University of California, Berkeley (2008)
7. Dimakis, A., Kar, S., Moura, J., Rabbat, M., Scaglione, A.: Gossip algorithms for distributed signal processing. Proceedings of the IEEE 98(11), 1847–1864 (2010)
8. Dumard, C., Riegler, E.: Distributed sphere decoding. In: International Conference on Telecommunications, ICT 2009, pp. 172–177 (2009)
9. Gansterer, W.N., Niederbrucker, G., Strakova, H., Schulze Grotthoff, S.: Scalable and fault tolerant orthogonalization based on randomized aggregation. To Appear in Journal of Computational Science
10. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. The Johns Hopkins University Press (1996)
11. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: FOCS 2003: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 482–491. IEEE Computer Society (2003)
12. Kempe, D., McSherry, F.: A decentralized algorithm for spectral analysis. Journal of Computer and System Sciences 74(1), 70–83 (2008)
13. Kielbasinski, A., Schwetlick, H.: Numeryczna algebra liniowa, 2nd edn. Wydawnictwo Naukowo-Techniczne, Warszawa (1994) (in Polish)
14. Ozgur, A., Leveque, O., Tse, D.: Hierarchical cooperation achieves optimal capacity scaling in ad hoc networks. IEEE Transactions on Information Theory 53(10), 3549–3572 (2007)
15. Rabbat, M., Nowak, R.: Distributed optimization in sensor networks. In: Third International Symposium on Information Processing in Sensor Networks, pp. 20–27 (2004)
16. Song, F., Ltaief, H., Hadri, B., Dongarra, J.: Scalable tile communication-avoiding QR factorization on multicore cluster systems. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2010)
17. Yu, Y., Krishnamachari, B., Prasanna, V.: Energy-latency tradeoffs for data gathering in wireless sensor networks. In: INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 1 (2004)

Static Load Balancing for Multi-level Monte Carlo Finite Volume Solvers

Jonas Šukys, Siddhartha Mishra, and Christoph Schwab

ETH Zürich, Switzerland
{jonas.sukys,smishra,schwab}@sam.math.ethz.ch

Abstract. The Multi-Level Monte Carlo finite volumes (MLMC-FVM) algorithm was shown to be a robust and fast solver for uncertainty quantification in the solutions of multi-dimensional systems of stochastic conservation laws. A novel load balancing procedure is used to ensure scalability of the MLMC algorithm on massively parallel hardware. We describe this procedure together with other arising challenges in great detail. Finally, numerical experiments in multi-dimensions showing strong and weak scaling of our implementation are presented.

Keywords: uncertainty quantification, conservation laws, multi-level Monte Carlo, finite volumes, static load balancing, linear scaling.

1 Introduction

A number of problems in physics and engineering are modeled in terms of systems of conservation laws:

$$\begin{cases} \mathbf{U}_t + \operatorname{div}(\mathbf{F}(\mathbf{U})) = \mathbf{S}(\mathbf{x}, \mathbf{U}), \\ \mathbf{U}(\mathbf{x}, 0) = \mathbf{U}_0(\mathbf{x}), \end{cases} \quad \forall (\mathbf{x}, t) \in \mathbb{R}^d \times \mathbb{R}_+. \tag{1}$$

Here, $\mathbf{U} : \mathbb{R}^d \to \mathbb{R}^m$ denotes the vector of conserved variables, $\mathbf{F} : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^{m \times d}$ is the collection of directional flux vectors and $\mathbf{S} : \mathbb{R}^d \times \mathbb{R}^m \to \mathbb{R}^m$ is the source term. The partial differential equation is augmented with initial data $\mathbf{U}_0$.

Examples for conservation laws include the shallow water equations of oceanography, the Euler equations of gas dynamics, the Magnetohydrodynamics (MHD) equations of plasma physics and the equations of non-linear elasticity.

As the equations are non-linear, analytical solution formulas are only available in very special situations. Consequently, numerical schemes such as finite volume methods [7] are required for the study of systems of conservation laws.

Existing numerical methods for approximating (1) require the initial data $\mathbf{U}_0$ and source $\mathbf{S}$ as the input. However, in most practical situations, it is not possible to measure this input precisely. This uncertainty in the inputs for (1) propagates to the solution, leading to the *stochastic* system of conservation laws:

$$\begin{cases} \mathbf{U}(\mathbf{x}, t, \omega)_t + \operatorname{div}(\mathbf{F}(\mathbf{U}(\mathbf{x}, t, \omega))) = \mathbf{S}(\mathbf{x}, \omega), \\ \mathbf{U}(\mathbf{x}, 0, \omega) = \mathbf{U}_0(\mathbf{x}, \omega), \end{cases} \quad \mathbf{x} \in \mathbb{R}^d, \quad t > 0, \quad \forall \omega \in \Omega. \tag{2}$$

where $(\Omega, \mathcal{F}, \mathbb{P})$ is a complete probability space; the initial data $\mathbf{U}_0$ and the source term $\mathbf{S}$ are random fields [8,10]. The solution is also realized as a random field; its statistical moments (e.g. expectation and variance) are the quantities of interest. An estimate of the expectation can be obtained by the so-called Monte Carlo finite volume method (MC-FVM) consisting of the following three steps:

1. **Sample:** We draw M independent identically distributed (i.i.d.) initial data and source samples $\{\mathbf{U}_0^i, \mathbf{S}_0^i\}_{i=1}^M$ from the random fields $\{\mathbf{U}_0, \mathbf{S}_0\}$ and then compute cell averages $\mathbf{U}_K^{i,0}, \mathbf{S}_K^{i,0}$ of $\mathbf{U}_0^i, \mathbf{S}_0^i$ in each cell K of a given mesh $\mathcal{T}$.
2. **Solve:** For each sample $\{\mathbf{U}_{\mathcal{T}}^{i,0}, \mathbf{S}_{\mathcal{T}}^{i,0}\}$, the underlying conservation law (1) is solved numerically by the finite volume method [7,5,4]. We denote the FVM solutions by $\mathbf{U}_{\mathcal{T}}^{i,n}$, i.e. by cell averages $\{\mathbf{U}_K^{i,n} : K \in \mathcal{T}\}$ at the time level t^n.
3. **Estimate Statistics:** We estimate the expectation of the random solution field with the sample mean (ensemble average) of the approximate solution:

$$E_M[\mathbf{U}_{\mathcal{T}}^n] := \frac{1}{M} \sum_{i=1}^{M} \mathbf{U}_{\mathcal{T}}^{i,n}, \quad M = \mathcal{O}(\Delta x^{-s/2}), \tag{3}$$

   where s denotes the convergence rate of the FVM solver.

The error of MC-FVM asymptotically scales [10] as $(\text{Work})^{-s/(d+1+2s)}$, making MC-FVM method computationally infeasible when high accuracy is needed.

The *multi-level* Monte Carlo finite volume method (MLMC-FVM) was recently proposed in [10,8]. The key idea behind MLMC-FVM is to simultaneously draw MC samples on a hierarchy of nested grids. There are four main steps:

1. **Nested Meshes:** Consider *nested* triangulations $\{\mathcal{T}_\ell\}_{\ell=0}^{\infty}$ of the spatial domain with corresponding mesh widths Δx_ℓ that satisfy $\Delta x_\ell = \mathcal{O}(2^{-\ell}\Delta x_0)$, where Δx_0 - mesh width of the coarsest resolution at the lowest level $\ell = 0$.
2. **Sample:** For each level of resolution $\ell \in \mathbb{N}_0$, we draw M_ℓ independent identically distributed (i.i.d) samples $\{\mathbf{U}_{0,\ell}^i, \mathbf{S}_{0,\ell}^i\}$ with $i = 1, 2, \ldots, M_\ell$ from the random fields $\{\mathbf{U}_0, \mathbf{S}_0\}$ and approximate these by cell averages.
3. **Solve:** For each resolution level ℓ and each realization $\{\mathbf{U}_{0,\ell}^i, \mathbf{S}_{0,\ell}^i\}$, the underlying balance law (1) is solved by the finite volume method [7,5,4] with mesh width Δx_ℓ; denote solutions by $\mathbf{U}_{\mathcal{T}_\ell}^{i,n}$ at the time t^n and mesh level ℓ.
4. **Estimate Solution Statistics:** Fix the highest level $L \in \mathbb{N}_0$. We estimate the expectation of the solution (random field) with the following estimator:

$$E^L[\mathbf{U}(\cdot, t^n)] := \sum_{\ell=0}^{L} E_{M_\ell}[\mathbf{U}_{\mathcal{T}_\ell}^n - \mathbf{U}_{\mathcal{T}_{\ell-1}}^n], \tag{4}$$

   with E_{M_ℓ} being the MC estimator defined in (3) for the level ℓ. Higher statistical moments can be approximated analogously (see, e.g., [10]).

In order to equilibrate statistical and spatio-temporal discretization errors in (4), the following number of samples on each discretization level ℓ is needed [10]:

$$M_\ell = \mathcal{O}(2^{2(L-\ell)s}). \tag{5}$$

Notice that most of MC samples are computed on the coarsest mesh level $\ell = 0$, whereas only a small fixed number of MC samples are needed on the finest mesh $\ell = L$. The error vs. work estimate for MLMC-FVM is given by [8,10],

$$\text{error} \lesssim (\text{Work})^{-s/(d+1)} \log(\text{Work}). \tag{6}$$

The above estimate shows that MLMC-FVM is superior to MC-FVM. In particular, at the relative error level of 1%, MLMC-FVM was shown to be approximately two orders of magnitude faster than MC-FVM [8,10].

2 Highly Scalable Implementation of MLMC-FVM

MLMC-FVM is *non-intrusive* as any standard FVM solver can be used in step 3. Furthermore, MLMC-FVM is amenable to *efficient parallelization* as most of data interacts only in step 4. We describe parallelization of nontrivial steps 2-4.

2.1 Robust Pseudo Random Number Generation

In step 2, we draw samples for $\{\mathbf{U}_0, \mathbf{S}_0\}$ with a given probability distribution. Here, a robust random number generator (RNG) is needed. Inconsistent seeding and insufficient period length of the RNG might cause correlations in presumably i.i.d. draws which might potentially lead to biased solutions, see Figure 6 of [8].

For the numerical simulations reported below, we used `WELL512a` RNG (with period $2^{512} - 1$) from the `WELL`-series [6] of pseudo random number generators.

2.2 A Priori Estimates for Computational Work

In step 3 of the MLMC-FVM algorithm, the conservation law (2) is solved for each draw of the initial data. This is done by deterministic solver `ALSVID` [1]. The key issue in the parallel implementation of the multiple concurrent solve steps is to distribute computational work evenly among the cores. The FVM algorithm consists of computing fluxes across all cell interfaces and then updating cell averages via the explicit stable time stepping routine [7]. The computational complexity of (explicit) numerical flux approximations is of order equal to the number of cells $N = \#\mathcal{T}$ in the mesh $\mathcal{T}$,

$$\text{Work}^{\text{step}}_{\mathcal{T}} = \text{Work}^{\text{step}}(\Delta x) = \mathcal{O}(N) = \mathcal{O}(\Delta x^{-d}), \tag{7}$$

where Δx denotes the mesh width of triangulation $\mathcal{T}$.

To ensure the stability of the FVM scheme, a CFL condition [7] is imposed on the time step size $\Delta t := t^{n+1} - t^n$, which forces

$$\Delta t = \mathcal{O}(\Delta x). \tag{8}$$

Hence, the computational work $\text{Work}^{\text{det}}_{\mathcal{T}}$ for *one* complete *deterministic* solve using the FVM method on the triangulation $\mathcal{T}$ with mesh width Δx is given by multiplying the work for one step (7) by the total number of steps $\mathcal{O}(\Delta t^{-1})$,

$$\text{Work}^{\text{det}}_{\mathcal{T}} = \text{Work}^{\text{step}}_{\mathcal{T}} \cdot \mathcal{O}(\Delta t) \overset{(8)}{=} \mathcal{O}(\Delta x^{-d}) \cdot \mathcal{O}(\Delta x^{-1}) = \mathcal{O}(\Delta x^{-(d+1)}). \tag{9}$$

In most explicit FVM schemes [7], lower order terms $\mathcal{O}(\Delta x^{-d})$ in (9) are negligible, even on a very coarse mesh. Hence, we assume a stricter version of (9),

$$\mathrm{Work}_{\mathcal{T}}^{\mathrm{det}} = K\Delta x^{-(d+1)}, \tag{10}$$

where constant K depends on FVM that is used and on the time horizon $t > 0$, but does *not* depend on mesh width Δx. Finally, by $\mathrm{Work}_M(\Delta x)$ we denote the computational work needed for a MC-FVM algorithm performing M solves,

$$\mathrm{Work}_M(\Delta x) = M \cdot \mathrm{Work}^{\mathrm{det}}(\Delta x) = M \cdot K\Delta x^{-(d+1)}. \tag{11}$$

Next, we describe our load balancing strategy needed for the step 3.

2.3 Static Load Balancing

In what follows, we assume *a homogeneous computing environment* meaning that all cores are assumed to have identical CPUs and RAM per node, and equal bandwidth and latency to all other cores. There are 3 levels of parallelization: across mesh resolution levels, across MC samples and *inside* the deterministic solver using domain decomposition (see example in Figure 1). Domain decomposition is used only in the few levels with the finest mesh resolution. On these levels, the number of MC samples is small. However, these levels require most of the computational effort. For the finest level $\ell = L$ we *fix* the number of cores C_L,

$$C_L \quad = \quad D_L \quad \times \quad P_L, \tag{12}$$

where for every level $0 \leq \ell \leq L$, D_ℓ denotes the number of subdomains and P_ℓ denotes the number of "samplers" - groups of cores, where every such group computes some portion of required M_ℓ Monte Carlo samples at level ℓ. We assume that each subdomain is computed on exactly one core and denote the total number of cores at level $0 \leq \ell \leq L$ by C_ℓ, i.e.

$$C_\ell = D_\ell P_\ell, \quad \forall \ell \in \{0, \dots, L\}. \tag{13}$$

The total computational work for $E_{M_\ell}[\mathbf{U}^n_{\mathcal{T}_\ell} - \mathbf{U}^n_{\mathcal{T}_{\ell-1}}]$ is then given by (11),

$$\mathrm{Work}_\ell := \mathrm{Work}_{M_\ell}(\Delta x_\ell) + \mathrm{Work}_{M_\ell}(\Delta x_{\ell-1}). \tag{14}$$

Hence the ratio of computational work for the remaining levels $\ell \in \{L-1, \dots, 1\}$ can be obtained recursively by inserting (5) into a priori work estimates (11):

$$\begin{aligned} \frac{\mathrm{Work}_\ell}{\mathrm{Work}_{\ell-1}} &= \frac{M_L 2^{2(L-\ell)s} K \left(\Delta x_\ell^{-(d+1)} + \Delta x_{\ell-1}^{-(d+1)}\right)}{M_L 2^{2(L-(\ell-1))s} K \left(\Delta x_{\ell-1}^{-(d+1)} + \Delta x_{\ell-2}^{-(d+1)}\right)} \\ &= \frac{2^{-2\ell s}}{2^{-2(\ell+1)s}} \frac{\Delta x_\ell^{-(d+1)} \left(1 + 2^{-(d+1)}\right)}{\Delta x_\ell^{-(d+1)} \left(2^{-(d+1)} + 2^{-2(d+1)}\right)} = 2^{d+1-2s}. \end{aligned} \tag{15}$$

For level $\ell = 0$, the term $\mathbf{U}^n_{\mathcal{T}_{-1}}$ in $E_{M_0}[\mathbf{U}^n_{\mathcal{T}_0} - \mathbf{U}^n_{\mathcal{T}_{-1}}]$ is known ($\equiv 0$), hence (15) provides a lower bound rather than an equality, i.e. $\mathrm{Work}_0 \le \mathrm{Work}_1/(2^{d+1-2s})$.

Consequently, the positive integer parameters D_L and $P_L \le M_L$ recursively determine the number of cores needed for each level $\ell < L$ via the relation

$$C_\ell = \left\lceil \frac{C_{\ell+1}}{2^{d+1-2s}} \right\rceil, \quad \forall \ell < L. \tag{16}$$

Notice, that the denominator 2^{d+1-2s} in (16) is a *positive integer* (a power of 2) provided $s \in \mathbb{N}/2$ and $s \le (d+1)/2$. However, in case $s < (d+1)/2$, we have

$$2^{d+1-2s} \ge 2, \tag{17}$$

which (when L is large) leads to inefficient load distribution for levels $\ell \le \ell^*$, where each successive level needs *less* than one core:

$$\ell^* := \min\{0 \le \ell \le L : C_{\ell+1} < 2^{d+1-2s}\}, \quad \text{i.e.} \quad C_{\ell^*} \le 1/2. \tag{18}$$

We investigate the amount of *total* computational work ($\mathrm{Work}_{\{0,\dots,\ell^*\}}$) required for such "inefficient" levels $\ell \in \{0,\dots,\ell^*\}$:

$$\begin{aligned} \mathrm{Work}_{\{0,\dots,\ell^*\}} &:= \sum_{\ell=0}^{\ell^*} \mathrm{Work}_\ell \overset{(15)}{=} \sum_{\ell=0}^{\ell^*} \frac{\mathrm{Work}_{\ell^*}}{2^{(d+1-2s)(\ell^*-\ell)}} \le \sum_{\ell=-\infty}^{\ell^*} \frac{\mathrm{Work}_{\ell^*}}{2^{(d+1-2s)(\ell^*-\ell)}} \\ &= \frac{\mathrm{Work}_{\ell^*}}{1-(2^{d+1-2s})^{-1}} \overset{(17)}{\le} \frac{\mathrm{Work}_{\ell^*}}{1-\frac{1}{2}} = 2\cdot \mathrm{Work}_{\ell^*}. \end{aligned} \tag{19}$$

For the sake of simplicity, assume that P_L and D_L are nonnegative integer powers of 2. Under this assumption, definition (18) of ℓ^* together with recurrence relation (16) *without* rounding up ($\lceil\cdot\rceil$) implies that $C_{\ell^*} \le 1/2$. Hence, total work estimate (19) for *all* levels $\ell \in \{0,\dots,\ell^*\}$ translates into an estimate for sufficient number of cores, which, instead of ℓ^*+1, turns out to be *only* 1:

$$\mathrm{Work}_{\{0,\dots,\ell^*\}} \le 2\cdot \mathrm{Work}_{\ell^*} \quad \longrightarrow \quad C_{\{0,\dots,\ell^*\}} \le 2\cdot C_{\ell^*} \le 2\cdot\frac{1}{2} = 1. \tag{20}$$

The implementation of (20) (i.e. multiple levels per 1 core) is *essential* to obtain efficient and highly scalable parallelization of MLMC-FVM when $s < \frac{d+1}{2}$.

The example of static load distribution for MLMC-FVM is given in Figure 1. Next, we describe our implementation of this load balancing using `C++` and `MPI`.

2.4 Implementation Using C++ and MPI

In what follows, we assume that each MPI process is running on its own core. Simulation is divided into 3 main phases - initialization, simulation and data collection; key concepts for the implementation of the static load balancing algorithm for each phase is described below.

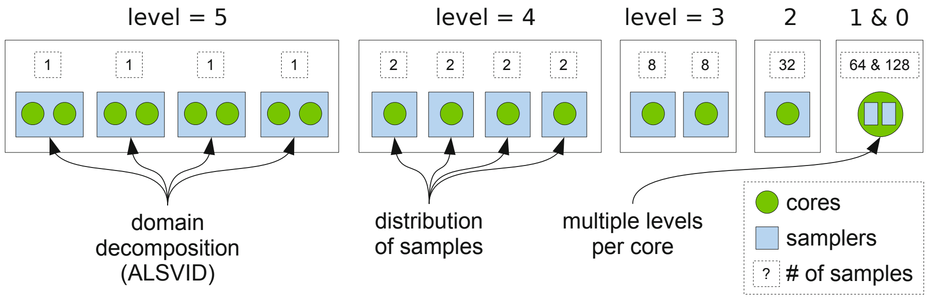


Fig. 1. Static load distribution structure: $L = 5, M_L = 4, d = 1, s = \frac{1}{2}, D_L = 2, P_L = 4$

Phase 1 - Initialization:

– **MPI Groups and Communicators.** By default, message passing in `MPI` is done via the main communicator `MPI_COMM_WORLD` which connects *all* processes. Each process has a prescribed unique non-negative integer called *rank*. The process with the rank 0 is called *root*. Apart from `MPI_COMM_WORLD`, we use `MPI_Group_range_incl()` and `MPI_Comm_create()` to create sub-groups and corresponding inter-communicators, this way empowering message passing within particular *subgroups* of processes. Such communicators ease the use of collective reduction operations within some particular *subgroup* of processes. We implemented 3 types of communicators (see Figure 2):

  1. **Domain** communicators connect processes within each sampler; these precesses are used for domain decomposition of one physical mesh.
  2. **Sampler** communicators connect processes that work on the MC samples at the *same* mesh level.
  3. **Level** communicators connect only the processes (across *all* levels) that are roots of *both* domain and sampler communicators,

  where, analogously to `MPI_COMM_WORLD`, every process has a unique rank in each of the communicators 1-3; processes with rank 0 in domain communicators are called *domain roots*, in sampler communicators - *sampler roots*, and in level communicators - *level roots*. `MPI_COMM_WORLD` is used *only* in `MPI_Init()`, `MPI_Finalize()` and `MPI_Wtime()`. Figure 2 depicts all non-trivial communicators and roots for the example setup as in Figure 1.

– **Seeding RNG.** To deal with the seeding issues mentioned in subsection 2.1, we *injectively* (i.e. one-to-one) map the *unique* rank (in `MPI_COMM_WORLD`) of each process that is root in *both* domain and sampler communicators to an element in the hardcoded array of prime seeds. Then *each* core generates random vectors of real numbers for MC samples $\{\mathbf{U}^i_{0,\ell}, \mathbf{S}^i_{0,\ell}\}$. Generating the full *sequence* of samples on domain and sampler roots and scattering/broadcasting samples via sampler/domain communicators should be avoided; this introduces unnecessary communication and memory overheads and makes simulations with large stochastic dimensions infeasible [9].

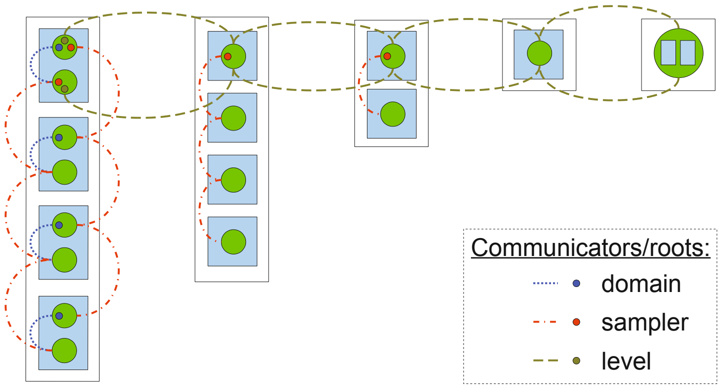


Fig. 2. Structure and root processes of the communicators for setup as in Figure 1

Phase 2 - Simulation:

– **FVM solves for 2 mesh levels.** FVM solves of each sample are performed for $E_{M_\ell}[\mathbf{U}^n_{\mathcal{T}_\ell}]$ and for $E_{M_\ell}[\mathbf{U}^n_{\mathcal{T}_{\ell-1}}]$, and then combined into $E_{M_\ell}[\mathbf{U}^n_{\mathcal{T}_\ell} - \mathbf{U}^n_{\mathcal{T}_{\ell-1}}]$.
– **Inter-domain communication.** Cell values near interfaces of adjacent subdomains are exchanged asynchronously with `MPI_Isend()` and `MPI_Recv()`.

Phase 3 - Data Collection and Output:

– **MC Estimator.** For each level, statistical estimates are collectively reduced with `MPI_Reduce()` into sampler roots using sampler communicators; then MC estimators (3) for mean and variance (see subsection 2.5) are obtained.
– **MLMC Estimator.** MC estimators from different levels are finally combined via level communicators to level roots to obtain MLMC estimator (4).
– **Parallel Data Output.** Each process that is *both* sampler root and level root writes out the final result. Hence, the number of parallel output files is equal to D_L, i.e. the number of subdomains on the finest mesh level.

This concludes the discussion of static load balancing and of step 3 of MLMC-FVM. In step 4, the results are combined to compute sample mean and variance.

2.5 Variance Computation for Parallel Runs

A numerically *stable* serial variance computation algorithm (so-called "online") is given as follows [11]: set $\bar{u}^0 = 0$ and $\Phi^0 = 0$; then proceed recursively,

$$\bar{u}^i = \sum_{j=1}^{i} u^j/i, \quad \Phi^i := \sum_{j=1}^{i}(u^j - \bar{u}^i)^2 = \Phi^{i-1} + (u^i - \bar{u}^i)(u^i - \bar{u}^{i-1}). \tag{21}$$

Then, the unbiased mean and variance estimates are given by:

$$E_M[u] = \bar{u}^M, \quad \mathrm{Var}_M[u] = \Phi^M/(M-1). \tag{22}$$

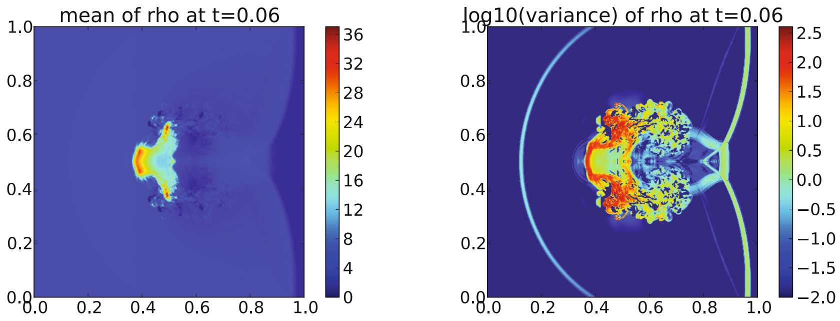
For parallel architectures, assume we have 2 cores, A and B, each computing M_A and M_B ($M = M_A + M_B$) number of samples, respectively. Then an *unbiased* estimate for mean and variance can be obtained by [3]

$$E_M[u] = \frac{M_A E_{M_A}[u] + M_B E_{M_B}[u]}{M}, \quad \mathrm{Var}_M[u] = \frac{\Phi^M}{M-1}, \tag{23}$$

$$\text{where:} \quad \Phi^M = \Phi^{M_A} + \Phi^{M_B} + \delta^2 \cdot \frac{M_A \cdot M_B}{M}, \quad \delta = E_{M_B}[u] - E_{M_A}[u]. \tag{24}$$

Then, for any finite number of cores formula (23) is applied recursively.

This finishes the discussion of the parallel implementation issues for MLMC-FVM algorithm. In Figure 3 we provide an example of solution obtained using MLMC-FVM solver, i.e. the mean and the variance at simulation time $t = 0.06$ of the cloud shock problem of Euler equations of compressible gas dynamics [8]. Next, we investigate the efficiency of the parallelization.



L	M_L	grid size	CFL	cores	runtime	efficiency
9	8	4096x4096	0.4	1023	5:38:17	96.9%

Fig. 3. Uncertain solution to the cloud shock problem of Euler equations of compressible gas dynamics using MLMC-FVM solver. Mean of the random mass density is plotted on the left and the logarithm of variance is plotted on the right. For the detailed description of the uncertain initial data $\mathbf{U}_0$ and further comments refer to [8].

3 Efficiency and Linear Scaling in Numerical Simulations

The static load balancing algorithm was tested on a series of benchmarks for hyperbolic solvers; refer to [8] for Euler equations of gas dynamics, [8,5] for Magnetohydrodynamics (MHD) equations of plasma physics, and [9] for shallow water equations with uncertain bottom topography. The runtime of all aforementioned simulations was measured by the *wall clock time*, accessible by `MPI_Wtime()` routine [12]. Parallel efficiency is defined as a fraction of *simulation time* (which excludes time spent for `MPI` communications and idling) over *wall clock time*,

$$\text{efficiency} := 1 - \frac{(\text{total clock time of all MPI routines})}{(\#\text{cores}) \times (\text{wall clock time})}. \tag{25}$$

In Figure 4 we verify *strong scaling* (fixed discretization and sampling parameters while increasing #cores) and in Figure 5 we verify *weak scaling* (problem size is proportional to #cores) of our implementation. In both cases, we maintained scaling up to almost 4000 cores at high efficiency. Simulations were executed on Cray XE6 (see [13]) with 352 12-core AMD Opteron CPUs (2.1 GHz), 32 GB DDR3 memory per node, 10.4 GB/s Gemini interconnect. We believe that our parallelization algorithm will scale linearly for a much larger number of cores. Labels "MLMC" and "MLMC2" in Figures 4 - 5 indicate that first and second order accurate FVM solvers were used, i.e. $s = 1/2$ and $s = 1$ in (5), respectively.

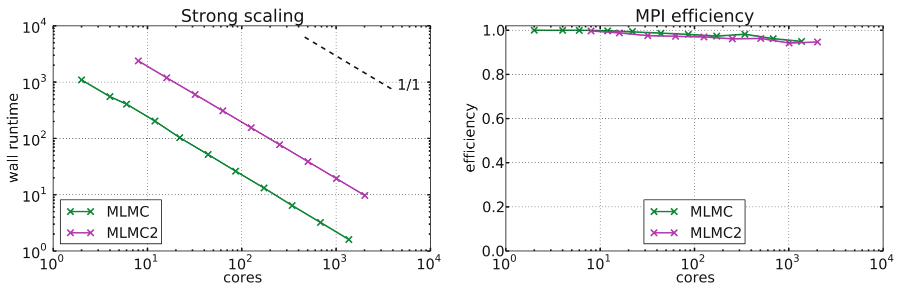


Fig. 4. Strong scaling up to 4000 cores. The inferior scalability of the domain decomposition method (DDM) due to additional networking between sub-domain boundaries has no significant influence on the overall scaling.

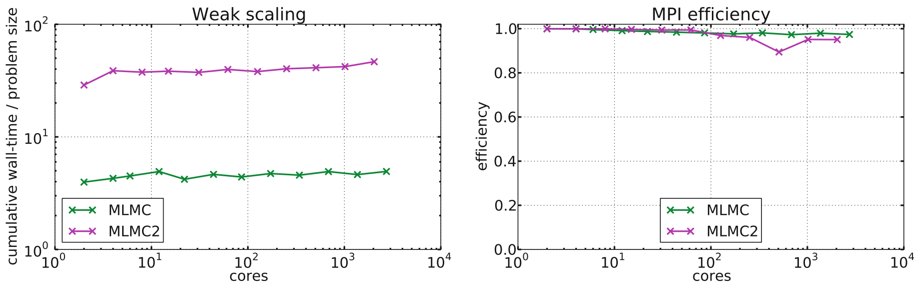


Fig. 5. Weak scaling up to 4000 cores. Analogously as in Figure 4, the inferior scalability of DDM has no significant influence on the overall scaling.

4 Conclusion

MLMC-FVM algorithm is superior to standard MC algorithms for uncertainty quantification in hyperbolic conservation laws, and yet, as most sampling algorithms, it still scales linearly w.r.t. number of uncertainty sources. Due to its non-intrusiveness, MLMC-FVM was efficiently parallelized for multi-core architectures. Strong and weak scaling of our implementation `ALSVID-UQ` [2] of the proposed static load balancing was verified on the high performance clusters [14,13] in multiple space dimensions. The suite of benchmarks included Euler equations of gas dynamics, MHD equations [8], and shallow water equations [9].

Acknowledgments. This work is performed under ETH interdisciplinary research grant CH1-03 10-1. CS acknowledges partial support by the European Research Council under FP7 grant ERC AdG 247277-STAHDPDE. JŠ is grateful to Stefan Pauli and Peter Arbenz for their contributions. We thank teams of Brutus [14] and CSCS [13] for their extensive support.

References

1. ALSVID, `http://folk.uio.no/mcmurry/amhd`
2. ASLVID, `http://www.sam.math.ethz.ch/alsvid-uq`
3. Chan, T.F., Golub, G.H., LeVeque, R.J.: Updating Formulae and a Pairwise Algorithm for Computing Sample Variances. STAN-CS-79-773 (1979)
4. Fjordholm, U.S., Mishra, S., Tadmor, E.: Well-balanced, energy stable schemes for the shallow water equations with varying topology. J. Comput. Phys. 230, 5587–5609 (2011)
5. Fuchs, F., McMurry, A.D., Mishra, S., Risebro, N.H., Waagan, K.: Approximate Riemann solver based high-order finite volume schemes for the Godunov-Powell form of ideal MHD equations in multi-dimensions. Comm. Comp. Phys. 9, 324–362 (2011)
6. L'Ecuyer, P., Panneton, F.: Fast Random Number Generators Based on Linear Recurrences Modulo 2. ACM Trans. Math. Software 32, 1–16 (2006)
7. LeVeque, R.A.: Numerical Solution of Hyperbolic Conservation Laws. Cambridge Univ. Press (2005)
8. Mishra, S., Schwab, Ch., Šukys, J.: Multi-level Monte Carlo finite volume methods for nonlinear systems of conservation laws in multi-dimensions. J. Comp. Phys. 231(8), 3365–3388 (2011)
9. Mishra, S., Schwab, Ch., Šukys, J.: Multi-level Monte Carlo Finite Volume methods for shallow water equations with uncertain topography in multi-dimensions. SIAM Journal of Scientific Computing (2011) (submitted), `http://www.sam.math.ethz.ch/reports/2011/70`
10. Mishra, S., Schwab, Ch.: Sparse tensor multi-level Monte Carlo Finite Volume Methods for hyperbolic conservation laws with random initial data. Math. Comp. (2011) (to appear), `http://www.sam.math.ethz.ch/reports/2010/24`
11. Welford, B.P.: Note on a Method for Calculating Corrected Sums of Squares and Products. Technometrics 4, 419–420 (1962)
12. MPI: A Message-Passing Interface Standard. Version 2.2 (2009), `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`
13. Cray XE6, Swiss National Supercomputing Center (CSCS), Manno, `www.cscs.ch`
14. Brutus, ETH Zürich, de.wikipedia.org/wiki/Brutus_(Cluster)

A Parallel Algorithm for Minimizing the Number of Routes in the Vehicle Routing Problem with Time Windows

Mirosław Błocho[1,3] and Zbigniew J. Czech[1,2]

[1] Silesia University of Technology, Gliwice, Poland
Zbigniew.Czech@polsl.pl
[2] Silesia University, Sosnowiec, Poland
Zbigniew.Czech@polsl.pl
[3] ABB ISDC, Krakow, Poland
blochom@gmail.com

Abstract. A parallel algorithm for minimizing the number of routes in the vehicle routing problem with time windows (VRPTW) is presented. The algorithm components cooperate periodically by exchanging their best solutions with the lowest number of routes found to date. The objective of the work is to analyze speedup, achieved accuracy of solutions and scalability of the MPI implementation. For comparisons the selected VRPTW tests are used. The derived results justify the proposed parallelization concept. By making use of the parallel algorithm the twelve new best-known solutions for Gehring and Homberger's benchmarking tests were found.

Keywords: parallel algorithm,vehicle routing problem with time windows, guided local search, heuristics, approximation algorithms, MPI library.

1 Introduction

This work presents a parallel algorithm for minimizing the number of routes in the vehicle routing problem with time windows (VRPTW). It extends our previous efforts concerning the improved version of the algorithm [2] originated from the heuristic by Nagata and Bräysy [19]. The sequential version of the improved algorithm is based on the notion of the ejection pool used in the heuristic by Lim and Zhang [16], combined with the guided local searches and the diversification strategy [30]. The components of the parallel algorithm cooperate periodically by exchanging their best solutions to the VRPTW.

The VRPTW belongs to a large family of the vehicle routing problems (VRP) which have numerous practical applications such as: the school bus routing, newspapers and mail delivery, armoured car routing, rail distribution, airline fleet routing and many others [29]. Apart from the practical significance, the VRPTW is also NP-hard in the strong sense as it contains several NP-hard optimisation problems such as TSP and bin packing [15].

The current state-of-the-art heuristics for the VRPTW can be divided into improvement heuristics (local searches), construction heuristics and metaheuristics. The improvement heuristics change the current solution iteratively by executing local searches for finding better neighboring solutions. The neighborhood contains the set of solutions that can be obtained from the base solution by exchanging customers or edges. The most successful applications of the improvement heuristics can be found in Bräysy [3], Potvin and Rousseau [20], Thompson and Psaraftis [28], Caseau and Laburthe [4], and Russell [24].

The construction heuristics generate a feasible solution by inserting customers iteratively into the partial routes due to specific criteria, such as maximum savings or minimum additional distance never violating the solution feasibility. The examples of such heuristics are described in Solomon [26], Potvin and Rousseau [21], Dullaert and Bräysy [8] and Mester [17].

The metaheuristics are based on exploring the solution space in order to identify better solutions and they often contain standard route construction as well as the improvement heuristics. As distinct from the classical approaches, the metaheuristics allow infeasible intermediate solutions and a solution deteriorating in the search process to escape a local minima. The best performance for the VRPTW was obtained by the application of the evolution strategies [22], tabu searches [11] and genetic algorithms [12]. For example the evolution strategies were applied by Gehring and Homberger ([9], [10], [13]) and Mester [18] while the tabu searches with excellent results were applicated by Taillard et al. [27], Schulze and Fahle [25], Chiang and Russell [5], Rochat and Taillard [23] and Cordeau et al. [6]. In turn, the genetic algorithms were successfully applied by Jung and Moon [14] and Berger et al. [1].

The VRPTW is defined on a complete graph $G = (V, E)$ with a set of vertices $V = \{v_0, v_1, ..., v_N\}$ and a set of edges $E = \{(v_i, v_j)\colon v_i, v_j \in V, i \neq j\}$. Node v_0 represents a depot and the set of nodes $\{v_1, ..., v_N\}$ represents the customers to be serviced. With each node $v_i \in V$ are associated a non-negative service time s_i ($s_0 = 0$), a non-negative load q_i ($q_0 = 0$) and a time window $[e_i, l_i]$. Each edge (v_i, v_j) contains a non-negative travel time c_{ij} and a travel distance d_{ij}. A solution to the VRPTW is the set of m routes on graph G such that (1) each route starts and ends at the depot; (2) every customer v_i belongs to exactly one route; (3) the total load of each route does not exceed Q; (4) the service at each customer v_i begins between e_i and l_i; (5) every vehicle leaves the depot and returns to it between e_0 and l_0; (6) the number of routes and total travelled distance are minimized.

The MPI (Message Passing Interface) implementation of the parallel algorithm for minimizing the number of routes in the VRPTW is presented. The objective of the work is to analyze speedup, achieved accuracy of solutions and scalability of this parallel heuristic implementation. The remainder of this paper is arranged as follows. In section 2 the parallel algorithm is presented. Section 3 describes the MPI implementation of the algorithm. Section 4 contains the discussion of the experimental results. Section 5 concludes the paper.

2 Parallel Algorithm

The parallel algorithm for minimizing the number of routes in the VRPTW extends the improved sequential version of the algorithm by Błocho and Czech [2]. The parallel algorithm contains N components which are executed as processes P_1, P_2, ..., P_N (Alg. 1).

```
1  for $P_i \leftarrow P_1$ to $P_N$ do in parallel
2      initialize the communication requests
3      create the initial solution $\sigma$
4      $TC \leftarrow false$ // termination condition
5      while not $TC$ do
6          $\sigma \leftarrow$ RemoveRoute($\sigma$)
           // cooperation during RemoveRoute function execution
7      end
8      gather solutions from all processes in process $P_1$ and produce
       the best solution $\sigma_{best}$
9      free the communication requests
10 end
```

Algorithm 1. Parallel algorithm (TC- termination condition)

The framework of the heuristic (Alg. 1) consists of the consecutive remove route steps performed until the total computation time reaches a specified limit T_{max}.

The *RemoveRoute* function (Alg. 2) is started by choosing and removing a route randomly from the current solution σ. The customers from this route are used to initialize the *Ejection Pool* (EP) that is used to hold the set of customers missing from σ. The penalty counters $p[v_i]$ initialized at the beginning of the function indicate how many times the attempts to insert the given customers failed. The bigger value of the penalty counter for a specified customer, the more difficult is to reinsert it into the solution [19].

After the random route is removed from the current solution, continuous attempts to include the ejected customers into the rest of the routes are performed (Alg. 3). These attempts are carried out until all customers from the *Ejection Pool* are inserted or the execution time of the algorithm reaches a specified time limit T_{max} [2]. $S^{fe}_{insert}(v_i, \sigma)$ is defined as a set of the feasible partial solutions that are obtained by inserting the customer v_i into all insertion positions in σ. If $S^{fe}_{insert}(v_i, \sigma)$ is empty then the function *Squeeze* (line 7 in Alg. 3) is called in order to help the insertion of the selected customer into σ. The idea of this function is to choose the temporarily infeasible insertion with the minimal penalty function value $F_p(\sigma)$ defined in [19].

If the *Squeeze* function fails then it informs that the selected customer v_i was not inserted into the solution σ. The value of the penalty counter for a chosen customer is then increased (line 10 in Alg. 3) and after that the ejections of the

```
begin
    $\sigma \leftarrow$ RemoveRouteInitStep($\sigma$)
    choose and remove a route randomly from the solution $\sigma$
    initialize $EP$ with the customers from the removed route
    initialize the penalty counters $p[v_i] \leftarrow 1 (i = 1, ..., N)$
    if $T_{curr} < T_{max}$ then
        while $EP \neq \emptyset$ do
            $\sigma \leftarrow$ RouteEliminationStep($\sigma$)
            if $c_{re}$ mod $f_c = 0$ then
                $TC \leftarrow$ CooperationStep ($\sigma$)
                if $TC = true$ then
                    break
            end
        end
    else
        $TC \leftarrow$ CooperationStep ($\sigma$)
    end
    if $EP \neq \emptyset$ then
        restore $\sigma$ to the beginning solution
    end
    return $\sigma$
end
```

Algorithm 2. RemoveRoute(σ) - minimizing the number of routes (c_{re} - route elimination steps count; f_c - frequency of the processes communication)

```
begin
    $c_{re} \leftarrow c_{re} + 1$ // increase route elimination steps count
    select and eject the customer $v_i$ from EP using LIFO stack
    if $S^{fe}_{insert}(v_i, \sigma) \neq \emptyset$ then
        $\sigma \leftarrow$ the new solution $\sigma'$ selected randomly from $S^{fe}_{insert}(v_i)$
    else
        $\sigma \leftarrow$ Squeeze ($v_i$, $\sigma$)
    end
    if $v_i$ is not inserted into $\sigma$ then
        $p[v_i] \leftarrow p[v_i] + 1$
        select $\sigma'$ from $S_{ej}(v_i)$ with min. $P_{sum} = p[v^{(1)}_{out}] + ... + p[v^{(k_m)}_{out}]$
        update $\sigma \leftarrow \sigma'$
        add the ejected customers: $v^{(1)}_{out}, v^{(2)}_{out}, ..., v^{(k_m)}_{out}$ to the EP
        $\sigma \leftarrow$ Perturb ($\sigma$)
    end
    return $\sigma$
end
```

Algorithm 3. RouteEliminationStep(σ) - single elimination step

customers are tested (up to the limit k_m [19]). For this reason, the set $S_{ej}(v_i)$ is being constructed, that contains the solutions with various ejected customers and the customers inserted at different route positions. The *Perturb* function (line 14 in Alg. 3) is used to diversify the search and only the feasible partial solutions are accepted in that step [2].

The parameter f_c indicates how often the processes communicate with each other to exchange the number of routes and the best solutions found to date. This parameter is compared with the route elimination steps count c_{re} of the current process (line 6 in Alg. 2). The chain of the route elimination steps of the process P_2 is updated to the better solution between the best solutions (in our algorithm only with respect to the number of routes) found by processes P_1 and P_2. Such a cooperation is continued until the process P_N obtains the best solution from all the processes P_1 to P_{N-1} and after that, the best result is sent back to the first process P_1. This way of the cooperation between the processes guarantees that after each cooperation step the process P_1 holds the best solution found to date by all the processes used in the communication chain. It also guarantees that each process P_i holds the best solution found in the previous cooperation step by all the processes. The scheme of the cooperation of the processes is presented in Fig. 1.

In order to minimize the cost of the cooperation step during an algorithm execution, the nonblocking operations for both *receive* and *send* functions were introduced. Because of the asynchronous communication, each time at the end of a cooperation step the process has to wait for already started operations to be finished. The successive route elimination steps are performed during that step to take advantage of a waiting time in the beneficial way. A two-grading system of sending data from one process to another was introduced. In the first step only a number of routes is sent to the next process, along with the information whether a solution will be sent as well or not. The full solution is sent only if the number of routes of the current solution is lower than in last cooperation step.

$$\begin{tikzcd}
\sigma_1^{(init)} \arrow[r, Rightarrow] \arrow[d] & \sigma_2^{(init)} \arrow[r, Rightarrow] \arrow[d] & \sigma_3^{(init)} \arrow[r, Rightarrow] \arrow[d] & \bullet\bullet\bullet \arrow[r, Rightarrow] & \sigma_{N-1}^{(init)} \arrow[r, Rightarrow] \arrow[d] & \sigma_N^{(init)} \arrow[r, Rightarrow] \arrow[d] & \sigma_1^{(init)} \arrow[d] \\
\sigma_1^{(1f_c)} \arrow[r, Rightarrow] \arrow[d] & \sigma_2^{(1f_c)} \arrow[r, Rightarrow] \arrow[d] & \sigma_3^{(1f_c)} \arrow[r, Rightarrow] \arrow[d] & \bullet\bullet\bullet \arrow[r, Rightarrow] & \sigma_{N-1}^{(1f_c)} \arrow[r, Rightarrow] \arrow[d] & \sigma_N^{(1f_c)} \arrow[r, Rightarrow] \arrow[d] & \sigma_1^{(1f_c)} \arrow[d] \\
\sigma_1^{(2f_c)} \arrow[r, Rightarrow] \arrow[d] & \sigma_2^{(2f_c)} \arrow[r, Rightarrow] \arrow[d] & \sigma_3^{(2f_c)} \arrow[r, Rightarrow] \arrow[d] & \bullet\bullet\bullet \arrow[r, Rightarrow] & \sigma_{N-1}^{(2f_c)} \arrow[r, Rightarrow] \arrow[d] & \sigma_N^{(2f_c)} \arrow[r, Rightarrow] \arrow[d] & \sigma_1^{(2f_c)} \arrow[d] \\
\bullet \arrow[d] & \bullet \arrow[d] & \bullet \arrow[d] & \bullet\bullet\bullet & \bullet \arrow[d] & \bullet \arrow[d] & \bullet \arrow[d] \\
\sigma_1^{(xf_c)} \arrow[r, Rightarrow] & \sigma_2^{(xf_c)} \arrow[r, Rightarrow] & \sigma_3^{(xf_c)} \arrow[r, Rightarrow] & \bullet\bullet\bullet \arrow[r, Rightarrow] & \sigma_{N-1}^{(xf_c)} \arrow[r, Rightarrow] & \sigma_N^{(xf_c)} \arrow[r, Rightarrow] & \sigma_1^{(xf_c)}
\end{tikzcd}$$

Fig. 1. Scheme of cooperation of the processes ($\sigma_i^{(init)}$- initial solution of the process P_i; $\sigma_i^{(jf_c)}$ - j cooperation step of the process P_i; x - cooperation steps count)

```
begin
    if $processID = 0$ then
        $TC \leftarrow (T_{curr} > T_{max})$
        PerformSendStep ($\sigma$)
        PerformReceiveStep ($\sigma$)
    else
        PerformReceiveStep ($\sigma$)
        PerformSendStep ($\sigma$)
    end
    WaitForSentRequests ($\sigma$)
    return $TC$
end
```

Algorithm 4. CooperationStep(σ) - function of cooperation of processes

3 The MPI Implementation

In the MPI implementation of the parallel algorithm all processes P_i, $i = 1, 2, \ldots,$ N execute the parallel loop shown in Alg. 1. The processes communicate repeatedly with the same parameters list and for this reason the persistent communication requests have been created for all *send* and *receive* operations:

- *SendNoReq* {a persistent request of sending number of routes},
- *SendSolReq* {a persistent request of sending full solution},
- *RecvNoReq* {a persistent request of receiving number of routes},
- *RecvSolReq* {a persistent request of receiving full solution}.

The list of the communication parameters is binded to the persistent requests only once using *MPI_Send_init* or *MPI_Recv_init* function (line 2 in Alg. 1). When the cooperation between the processes is finished the persistent requests have to be freed (line 9 in Alg. 1) using *MPI_Request_free* function.

In the MPI version of the algorithm, *MPI_Start*() function is used to initiate the communication with a persistent request handle and in the algorithm it is used for both *send* and *receive* operations.

Only after both *send* and *receive* operations were invoked, it is necessery to wait for the sent requests to be finished. By taking advantage of the *MPI_Test* function it was possible to test if they are already completed. If not, the successive route elimination steps are executed.

4 Experimental Results

The MPI implementation was run on the Galera (TASK) supercomputer which nodes consisted of 2 Intel Xeon Quad Core 2.33 GHz processors, each with 12

MB level 3 cache. The nodes were connected by the Infiniband DDR fat-free network (throughput 20 Gbps, delay 5 s). The computer was executing Linux operating system. The source code was compiled using Intel 10.1 compiler and MVAPICH1 v. 0.9.9 MPI library.

The selected benchmarking tests proposed by Gehring and Homberger (GH) were used for testing the MPI implementation of the parallel algorithm. The investigations reported in [2] indicated that using the sequential version of the algorithm most of the GH tests can be solved quickly to good accuracy except of tests from C1, C2, RC1, RC2 groups having 400-1000 customers. In order to analyze speedup of the parallel version of this algorithm only those of the GH tests were chosen for which the execution time was one of the longest. The results of the experiments are gathered in Tab. 1 and illustrated in Fig. 2. Overall 150 experiments for RC tests (RC2_6_2, RC2_8_5, RC2_10_5) and 100 experiments for C tests (C2_6_2, C2_8_9, C2_10_7) for 1..16 and for 20, 24, 28, ..., 64 processors were carried out. The results in the table show speedups and the average execution times only for 1, 16, 32, 48 and 64 processors. It can be noticed that the algorithm achieves slightly worse speedups for tests from C1/C2 groups than for tests from RC1/RC2 groups. It may indicate that the additional operations such as communication, idling and synchronization of processes take the larger part for the C1/C2 tests than for RC1/RC2 tests.

By running the parallel algorithm on 32/64 processors 12 new best-known solutions were found and they are reported in Tab. 2.

Table 1. Results for selected GH tests (p – number of processors, $\bar{\tau}$ – average execution time [s], $\mathcal{S}$ – speedup, exp. – number of experiments)

	RC2_6_2			RC2_8_5			RC2_10_5		
p	$\bar{\tau}$	$\mathcal{S}$	exp.	$\bar{\tau}$	$\mathcal{S}$	exp.	$\bar{\tau}$	$\mathcal{S}$	exp.
1	2573.1	1.00	150	369.7	1.00	150	454.7	1.00	150
16	203.9	12.62	150	30.4	12.16	150	35.6	12.64	150
32	91.1	28.24	150	13.3	27.81	150	16.8	27.02	150
48	62.8	40.99	150	9.5	39.01	150	11.9	38.21	150
64	52.4	49.12	150	7.6	48.97	150	9.5	47.88	150
	C2_6_2			C2_8_9			C2_10_7		
p	$\bar{\tau}$	$\mathcal{S}$	exp.	$\bar{\tau}$	$\mathcal{S}$	exp.	$\bar{\tau}$	$\mathcal{S}$	exp.
1	492.1	1.00	100	1564.6	1.00	100	852.7	1.00	100
16	49.2	10.01	100	152.5	10.27	100	90.1	9.47	100
32	19.8	24.92	100	64.6	24.21	100	40.6	21.02	100
48	14.3	34.48	100	49.2	31.82	100	26.1	32.70	100
64	11.0	44.66	100	36.5	42.92	100	21.4	39.92	100

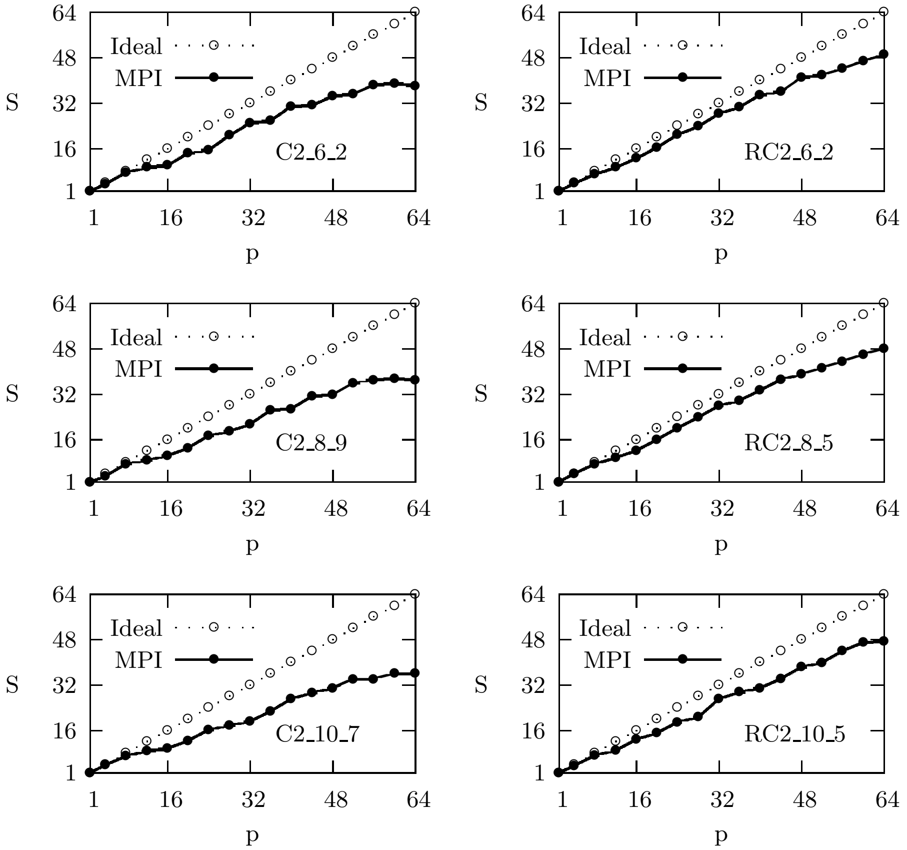


Fig. 2. Speedup $\mathcal{S}$ vs. number of processors p for tests C2_6_2, C2_8_9, C2_10_7, RC2_6_2, RC2_8_5 and RC2_10_5

Table 2. New best-known solutions (N – number of customers, p – number of processors)

Instances	N	p	Best-known	New-best	Time [s]
C2_6_9	600	64	18	17	113
RC2_6_1	600	64	15	14	192
RC2_6_5	600	32	12	11	378
RC2_8_1	800	32	19	18	360
RC2_8_2	800	64	17	16	80
RC2_8_5	800	32	16	15	232
C1_10_2	1000	32	91	90	61
C2_10_3	1000	64	29	28	9980
C2_10_4	1000	32	29	28	463
C2_10_6	1000	32	30	29	394
C2_10_7	1000	32	30	29	3005
C2_10_10	1000	64	29	28	542

5 Conclusions

The parallel algorithm for minimizing the number of routes in the VRPTW along with its MPI implementation were presented. The main advantage of the proposed algorithm is a very short time of obtaining solutions which contain the number of routes equal to or better than the best known solutions. The parallel algorithm displays very good performance in terms of the quality of solutions, robustness and computational effort. The experimental results obtained for the selected tests by Gehring and Homberger indicated that the algorithm scales better for the tests from groups RC1/RC2 than for the tests from groups C1/C2. It may be expected, that the proposed way of parallelization will lead to similar results also in case of other combinatorial optimization problems.

Acknowledgments. We thank the following computing centers where the computations of our project were carried out: Academic Computer Centre in Gdańsk TASK, Academic Computer Centre CYFRONET AGH, Kraków (computing grant 027/2004), Poznań Supercomputing and Networking Center, Interdisciplinary Centre for Mathematical and Computational Modeling, Warsaw University (computing grant G27-9), Wrocław Centre for Networking and Supercomputing (computing grant 30).

References

1. Berger, J., Barkaoui, M., Bräysy, O.: A route-directed hybrid genetic approach for the vehicle routing problem with time windows. INFOR 41, 179–194 (2003)
2. Błocho, M., Czech, Z.J.: An improved route minimization algorithm for the vehicle routing problem with time windows. Studia Informatica 32, No. 3B(99), 5–19 (2011)

3. Bräysy, O., Hasle, G., Dullaert, W.: A multi-start local search algorithm for the vehicle routing problem with time windows. European Journal of Operational Research (2002)
4. Caseau, Y., Laburthe, F.: Heuristics for large constrained vehicle routing problems. Journal of Heuristics 5, 281–303 (1999)
5. Chiang, W.C., Russell, R.A.: A reactive tabu search metaheuristic for the vehicle routing problem with time windows. INFORMS Journal on Computing 9, 417–430 (1997)
6. Cordeau, J.-F., Laporte, G., Mercier, A.: A unified tabu search heuristic for vehicle routing problems with time windows. Journal of the Operational Research Society 52, 928–936 (2001)
7. Czech, Z.J., Mikanik, W., Skinderowicz, R.: Implementing a Parallel Simulated Annealing Algorithm. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 146–155. Springer, Heidelberg (2010)
8. Dullaert, W., Bräysy, O.: Routing with relatively few customers per route. Top (2002)
9. Gehring, H., Homberger, J.: A parallel hybrid evolutionary metaheuristic for the vehicle routing problem with time windows. In: Miettinen, K., Mkel, M., Toivanen, J. (eds.) Proceedings of EUROGEN 1999, pp. 57–64. University of Jyvskyl, Jyvskyl (1999)
10. Gehring, H., Homberger, J.: Parallelization of a two-phase metaheuristic for routing problems with time windows. Asia-Pacific Journal of Operational Research 18, 35–47 (2001)
11. Glover, F.: Future paths for integer programming and links to artificial intelligence. Computers and Operations Research 13, 533–549 (1986)
12. Holland, J.: Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor (1975)
13. Homberger, J., Gehring, H.: Two evolutionary meta-heuristics for the vehicle routing problem with time windows. INFOR 37, 297–318 (1999)
14. Jung, S., Moon, B.-R.: A hybrid genetic algorithm for the vehicle routing problem with time windows. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 1309–1316. Morgan Kaufmann, San Francisco (2002)
15. Kohl, N.: Exact Methods for Time Constrained Routing and Related Scheduling Problems, PhD. Institut for Matematisk Modellering, Danmarks Tekniske Universitet
16. Lim, A., Zhang, X.: A two-stage heuristic with ejection pools and generalized ejection chains for the vehicle routing problem with time windows. Informs Journal on Computing 19, 443–457 (2007)
17. Mester, D.: A parallel dichotomy algorithm for vehicle routing problem with time windows, Working paper, Minerva Optimization Center, Technion, Israel (1999)
18. Mester, D.: An evolutionary strategies algorithm for large scale vehicle routing problem with capacitate and time windows restrictions, Working paper, Institute of Evolution, University of Haifa, Israel (2002)
19. Nagata, Y., Bräysy, O.: A Powerful Route Minimization Heuristic for the Vehicle Routing Problem with Time Windows. Operations Research Letters 37, 333–338 (2009)
20. Potvin, J.-Y., Rousseau, J.-M.: An exchange heuristic for routeing problems with time windows. Journal of the Operational Research Society 46, 1433–1446 (1995)

21. Potvin, J.-Y., Rousseau, J.-M.: A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. European Journal of Operational Research 66, 331–340 (1993)
22. Rechenberg, I.: Evolutionsstrategie. Fromman-Holzboog, Stuttgart (1973)
23. Rochat, Y., Taillard, E.: Probabilistic diversification and intensification in local search for vehicle routing. Journal of Heuristics 1, 147–167 (1995)
24. Russell, R.A.: Hybrid heuristics for the vehicle routing problem with time windows. Transportation Science 29, 156–166 (1995)
25. Schulze, J., Fahle, T.: A parallel algorithm for the vehicle routing problem with time window constraints. Annals of Operations Research 86, 585–607 (1999)
26. Solomon, M.M.: Algorithms for the vehicle routing and scheduling problems with time window constraints. Operations Research 35, 254–265 (1987)
27. Taillard, E., Badeau, P., Gendreau, M., Guertin, F., Potvin, J.-Y.: A tabu search heuristic for the vehicle routing problem with soft time windows. Transportation Science 31, 170–186 (1997)
28. Thompson, P., Psaraftis, H.: Cyclic transfer algorithms for multivehicle routing and scheduling problems. Operations Research 41, 935–946 (1993)
29. Toth, P., Vigo, D. (eds.): The vehicle routing problem. SIAM, PA (2002)
30. Voudouris, C., Tsang, E.: Guided local search. In: Glover, F. (ed.) Handbook of Metaheuristics, pp. 185–218. Kluwer (2003)

Towards Parallel Direct SAT-Based Cryptanalysis

Paweł Dudek[1], Mirosław Kurkowski[1], and Marian Srebrny[2,⋆]

[1] Institute of Computer and Information Sciences,
Czestochowa University of Technology,
Dabrowskiego 73, 42-200 Czestochowa, Poland
{mkurkowski,pdudek}@icis.pcz.pl

[2] Institute of Computer Science, Polish Academy of Sciences,
Ordona 21, 01-237 Warsaw, Poland
marians@ipipan.waw.pl

Abstract. In this paper we show a new approach of parallelised and optimised direct SAT-based cryptanalysis for symmetric block ciphers. It is shown how one can code directly in SAT each bit of the plaintext together with its 'route' through the enciphering algorithm steps, code the round key schedule and S-boxes, and eliminate all simple Boolean equivalences and redundancies. We show Boolean coding directly from the analysed cipher's source code, with no intermediate step of generating any auxiliary system of multivariate low-degree equations, as it was the case in SAT-enhanced algebraic cryptanalysis of [4]. This contributes to the results in much shorter formulae. Another speed-up effect we get by parallelising the cryptanalytic effort to some 2^n available processing cores. We report some experimental results on two basic well known symmetric ciphers.

Keywords: cryptanalysis, satisfiability, parallel computation.

1 Introduction

Boolean (or propositional) SATisfiability is a celebrated NP-complete problem. There is no known algorithm that efficiently solves all instances of SAT. It is generally believed that no such algorithm can exist. Still a lot of Boolean formula instances can be solved surprisingly efficiently, even very large formulas emerging in various industrial scale but naturally-occurring decision and optimisation contexts (see [2]). There are many competing algorithms searching for a satisfying valuation for a given Boolean formula. Most of them are highly optimised versions of the DPLL procedure of [5] and [6]. Usually SAT-solvers take input formulas in the conjunctive normal form, CNF. It is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is a propositional variable or the complement of a propositional variable. The Boolean conjunctive normal form is similar to the canonical product of sums form used in circuit theory.

⋆ The third author gratefully acknowledges many useful conversations with Dr. Paweł Morawiecki and his comments.

In this paper we propose a new approach to SAT-based cryptanalysis of symmetric cryptographic algorithms, its optimisation and parallel realization. The main idea of SAT-based cryptanalysis is translating the problem of recovering a secret cryptographic key into a satisfiability instance of some Boolean (propositional) formula which encodes the cipher being analysed. We translate the whole given enciphering algorithm directly into a Boolean formula without any intermediate algebraic system of equations or extra application of any automatic or semiautomatic tools. Some of the propositional variables in this formula represent a plaintext, a cryptographic key and the corresponding ciphertext, respectively. We take an arbitrary plaintext and a key, and compute the corresponding ciphertext. Now, given a plaintext and its ciphertext, we run a SAT-solver to search for the secret key. Our optimisation and parallelisation is given in Section 4. As one of our improvements, we propose parallelisation of the cryptanalityc effort to some available 2^n processing cores, with a random choice of n bits, for some small n, out of those searched for, and assigning them with all the possible combinations of $0-1$ values.

In this work we show the efficiency of our method on two well known symmetric-key block ciphers: the Feistel Network and the DES algorithm. A block cipher is an encryption method that processes the input plaintext in blocks of bits that are fixed in size, typically 64 bits long. The state of a block cipher is reset before processing each block, with an unvarying transformation. In a symmetric-key cipher the transformation is controlled using a second input – the secret key, and the same (or trivially-related) key is used for both encryption and decryption. The key should be a secret known only for two entities maintaining a private communication channel (see [9]).

Choosing those two ciphers for our experiments reported here (although, attacking some other carefully chosen ciphers is our future research goal) we wanted to investigate and illustrate how our method works in the case of well known kind of benchmark ciphers - very often used in practice, with a lot of cryptanalytic research that has been devoted to them, what enables a comparison of our technique and results with related work.

The contribution of this paper is the following: we propose some modifications to the SAT-based cryptanalysis technique and show they turn out giving (slightly) better results than the best ones so far in SAT-based approach (due to [7] and [4]). It has to be emphasised that in general no SAT-based method has broken DES so far. Besides the brute-force, today also differential and linear analysis provide attacks with practical complexity; i.e., attacks that can be experimentally verified (see [3,8]).

The rest of this paper is organised as follows. In section 2, we introduce all basic information on both ciphers mentioned, to the extent necessary for explaining our Boolean encoding method. Section 3 gives a process of a direct Boolean encoding of the ciphers we consider. In section 4, we introduce several optimisation and parallelisation ideas used in our method. In section 5, we present some experimental results we have obtained. Finally, some conclusion and future directions are indicated in the last section.

2 Feistel Network and DES Cipher

In this section, we introduce all the basic information on the Feistel and the DES ciphers needed for understanding our methodology of SAT based cryptanalysis of symmetric cryptographic algorithms.

2.1 Feistel Network

The Feistel cipher is a symmetric-key block algorithm widely used as a design principle of many symmetric ciphers, including the famous Data Encryption Standard (DES). This framework is also commonly known as the Feistel Network (FN). Its algorithm has the advantage that its encryption and decryption procedures are very similar, requiring only a reversal of the key schedule. FN is an iterated algorithm which is executed many times on the same input. FN was the first commercial cipher used in IBM's cipher named Lucifer, designed by Horst Feistel and Don Coppersmith. FN has gained a lot of attention since the U.S. Federal Government adopted the DES algorithm as symmetric encryption standard. Like other parts of the DES, the iterative nature of FN makes implementing the cipher in hardware easier. Due to a simple structure and easy hardware implementation, Feistel-like networks are widely used as a component of various cipher designs. Some examples are these: MISTY1 which is a FN using a three-round Feistel network in its round function, Skipjack uses FN in its permutation, Threefish (part of Skein) uses a Feistel-like MIX function, Blowfish, Camellia, CAST-128, FEAL, ICE, LOKI97, Lucifer, MARS, MAGENTA, RC5, TEA, Twofish, XTEA, GOST 28147-89 (see [9]).

Let F denote the round function and $K_1, \ldots, K_n$ denote a sequence of keys obtained in some way from the main key K for the rounds $1, \ldots, n$, respectively. We use symbol $\oplus$ for denoting the exclusive-OR (XOR) operation.

The basic operations of FN are specified as follows:

1. break the plaintext block into two equal length parts denoted by (L_0, R_0),
2. for each round $i = 0, \ldots, n$, compute $L_{i+1} = R_i$ and $R_{i+1} = L_i \oplus F(R_i, K_i)$.

Then the ciphertext sequence is (R_{n+1}, L_{n+1}).

The structure of FN allows easy method of decryption. Lets recall basic properties of operation $\oplus$: $x \oplus x = 0$, $x \oplus 0 = x$, $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ for all $x, y, z \in \{0, 1\}$.

A given ciphertext (R_{n+1}, L_{n+1}) is decrypted by computing $R_i = L_{i+1}$ and $L_i = R_{i+1} \oplus F(L_{i+1}, K_i)$, for $i = n, \ldots, 0$. It is easy to observe that (L_0, R_0) is the plaintext again. Observe additionally that we have the following equations:

$$\begin{aligned} R_{i+1} \oplus F(L_{i+1}, K_i) &= (L_i \oplus F(R_i, K_i)) \oplus F(L_i, K_i) = \\ &= L_i \oplus (F(R_i, K_i) \oplus F(L_i, K_i)) = L_i \oplus 0 = L_i. \end{aligned}$$

2.2 DES Cipher

The well known Data Encryption Standard (DES) algorithm is a symmetric block cipher that uses a 56-bit key. In 1976 DES was chosen by the US National

Bureau of Standards as an official Federal Information Processing Standard. Since 1976 it had been widely used around the whole world until the late '90s when it was found insecure. This is due to the fact that the 56-bit key size has been too small. In 1999 the organisation *distributed.net* and the *Electronic Frontier Foundation* collaborated to break a DES key in about 22 hours.

There are several attacks known that can break the full 16 rounds of DES with less complexity than a brute-force search: differential cryptanalysis, linear cryptanalysis. In a few recent years, due to tremendous progress in hardware design, they are of practical complexity; i.e, they can be verified experimentally. Differential cryptanalysis, (re)discovered in the late 1990s by Eli Biham and Adi Shamir [3], requires 2^{47} chosen plaintexts to break the full 16 rounds. Linear cryptanalysis, discovered by Mitsuru Matsui [8], needs 2^{43} known plaintexts (see [9]).

Now DES is believed to be strongly secure only in its modified form named Triple DES. In 2002 DES was definitely superseded by another cipher named Rijndael that has been approved as the new Encryption Standard (AES). However, DES and especially its modifications can be still treated as a strong cipher for private applications.

The algorithm consists of 16 identical stages of processing, named rounds. Additionally, an initial and final permutation is used in the beginning and at the end of the algorithm, respectively. Before all the rounds, the block is broken into two 32-bit portions that are processed alternately using some modification of FN. FN structure guarantees that the decryption and encryption are very similar processes from the computation time and practical realization point of view. Like in the FN case, the only difference between encryption and decryption process is that the subkeys are given in computation in the reverse order. The rest of the algorithm is identical, what really simplifies implementation, especially in hardware, as there is no need for separate different encryption and decryption units.

The F-function mixes the portions of the main block together with one of the sub-keys. The output from the F-function is then combined with the second portion of the main block, and both portions are swapped before the next round. After the last round, the portions are not swapped.

The F-function works on half a block (32 bits) at a time and consists of the following four steps:

1. Expansion. The 32-bit half-block is enlarged into 48 bits using some special function by duplicating half of the bits. The output consists of eight 6-bit ($8 \cdot 6 = 48$ bits) pieces, each containing a copy of 4 corresponding input bits, plus a copy of the immediately adjacent bit from each of the input pieces to either side.

2. Key Mixing. The result is combined with a sub-key using operation $\oplus$. Sub-keys are obtained from the main initial encryption key using a special key schedule - one for each round. The schedule used consists of some rotations of bits. For each round a different subset of key bits is chosen.

3. Substitution. After mixing with the subkey, the block is divided into eight 6-bit portions, before processing using the S-boxes. Each of the eight S-boxes is a matrix with four rows and six columns. It can be treated as a non-linear function from $\{0,1\}^6$ into $\{0,1\}^4$. Each S-box replaces a six-tuple input bits with some four output bits. The S-boxes provide a high level of security - without them, the cipher would be linear, and easily susceptible to be broken.

4. Permutation. Finally, the 32 output bits $(8 \cdot 4)$ from the S-boxes are mixed with a next fixed permutation. Called P-box. This is designed in such a way that after expansion, each S-box's output bits go across 6 different S-boxes in the next round of the algorithm.

The key schedule for decryption procedure is similar. The subkeys are in the reversed order than in the encryption procedure.

From the Boolean encoding point of view, it is important to note that all the elementary operations in DES, except the S-boxes can be described as some equivalences (permutations, expansions, rotations). In the case of S-box, it can be encoded as proper implication. The full encoding process will be described in the next section of the paper.

3 Boolean Encoding for Cryptanalysis

This section presents our method of direct Boolean encoding of the two benchmark ciphers. As mentioned above FN constitutes the basic structure for many well respected symmetric ciphers. Hence its Boolean encoding will be helpfull in SAT-based cryptanalysis we want to pursue in the future. As indicated above, each of the elementary operations of FN and DES can be presented as a conjunction of Boolean equivalences and implications. To explain our method of encoding, we start with encoding FN. After that, we show the encoding of the main steps of DES, including the permutations and S-box computations.

3.1 Encoding Feistel Network

For our explanation we consider the Feistel Network with a 64-bit block of a plaintext and a 32-bit key. Let $p_1, \ldots, p_{64}$, $k_1, \ldots, k_{32}$ and $c_1, \ldots, c_{64}$ are the propositional variables representing a plaintext, a key, and the ciphertext, respectively. Observe that following the Feistel algorithm for the first half of ciphertext we have:

$$\bigwedge_{i=1}^{32} (c_i \Leftrightarrow p_{i+32}).$$

As a simple instantiation of function F we use function XOR, denoted by $\oplus$ as before. (Clearly this is a simplest possible example of function F, but at this point we only show our encoding method for the FN structure.) It is easy to observe that for the second half of ciphertext we have:

$$\bigwedge_{i=33}^{64} (c_i \Leftrightarrow (p_i \oplus k_{i-32} \oplus p_{i+32}).$$

Hence, the encoding formula for one round of FN is this:

$$\Phi^1_{Feistel} : \bigwedge_{i=1}^{32} (c_i \Leftrightarrow p_{i+32}) \ \wedge \ \bigwedge_{i=33}^{64} (c_i \Leftrightarrow (p_i \oplus k_{i-32} \oplus p_{i+32})$$

In the case of t rounds of FN we have the following. Let $(p^1_1, \ldots, p^1_{64})$, $(k_1, \ldots, k_{32})$ are a plaintext and a key vectors of variables, respectively. By $(p^j_1, \ldots, p^j_{64})$ and $(c^i_1, \ldots, c^i_{64})$ we describe vectors of variables representing input of j-th round for $j = 2, \ldots, t$ and output of i-th round for $i = 1, \ldots, t-1$. We denote by $(c^t_1, \ldots, c^t_{64})$ the variables of a cipher vector after t-th round, too.

The formula which encodes the whole t-th round of a Feistel Network is as follows:

$$\Phi^t_{Feistel} : \bigwedge_{i=1}^{32} \bigwedge_{s=1}^{t} (c^s_i \Leftrightarrow p^s_{i+32}) \ \wedge \ \bigwedge_{i=1}^{32} \bigwedge_{s=1}^{t} [c^s_{i+32} \Leftrightarrow (p^s_i \ \oplus \ p^s_{i+32} \ \oplus \ k_i)] \wedge$$

$$\wedge \bigwedge_{i=1}^{64} \bigwedge_{s=1}^{t-1} (p^{s+1}_i \Leftrightarrow \ c^s_i).$$

Observe that the last part of $\Phi^t_{Feistel}$ states that the outputs from s-th rounds are the inputs of the $(s+1)$-th.

As we can see, the formula obtained is a conjunction of ordinary, or rather simple, equivalences. It is important from the translating into CNF point of view. The second advantage of this description is that we can automatically generate the formula for many investigated rounds.

3.2 Encoding DES

In the case of DES, we show an encoding procedure in some detail of the most important parts only for the cipher. An advantage of our method is a direct encoding of each bit in the process of a DES execution, with no redundancy from the size of the encoding formula point of view. For describing each bit in this procedure we use one propositional variable. We encode directly all parts of DES.

The whole structure of the encoding formula is similar to FN. We can consider DES as a sequence of permutations, expansions, reductions, XORs, S-box computations and key bits rotations. Each of these operations can be encoded as a conjunction of propositional equivalences or implications.

For example, consider P - the initial permutation function of DES. Let $(p_1, \ldots, p_{64})$ be a sequence of variables representing the plaintext bits. Denote by $(q_1, \ldots, q_{64})$ a sequence of variables representing the block bits after permutation P. Easy to observe that we can encode P as the following formula: $\bigwedge_{i=1}^{64} (p_i \Leftrightarrow q_{P(i)})$.

In a similar way, we can encode all the permutations, expansions, reductions, and rotations of DES. In the case of S-box encoding, observe that S-box is the matrix with four rows and sixteen columns where in each row we have one different permutation of numbers belonging to Z_{16}. These numbers are denoted in binary form as four-tuples of bits. Following the DES algorithm we can consider each S-box as a function of type $S_{box}:\ \{0,1\}^6 \rightarrow \{0,1\}^4$.

For simplicity let us denote a vector $(x_1, \ldots, x_6)$ by $\overline{x}$ and by $S^k_{box}(\overline{x})$ the k-th coordinate of value $S_{box}(\overline{x})$, for $k = 1, 2, 3, 4$.

We can encode each S-box as the following Boolean formula:

$$\Phi_{S_{box}} : \bigwedge_{\overline{x}\in\{0,1\}^6} (\bigwedge_{i=1}^{6} (\neg)^{1-x_i} p_i \ \Rightarrow \ \bigwedge_{j=1}^{4} (\neg)^{1-S^j_{box}(\overline{x})} q_j),$$

where $(p_1, \ldots, p_6)$ is the input vector of S-box and $(q_1, \ldots, q_4)$ the output one. Additionally, by $(\neg)^0 p$ and $(\neg)^1 p$ we mean p and $\neg p$, respectively. Using this we can encode each of the S-boxes used in all considered rounds of DES as 256 simple implication. This number is equal to the size of S-box matrix. Due to the strongly irregular and random character of S-boxes, we are sure that this is the simplest method of Boolean encoding of the S-boxes.

Having these procedures, we can encode any given number of rounds of DES algorithm as a Boolean formula. Our encoding gave formulas shorter than those of Massacci [7]. We got 3 times less variables and twice less clauses. Observe that from the computational point of view, it is important to decrease as far as possible the number of variables and connectives used in the formula. In the next section we briefly describe a method of decreasing the parameters of the formula obtained, preserving its equivalences.

3.3 Cryptanalysis Procedure

The cryptanalysis procedure we propose in this paper is the following:

1. encode a single round of the cipher considered as a Boolean propositional formula;
2. automatic generation of the formula encoding a desired number of iteration rounds (or the whole cipher);
3. convert the formula obtained into its CNF;
4. (randomly) choose a plaintext and the key vector as a $0,1$-valuation of the variables representing them in the formula;
5. insert the chosen valuation into the formula;
6. calculate the corresponding ciphertext using an appropriate key and insert it into the formula;
7. run your favourite SAT-solver with the plaintext and its ciphertext bits inserted, to find a satisfying valuation of the key variables.

4 Optimisation and Parallel Computation

In this section we show some optimisation and parallelisation of various stages the above procedure. The results of these optimisation and parallelisation are shown in the tables presented below. Our implementation is made in C++. For the parallelisation process *OpenMP* library is used. Today, many compilers already have it by default. For our tests we used the *g++* compiler. Our experiments were carried out on an Intel Core 2 Quad CPU Q8200 2.33GHz machine, Linux operating system, and SAT-solver MiniSat.

The first parallelisation in our work occurs in the process of generation of the encoding formula. The main loop of this process can work parallely. Independent operations of each iteration of the formula generation are ideal for dividing them between the working threads. In this case we obtained an acceleration about 15% of the time.

The second improvement is optimising the formula obtained by removing all not necessary equivalences. In the case of a subformula that encode permutations used in the algorithm that are conjunctions of simple equivalences $p_n \Leftrightarrow p_k$, we remove these fragments of the formula and substitute the propositional variables used in encoding in the following way. Consider the equivalence $p_n \Leftrightarrow p_k$, if $n < k$ into the whole formula we put a variable p_n in all places where p_k occurs. After all these removing and substitutions we reenumerate all variables in order to get a new smaller set of variables. After that we have a shorter formula equivalent with the initial one. In this way, we can decrease the number of variables used sometimes by 50% and the number of clauses by 10%. A good example of the quality of our formula optimisation is presented in Table 1. We can see that our method allows a significant decrease of encoding formula with preservation of equivalence. The third improvement is parallel realization of translating formula into a CNF form. The data structure constructed for the generated encoding formula allows us to modify each of its members separately. This is useful when converting a formula to CNF. Elimination of the logical operators: equivalence, implication, XOR, can be divided into multiple threads. Other transformations such as the use of de Morgan's laws, or resolution alternative to a conjunction law, also can be divided in the same way. In the outcome, it decreases speed of realisation of the whole conversion process. In this case we obtained about 10% speedup in the translation time.

Table 1. Results of the formula first optimisation

Rounds	Variables	Variables after 1st optimisation	Clauses	Clauses after 1st optimisation
4	1024	568	10496	9472
8	1976	1016	20866	18944
16	3768	1912	41601	37888

The next step is some optimisation of the CNF formula after inserting the plaintext and the ciphertext values. We insert into the formula the valuation representing a plaintext and its ciphertext, respectively, as a set of one-literal clauses. After that we can optimise the formula with the following scheme. All clauses that include a literal considered are removed from CNF form. In the case of clauses that include a negation of a literal, this negation with a literal is removed. Table 2 shows this type of optimisation in the case of some formula for 4, 8, and 16 rounds and randomly chosen input and its output computed with some key. The last step is a parallel realization of the SAT-solver's work. In our

Table 2. Results of the second optimisation

Rounds	Variables	Variables after 2nd optimisation	Clauses	Clauses after 2nd optimisation
4	568	440	9472	6642
8	1016	888	18944	17405
16	1912	1784	37888	36350

first attempt we used the winner of Special Parallel Track of the SAT-Race 2010 Conference called *Plingeling* (see [1]). Unfortunately it worked about four times worse than MiniSat we used in our earlier investigations without parallelisation. We are sure that the problem in this case lies in the very complicated structure of the formula encoding the cipher considered. Having this we tried to modify the implementation of MiniSat introducing parallelisation in the proper places of the solver's source code. The results for a few rounds were rather average. In four rounds we obtained about 10% speedup. Unfortunately, in the case of more rounds, the results were similar to those with the original version of MiniSat. In what follows below we report another approach to parallelisation. Given a number of CPUs which is some power of 2, searching for all the key bits values, we can create different formulas for each unit inserting all possible different valuations of some chosen set of the key variables. Since we have experimented on a quad-core machine, we chose some two of the key bits and put in extra four different with a valuation of the two bits. The results obtained using this approach are shown in Table 3. In the last row of Table 3, for six rounds of DES with added valuations of 20 key bits, the whole key was solved in 2.2 secs with all our optimisation and parallelisation. In the same case but without the improvements presented above, the whole key was solved in 145 secs. The best result known so far in SAT-based cryptanalysis for this case was 68 secs reported in [4].

Table 3. Solving time speed-up

Rounds	Sequential time (s.)	Parallel time (s.)	Speed-up
3	0,052	0,036	31%
4	34,686	6,542	81%
5	12762	2438	81%
6 + 20 key bits	47,539	2,216	95%

5 Conclusion and Future Directions

In this paper we have proposed some optimisation and parallelisation improvements to direct SAT-based cryptanalysis of symmetric block ciphers. We have reported some experimental results, encouraging as compared with the so far best ones in the SAT-based approach. For sake of comparison, we have carried out the experiments reported here on two very famous ciphers that have attracted extensive research over the last 4 decades. Clearly, a success of our new method depends on finding a cipher which we can break.

Our experiments were carried out on a quad-core device. It seems interesting to experiment on much more advantageous technology of over some many-core threshold; e.g., on a hundred-core processor, as manufactured on a single multiprocessor chip and widely used these days across many application domains including general-purpose, graphics, *et cetera*. An interesting question remains what architecture might fit best to our purpose: the cores may or may not share caches, they may implement message passing or shared memory, and other intercore communication designs. The parallelisation of SAT-solving software is a significant ongoing topic of research.

References

1. Biere, A.: Lingeling, Plingeling, Picosat and Precosat at SAT Race 2010. Technical Report FMV Reports Series 10/1, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2010)
2. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (February 2009)
3. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. J. Cryptology 4(1), 3–72 (1991)
4. Courtois, N.T., Bard, G.V.: Algebraic Cryptanalysis of the Data Encryption Standard. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 152–169. Springer, Heidelberg (2007)
5. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
6. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)
7. Massacci, F.: Using Walk-SAT and Rel-SAT for cryptographic key search. In: Dean, T. (ed.) IJCAI, pp. 290–295. Morgan Kaufmann (1999)
8. Matsui, M.: The First Experimental Cryptanalysis of the Data Encryption Standard. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 1–11. Springer, Heidelberg (1994)
9. Menezes, A., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1996)

Parallel Version of Image Segmentation Algorithm Using Polygonal Markov Fields

Rafał Kluszczyński[1] and Piotr Bała[1,2]

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

Abstract. In this paper we present an application of parallel simulated annealing method to a segmentation algorithm using polygonal Markov fields. After a brief presentation of the algorithm and a general scheme of parallelization methods using simulated annealing technique, there is presented parallel approach to the segmentation algorithm with different synchronization scenarios.

Authors also present results of the parallelization of the segmentation algorithm. There is discussed comparison between simulations with different synchronization scenarios applied to the multiple-trial approach of simulated annealing technique. Some simulations based on the number of threads are presented as well.

Keywords: Image segmentation, Polygonal Markov field, Parallel simulated annealing.

1 Introduction

Segmentation is one of the fundamental processes in image processing and analysis. It partitions the image into relatively homogeneous areas [19] with respect to some characteristic or computed property, such as color or intensity. This process is usually used as a one of the first steps during image analysis. Based on its results further methods of visual data interpretation focused on selected areas can be applied.

A plenty of applications can serve as an example of importance of the segmentation process. Segmentation is used in medical imaging to locate tumors or measure tissue volume. It is used for example to fingerprint images, visual face recognition and in satellite imaging to determine the area of forests for example. In these scenarios different segmentation methods are usually used. In this paper authors focused on the algorithm using polygonal Markov fields and present its parallel version based on multi-trial parallelism of simulated annealing technique.

The goal of this paper is to present efficient parallelization of the segmentation algorithm. Authors also presents achieved execution time speed up based on different synchronization scenarios and a number of threads. The paper is organized as follows. Model of polygonal Markov fields is described in the next section of the paper. The third section presents the segmentation algorithm using the model. There is also described energy term used in simulated annealing (SA) cooling scheme. Next, there are presented parallel implementations of SA method. The following section describes scenarios used by authors in their simulations and a comparison of achieved results for different scenarios and number of threads. The paper is concluded with the summary.

2 Polygonal Markov Fields

Polygonal Markov fields construction was introduced for the first time by Arak in 1982 [2]. In paper [3] there were presented several equivalent techniques of construction. One of them, called *dynamic representation*, occurs to be very well suited for computer simulations. Based on that description there has been developed algorithm generating polygonal configurations, which enabled to implement segmentation algorithm using polygonal Markov fields model [12,13].

In the case of image segmentation we consider polygonal configuration on rectangular domain $D \in R^2$ (in general it can be any convex area). The crucial idea is to interpret the polygonal segments of configuration as the trace left by a particle traveling in two-dimensional time-space. Thus, points of domain D are treated as a set of time-space points (t, y), where t represents time coordinate and y position in a one-dimensional space, see Figure 1.

In this approach, segment is a trajectory of evolving particle and admissible configurations [3] are represented by following particle evolution rules:

- every particle moves freely with constant velocity except the critical events listed below,
- when a particle touches the boundary of domain D, it dies,
- in case of collision of two particles (equal y coordinate at some moment t), both of them die in this point,
- the time evolution of the particle's velocity is given by a pure-jump Markov process (see [13] for details).

The above description is very convenient for simulation and can be easily implemented. Application of sweep method introduced by Bentley and Ottoman [4] can easily lead to an optimal algorithm generating and modifying configurations. Moreover, polygonal Markov fields have many numerical characteristics, which allow evaluation of construction algorithm. Some of them are presented in [7].

3 Segmentation Using Polygonal Markov Fields

The first application of polygonal Markov fields model in image processing was presented in 1989 [6]. Later, Clifford and Nicholls developed a Metropolis-Hastings sampler and applied it to an image reconstruction problem within

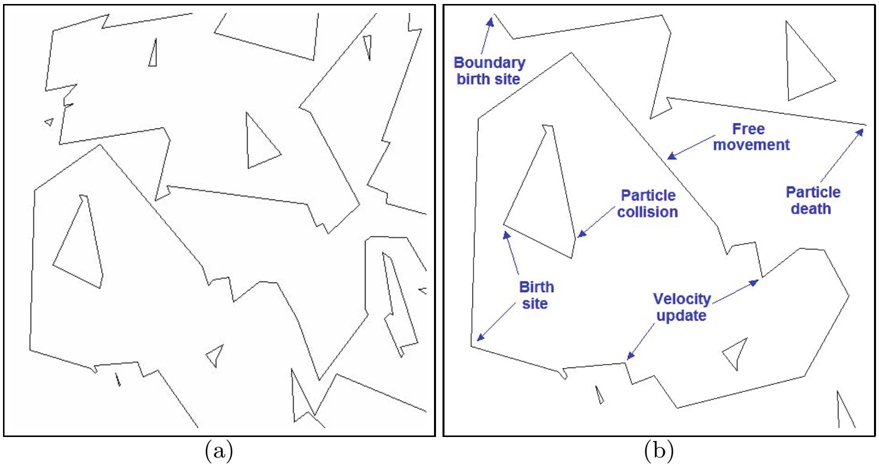


Fig. 1. Example of polygonal configuration (a) and presentation of evolution events of dynamic representation (b). Assuming that configuration's simulation time flows from left to right.

Bayesian framework. Segmentation algorithm using polygonal configurations [13] was developed based on the work of Schreiber who introduced a Gibbsian modification of polygonal random fields [21].

The main concept of the algorithm is a *disagreement loop* [20]. It enables to modify polygonal configuration by adding a new birth site, removing existing one or changing velocity of randomly chosen critical moment of particle's evolution (see [12,13] for more details). Disagreement loop is a symmetric difference between two polygonal random fields: before and after applying one of the modifications. In other words, it is a single loop (a closed polygonal curve), possibly self-intersecting and possibly chopped off by the boundary. This observation allows to describe configuration modifications (a single Monte Carlo move) in dynamic representation language suitable for implementation. An example of disagreement loop is presented in Fig. 2.

Another important piece of the segmentation algorithm is an energy function for Gibbsian modification of polygonal Markov fields. For this purpose we used:

$$H(\hat{\gamma}, I) := H_1(I, \hat{\gamma}) + H_2(\gamma), \tag{1}$$

where I denotes observed image, γ – polygonal configuration and $\hat{\gamma}$ colored version (black and white) of γ interpreted as foreground and background regions. The first term is responsible for a measure how well configuration γ is suited for segmentation of image I. It is the sum of absolute differences between the actual pixel values and those assigned by $\hat{\gamma}$. It is later denoted as PMR (*pixel misclassification rate*).

The second term of energy function (1) is responsible for geometrical characteristics of polygonal configuration γ. There can be added some regularization

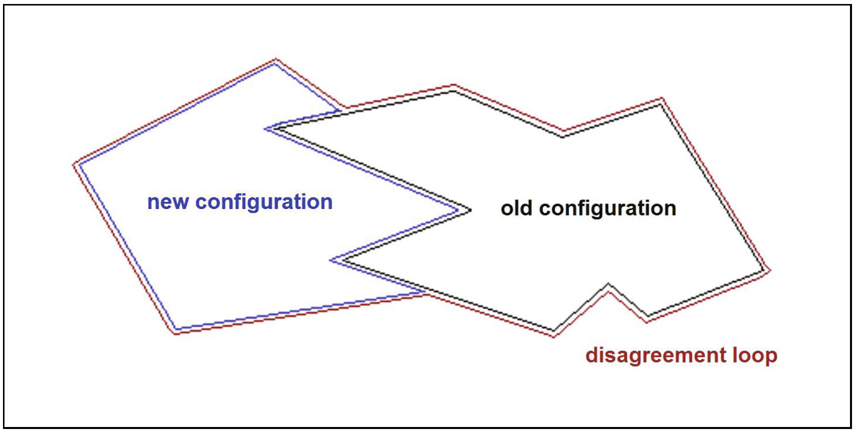


Fig. 2. Example of dissagreement loop (time flows from left to right during the update)

terms like Lennard-Jones potential to penalize short segments. During our simulations we decided to use penalization term

$$H_2(\gamma) = \sum_{s \in \gamma} V(\text{length}(s)), \tag{2}$$

where s means a segment of configuration γ and $V(r) = R - log(r)$ if $r < r_{cut}$ and 0 otherwise. Constant R equals $log(r_{cut})$ and r_{cut} denotes minimal, not penalized segment length.

At this point, problem of image segmentation is equivalent to the task of finding a good approximation of the global minimum. For this purpose simulated annealing (SA) technique is used with energy function (1). By theorems in [21,13], dynamics based on disagreement loop update mechanism of admissible configurations converges in distribution to the Gibbsian modification. Figure 3 presents pseudo-code of the segmentation algorithm. More detailed description can be found in [12,13].

4 Parallel Simulated Annealing Method

Simulated annealing technique has been considered as a good tool for complex nonlinear optimization problems and is widely applied to a variety of problems [8,18,5,15]. One of the drawbacks is its slow convergence, which causes that the method is time-consuming. That is why there were many attempts to develop parallel versions which could be applied to different kinds of problems. There were some achievements close to ideal speedup on small processors arrays [18].

There exist two different approaches to parallelization of SA method [8]:

- single-trial parallelism,
- multiple-trial parallelism.

```
1. Generate an initial configuration $\gamma_0$ and calculate $H_0$;
2. Set $i = 1$;
3. Generate new configuration $\gamma_{new}$ based on $\gamma_{i-1}$ and choose randomly one of the modifications;
4. Calculate new energy $H_{new}$ and $\delta = H_{new} - H_{i-1}$;
5. If $\delta < 0$, then $\gamma i = \gamma_{new}$, else
       $\gamma i = \begin{cases} \gamma_{new} & \text{with probability} \exp(\delta), \\ \gamma_{i-1} & \text{otherwise;} \end{cases}$
6. Calculate $PMR$ value of configuration $\gamma_i$;
7. If stopping criterion is not satisfied (still too high $PMR$), back to step 3.
```

Fig. 3. Pseudo-code of the sequential segmentation algorithm

The first approach involves the use of multiple processes by a single simulated annealing operation. It is clear that implementation and speedup highly depends on the problem and its characteristics. On the other hand, multiple-trial parallelism is a strategy that can be applied to all problems using simulated annealing technique. Increasing availability of multi-core processors ideal for concurrent computations, is a strong motivation.

Simulated annealing technique has been used in molecular modeling [14,16]. This and many other applications usually make use of multiple-trial parallelism in distributed environment to speed up calculations (execution time). Its general nature makes it easy to apply in the case of an algorithm for segmentation using polygonal Markov fields. The method consists of concurrent independent simulations (in threads) under different conditions (temperature). Besides standard Monte Carlo operations, there is also introduced an additional move which exchanges configurations between threads. This approach is well known as the "parallel tempering" method, first introduced by Geyer [9], and later rediscovered through independent work by Hukushima and Nemoto under the name of "replica exchange" [10].

It can be shown that such operation can be applied to any Markov chain Monte Carlo simulations [1]. That is why it fits in our segmentation method very easily. We run independent segmentations, and based on the exchange scenario, we synchronized them to apply exchange between replicas. By running such simulation on multi-core processor we were able to get results in shorter execution time than in a sequential approach.

5 Exchange Scenarios and Results

In this section we present some scenarios applied to the exchange move of segmentation algorithm using multiple-trial parallelism. In order to compare different scenarios we decided to measure the number of all performed operations

(Monte Carlo moves) by all threads, denoted later as an *overall work* of a simulation. In this way we have assumed the same average execution time of a single move during all simulations, which does not have to be true in general. Segmentations were run on a cluster of machines with six-Core AMD Opteron(tm) Processor 2.6 GHz and 32 GB memory using the same input image. We have chosen such measure, because we noticed that execution times depend on the configuration and a load of machines during simulations.

The algorithm was implemented in `C++`. For parallelization there have been used `OpenMP` directives included in `GCC` compiler. Figure 4 presents pseudo-code of parallel version of the segmentation algorithm using polygonal Markov fields. There can be seen, that only synchronization step depends on the scenario of exchanging configurations between threads. For all simulations presented in this section there has been used function (1) with

$$H_1 = (2000 + 0.01 \times i) \times PMR_i,$$

where i denotes iteration number, and H_2 defined by equation (2) with $r_{cut} = 0.00001$ for polygonal field size 3×3.

```
1. Generate all initial configurations in all threads;
2. Set shared variable $sync = False$;
3. For each thread in parallel do:
   (a) Generate new configuration by applying randomly chosen modification and
       accepting it or not according to the scheme presented in sequential algorithm;
   (b) Check the synchronization criterion and if it is satisfied set $sync$ to $True$;
   (c) If $sync$ equals $True$, then first thread:
         – Waits for other threads and blocks their execution;
         – Makes exchange operation between threads according to a specific scenario;
         – Sets $sync$ to $False$ and releases all threads.
```

Fig. 4. Pseudo-code of parallel version of the segmentation algorithm

5.1 Results for Different Synchronization Scenarios

As it was mentioned in the previous section, multiple-trial parallelism introduces new operation of exchanging partial solutions between two replicas. We decided to test following different scenarios of exchanging replicas:

- exchange with a probability,
- weighted replication with fixed step,
- weighted replication with increasing step,
- replication of the best partial solution with increasing step.

In the case of parallel SA technique the first approach is usually presented in literature [1]. Based on that experience, authors decided to investigate three other techniques presented above. All scenarios were run on the same input image with four threads.

In the case of exchange with a probability scenario, exchange operation can be used at any time. During our simulation we have used probability $p = 0.001$. As presented in the Table 1, this scenario has decreased execution time (running in parallel environment) and also decreased overall work compare to standard sequential approach.

Table 1. Comparison of different parallel SA scenarios based on 30 segmentations ($PMR = 2\%$). Average number of operations per thread as well as an average overall work are presented.

Strategy	Thread 1	Thread 2	Thread 3	Thread 4	Overall work
Single thread run	14776	–	–	–	14776
Exchange with probability $p = 0.001$	3236	3159	3190	3220	12805
Weighted replication with fixed step	2618	2652	2611	2609	10490
Weighted replication with increasing step	2495	2521	2517	2533	10066
Best solution replication with increasing step	2214	2235	2267	2235	8950

This improvement was the inspiration for another scenario to apply. The second approach assumed synchronization every fixed number of steps. Then, randomly selected configuration of one thread was replicated among the others. To promote solutions closer to the optimal one, we used weighted probability. To be sure, that replicas with higher energy are less probably to be chosen, we applied following probability formula for choosing thread i to be replicated:

$$p_i = \frac{\exp(E_i - E_m)}{\sum_k \exp(E_i - E_m)}, \tag{3}$$

where E_i is the energy of replica i, while $E_m = \min_k(E_k)$. It can be seen (Table 1) that solution at PMR = 2% was obtained with a better overall work than the first approach.

In the third scenario there has been also used synchronization with probability defined by the formula (3). We noticed that the fixed period of synchronization is impractical in the case of exponential convergence of SA technique. Therefore, we modified previous scheme with linear increase of synchronization point at every time it occurs. In our simulations we used periods of $750 + 250 \times i$ ($i = 0, 1, \ldots$) iterations between consecutive synchronizations. This strategy did not make much improvements, what can be seen in the Table 1.

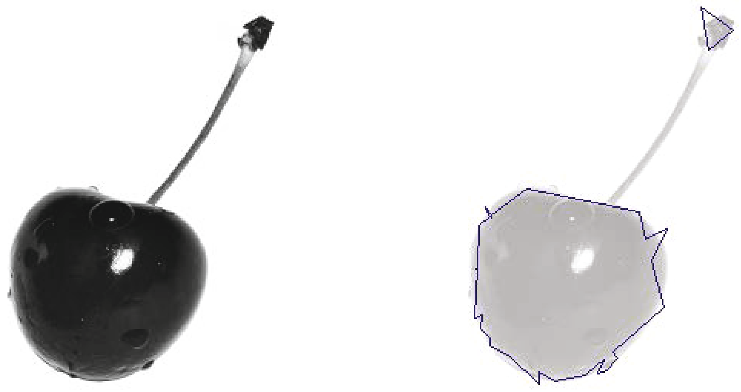


Fig. 5. Cherry image used for simulations (on the left) and its best achived result (on the right) with $PMR = 1.66\%$

The last applied scenario was propagation of the best solution got from all replicas during synchronization. There has been also used increasing synchronization point which is more suitable for SA technique. It has occurred, that this approach has been very effective. It reduced the execution time and overall work of the previous scenario. Average numbers of threads' operations are presented in Table 1.

5.2 Results for Different Number of Threads

In this subsection we present comparison of segmentations using different numbers of threads. For this purpose we used the same image as in previous simulations, which is presented together with its best achieved result in Figure 5. There was used brute-force scenario (propagation of minimum) with increasing synchronization point. The first point was set to 750th iteration and after each synchronization its period was increased by 250. The pixel misclassification rate to achieve was set to 1.8%.

Table 2 presents average segmentations' results together with its average overall work using threads number from 1 to 6. Results show that increasing number of threads decreases the time of execution needed to achieve PMR. Of course, this is not a deterministic method, so one particular simulation can run much longer or end right after it starts. Nevertheless, average overall work made by simulations increases with the number of threads, but not proportional to the threads number. This is acceptable as far as we are using multi-core environment, where number of threads does not exceed the number of cores and we can speedup execution time.

Table 2. Comparison of average work for different number of threads. Table presents number of operations and an interval of achieved PMRs based on a run of 100 simulations with $PMR = 1.8\%$.

Number of threads	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Overall work
1	12119 (1.72% - 1.80%)	-	-	-	-	-	12119
2	7159 (1.74% - 2.98%)	7173 (1.66% - 2.49%)	-	-	-	-	14332
3	5977 (1.77% - 2.54%)	5992 (1.75% - 2.99%)	5977 (1.73% - 2.43%)	-	-	-	17946
4	4814 (1.77% - 2.62%)	4820 (1.75% - 2.67%)	4824 (1.73% - 2.51%)	4808 (1.75% - 2.48%)	-	-	19266
5	4530 (1.75% - 2.61%)	4527 (1.76% - 2.47%)	4523 (1.77% - 2.68%)	4522 (1.73% - 2.62%)	4522 (1.74% - 2.54%)	-	22625
6	4225 (1.77% - 2.37%)	4224 (1.77% - 2.36%)	4209 (1.77% - 2.33%)	4233 (1.79% - 2.63%)	4221 (1.75% - 2.46%)	4216 (1.76% - 2.34%)	25328

6 Conclusions

Nowadays, multi-core processors are easily available. That is why parallel computing is more important then couple years ago. Increasing availability of multi-cores processors encourages us to use algorithms designed for such architecture. This also includes other methods like simulated annealing technique which can determine a good approximation of a global minimum. For example, we have presented the segmentation algorithm using polygonal Markov field model which we have easily parallelized to take advantage of multi-core computers decreasing its execution time. We have also shown that in case of using parallel SA technique, it is important to investigate synchronization scenario which may affect significantly execution time of simulations.

Acknowledgements. This research was supported in part by PL-Grid Infrastructure [17].

References

1. Adib, A.B.: The theory behind tempered Monte Carlo methods (2005), `http://arturadib.googlepages.com/tempering.pdf`
2. Arak, T.: On Markovian random fields with finite number of values. In: 4th USSR-Japan Symposium on Probability Theory and Mathematical Statistics, Abstracts of Communications, Tbilisi (1982)

3. Arak, T., Surgailis, D.: Markov fields with polygonal realisations. Probabability Theory and Related Fields, 80 (1989)
4. Bentley, J.L., Ottmann, T.A.: Algorithms for reporting and counting geometric intersections. IEEE Transactions on Computers C-28(9) (1979)
5. Chu, K.-W., Deng, Y., Reinitz, J.: Parallel Simulated Annealing by Mixing of States. Journal of Computational Physics 148 (1999)
6. Clifford, P., Middleton, R.D.: Reconstruction of polygonal images. Journal of Applied Statistics 16 (1989)
7. Clifford, P., Nicholls, G.K.: A Metropolis sampler for polygonal image reconstruction (1994)
8. Eglese, R.W.: Simulated Annealing: A tool for Operational Research. European Journal of Operational Research 46 (1994)
9. Geyer, C.J.: Markov chain Monte Carlo maximum likelihood. In: Keramidas, E.M. (ed.) Computing Science and Statistics: Proceedings of the 23rd Symposium on the Interface. Interface Foundation, Fairfax Station (1991)
10. Hukushima, K., Nemoto, K.: Exchange Monte Carlo Method and Application to Spin Glass Simulations. J.Phys. Soc. Japan 65 (1996)
11. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by Simulated Annealing. Science, 220 (1983)
12. Kluszczyński, R., van Lieshout, M.-C., Schreiber, T.: An Algorithm for Binary Image Segmentation Using Polygonal Markov Fields. In: Roli, F., Vitulano, S. (eds.) ICIAP 2005. LNCS, vol. 3617, pp. 383–390. Springer, Heidelberg (2005)
13. Kluszczyński, R., van Lieshout, M.N.M., Schreiber, T.: Image segmentation by polygonal Markov fields. Journal Annals of the Institute of Statistical Mathematics 59(3) (2007)
14. Li, H., Tejero, R., Monleon, D., Bassolino-Klimas, D., Abate-Shen, C., Bruccoleri, R.E., Montelione, G.T.: Homology modeling using simulated annealing of restrained molecular dynamics and conformational search calculations with CONGEN: Application in predicting the three-dimensional structure of murine homeodomain Msx-1. Protein Science 6 (1997)
15. Miki, M., Hiroyasu, T., Kasai, M., Ono, K., Jitta, T.: Temperature Parallel Simulated Annealing with Adaptive Neighborhood for Continuous Optimization Problem. Computational Intelligence and Applications (2002)
16. Moglich, A., Weinfurtner, D., Maurer, T., Gronwald, W., Kalbitzer, H.R.: A restraint molecular dynamics and simulated annealing approach for protein homology modeling utilizing mean angles. BMC Bioinformatics 6, 91 (2005)
17. PL-Grid project home page, `http://plgrid.pl`
18. Ram, D.J., Sreenivas, T.H., Subramaniam, K.G.: Parallel Simulated Annealing Algorithms. Journal of Parallel and Distributed Computing 37 (1996)
19. Rosenfeld, A., Kak, A.C.: Digital picture processing, 2nd edn., vol. 2. Academic Press, Orlando (1982)
20. Schreiber, T.: Mixing properties of polygonal Markov fields in the plane (2003), `http://www.mat.uni.torun.pl/preprints`
21. Schreiber, T.: Random dynamics and thermodynamic limits for polygonal Markov fields in the plane. Advances in Applied Probability 37(4) (2004)

Parallel Community Detection for Massive Graphs

E. Jason Riedy[1], Henning Meyerhenke[1,2], David Ediger[1], and David A. Bader[1]

[1] Georgia Institute of Technology, 266 Ferst Drive, Atlanta, Georgia, 30332, USA
[2] Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany

Abstract. Tackling the current volume of graph-structured data requires parallel tools. We extend our work on analyzing such massive graph data with the first massively parallel algorithm for community detection that scales to current data sizes, scaling to graphs of over 122 million vertices and nearly 2 billion edges in under 7300 seconds on a massively multithreaded Cray XMT. Our algorithm achieves moderate parallel scalability without sacrificing sequential operational complexity. Community detection partitions a graph into subgraphs more densely connected within the subgraph than to the rest of the graph. We take an agglomerative approach similar to Clauset, Newman, and Moore's sequential algorithm, merging pairs of connected intermediate subgraphs to optimize different graph properties. Working in parallel opens new approaches to high performance. On smaller data sets, we find the output's modularity compares well with the standard sequential algorithms.

Keywords: Community detection, parallel algorithm, graph analysis.

1 Communities in Graphs

Graph-structured data inundates daily electronic life. Its volume outstrips the capabilities of nearly all analysis tools. The Facebook friendship network has over 800 million users each with an average of 130 connections [9]. Twitter boasts over 140 million new messages each day [24]. The NYSE processes over 300 million trades each month [20]. Applications of analysis range from database optimization to marketing to regulatory monitoring. Global graph analysis kernels at this scale tax current hardware and software due to the size *and* structure of typical inputs.

One such useful analysis kernel finds smaller communities, subgraphs that locally optimize some connectivity criterion, within these massive graphs. We extend the boundary of current complex graph analysis by presenting the first algorithm for detecting communities that scales to graphs of practical size, over 120 million vertices and nearly two billion edges in less than 7300 seconds on a shared-memory parallel architecture with 1 TiB of memory.

Community detection is a graph clustering problem. There is no single, universally accepted definition of a community within a social network. One popular definition is that a community is a collection of vertices more strongly connected than would occur from random chance, leading to methods based on modularity [17]. Another definition [22] requires vertices to be more connected to others within the community than those outside, either individually or in aggregate. This aggregate measure leads to minimizing the communities' conductance. We consider disjoint partitioning of a graph into connected communities guided by a local optimization criterion. Beyond obvious visualization applications, a disjoint partitioning applies usefully to classifying related genes by primary use [26] and also to simplifying large organizational structures [14] and metabolic pathways [23]. We report results for maximizing modularity, although our implementation also supports minimizing conductance.

Contributions. We present the first published parallel agglomerative community detection algorithm. Our algorithm scales to practical graph sizes on available multithreaded hardware but with the same sequential operation complexity as current state-of-the-art algorithms. Our approach is both natively parallel and simpler than most current sequential community detection algorithms. Also, our algorithm is agnostic towards the specific criterion; any criterion expressible as individual edge scores can be maximized (or minimized) locally with respect to edge contractions. Our implementation supports four different criteria, and here we report on two modularity-maximizing criteria. Validation experiments show that our implementation yields solutions comparable in quality to the state-of-the-art sequential agglomerative algorithm.

Capability and Performance. On a 128 processor Cray XMT with 1 TiB of memory, our algorithm extracts communities from a graph of 122 million vertices and 1.99 billion edges into communities by maximizing modularity in under 7300 seconds. Currently, our method for *generating* artificial test data rather than our community detection algorithm limits input sizes. Our edge-list implementation scales in execution time up to 128 processors on sufficiently large graphs.

2 Agglomerative Community Detection

Agglomerative clustering algorithms start with every vertex in a singleton community. Edges are scored through some metric, and a local optimization heuristic chooses the next edge(s) to contract. To increase available parallelism, we choose multiple contraction edges simultaneously as opposed to Clauset, Newman, and Moore [8]'s sequential algorithm and the method proposed by Blondel *et al.* [5]. Chosen edges form a maximal cardinality matching that approximates the maximum weight, maximal cardinality matching. We consider maximizing metrics (without loss of generality) and also target a local maximum rather than a global, possibly non-approximable, maximum. There are a wide variety of metrics in use for optimizing and evaluating communities [11]. We focus on the established measure modularity defined in Section 2.2.

2.1 Defining the Algorithm

We take an input graph $G_0 = (V_0, E_0)$ and re-interpret G_0 as a *community graph* $G = (V, E)$. Each vertex in a community graph is a disjoint subset of the input graph's vertices, and we begin with $V = \{\{i\} | i \in V_0\}$. Each edge i, j in E corresponds to an edge between communities, $E = \{\{i, j\} | \exists i_0 \in i \, \exists j_0 \in j$ such that $\{i_0, j_0\} \in E_0\}$. Assign edge weights $w(\{i, j\})$ to count the number of edges between communities i and j. To make expressions simpler, let self-edge weights $w(\{i, i\})$ count the volume (sum of degrees) where both end vertices lie in community i. Our algorithm contracts edges $\{i, j\}$ in G to form a new community graph G' with vertices representing the union of disjoint communities i and j.

We define our agglomerative algorithm with matrix operations. Consider the typical mapping of an undirected graph $G = (V, E)$ with edge weights $w(\{i, j\})$ to a sparse, symmetric adjacency matrix A. The matrix A has dimension $|V| \times |V|$, where $|V|$ is the number of vertices in G. The matrix has two non-zero entries for each edge $\{i, j\} \in E$ with $i \neq j$, $A(i, j) = A(j, i) = w(\{i, j\})$. Self edges appear along the diagonal, $A(i, i) = w(\{i, i\})$.

All of Section 2.2's metrics use additive edge weights. Represent a set of edge contractions with a matching matrix M of dimension $|V'| \times |V|$, where $|V'|$ is the number of vertices of the *contracted* graph, and entries $M(i, j) = 1$ when vertex $j \in V$ contracts into vertex $i \in V'$. A maximal matching should produce a matrix M with as many degree-two columns as possible, and ideally $|V'| \approx |V|/2$. With M and additive weights, we represent the graph contraction from A to A' with $A' = M \cdot A \cdot M^T$.

Our agglomerative algorithm is agnostic of local maximization criteria. Given a metric of interest, our algorithm works abstractly as follows:

1. Construct the symmetric sparse matrix A representing the undirected multi-graph G and its edge multiplicities.
2. Set the initial trivial community mapping $C := I$, the $|V|^2$ identity matrix.
3. While the number of communities and the largest community size have not reached a pre-set limit, repeat:
   (a) Compute a sparse matrix of edge scores according to the optimization metric given A and v.
   (b) Compute a matching matrix M on the score matrix to maximize the metric of interest.
   (c) If the matching is empty, quit.
   (d) Contract $A := M \cdot A \cdot M^T$.
   (e) Update the community mapping $C := M \cdot C$.

When a matching is empty, no edges increase the metric of interest. Section 3's parallel implementation works with an edge list data structure rather than typical compressed formats and implements the sparse operations directly.

Typical sequential agglomerative algorithms like Clauset *et al.* (CNM) [8]'s method contract a single edge in each iteration. Our algorithm generalizes Blondel *et al.*'s sequential approach [5] by identifying many contraction edges simultaneously. If the matching M contracts most edges of the graph at once, most edge scores will need recomputation. This differs from CNM [8]'s incremental edge scoring but does not affect the algorithm's asymptotic complexity.

Assuming all edges are scored in a total of $O(|E|)$ operations and some heavy weight maximal matching is computed in $O(|E|)$ [21] where E is the edge set of the current community graph, each iteration of our algorithm's inner loop requires $O(|E|)$ operations. As with other algorithms, the total operation count depends on the community growth rates. If our algorithm halts after K contraction phases, our algorithm runs in $O(|E| \cdot K)$ operations. If the community graph is halved with each iteration, our algorithm requires $O(|E| \cdot \log |V|)$ operations. If the graph is a star, only two vertices are contracted per step and our algorithm requires $O(|E| \cdot |V|)$ operations. This matches experience with the CNM algorithm [25].

2.2 Local Optimization Metrics

Here we score edges for contraction by modularity, an estimate of a community's deviation from random chance [17,3]. We maximize modularity by choosing the largest independent changes from the current graph to the new graph by one of two heuristics. Minimization measures like conductance involve maximizing changes' negations.

Modularity. Newman [16]'s modularity metric compares the connectivity within a collection of vertices to the expected connectivity of a random graph with the same degree distribution. Let m be the number of edges in an undirected graph $G = G(V, E)$ with vertex set V and edge set E. Let $S \subset V$ induce a graph $G_S = G(S, E_S)$ with $E_S \subset E$ containing only edges where both endpoints are in S. Let m_S be the number of edges $|E_S|$, and let $\overline{m_S}$ be an expected number of edges in S given some statistical background model. Define the modularity of the community induced by S as $Q_S = \frac{1}{m}(m_S - \overline{m_S})$. Modularity represents the deviation of connectivity in the community induced by S from an expected background model. Given a partition $V = S_1 \cup S_2 \cup \cdots \cup S_k$, the modularity of that partitioning is $Q = \sum_{i=1}^{k} Q_{S_i}$.

Newman [16] considers the specific background model of a random graph with the same degree distribution as G where edges are independently and identically distributed. If x_S is the total number of edges in G where either endpoint is in S, then we have $Q_S = (m_S - x_S^2/4m)/m$ as in [3]. A subset S is considered a module when there are more internal edges than expected, $Q_S > 0$. The m_S term encourages forming large modules, while the x_S term penalizes modules with excess external edges. Maximizing Q_S finds communities with more internal connections than external ones. Expressed in matrix terms, optimizing modularity is a quadratic integer program and an NP-hard optimization problem [6]. We compute only a local maximum, which depends on the operation order.

Section 3's implementation scores each edge e by the change in modularity contracting e would produce, analogous to the sequential CNM algorithm.

Merging the vertex U into a disjoint set of vertices $W \in C$, requires that the change $\Delta Q(W, U) = Q_{W \cup U} - (Q_W + Q_U) > 0$. Expanding the expression for modularity,

$$\begin{aligned} m \cdot \Delta Q(W, U) &= m\,(Q_{W \cup U} - (Q_W + Q_U)) \\ &= (m_{W \cup U} - (m_W + m_U) - (\overline{m_{W \cup U}} - (\overline{m_W} + \overline{m_U})) \\ &= m_{W \leftrightarrow U} - \overline{(m_{W \cup U} - (m_W + m_U))}, \end{aligned}$$

where $m_{W \leftrightarrow U}$ is the number of edges between vertices in sets W and U. Assuming the edges are independent and identically distributed across vertices respecting their degrees [8],

$$\begin{aligned} \overline{(m_{W \cup U} - (m_W + m_U))} &= m \cdot \frac{x_W}{2m} \cdot \frac{x_U}{2m}, \text{ and} \\ \Delta Q(W, U) &= \frac{m_{W \leftrightarrow U}}{m} - \frac{x_W}{2m} \cdot \frac{x_U}{2m}. \end{aligned} \tag{1}$$

We track $m_{W \leftrightarrow U}$ and x_W in the contracted graph's edge and vertex weights, respectively. The quantity x_W equals the volume of W. In Section 2's matrix notation, ΔQ is the rank-one update $A/m - (v/2m) \cdot (v/2m)^T$ restricted to non-zero, off-diagonal entries of A. The data necessary for computing the score of edge $\{i, j\}$ are $A(i, j)$, $v(i)$, and $v(j)$, similar in spirit to a rank-one sparse matrix-vector update.

Modularity has known limitations. Fortunato and Barthélemy [10] demonstrate that global modularity optimization cannot distinguish between a single community and a group of smaller communities. Berry *et al.* [4] provide a weighting mechanism that overcomes this resolution limit. Instead of this weighting, we compare CNM with the modularity-normalization of McCloskey and Bader [3].

McCloskey and Bader's algorithm (MB) only merges vertices into the community when the change is deemed statistically significant against a simple statistical model assuming independence between edges. The sequential MB algorithm computes the mean $\overline{\Delta Q(W, :)}$ and standard deviation $\sigma(\Delta Q(W, :))$ of all changes adjacent to community W. Rather than requiring only $\Delta Q(W, U) > 0$, MB requires a tunable level of statistical significance with $\Delta Q(W, U) > \overline{\Delta Q(W, :)} + k \cdot \sigma(\Delta Q(W, :))$. Section 4 sets $k = -1.5$. Sequentially, MB considers only edges adjacent to the vertex under consideration and tracks a history for wider perspective. Because we evaluate merges adjacent to all communities at once by matching, we instead filter against the threshold computed across all current potential merges.

3 Mapping Our Algorithm to the Cray XMT

Our algorithm matches the sequential CNM algorithm's operation complexity while avoiding potential bottlenecks in priority queues. Here we outline the mapping from our algorithm to a massively multithreaded shared-memory platform, the Cray XMT. The Cray XMT provides a flat, shared-memory execution environment; Section 5 discusses other possibilities. The parallel mapping is straightforward for this environment and still scales to massive graphs.

The Cray XMT is a supercomputing platform designed to accelerate massive graph analysis codes. The architecture tolerates high memory latencies using massive multithreading. There is no cache in the processors; all latency is handled by threading. Each Threadstorm processor within a Cray XMT contains user-available 100 hardware streams each maintaining a thread context. Context switches between threads occur every cycle, selecting a new thread from the pool of streams ready to execute.

A large, globally shared memory enables the analysis of graphs using a simple shared-memory programming model. Physical address hashing breaks up locality and ensures every node contributes to the aggregate memory bandwidth. Synchronization occurs at the level of 64-bit words through full/empty bits and primitives like an atomic fetch-and-add. The cost of synchronization is amortized over the cost of memory access. Combined, these features assist developing scalable parallel implementations for massive graph analysis.

Within our implementation, the edge scoring heuristics (CNM and MB) parallelize evenly across the edges. Evaluating the scores for all $|E|$ edges requires access to $O(|E|)$ scattered memory locations. Our implementation stores the graph as a vector of self-edge weights and an array of edges $\{i, j\}$ with $i > j$, equivalent to an unpacked lower-triangular sparse matrix representation. Given a matching M, we implement the sparse projection $M \cdot A \cdot M^T$ in-place. Vertices are relabeled and duplicate edges eliminated using $|V| + |E|$ workspace. The implementation forms linked lists of potential duplicates and walks that list; we will see that this list-walking limits our concurrency and ultimate scalability.

To compute the matching M, we begin with a greedy, non-maximal algorithm. We iterate in parallel across the edge array. Each edge $\{i, j\}$ checks its end points i and j. If the current edge is the best possible match seen so far for *both* i and j, the edge registers itself with both endpoints. The Cray XMT's full/empty word synchronization ensures that edge registration occurs without race conditions. Because there is no ordering enforced between edges, this provides neither a maximal nor an approximately maximum weight matching but uses only $2|V|$ working space and $O|E|$ operations. To compute a maximal matching, we run the non-maximal passes until they converge. On convergence, we have a maximal matching where every edge dominates its neighbors, ensuring a 1/2 approximation to the maximum weight [13,15]. We have not analyzed the convergence rate, but our test cases converge in fewer than ten iterations.

Our implementation currently does not track the dendogram, or history of vertex contractions. The dendogram is a tree and can be represented by a $|V|$-long vector of parent pointers updated in $O(|V|)$ time per contraction step with no additional memory use beyond the tree itself.

4 Evaluating Parallel Community Detection

4.1 Parallel Performance

We evaluate parallel scalability using artificial R-MAT [7,1] input graphs derived by sampling from a perturbed Kronecker product. R-MAT graphs are scale-free and reflect many properties of real social networks. We generate each R-MAT graph with parameters $a = 0.55$, $b = c = 0.1$, and $d = 0.25$ and extract the largest component. An R-MAT generator takes a scale s and edge factor f as input and generates a sequence of $2^s \cdot f$ edges over 2^s vertices, including self-loops and repeated edges. We accumulate multiple edges within edge weights.

Our implementation scales to massive graphs, but evaluating strong scalability against a single Cray XMT processor requires using a smaller data set. We generate R-MAT graphs of scale 18 and 19 and with edge factors 8, 16, and 32. Table 1 shows the size of the largest component in each case. We use the largest component to investigate performance of the core algorithm and not heuristics for filtering the many singleton vertices and tiny components not connected to the largest component. The Cray XMT used for these experiments is located at Pacific Northwest National Lab and contains 128 Threadstorm processors running at 500 MHz. These 128 processors support over 12 000 hardware thread contexts. The globally addressable shared memory totals 1 TiB.

Table 1. We evaluate performance against multiple graphs generated by R-MAT with the given scale and edge factor. The graphs are lumped into rough categories by the number of R-MAT generated edges.

Scale	Fact.	$\lvert V\rvert$	$\lvert E\rvert$	Avg. degree	Edge group
18	8	236 605	2 009 752	8.5	2M
	16	252 427	3 936 239	15.6	4M
	32	259 372	7 605 572	29.3	8M
19	8	467 993	3 480 977	7.4	4M
	16	502 152	7 369 885	14.7	8M
	32	517 452	14 853 837	28.7	16M

Figure 1 shows the speed-up against a single Cray XMT processor from three runs on each of Table 1's graphs. The speed-up plot shows some performance variation from the parallel, non-deterministic matching procedure. Unlike sequential experience, performance for the CNM and MB scoring methods is roughly similar. Figure 2 shows that performance plateaus when the matching phase takes as long as contraction. We are investigating why the phases' fractions of time change with more processors. To test scalability to large data sets, applying our algorithm to a 122 million vertex and 1.99 billion edge graph with scale 27 and edge factor 16 requires 7258 (CNM) and 7286 (MB) seconds, respectively.

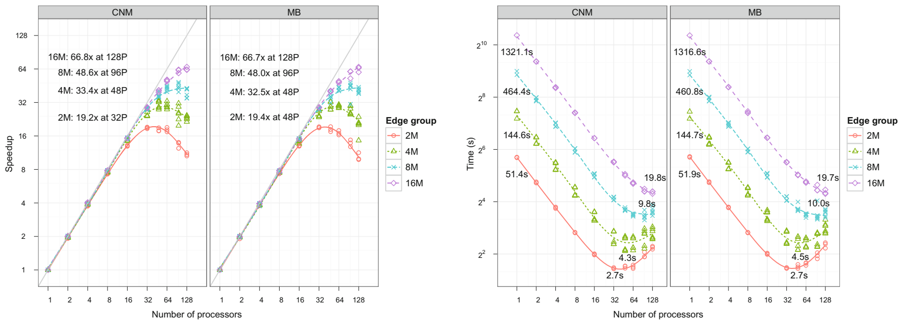


Fig. 1. Execution time on the largest of Table 1's graphs scales up to 128 processors. The left plot shows the best speed-up achieved for each edge group. The right plot shows both the best overall execution time and the best single-processor execution time.

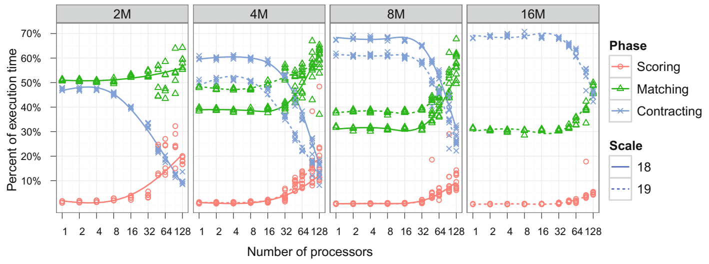


Fig. 2. Execution time fractions show that performance flattens where the matching phase takes as much execution time as the graph contraction

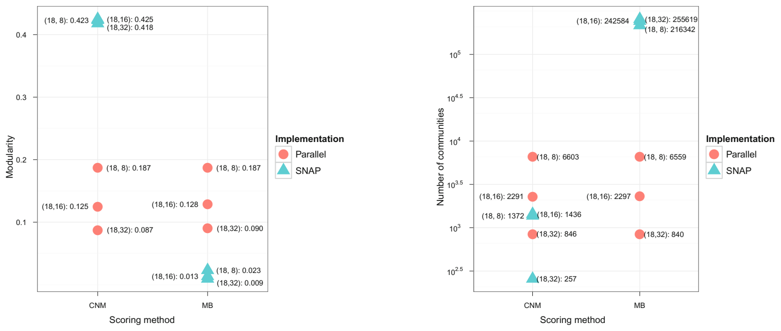


Fig. 3. Comparing modularity values and the number of communities between our parallel agglomerative community detection and a separate, sequential implementation in SNAP shows that ours finds an interesting trade-off between community sizes and modularity. Graph points are labeled by (scale, edge factor) and show either the modularity (left) or number of communities (right).

4.2 Community Quality

Computing communities quickly is only good if the communities themselves are useful. We compare the modularity results from Table 1's scale 18 graphs between our parallel implementation and the state-of-the-art implementation in SNAP[2]. Because our parallel matching algorithm is non-deterministic, we use three runs for each P value. All evaluations are run sequentially through SNAP using the output community maps.

Figure 3 shows the modularity values from our parallel community detection implementation against those returned by SNAP. Forcing a more balanced merge though a maximal matching produces communities not as modular as sequential CNM optimization, but more modular than sequential MB. The number of communities also finds a compromise between the different sequential methods.

5 Related Work

Graph partitioning, graph clustering, and community detection are tightly related topics. A recent survey [11] covers many aspects of community detection with an emphasis on modularity maximization. Nearly all existing work of which we know is sequential and targets specific contraction edge scoring mechanisms.

Zhang *et al.* [27] recently proposed a parallel algorithm that identifies communities based on a custom metric rather than modularity. Gehweiler and Meyerhenke [12] proposed a distributed diffusive heuristic for implicit modularity-based graph clustering. Classic work on parallel modular decompositions [19] finds a different kind of module, one where any two vertices in a module have identical neighbors and somewhat are indistinguishable. This could provide a scalable preprocessing step that collapses vertices that will end up in the same community, although removing the degree-1 fringe may have the same effect.

Work on sequential multilevel agglomerative algorithms like [18] focuses on edge scoring and local refinement. Our algorithm is agnostic towards edge scoring methods and can benefit from any problem-specific methods. The Cray XMT's word-level synchronization may help parallelize refinement methods, but we leave that to future work. Outside of the edge scoring, our algorithm relies on well-known primitives that exist for many execution models. The matching matrix M is equivalent to an algebraic multigrid restriction operator; implementations for applying restriction operators are widely available.

6 Observations

Our algorithm and implementation, the first parallel algorithm for agglomerative community detection, scales to 128 processors on a Cray XMT and can process massive graphs in a reasonable length of time. Finding communities in graph with 122 million vertices and nearly two billion edges requires slightly more than two hours. Our implementation can optimize with respect to different local optimization criteria, and its modularity results are comparable to a state-of-the-art sequential implementation. As a twist to established sequential algorithms for agglomerative community detection, our parallel algorithm takes a novel and naturally parallel approach to agglomeration with maximum weighted matchings. That difference appears to reduce differences between the CNM and MB edge scoring methods. The algorithm is simpler than existing sequential algorithms and opens new directions for improvement. Separating scoring, choosing, and merging edges may lead to improved metrics and solutions.

Acknowledgments. This work was supported in part by the Pacific Northwest National Laboratory (PNNL) CASS-MT Center and NSF Grants CNS-0708307 and IIP-0934114. We also thank PNNL and Cray, Inc. for providing access to Cray XMT hardware.

References

1. Bader, D., Gilbert, J., Kepner, J., Koester, D., Loh, E., Madduri, K., Mann, W., Meuse, T.: HPCS SSCA#2 Graph Analysis Benchmark Specifications v1.1 (July 2005)
2. Bader, D., Madduri, K.: SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In: Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS 2008), Miami, FL (April 2008)
3. Bader, D., McCloskey, J.: Modularity and graph algorithms (September 2009), presented at UMBC
4. Berry, J., Hendrickson, B., LaViolette, R., Phillips, C.: Tolerating the community detection resolution limit with edge weighting. Phys. Rev. E 83, 056119 (2011)
5. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment 2008(10), P10008 (2008)
6. Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., Wagner, D.: On modularity clustering. IEEE Trans. Knowledge and Data Engineering 20(2), 172–188 (2008)
7. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: Proc. 4th SIAM Intl. Conf. on Data Mining (SDM). SIAM, Orlando (2004)
8. Clauset, A., Newman, M., Moore, C.: Finding community structure in very large networks. Physical Review E 70(6), 66111 (2004)
9. Facebook, Inc.: User statistics (October 2011), `http://www.facebook.com/press/info.php?statistics`
10. Fortunato, S., Barthélemy, M.: Resolution limit in community detection. Proc. of the National Academy of Sciences 104(1), 36–41 (2007)
11. Fortunato, S.: Community detection in graphs. Physics Reports 486(3-5), 75–174 (2010)
12. Gehweiler, J., Meyerhenke, H.: A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In: Proc. 7th High-Performance Grid Computing Workshop (HGCW 2010) in Conjunction with 24th Intl. Parallel and Distributed Processing Symposium (IPDPS 2010). IEEE Computer Society (2010)
13. Hoepman, J.H.: Simple distributed weighted matchings. CoRR cs.DC/0410047 (2004)

14. Lozano, S., Duch, J., Arenas, A.: Analysis of large social datasets by community detection. The European Physical Journal - Special Topics 143, 257–259 (2007)
15. Manne, F., Bisseling, R.: A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 708–717. Springer, Heidelberg (2008)
16. Newman, M.: Modularity and community structure in networks. Proc. of the National Academy of Sciences 103(23), 8577–8582 (2006)
17. Newman, M., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E 69(2), 026113 (2004)
18. Noack, A., Rotta, R.: Multi-level Algorithms for Modularity Clustering. In: Vahrenhold, J. (ed.) SEA 2009. LNCS, vol. 5526, pp. 257–268. Springer, Heidelberg (2009)
19. Novick, M.B.: Fast parallel algorithms for the modular decomposition. Tech. rep., Cornell University, Ithaca, NY, USA (1989)
20. NYSE Euronext: Consolidated volume in NYSE listed issues, 2010 - current (March 2011), http://www.nyxdata.com/nysedata/asp/factbook/viewer_edition.asp?mode=table&key=3139&category=3
21. Preis, R.: Linear Time $\frac{1}{2}$-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In: Meinel, C., Tison, S. (eds.) STACS 1999. LNCS, vol. 1563, pp. 259–269. Springer, Heidelberg (1999)
22. Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., Parisi, D.: Defining and identifying communities in networks. Proc. of the National Academy of Sciences 101(9), 2658 (2004)
23. Ravasz, E., Somera, A.L., Mongru, D.A., Oltvai, Z.N., Barabási, A.L.: Hierarchical organization of modularity in metabolic networks. Science 297(5586), 1551–1555 (2002)
24. Twitter, Inc.: Happy birthday Twitter! (March 2011), http://blog.twitter.com/2011/03/happy-birthday-twitter.html
25. Wakita, K., Tsurumi, T.: Finding community structure in mega-scale social networks. CoRR abs/cs/0702048 (2007)
26. Wilkinson, D.M., Huberman, B.A.: A method for finding communities of related genes. Proceedings of the National Academy of Sciences of the United States of America 101(suppl. 1), 5241–5248 (2004)
27. Zhang, Y., Wang, J., Wang, Y., Zhou, L.: Parallel community detection on large networks with propinquity dynamics. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2009, pp. 997–1006. ACM, New York (2009)

Is Your Permutation Algorithm Unbiased for $n \neq 2^m$?

Michael Waechter[1], Kay Hamacher[2], Franziska Hoffgaard[2], Sven Widmer[1], and Michael Goesele[1]

[1] GRIS, TU Darmstadt, Germany
[2] Bioinformatics and Theoretical Biology, TU Darmstadt, Germany

Abstract. Many papers on parallel random permutation algorithms assume the input size n to be a power of two and imply that these algorithms can be easily generalized to arbitrary n. We show that this simplifying assumption is not necessarily correct since it may result in a bias. Many of these algorithms are, however, consistent, i.e., iterating them ultimately converges against an unbiased permutation. We prove this convergence along with proving exponential convergence speed. Furthermore, we present an analysis of iterating applied to a butterfly permutation network, which works in-place and is well-suited for implementation on many-core systems such as GPUs. We also show a method that improves the convergence speed even further and yields a practical implementation of the permutation network on current GPUs.

Keywords: parallel random permutation, butterfly network, bias, consistency, GPU.

1 Introduction

Parallel generation of random permutations is an important building block in parallel algorithms. It can, e.g., be used to perturb the input of a subsequent algorithm making worst-case behavior unlikely [3]. It is also useful in statistical applications where a sufficient number of sample permutations is generated to draw conclusions about every possible input order. This is, e.g., the most important step in bootstrapping often applied in statistical science and modeling [16,17], in particular in bioinformatical phylogenetic reconstruction [14,7].

Using the divide and conquer design paradigm it is convenient to assume that the input array size n is a power of two. This assumption is frequently used (e.g., in [15,5]) to simplify the notation of the algorithm or its proof of unbiasedness, without pointing out an unbiased method for generalization to arbitrary n. In this paper we argue that this simplification may be too strong and inadmissible.

We demonstrate a butterfly style [10, Sec. 3.2] permutation network well-suited for parallelization on many-core machines with lots of processing elements (i.e., a number close to the problem size). If this algorithm is generalized to arrays, whose size is not a power of two, it does not generate all possible permutations equally likely. As this algorithm and its generalization to arbitrary n is not pathological but seems rather natural, this needs to be resolved.

Our main contribution is to demonstrate and resolve this issue by showing that iterative application of any permutation algorithm, whose corresponding permutation matrix is positive, converges against an unbiased permutation, i.e., the algorithm is consistent. Furthermore we show that with an increasing number of iterations the bias diminishes exponentially. We present a method for improving the convergence behavior of the butterfly network even further and demonstrate a GPU implementation that is competitive to or even faster than a highly optimized state-of-the-art GPU algorithm.

2 Related Work

Most random permutation algorithms belong to one of five categories listed below, the first four of which are described and analyzed by Cong and Bader [3].

Rand_Sort assigns a random key to every value and sorts the array in parallel using these keys. Hagerup [6] gives a sorting based algorithm that runs in $\mathcal{O}(\log n)$ with n processing elements and $\mathcal{O}(n)$ space. The approach's general drawback is, that due to the sorting keys the array is effectively doubled in size.

Rand_Dart randomly maps elements into an array of size kn with $k > 1$ (e.g., $k = 2$ [3]) and compacts the resulting array. There are two obvious drawbacks: First, space consumption is kn, and second, memory conflicts need to be resolved. This can be done by using memory locks and either re-throwing "darts" that had a collision into the same "dart board" until they find an empty cell or by throwing them onto a new "board" of smaller size.

Rand_Shuffle is a parallel version of Knuth's sequential algorithm [8, Sec. 3.4.2]. Memory conflicts are resolved by sequentializing conflicting accesses. Anderson [1] analyzes this algorithm for a small number of processing elements. For an increasing number of processing elements the likelihood of memory conflicts increases drastically, especially if the number of processing elements is of the same order as the problem size. Hagerup [6] gives a variant of this algorithm for n processing elements which runs in $\mathcal{O}(\log n)$, but requires $\mathcal{O}(n^2)$ space.

Rand_Dist assigns each of the p processing elements n/p elements, each processing element sends all of its elements to random processing elements, sequentially permutes its received elements and all subsets are concatenated. For small numbers of processing elements and with fast random generators this can outperform other algorithms [3], but for large p the algorithm's work is mostly about redistributing the elements among the processing elements and load-balancing the work must be traded off against implementing the algorithm in-place.

Permutation Networks: Knuth [9, Sec. 5.3.4] points out that sorting networks can be turned into permutation networks by replacing comparers with random exchangers. Waksman [15] gives a network of size $\mathcal{O}(n \log n)$ and time $\mathcal{O}(\log n)$.

Most of the above approaches suffer from several shortcomings. If they are implemented on a system whose number of processing elements is of the same order as the problem size (e.g., many-core systems like GPUs), they either do

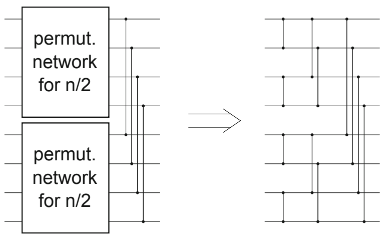


Fig. 1. Recursive construction scheme of the butterfly permutation network

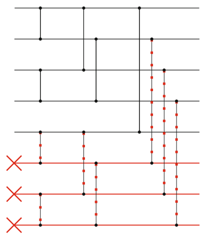


Fig. 2. Network for $n = 5$. Omitted permutations are marked dotted red

not work in-place or require too much synchronization or contention resolving like memory locks. Only permutation networks seem suitable in this setting. In a permutation network each of the random exchanges, that happen in parallel in one layer of the network, can be done by one of the processing elements. This form of parallelization is extremely fine-grained, which makes it very suitable for GPU implementation. Also, the memory access pattern is determined only by the network structure and not by the result of previous computations. Therefore, it is guaranteed that no two processing elements try to access the same memory address at the same time and no contention resolving mechanism is needed.

3 Critical Analysis of the Butterfly Permutation Network

Our algorithm is based on the butterfly network [10, Sec. 3.2]. Given an input array with size $n = 2^m$ it recursively divides the input into two subarrays until a single element is left, permutes the subarrays, and combines them by randomly exchanging element i with element $i + \frac{n}{2}$ for all $i \in \{1, ..., \frac{n}{2}\}$ (see Fig. 1). This network has depth $\log_2 n$ and size $\frac{n}{2} \log_2 n$. It can be executed in parallel using $\frac{n}{2}$ processing elements, requires no memory locks and only $\log_2 n$ thread synchronizations. Compared to Knuth's sequential shuffling [8] the speed-up is in $\mathcal{O}(p/\log n)$ and the efficiency is in $\mathcal{O}(1/\log n)$. An important property of this algorithm is, that it works in-place (in contrast to many other algorithms, see Sec. 2). This simplifies the implementation on modern many-core systems such as GPUs, which typically have very limited shared memory with fast access.

In the following we will first prove that, if n is a power of two, the butterfly network permutes unbiased, i.e., the probability for an element from some origin to be placed at some destination is equal for all origin/destination combinations. Afterwards we demonstrate that this may not be the case for arbitrary n.

3.1 Unbiasedness for $n = 2^m$

A random permutation algorithm works on an array of size n. p_{ij} is the probability that, after some number of steps, element j of the original array can be found at position i. The matrix $\mathbf{M}_n = [p_{ij}]_{1 \le i,j \le n}$ contains all such probabilities. An algorithm is unbiased if $p_{ij} = \frac{1}{n}$ $\forall i, j$ after the algorithm's termination. We show that our algorithm is unbiased for all $n = 2^m$ using induction over m.

Base case: For $m = 0$, i.e., $n = 1$ the algorithm terminates immediately. Since $i = j = 1$, $\mathbf{M}_n = [p_{1,1}] = [1] = \left[\frac{1}{n}\right]$ holds, which is obviously unbiased.
Induction step: The induction hypothesis is, that after $(\log_2 n) - 1$ steps the array is composed of two equally sized subarrays, both of which are permuted unbiasedly themselves. Hence, we are concerned with the following matrix:

$$\mathbf{M}_{\text{partial}} = \begin{bmatrix} 2/n & \dots & 2/n & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 2/n & \dots & 2/n & 0 & \dots & 0 \\ 0 & \dots & 0 & 2/n & \dots & 2/n \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 2/n & \dots & 2/n \end{bmatrix}$$

In the $\log_2 n$-th step, the algorithm exchanges element i and $i+\frac{n}{2}$ with probability $\frac{1}{2}$. It follows that in the final matrix $\mathbf{M}_n$ the rows i and $i+\frac{n}{2}$, for $i \in \{1, \dots, \frac{n}{2}\}$, are the arithmetic mean of the corresponding rows from $\mathbf{M}_{\text{partial}}$:

$$[\mathbf{M}_n]_i = [\mathbf{M}_n]_{i+\frac{n}{2}} = \frac{1}{2}\left(\left[\frac{2}{n}, ..., \frac{2}{n}, 0, ..., 0\right] + \left[0, ..., 0, \frac{2}{n}, ..., \frac{2}{n}\right]\right) = \left[\frac{1}{n}, ..., \frac{1}{n}\right] \square$$

3.2 Bias for $n \neq 2^m$

For generalizing the algorithm to arbitrary n two methods come to mind:

The first is to pad the array to the next larger power of two, do the permutation, and remove the padding in any way that preserves the relative order of the non-padding data, e.g., by using parallel prefix-sum [2]. In fact any compaction algorithm could be used, but in case it does not preserve the order it would qualify as permutation and would thus need to be analyzed as well.

On close inspection padding proves to be biased. Via simulations with 10^7 independent runs we obtained the following clearly biased matrices for $n \in \{3, 5\}$:

$$\mathbf{M}_{\text{Butter.\&pad.},3} \approx \begin{bmatrix} 0.313 & 0.313 & 0.375 \\ 0.375 & 0.375 & 0.250 \\ 0.312 & 0.312 & 0.375 \end{bmatrix} \quad \mathbf{M}_{\text{Butter.\&pad.},5} \approx \begin{bmatrix} 0.191 & 0.191 & 0.191 & 0.191 & 0.235 \\ 0.203 & 0.203 & 0.203 & 0.203 & 0.188 \\ 0.211 & 0.211 & 0.211 & 0.211 & 0.156 \\ 0.203 & 0.203 & 0.203 & 0.203 & 0.188 \\ 0.192 & 0.191 & 0.191 & 0.192 & 0.234 \end{bmatrix}$$

One might argue that the butterfly network with padding is pathological, but this also happens in other algorithms: Performing 10^7 independent simulations of Waksman's permutation network [15] with padding shows the exact same effect:

$$\mathbf{M}_{\text{Waks.\&pad.},3} \approx \begin{bmatrix} 0.313 & 0.312 & 0.375 \\ 0.375 & 0.375 & 0.250 \\ 0.313 & 0.312 & 0.375 \end{bmatrix} \quad \mathbf{M}_{\text{Waks.\&pad.},5} \approx \begin{bmatrix} 0.191 & 0.191 & 0.191 & 0.191 & 0.234 \\ 0.203 & 0.203 & 0.203 & 0.203 & 0.188 \\ 0.211 & 0.211 & 0.211 & 0.211 & 0.156 \\ 0.203 & 0.203 & 0.203 & 0.203 & 0.187 \\ 0.191 & 0.191 & 0.192 & 0.191 & 0.234 \end{bmatrix}$$

The second method to generalize the butterfly network is to simply use the network for the next larger power of two and omit the network's exchanges that involve non-existing array elements (as can be seen for $n = 5$ in Fig. 2, where non-existing elements and omitted exchanges are marked in red). Using this method some elements in the network do not have a corresponding element they

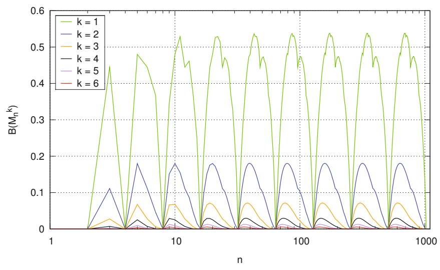


Fig. 3. Bias $\mathrm{B}({\mathbf{M}_n}^k)$ for $n \leq 1024$ after $k = 1$ to 6 rounds of the iterating approach

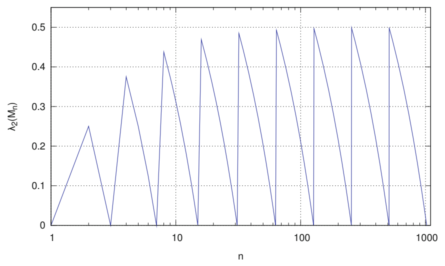


Fig. 4. Second largest eigenvalues λ_2 of $\mathbf{M}_n$ for $n \leq 1024$

could be exchanged with, which in turn leads to a bias. E.g., for $n = 5$ we can see that the corresponding matrix is not equal to the unbiased permutation matrix:

$$\begin{bmatrix} 1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1 \end{bmatrix} \rightarrow \begin{bmatrix} 1/2&1/2&0&0&0\\1/2&1/2&0&0&0\\0&0&1/2&1/2&0\\0&0&1/2&1/2&0\\0&0&0&0&1 \end{bmatrix} \rightarrow \begin{bmatrix} 1/4&1/4&1/4&1/4&0\\1/4&1/4&1/4&1/4&0\\1/4&1/4&1/4&1/4&0\\1/4&1/4&1/4&1/4&0\\0&0&0&0&1 \end{bmatrix} \rightarrow \begin{bmatrix} 1/8&1/8&1/8&1/8&1/2\\1/4&1/4&1/4&1/4&0\\1/4&1/4&1/4&1/4&0\\1/4&1/4&1/4&1/4&0\\1/8&1/8&1/8&1/8&1/2 \end{bmatrix} = \mathbf{M}_5$$

The probability for an element j to end up at position i is not uniformly distributed. Especially, element 5 can only be exchanged in the last step of the butterfly network and will thus end up at position 1 or 5.

Throughout the remainder of this paper the second generalization method will be used, because it is faster than the first and preserves in-placeness.

To quantify bias we define a bias measure, that gives the relative deviation from a uniform random permutation matrix averaged over all matrix elements:

$$\mathrm{B}(\mathbf{M}_n) = \frac{1}{n^2} \sum_{i,j} \frac{\left|\mathbf{M}_n(i,j) - \frac{1}{n}\right|}{\frac{1}{n}} = \frac{1}{n} \sum_{i,j} \left|\mathbf{M}_n(i,j) - \frac{1}{n}\right|$$

The butterfly network's bias w.r.t. n is shown in the topmost curve of Fig. 3.

4 Consistency

Since many applications deal with array sizes that are not a power of two, biased permutations may lead to severe problems that are relevant in practice. We therefore show in this section that a large group of algorithms including the butterfly network are consistent even though they may be biased. An algorithm is consistent, if iterative application ad infinitum eliminates its bias.

Formally, the matrices $\mathbf{M}$ described in the previous section are stochastic permutation matrices. To obtain the matrix that results from applying an algorithm k times, we raise its original matrix to the k-th power. Fig. 3 displays the resulting bias $\mathrm{B}({\mathbf{M}_n}^k)$ for $k \in \{1, \ldots, 6\}$ applications of the butterfly algorithm. The figure suggests, that for $k \to \infty$ the bias converges against 0.

In the following we will prove that this is the case for our algorithm as well as all algorithms whose stochastic permutation matrix $\mathbf{M}$ is positive ($m_{ij} > 0\ \forall i, j$)

and doubly (column and row) stochastic. Furthermore, we will demonstrate that the butterfly network's matrices are positive and doubly stochastic.

4.1 Convergence

To determine the bias for $k \to \infty$, we examine the Markov chain associated with $\mathbf{M}$: If the Markov chain's distribution converges against a uniform distribution, the algorithm is consistent. Because $\mathbf{M}$ is row-stochastic, a vector describing a uniform distribution is an eigenvector with corresponding eigenvalue 1:

$$\mathbf{M} \cdot \left(\tfrac{1}{n}, ..., \tfrac{1}{n}\right)^T = \left(\tfrac{1}{n} \textstyle\sum_j m_{1j}, ..., \tfrac{1}{n} \sum_j m_{nj}\right)^T = \left(\tfrac{1}{n}, ..., \tfrac{1}{n}\right)^T$$

Hence, the uniform distribution is stable in the Markov chain. Stability does, however, not imply that the Markov chain converges against this distribution.

Using not only $\mathbf{M}$'s row stochasticity but also column stochasticity and positiveness, the convergence can be shown using the Perron-Frobenius theorem [13,11]: It states, that all positive matrices $\mathbf{M}$ have an eigenvalue r that satisfies the condition $\min_i \sum_j m_{ij} \leq r \leq \max_i \sum_j m_{ij}$. For row stochastic matrices we obtain $r = 1$. This Perron-Frobenius eigenvalue is a simple eigenvalue and strictly greater than all other eigenvalues' absolute values. r has a corresponding right eigenvector v and a left eigenvector w. Because $\mathbf{M}$ is row stochastic,

$$\mathbf{M} \cdot \left(\tfrac{1}{\sqrt{n}}, ..., \tfrac{1}{\sqrt{n}}\right)^T = \left(\tfrac{1}{\sqrt{n}} \textstyle\sum_j m_{1j}, ..., \tfrac{1}{\sqrt{n}} \sum_j m_{nj}\right)^T = \left(\tfrac{1}{\sqrt{n}}, ..., \tfrac{1}{\sqrt{n}}\right)^T$$

holds, where $(\frac{1}{\sqrt{n}}, ..., \frac{1}{\sqrt{n}})^T$ is a solution for v and (with analogous reasoning using $\mathbf{M}$'s column stochasticity) for w. Then the Perron-Frobenius theorem states:

$$\lim_{k\to\infty} \frac{\mathbf{M}^k}{r^k} = \lim_{k\to\infty} \mathbf{M}^k = \frac{vw^T}{w^T v} = \begin{bmatrix} 1/n & \dots & 1/n \\ \vdots & \ddots & \vdots \\ 1/n & \dots & 1/n \end{bmatrix} \qquad \square$$

The constraint for $\mathbf{M}$ to be doubly stochastic is not a real constraint: Every permutation algorithm's matrix is doubly stochastic, because in the permutation process elements must not be lost and no new elements may be inserted into the list.

Therefore, any random permutation algorithm is consistent, if the probability for any element i to be moved to position j is positive.

The butterfly network's matrices are doubly stochastic, but not positive, as some entries are 0. However, we can show that $\mathbf{M}_n{}^2 > \mathbf{0}$: The algorithm's matrices contain entries greater than 0 in at least the first row and the first column (proof is omitted). It follows that (with $\mathbf{A} > \mathbf{B}$ meaning $a_{ij} > b_{ij}\ \forall i, j$)

$$\mathbf{M}_n{}^2 > \begin{bmatrix} m_{11} & m_{12} & \dots & m_{1n} \\ m_{21} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & 0 & \dots & 0 \end{bmatrix}^2 > \begin{bmatrix} m_{11} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & 0 & \dots & 0 \end{bmatrix} \cdot \begin{bmatrix} m_{11} & \dots & m_{1n} \\ 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix} = \begin{bmatrix} m_{11} \cdot m_{11} & \dots & m_{11} \cdot m_{1n} \\ \vdots & \ddots & \vdots \\ m_{n1} \cdot m_{11} & \dots & m_{n1} \cdot m_{1n} \end{bmatrix} > \mathbf{0}$$

for positive $m_{11}, \ldots, m_{1n}, m_{21}, \ldots, m_{n1}$. Because $\mathbf{M}_n{}^2$ is positive, $\lim\limits_{k\to\infty} (\mathbf{M}_n{}^2)^k$ is the uniform distribution matrix.

4.2 Convergence Speed

For algorithms with positive permutation matrices $\mathbf{M}$ the convergence speed can be shown as well:

Let $\mathbf{U}$ be the uniform distribution matrix, $u = (\frac{1}{\sqrt{n}}, ..., \frac{1}{\sqrt{n}})^T$, $\mathbf{V} = (v_1, ..., v_n)$ be the matrix of $\mathbf{M}$'s eigenvectors, and $\mathbf{V}^{-1} = \mathbf{W} = (w_1, ..., w_n)^T$ be its inverse. Furthermore we assume the eigenvalues λ_l to be in decreasing order. For positive, doubly stochastic matrices the Perron-Frobenius theorem ensures that $1 = r = \lambda_1 > |\lambda_l|\ \forall l \in \{2, ..., n\}$ and $v_1 = w_1^T = u$. By eigendecomposing $\mathbf{M}^k$, we obtain

$$\begin{aligned}
\|\mathbf{M}^k - \mathbf{U}\| &= \left\|v_1 \lambda_1^k w_1 + ... + v_n \lambda_n^k w_n - u \cdot 1 \cdot u^T\right\| \\
&= \|u \cdot 1^k \cdot u^T + v_2 \lambda_2^k w_2 + ... + v_n \lambda_n^k w_n - u \cdot 1 \cdot u^T\| \\
&\leq \sum_{l=2}^{n} |\lambda_l|^k \|v_l w_l\| \leq |\lambda_2|^k \sum_{l=2}^{n} \|v_l w_l\| \in \mathcal{O}(|\lambda_2|^k)
\end{aligned}$$

Note, that $\|v_l w_l\|$ only depends on $\mathbf{M}$ but not on k. Using $|\lambda_2| < 1$, it follows that $\|\mathbf{M}^k - \mathbf{U}\|$ decreases exponentially with k. □

Since our only assumptions were positiveness and double stochasticity, exponential convergence holds for all permutation algorithms with positive $\mathbf{M}$.

Fig. 4 shows all λ_2 of the butterfly network's $\mathbf{M}_n$ for $n \leq 1024$. The graph displays a periodical behavior. The maximum values in each $(2^m, 2^{m+1})$-interval converge against 0.5. We note, that the local maxima can be found at $n = 2^m + 1$. This is in agreement with the fact that the maxima in every $(2^m, 2^{m+1})$-interval in Fig. 3 are further to the left for larger k. Further analysis showed that for much larger k (e.g., $k = 51$) the bias maxima are indeed located at $n = 2^m + 1$.

5 Improvement for the Butterfly Permutation Network

The proportionality constant of the butterfly network's exponential convergence can be improved using *shifting*, i.e., the repeated application of the algorithm is interleaved with circular shifting of the data using some offset l. Circular shifting moves any element i of an array to position $(i + l) \bmod n$. The underlying idea is to choose l such that array positions with a big bias are shifted to positions with a smaller bias. For $\mathbf{M}_{\text{shift},l}$ being a regular, non-stochastic permutation matrix that circularly shifts by l positions we obtain the combined matrix $(\mathbf{M}_{\text{shift},l} \cdot \mathbf{M}_n)^k$.

Fig. 5 displays the bias for two array sizes and various shifting offsets. The graph for $n = 304$ reveals a global minimum for a shifting offset of 229 where the bias' magnitude is reduced to 1.8% of the bias without shifting ($l = 0$). The graph for $n = 400$ demonstrates that the bias can even exceed the bias without shifting. E.g., if we shift with an offset of 112, the bias is 3.4 times larger than without shifting. Comparing the graphs for the two different choices of n shows large differences in the bias' overall behavior. Therefore, l needs to be carefully selected for each n to achieve optimal results.

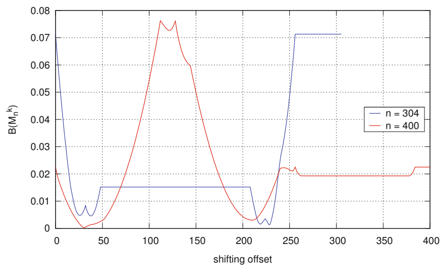


Fig. 5. Bias $\mathrm{B}(\mathbf{M}_n{}^k)$ for $k = 3$ iterations and array sizes of $n = 304$ (blue) and $n = 400$ (red) with shifting offsets from 0 to n

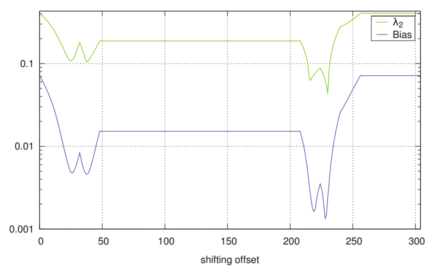


Fig. 6. Comparison of the behavior of λ_2 and bias for $n = 304$ and shifting offsets ranging from 0 to n

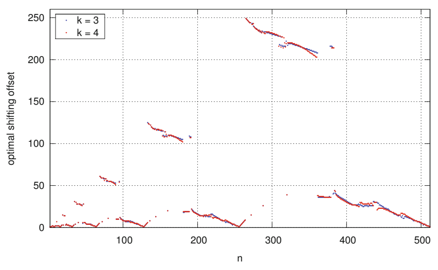


Fig. 7. Optimal shifting offsets for $n \in \{1, \ldots, 512\}$ and $k \in \{3, 4\}$ iterations

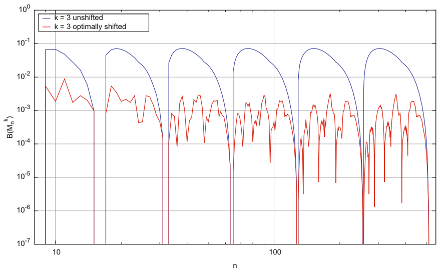


Fig. 8. Bias for 3 iterations with optimal (red) and without shifting (blue)

Optimal Shifting Offset. We empirically determined optimal shifting offsets for a range of array sizes by analyzing the permutation matrices $(\mathbf{M}_{\text{shift}} \cdot \mathbf{M}_n)^k$ for $k = 3$ and 4 (see Fig. 7). Note, that the optimal shifting offset is slightly different for different choices of k. The optimal offset can be precomputed at programming time for any combination of k and n. Using these optimal offsets, we obtained the bias graph for $k = 3$ as shown in Fig. 8, which shows a clear improvement over the basic algorithm without shifting.

Convergence of Shifting. Shifting is still consistent, because the convergence arguments from Sec. 4 also apply for the shifting extension (since $(\mathbf{M}_{\text{shift}} \cdot \mathbf{M}_n)^2 > \mathbf{0}$). Furthermore, Fig. 6 shows a strong correlation between the λ_2 and the bias of $\mathbf{M}_n \cdot \mathbf{M}_{\text{shift},1}$ for various shifting offsets. This shows, first, that the theoretically derived measure is valid in a practical application and second, that the bias (and thus the optimal shifting offset) can be predicted using the second eigenvalues.

6 Experimental Results

To demonstrate the butterfly network's speed and suitability for many-core systems we implemented our algorithm on GPUs using NVIDIA's CUDA framework

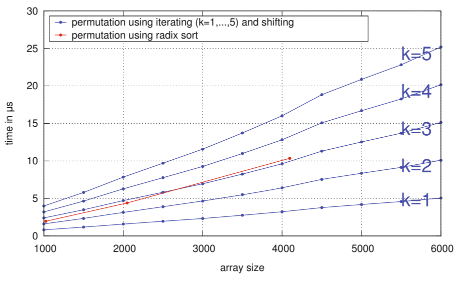


Fig. 9. Speed of shuffling arrays of various sizes with our algorithm using iterating ($k = 1, \ldots, 5$) and shifting, compared to a CUDPP-based radix sort shuffling

[12]. Fig. 9 shows the runtime for performing array permutations with sizes up to $n = 6000$ and $k = 1, ..., 5$ iterations with optimal shifting. Each array was permuted by a single CUDA thread block and stored in shared memory. All experiments were run on a GeForce GTX 480. The implementation is not limited to $n = 6000$ but can permute arrays of up to 12,288 4-byte or 49,152 1-byte values. 49,152 byte is the maximum shared memory that can be used by one thread block on current NVIDIA GPUs.

For comparison, we also implemented a second algorithm based on the Rand_Sort approach using radix sort for sorting. Rand_Sort is unbiased but cannot be implemented in-place since random sorting keys need to be stored for each array element. We used the CUDPP library's [4] radix sort implementation, which requires array sizes smaller than 4096. This is not a fundamental limitation but merely an implementation issue. The CUDPP code is highly optimized and can thus be legitimately regarded as suitable state-of-the-art performance reference.

Fig. 9 shows that the butterfly network with $k = 3$ and optimal shifting is about as fast as the reference approach while requiring less memory due to being in-place. If a higher bias is acceptable, using $k = 2$ or 1 yields significant speedups of roughly 1.5 and 3, respectively.

7 Conclusions and Future Work

We showed that not all random permutation algorithms can be easily generalized for n that are not a power of two without introducing some bias. We proved that any algorithm, whose corresponding stochastic permutation matrix is positive, is nevertheless consistent, i.e., iterative application reduces the bias at an exponential rate. We also gave a specific example of a biased algorithm, the butterfly network, whose convergence behavior is further improved by shifting with carefully chosen shifting offsets. Because the butterfly network is well-suited for implementation on a GPU (due to enabling fine-grained parallelism, needing few thread synchronizations and no contention resolving scheme), we implemented it on a current GPU and it proved to be competitive to or even faster than another highly optimized algorithm well suited for GPUs.

Further analysis on more involved shifting strategies is needed: In this paper we determined optimal offsets l as a function of n. There is room for further improvement, e.g., by applying varying shifting offsets after each iteration. This would require intensive pre-computations of lookup-tables for all n and k, which is beyond the scope of this work. Also, it would be interesting to analyze whether or not shifting can improve the convergence of other algorithms as well.

Acknowledgements. We thank Jörg Keller for originally directing our attention towards the paper by Cong and Bader. KH is grateful for financial support by the Fonds der Chemischen Industrie and through the Graduiertenkolleg 1657 of the Deutsche Forschungsgemeinschaft. Also, we would like to thank Dominik Wodniok for implementing the radix sort performance reference for Sec. 6.

References

1. Anderson, R.: Parallel algorithms for generating random permutations on a shared memory machine. In: Proc. SPAA 1990, pp. 95–102. ACM (1990)
2. Blelloch, G.E.: Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University (November 1990)
3. Cong, G., Bader, D.A.: An empirical analysis of parallel random permutation algorithms on SMPs. In: Oudshoorn, M.J., Rajasekaran, S. (eds.) ISCA PDCS, pp. 27–34 (2005)
4. CUDPP – CUDA data parallel primitives library, `http://code.google.com/p/cudpp/`
5. Czumaj, A., Kanarek, P., Kutylowski, M., Lorys, K.: Fast Generation of Random Permutations via Networks Simulation. In: Díaz, J. (ed.) ESA 1996. LNCS, vol. 1136, pp. 246–260. Springer, Heidelberg (1996)
6. Hagerup, T.: Fast Parallel Generation of Random Permutations. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 405–416. Springer, Heidelberg (1991)
7. Holmes, S.: Bootstrapping Phylogenetic Trees: Theory and Methods. Statistical Science 18(2), 241–255 (2003)
8. Knuth, D.E.: The art of computer programming, 3rd edn., vol. 2 (1997)
9. Knuth, D.E.: The art of computer programming, volume 3 (2nd ed.) (1998)
10. Leighton, F.: Introduction to parallel algorithms and architectures: arrays, trees, hypercubes, vol. (1). M. Kaufmann Publishers (1992)
11. Meyer, C.: Matrix Analysis and Applied Linear Algebra. SIAM (2000)
12. NVIDIA: NVIDIA CUDA C programming guide, version 3.2 (2011)
13. Perron, O.: Zur Theorie der Matrices. Mathematische Annalen 64, 248–263 (1907)
14. Soltis, P.S., Soltis, D.E.: Applying the bootstrap in phylogeny reconstruction. Statistical Science 18(2), 256–267 (2003)
15. Waksman, A.: A permutation network. J. ACM 15, 159–163 (1968)
16. Wu, C.F.J.: Jackknife, bootstrap and other resampling methods in regression analysis. Ann. Statist. 14(4), 1261–1295 (1986)
17. Zoubir, A.M.: Model selection: A bootstrap approach. In: Proc. ICASSP (1999)

Extracting Coarse–Grained Parallelism for Affine Perfectly Nested Quasi–uniform Loops

Włodzimierz Bielecki and Krzysztof Kraska

Faculty of Computer Science and Information Technology,
West Pomeranian University of Technology, ul.Żołnierska 49, 71-210 Szczecin, Poland
{wbielecki,kkraska}@wi.zut.edu.pl

Abstract. This paper presents a new approach for the extraction of coarse–grained parallelism available in program loops. The approach permits for extracting parallelism for both uniform and quasi–uniform perfectly nested parameterized loops, where the loop bounds and data accesses are affine functions of loop indices and symbolic parameters. It extracts a set of synchronization–free code fragments. The procedure has a polynomial time complexity except for one step of calculations. The effectiveness and time complexity of the approach are evaluated by means of loops of the NAS Parallel Benchmark suite.

Keywords: parallelizing compilers, loop transformation, quasi–uniform dependences, polyhedral model.

1 Introduction

A parallel computer requires adequate software that can take advantage of the computational power of multiple processors. Manual writing parallel programs is time and cost consuming. Therefore, we need a compiler that can be able to produce a parallel program automatically. Moreover, to effectively utilize the power of contemporary shared memory parallel machines, a compiler have to find coarse–grained parallelism that does not incur or significantly reduces synchronization.

In [1] a method is presented for the identification of independent subsets of operations in loops but it can be applied to uniform loops only. Over a decade later, the Affine Transformation framework was introduced permitting for extracting coarse–grained parallelism [2,3]. But it has large computational complexity. Other ones, for example [4], are computationally too complex to be widely used in commercial compilers. Therefore, solutions characterized by a polynomial complexity, such as that presented in [1], are desired.

The main purpose of this paper is to present a new approach for the extraction of coarse–grained parallelism available in program loops and being characterized mainly by a polynomial time complexity. It deals with perfectly nested static–control loops, where the loop bounds as well as array subscripts are affine functions of symbolic parameters and surrounding loop indices. The polyhedral model is adopted in the proposed algorithm.

2 Background

In this section, we briefly introduce necessary preliminaries which are used throughout this paper.

The following concepts of linear algebra are used in the approach presented in this paper: a polyhedron, lattice, the Hermite Normal Form of a matrix of full row rank and its uniqueness, Hermite decomposition, affine lattice canonical form. Details can be found in papers [5,6,7].

Definition 1 (Congruence relation, modulus matrix). *Let y and z be two d–dimensional integral vectors and D be some integral $d \times d$ matrix of full row rank. We say that y is congruent to z modulo the column image of D, written:*

$$y \equiv z \mod D,$$

if and only if the difference $z - y$ is equal to Dx for some d–dimensional integral vector $x \in \mathbb{Z}^d$. Matrix D is called the modulus matrix [8].

Definition 2 (Equivalence relation). *Matrix D yields an equivalence relation, denoted by $\backsim_D$, which is defined by $y \backsim_D z$ if and only if $y \equiv z \mod D$* [8].

Definition 3 (Equivalence class). *An equivalence class in a set is the subset of all elements which are in equivalence relation $\backsim_D$. The number of equivalence classes of $\backsim_D$ is denoted by $vol(D)$, the volume of D, which is the absolute value of the determinant of D. If the determinant of D is zero, then D does not have full row rank and thus $\backsim_D$ has the infinite number of equivalence classes. An equivalence class of $\backsim_D$ is also called a lattice* [8].

Definition 4 (Representatives). *The set of all integral vectors in the parallelepiped $R(D)$ defined by the columns of D:*

$$R(D) = \{x \in \mathbb{Z} \mid x = D \cdot \alpha,\ \alpha \in \mathbb{R}^d,\ 0 \leq \alpha \leq 1\},$$

defines a set of representatives for the equivalence classes of $\backsim_D$ [8].

In this paper, we deal with the following definitions concerned program loops: parameterized nested loops, iteration vector, loop domain (index set) whose explanations are given in papers [3,9].

Definition 5 (Dependence). *Two statement instances $S_1(I)$ and $S_2(I)$ are dependent if both access the same memory location and if at least one access is a write. Provided that $S_1(I)$ is executed before $S_2(I)$, $S_1(I)$ and $S_2(I)$ are called the source and destination of the dependence, respectively.*

Definition 6 (Dependence distance set, dependence distance vector). *We define a dependence distance set $D_{S,T}$ as a set of all differences of iteration vectors of T and S for their commonly surrounding loops. Each element of set $D_{S,T}$, we call a dependence vector and denote it as $d_{S,T}$* [3].

Definition 7 (Uniform dependence, non–uniform dependence). *If the difference of iteration vectors $\boldsymbol{i_t}$ and $\boldsymbol{i_s}$ is constant for dependent statement instances $T(\boldsymbol{i_t})$ and $S(\boldsymbol{i_s})$, we call the dependence uniform, otherwise the dependence is non–uniform* [3].

Definition 8 (Uniform loop, quasi–uniform loop). *We say that a parameterized loop is uniform if it induces dependences represented by the finite number of uniform dependence distance vectors* [3]. *A parameterized loop is quasi–uniform if all its dependence distance vectors can be represented by a linear combination of the finite number of linearly independent vectors with constant coordinates.*

Let us consider a parameterized dependence distance vector $(N, 2)$. It can be represented as $(0, 2) + a \times (1, 0)$, where $a \geq 1, a \in \mathbb{Z}$ (see Fig. 1).

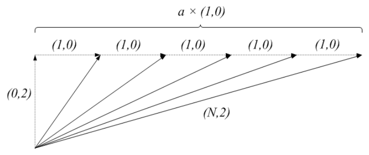


Fig. 1. The parameterized vector $(N, 2)$ represented as the linear combination of the two linearly independent vectors with constant coordinates

To find constant vectors representing a parameterized vector, we can apply the algorithm presented in [10] and implemented in [7]. As input it takes a polyhedron and returns: (1) an affine lattice canonical form for that polyhedron and (2) the number that determines the space of the polyhedron.

Definition 9 (Dependence Sources Polyhedron). *Given a dependence distance vector $d_{S,T}$, a dependence sources polyhedron $DSP(d_{S,T})$ is the set of all the values of iteration vector $\boldsymbol{i_S}$ such that there exist dependences between $S(\boldsymbol{i_S})$ and $T(\boldsymbol{i_S} + d_{S,T})$.*

Definition 10 (Polyhedral Reduced Dependence Graph). *A Polyhedral Reduced Dependence Graph (PRDG) is the graph where a vertex stands for every statement S and an edge connects statements S and T whose instances are dependent. The number of edges between vertices S and T is equal to the number of vectors $d_{S,T} \in D_{S,T}$. Every edge is labeled with a dependence distance vector $d_{S,T}$ and a dependence sources polyhedron $P(d_{S,T})$.*

3 Approach to Extracting Parallelism

It is well–known [1] that code fragments, representing different equivalence classes, are independent, hence they can be executed in parallel. The goal of the algorithm presented below is to extract equivalence classes for both uniform and quasi–uniform loops.

3.1 Algorithm Extracting Equivalence Classes for Both Uniform and Quasi–uniform Loops

Input: A set of dependence distance vectors for a given loop.
Output: Equivalence classes.

Method:

1. *Replace each parameterized dependence distance vector by a combination of linearly independent vectors.* For this purpose apply the algorithm presented in [10]. Skip this step for uniform loops.
2. *Form a dependence distance set.* Form matrix D, $D \in \mathbb{Z}^{n\times m}$, whose m columns are all non–parameterized dependence distance vectors $d_{S,T}$ corresponding to the edges of $PRDG$. Associate row k of $D^{n\times m}$ with a loop index i_k, $k = 1, 2, \ldots, n$ where n is the number of loop indices (i.e. surrounding loops).
3. *Form the basis of a lattice from the dependence distance set.* Transform matrix D into two sub–matrices D', $D' \in \mathbb{Z}^{l\times m}$ and D'', $D'' \in \mathbb{Z}^{(n-l)\times m}$, such that l rows of D', $1 \le l \le n$, are linearly independent and $(n-l)$ rows of D'' are linearly dependent. When interchanging two rows, interchange also the loop indices associated with these rows.
4. *Find lattice canonical form.* Transform sub–matrix D' to the Hermite Normal Form:
$$D' = HU = [B\ \ 0]U,\ B \in \mathbb{Z}^{l\times l},$$
preserving loop indices associated with the rows of D'. Note that the lattice canonical form represents the equivalence relation.
5. *Find representatives for equivalence classes.* Using B, calculate a set of representatives (one for each equivalence class) depending on the following cases:
   (a) $l = n$: define a set $R(B)$ of representatives for equivalence classes as the set of all integral vectors in the parallelepiped defined by the columns of B,
$$R\left(B\right) = \{x \in \mathbb{Z} \mid x = B\cdot\alpha,\ \alpha \in \mathbb{R}^d,\ 0 \le \alpha \le 1\}.$$
   (b) $l < n$: find the first l coordinates of representatives for equivalence classes as follows:
$$R\left(B^l\right) = \{x^l \in \mathbb{Z}^l \mid x^l = B^l\cdot\alpha,\ \alpha \in \mathbb{R}^l,\ 0 \le \alpha \le 1\},$$
   and enlarge matrix B^l to matrix B by inserting the last $n-l$ zero rows.
6. *Find equivalence classes.* Using representatives x, $x \in R\left(B\right)$, form the following polyhedra that specifies equivalence classes:
$$P\left(x\right) = \left\{\begin{array}{c} y = x + Bz \mid x \in R(B), z \in \mathbb{Z}^n, y \in P(d_{S,T})\cup \\ \cup\{J \mid J = I + d_{S,T} \wedge I \in P(d_{S,T})\}, S,T \in \text{vertices of}\, PRDG \end{array}\right\}.$$

Each equivalence class represents an independent subset of statement instances which are represented by an equivalence relation.

Having equivalence classes, we can apply any well-known code generation algorithm to generate parallel outer loops scanning a set of representatives $R(B)$ and sequential inner loops enumerating in the lexicographical order the elements of set $P(x)$ for every equivalence class represented by x.

3.2 Degree of Parallelism

The degree of parallelism is characterized by the number of equivalence classes. Let us remind that an integral $l \times l$ matrix B of full row rank defines a set of representatives $R(B)$. Thus, the number of equivalence classes is the absolute value of the determinant of B, the volume of B (see Definition 3). In [11] it is proven that if $B \in \mathbb{Z}^{l \times l}$, then the number of equivalence classes is $\prod_{i=1}^{l} b_{ii}$.

Now, similarly to [1], we can investigate available parallelism by a simple inspection of matrix B:

$$vol(B) = \prod_{i=1}^{l} b_{ii},\ B \in \mathbb{Z}^{l \times l}.$$

However, when $l < n$ then a set of representatives is defined by an enlarged matrix B. To find the degree of parallelism, we need to compute $vol(B^l)$ and take into consideration the number of iterations for loop indices corresponding to $n - l$ rows of the enlarged matrix B:

$$vol(B) = vol(B^l) \times \prod_{i=n-l}^{n} vol(D_i^S) = \prod_{i=1}^{l} b_{ii}^l \times \prod_{i=n-l}^{n} vol(D_i^S).$$

3.3 Time Complexity

Except for the first step, all the other steps of the algorithm can be accomplished in polynomial time. The proof is below.

1. The task of identifying a set of linearly independent rows of a matrix D, $D \in \mathbb{Z}^{n \times m}$ with constant coordinates and dependent ones can be done in polynomial time by the Gaussian elimination. According to [11], this computation can be done in $\mathcal{O}(ldm)$ arithmetic operations.
2. The task of transforming a matrix $D', D' \in \mathbb{Z}^{l \times m}$ to its Hermite Normal Form matrix $B, B \in \mathbb{Z}^{l \times l}$ can be accomplished, depending on the algorithm used, even in $\mathcal{O}\left(l^{\theta-1} m \log(2m/l) B(l \log(l \|D'\|))\right)$ operations [11], where $\theta < 2.376$.
3. A set $R(B)$ of representatives for equivalence classes is defined as a set of all integral vectors in the parallelepiped defined by the columns of B. According to [8] such a set of representatives can be found by enumerating the equivalence classes with nonnegative integral vectors in the lexicographical order ($y \prec z$ if there is some $i, 1 \le i \le l$, such that $y_i < z_i$ and for $j = 1, \ldots, i-1$ and $y_j = z_j$). Enumerating l diagonal coefficients of B requires $\mathcal{O}(l)$ operations and enumerating $n - l$ rows of D'' requires $\mathcal{O}(n \times (n-l))$ operations.

3.4 Example

Let us consider the following example:

```
for(i=2; i<N; i++)
  for(j=2; j<N; j++)
    for(k=0; k<N; k++)
      S: a[i][j][k]=a[i][j-1][k]+a[i-2][j][k]+a[i-2][P][k];
```

$PRDG$ describing dependences in the loop is represented by the following $DSPs$:

$$\begin{aligned}&DSP(0,1,0) = \{(i,j,k) \mid 2 \le i < N, 2 \le j \le N-2, 0 \le k < N\},\\ &DSP(2,0,0) = \{(i,j,k) \mid 2 \le i \le N-3, 2 \le j < N, 0 \le k < N\},\\ &DSP(2,M,0) = \{(i,j,k) \mid 1 \le M\}.\end{aligned}$$

The algorithm produces the following results.

1. *Replace each parameterized distance vector by a linear combination of constant vectors.*
   The parameterized vector:

$$P(2,M,0) = \begin{cases} 1 \times i + 0 \times M + 0 \times k - 2 & = 0 \\ 0 \times i + 1 \times M + 0 \times k - 1 & \ge 0 \\ 0 \times i + 0 \times M + 1 \times k + 0 & = 0 \end{cases},$$

   is replaced by the vectors: $\begin{pmatrix}0\\1\\0\end{pmatrix}$ and $\begin{pmatrix}2\\0\\0\end{pmatrix}$ (see Fig. 2a).

2. *Form the dependence distance set.*

$$\begin{pmatrix}i\\j\\k\end{pmatrix} = \left(\begin{pmatrix}0\\1\\0\end{pmatrix}\begin{pmatrix}2\\0\\0\end{pmatrix}\begin{pmatrix}0\\1\\0\end{pmatrix}\begin{pmatrix}2\\0\\0\end{pmatrix}\right) = \begin{pmatrix}0\,2\,0\,2\\1\,0\,1\,0\\0\,0\,0\,0\end{pmatrix}, \ n = 3.$$

3. *Form the basis of a lattice from the dependence distance set* (see Fig. 2b).

$$D'\begin{pmatrix}i\\j\end{pmatrix} = \begin{pmatrix}0\,2\,0\,2\\1\,0\,1\,0\end{pmatrix}, \ D''\left(k\right) = \left(0\,0\,0\,0\right).$$

4. *Find the lattice canonical form.*

$$B\begin{pmatrix}i\\j\end{pmatrix} = \begin{pmatrix}2\,0\\0\,1\end{pmatrix}, \ l = 2.$$

5. *Find representatives for equivalence classes.*
   In the case $l < n$, we find the first l coordinates of representatives for equivalence classes as follows (see Fig. 2c):

$$R\left(B^l\right) = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}.$$

The final set of representatives is of the form:

$$R(B) = \left\{ \begin{pmatrix} 0 \\ 0 \\ k \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ k \end{pmatrix} \right\}.$$

The enlarged matrix $B \in \mathbb{Z}^{l \times l}$ to matrix $B \in \mathbb{Z}^{n \times l}$ is as follows:

$$B \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 2\,0 \\ 0\,1 \\ 0\,0 \end{pmatrix}.$$

6. *Find equivalence classes* (see Fig. 2d).

$$P(x) = \left\{ \begin{array}{c} y = x + \begin{pmatrix} 2\,0 \\ 0\,1 \\ 0\,0 \end{pmatrix} z \mid x \in R(B), \\ (2,2,0) \prec y \prec (N-1, N-1, N-1), z \in \mathbb{Z}^2 \end{array} \right\}.$$

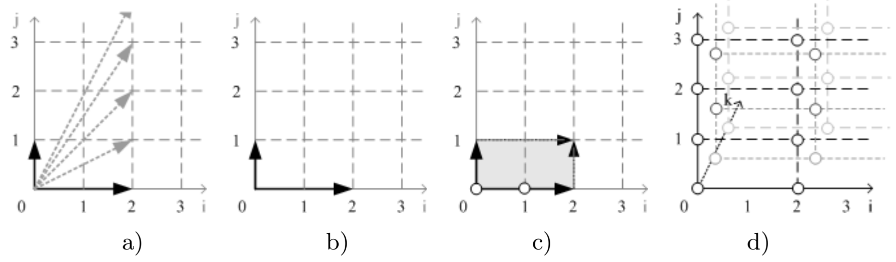


Fig. 2. Illustration of some selected steps in the method: a) replacement of the parameterized dependence distance vector, b) the basis of a lattice, c) representatives for equivalence classes, d) the equivalence classes for representatives $(0, 0, k)$

The degree of parallelism is as follows:

$$vol(B) = \prod_{i=1}^{l} b_{ii}^{l} \times \prod_{i=n-l}^{n} vol(D_i^S) = vol \begin{pmatrix} 2\,0 \\ 0\,1 \end{pmatrix} \times N = 2N.$$

Applying CLooG, the well–known code generation tool [12], to scan elements of $P(x)$ we get the following code:

```
if(N>=3)
 for(REPR_i=0; REPR_i<=1; REPR_i++)     /* parallel */
   for(k=0; k<=N-1; k++)                /* parallel */
     for(i=2; i<=N-1; i++)
       for(j=2; j<=N-1; j++)
         if((-REPR_i+i)%2==0)
           S: a[i][j][k]=a[i][j-1][k]+a[i-2][j][k]+a[i-2][P][k];
```

The two outer loops scan the set of representatives and can be executed in parallel. The inner two loops enumerate (sequentially) elements contained in the same equivalence class being defined by a representative pointed out by the indices of the first two outer loops.

4 Experiments

We implemented our algorithm as an ANSI–C++ software module using the well–known library PolyLib v5.22.5 [7]. Additionally, we used the following well–known tools: Petit (from Omega Project v2.1 [14]) for dependence analysis, and CLooG v0.14.1 [12] for code generation, to be able to operate directly on source codes and generate output ones.

In order to get a feeling of the performance of our approach, we carried out experiments with the well–known NAS Parallel Benchmark (NPB) suite from NASA [13]. We found 185 perfectly nested loops for which the proposed algorithm of identifying equivalence classes could be potentially applied. During the dependence analysis, Petit returns the results presented in Table 1.

Table 1. The quantitative distribution of loops in terms of dependence types

	Petit's dependence analysis results		No. loops
1)	Error reports during analysis (Petit's shortcomings)	:	28
2)	No dependences	:	123
3)	Uniform dependences, including:	:	14
	non loop–carried dependences	:	9
	loop–carried dependences	:	5
4)	Parameterized distance vectors	:	**20**
	The total number of perfectly nested loops	:	185

Quasi–uniform loops (contained parameterized distance vectors) were parallelized by means of the algorithm presented in this paper using the following machine: Intel PentiumM 1.5GHz with Linux openSUSE v11.1 32–bit operating system. The results of the experiments are included in Table 2 where time is presented in microseconds.

For loops exposing parallelism, we generated parallel OpenMP code being compiled by means of the gcc v4.5.1 compiler. To define the speed–up of parallel code, we have used an Intel Core i7–2630QM 2.00GHz machine with the Linux openSUSE v11.4 32–bit operating system. It is worth to note that speed–up, s, for several parallel loops is superlinear, i.e., $s > p$ (see Table 2).

The tool takes the most time to replace parameterized distance vectors by a linear combination of constant vectors. The other steps of the algorithm have performed several times faster. Under our experiment, the whole time, required for extracting equivalence classes, does not exceed 3 milliseconds. This fact permits us to conclude that the presented approach can be successfully applied for building optimizing compilers extracting automatically coarse–grained parallelism available in real–life loops.

Table 2. Effectiveness and time complexity of the proposed approach for NPB's quasi–uniform loops

#	Source loop	Degree of parallelism	Sizes $\forall N$	Speed-up $\frac{t_1}{t_p}, p = 4$	No. param. vectors	Time taken by each step of the algorithm $[\mu s]$					
						-1-	-2-	-3-	-4-	-5-	-6-
1)	UA_adapt.f2p_2.t	*N1*N3*N4*	99	4.7614	24	2416	2	267	51	1	221
2)	FT_auxfnct.f2p_1.t	*1*	-	-	1	56	1	6	5	-	-
3)	UA_diffuse.f2p_2.t	*1*	-	-	3	148	1	18	6	-	-
4)	UA_diffuse.f2p_3.t	*N1*N3*N4*	99	2.7457	3	346	1	23	6	2	204
5)	UA_diffuse.f2p_4.t	*N1*N3*N4*	99	7.4480	3	312	2	21	5	1	225
6)	UA_diffuse.f2p_5.t	*N2*N3*N4*	99	4.6477	3	348	1	21	5	1	238
7)	UA_precond.f2p_3.t	*N1*	699	1.0780	3	237	1	15	6	1	202
8)	UA_setup.f2p_16.t	*N1*N2*	999	2.9474	3	321	1	20	6	1	221
9)	UA_transfer.f2p_1.t	*1*	-	-	3	197	1	15	6	-	-
10)	UA_transfer.f2p_2.t	*1*	-	-	3	170	1	13	6	-	-
11)	UA_transfer.f2p_3.t	*1*	-	-	3	168	1	13	6	-	-
12)	UA_transfer.f2p_5.t	*1*	-	-	3	170	1	14	5	-	-
13)	UA_transfer.f2p_6.t	*1*	-	-	3	167	1	13	6	-	-
14)	UA_transfer.f2p_7.t	*N1*	699	1.0349	3	269	1	16	7	1	202
15)	UA_transfer.f2p_8.t	*1*	-	-	3	169	1	16	8	-	-
16)	UA_transfer.f2p_9.t	*N1*	699	1.4263	3	270	1	18	5	1	199
17)	UA_transfer.f2p_10.t	*1*	-	-	3	173	1	14	6	-	-
18)	UA_transfer.f2p_13.t	*N1*	699	2.1882	3	262	1	15	5	1	203
19)	UA_transfer.f2p_15.t	*N1*	699	1.4815	3	257	1	15	6	1	202
20)	UA_transfer.f2p_18.t	*N1*	699	2.2140	3	256	1	15	6	1	196

5 Conclusions

In this paper, we have presented a new approach that permits for the extraction of coarse–grained parallelism available not only in uniform loops (as the approach presented in [1]) but also in quasi–uniform perfectly nested parameterized loops. The experiments conducted on perfectly nested loops from the NAS Parallel Benchmark suite demonstrate that the presented approach is very fast.

In our next work, we plan to extend the approach to imperfectly nested loops and investigate its effectiveness and time complexity.

References

1. D'Hollander, E.H.: Partitioning and Labeling of Loops by Unimodular Transformations. IEEE Transactions on Parallel and Distributed Systems 3(4), 465–476 (1992)
2. Lim, A.W., Lam, M.S.: Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. Parallel Computing 24(3-4), 445–475 (1998)
3. Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Achitectures. Habilitation. Fakultät für Mathematik und Informatik Universität Passau (2004)

4. Beletska, A., Bielecki, W., Pietro, P.S.: Extracting Coarse-Grained Parallelism in Program Loops with the Slicing Framework. In: ISPDC, pp. 203–210 (2007)
5. Schrijver, A.: Theory of Linear and Integer Programming. Series in Discrete Mathematics (1999)
6. Nookala, P.K.V.V., Risset, T.: A Library for ℤ–Polyhedral Operations. Publication interne n.1330, Institut de Recherche en Informatique et Systèmes Aléatoires (2000)
7. Polylib User's Manual. The Polylib Team (2002)
8. Höfting, F., Wanke, E.: Polynomial-Time Analysis of Toroidal Periodic Graphs. Journal of Algorithms 34, 14–39 (2000)
9. Bondhugula, U.K.R.: Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model. Dissertation. The Ohio State University (2010)
10. Quinton, P., Rajopadhye, S., Risset, T.: On Manipulating ℤ–Polyhedra. Publication interne n.1016, Institut de Recherche en Informatique et Systèmes Aléatoires (1996)
11. Cohen, E., Megiddo, N.: Recognizing Properties of Periodic Graphs. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 4, 135–146 (1991)
12. Bastoul, C.: Code Generation in the Polyhedral Model Is Easier Than You Think. In: PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques, pp. 7–16 (2004)
13. NASA Advanced Supercomputing Division, http://www.nas.nasa.gov
14. https://github.com/davewathaverford/the-omega-project

Polish Computational Research Space for International Scientific Collaborations

Jacek Kitowski[1,2], Michał Turała[2], Kazimierz Wiatr[2], Łukasz Dutka[2], Marian Bubak[1,2], Tomasz Szepieniec[2], Marcin Radecki[2], Mariusz Sterzel[2], Zofia Mosurska[2], Robert Pająk[2], Renata Słota[1], Krzysztof Kurowski[3], Bartek Palak[3], Bartłomiej Balcerek[4], Piotr Bała[5], Maciej Filocha[5], and Rafał Tylman[6]

[1] AGH University, Department of Computer Science, al. Mickiewicza 30, 30-059, Krakow, Poland
[2] AGH University, ACC Cyfronet AGH, ul. Nawojki 11, 30-950, Krakow, Poland
[3] Poznan Supercomputing and Networking Center, ul. Noskowskiego 10, 61-704 Poznan, Poland
[4] Wroclaw Centre for Networking and Supercomputing, Wybrzeze Wyspianskiego 27, 50-370 Wroclaw, Poland
[5] Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw, ul. Pawinskiego 5a, 02-106 Warsaw, Poland
[6] Gdansk University of Technology, Academic Computer Centre in Gdansk – TASK, ul. G. Narutowicza 11/12, 80-233 Gdansk, Poland

Abstract. The Polish Grid Initiative commenced in 2009 as part of the PL-Grid project funded under the framework of the Innovative Economy Operational Programme [1]. The main purpose of this Project is to provide the Polish scientific community with an IT basic platform making use of Grid computer clusters, enabling e-science research in various fields. The Project is establishing a country-wide computing platform, which supports scientific research through integration of experimental data and results of advanced computer simulations carried out by geographically-dispersed teams. The solutions applied in setting up this e-infrastructure will ensure integration with other, similar platforms around the world. In the paper some basic facts concerning the Project history are given, PL-Grid goals are described and several examples of innovative grid services and software as well as support procedures developed to-date are presented.

Keywords: grid, computing clusters, IT platform, e-science, NGI, Virtual Organization.

1 Introduction

At the end of XX and the beginning of XXI century a number of "grid" projects developed in USA (Globus, Condor, GriPhiN, PPDG, iVDGL) and in Europe (DataGrid, NorduGrid, CrossGrid, DataTAG, GridPP, etc.), largely motivated by the physics communities, which desperately looked for new solutions to the globalization of computing, to solve very demanding requirements of LHC experiments [2]. The derivative of those projects included Open Science Grid and TeraGrid in USA, three generations of EGEE projects and two of BalticGrid in Europe, and ApGrid and Grid@Asia in Asia; in addition many application projects have been launched and executed. In many countries national Grid projects were developing.

Poland, especially computing centers of Krakow (Academic Computer Center Cyfronet AGH), Poznan (Poznan Supercomputing and Networking Center, PSNC) and Warsaw (Interdisciplinary Centre for Mathematical and Computational Modeling, ICM Warsaw) participated in number of European Grid projects – this way an experience and close links to the Grid communities have been established. The CrossGrid project [3], coordinated by ACC Cyfronet AGH Krakow, demonstrated the use of Grid for interactive applications; the GridLab project [4], coordinated by PSNC Poznan, developed Grid Application Toolkit to simplify access to Grid resources and services. In 2007 a PL-Grid Consortium has been formed, which apart of the three institutions listed above included computing centers of Gdansk (Academic Computer Centre, CI TASK) and Wroclaw (Wroclaw Centre for Networking and Supercomputing, WCNS), with a goal "...to provide the Polish scientific community with an IT platform based on Grid computer clusters, enabling e-science research in various fields"; it was strongly felt that such infrastructure should be compatible with existing European and worldwide Grid frameworks.

2 Goals and Platform Overview

Our activity aims at significantly extending the amount of computing resources provided to the Polish scientific community (by approximately 215 TFlops of computing power and 2500 TB of storage capacity) and constructing a Grid system that will facilitate effective and innovative use of the available resources. In this aspect, we have been focusing on exploitation of computational and storage facilities by virtual organization paradigm and on implementing a comprehensive grid resource management suite, comprising many individual services.

According to the overall layered architecture of the proposed platform (see Fig. 1) the scientific novelty in the scope of Grid services and Grid Application Programming Interfaces has been achieved with special interest on efficient resources allocation, experimental workbench, novel grid middleware and tools, as further outlined in section 3.

The software development is closely related to achievements in the fields of hardware resources and operational issues, supported by careful coordination of scientific groups performing the research. Extension of the hardware resources enabled our computer centres to be highly located at the June 2011 TOP500 list (with ranks: ACC Cyfronet AGH: 81, CI TASK: 163 and WCNS: 194) with the aggregated ca. 2000 TB of storage, thus offering substantial amount of resources to the users already. To make the usage efficient the Operations Centre coordinates every-day infrastructure handling, while hardware, software and

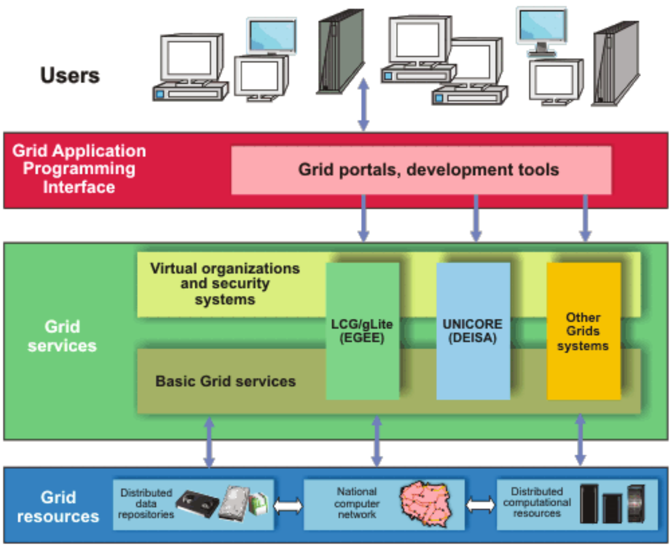

Users
Grid Application Programming Interface
Grid portals, development tools
Grid services
Virtual organizations and security systems
Basic Grid services
LCG/gLite (EGEE)
UNICORE (DEISA)
Other Grids systems
Grid resources
Distributed data repositories
National computer network
Distributed computational resources

Fig. 1. Structure of the PL-Grid Computing Platform

tools development, security aspects, training and user support are performed by cross-organizational groups participating in platform progress.

3 Innovation in International Context

The established Grid system must enable efficient and innovative use of the newly built hardware infrastructure. This goal has been achieved by elaboration of rules for efficient infrastructure usage by virtual organizations and deployment of a set of tools and Grid management services at various levels of the infrastructure.

Below a survey of the example software and services supporting grid users and administrators is presented.

3.1 Efficient Resource Allocation

Allocation of resources available in the e-infrastructure is a vital process in each Grid infrastructure. At the same time, this process is directly related with the way how customers' needs are fulfilled. The process itself is pretty complex as users and multiple providers are involved in it. Therefore, PL-Grid implemented the process of resources allocation as well as a collaborative platform that supports both users and providers with dealing with complexity of this process. The result of resource allocation process is a document called Service Level Agreement (SLA), which is a basis for grid services configuration, monitoring and accounting. This makes an SLA the core concept in the whole PL-Grid operation framework.

This principle and many details of a design follow the best practices related to Service Level Managements (SLM), which is a relevant part of ITIL specification [5] and ISO2000 standard [6]. The design was verified in the collaboration with some National Grid Initiatives (NGIs) in the frame of EGI [7] and with gSLM project [8]. Solutions of this kind are still in a research phase and PL-Grid is the first NGI that provides such a solution into production environment.

PL-Grid SLAs are mainly aimed at proper resource allocation for users and at the related issue of usage capacity planning. Additionally, SLAs between a service provider and an infrastructure operator define the minimal level of service parameters. This type of SLAs is used both as en entry agreement when joining the infrastructure, and as a level of service which is guaranteed even if the same quality parameters are not included in the user-provider SLA.

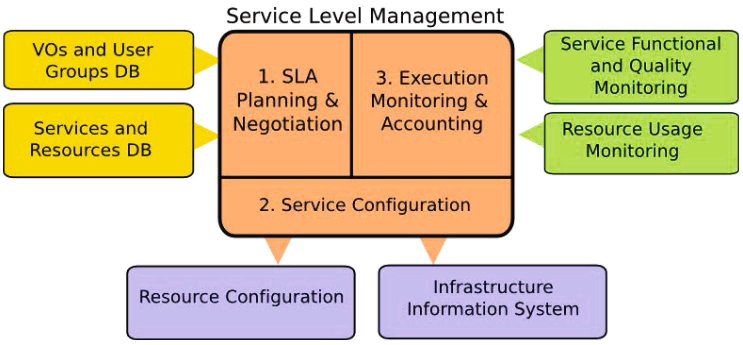

Service Level Management
VOs and User Groups DB
Services and Resources DB
1. SLA Planning & Negotiation
3. Execution Monitoring & Accounting
2. Service Configuration
Service Functional and Quality Monitoring
Resource Usage Monitoring
Resource Configuration
Infrastructure Information System

Fig. 2. Operational Architecture Grid

SLA-aware Architecture. Our proposal of operational architecture, as seen in Fig. 2, is designed to face SLM principles. Many components, including monitoring and operational data-bases, are inherited from the existing EGEE/EGI operational architecture, which – to our knowledge – is the most mature operational architecture deployed in contemporary production grids. The novelty of PL-Grid architecture lays in the integrated SLM component. The process of providing reliable resources to users according to SLAs is recognized as the most important service of the infrastructure. Therefore, a related component, namely the SLM, was placed in the center of the architecture. This glues together all the other components, which are operational data producers and consumers.

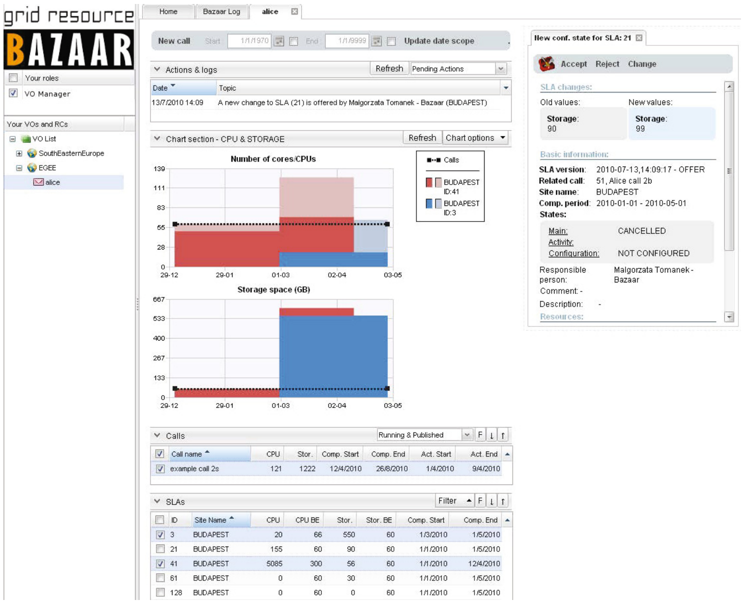
The components responsible for SLM are as follows:

– **SLA Planning & Negotiation.** It enables specification of user requirements and negotiation of the SLA properties with resource providers. They are specified in the form of one- or multi–providers SLA. Both users and providers have a clear view on how the resources usage is planned, and can take appropriate actions. This component uses information about user accounts, groups and roles. The information about available services and people responsible for SLA signing is taken from a related repository.

– **Service Configuration.** Signed SLAs as well as SLA monitoring results influence the execution process. Firstly, resource configuration can be done according to signed SLAs. In case of fine-grained SLAs, some automation needs to be deployed. Secondly, information published in the Grid information services should be controlled (filtered) by the SLA Execution component. This might prevent from enabling support for a VO on resources without informing related managers, and gives a possibility to filter out resources violating an SLA.
– **SLA Execution Monitoring & Accounting.** In this component, raw monitoring data related to functionality monitoring, quality monitoring (e.g. reliability and availability metrics) and usage monitoring (site accounting data) are analyzed in the light of SLAs that were signed. Additionally, information about maintenance work is included. The results of this analysis enable to monitor how an SLA is executed and give the data about possible violations of SLAs. That data is presented to users and resource providers. Accounting can be implemented on the base of a full view on SLA execution, according to the infrastructure business model.

The presented architecture provides full support for SLM in computational infrastructure. Please note that most of the components built for grid infrastructures can be reused. However, enabling a component for SLM, introduces qualitative change to the operation architecture.

Bazaar – Collaboration Platform. To enable presented solution into production infrastructure, we needed a collaboration platform that would support SLM



Date	Topic
13/7/2010 14:09	A new change to SLA (21) is offered by Malgorzata Tomanek - Bazaar (BUDAPEST)

Calls

	Call name	CPU	Stor.	Comp. Start	Comp. End	Act. Start	Act. End
✓	example call 2s	121	1222	12/4/2010	26/8/2010	1/4/2010	9/4/2010

SLAs

	ID	Site Name	CPU	CPU BE	Stor.	Stor. BE	Comp. Start	Comp. End
✓	3	BUDAPEST	20	66	550	60	1/3/2010	1/5/2010
	21	BUDAPEST	155	60	90	60	1/1/2010	1/5/2010
✓	41	BUDAPEST	5085	300	56	60	1/1/2010	12/4/2010
	61	BUDAPEST	0	60	30	60	1/1/2010	1/5/2010
	128	BUDAPEST	0	60	0	60	1/1/2010	1/5/2010

New conf. state for SLA: 21

SLA changes:

Old values:	New values:
Storage: 90	**Storage:** 99

Basic information:

SLA version: 2010-07-13,14:09:17 - OFFER
Related call: 51, Alice call 2b
Site name: BUDAPEST
Comp. period: 2010-01-01 - 2010-05-01
States:
Main: CANCELLED
Activity:
Configuration: NOT CONFIGURED
Responsible person: Malgorzata Tomanek - Bazaar
Comment: -
Description: -
Resources:

Fig. 3. Graphical user interface of Grid Resource Bazaar

process. The work started with creating an SLA negotiation platform named Grid Resource Bazaar [9]. The main goal of the platform is to cover complex process of communication between remote SLA parties. There is a lot of communication to define in the process of negotiating and signing an SLA. Our experience shows that a smooth and well-defined communication process is a key factor in successful SLM adoption. A collaborative tool, which Bazaar is, simplifies the communication by implementing communication patterns crafted according to SLM model. Bazaar graphical interface was designed for easy SLA management with support of complexity management. As we can see on Fig. 3, the portal is organized in a form of a dashboard with views of resource allocation for a given user or VO. An equivalent dashboard is available for a resource provider as well. The resources are visualized on a chart and SLAs are listed beneath. The user can manage SLA proposals and examine the influence of a new SLA on the resources. Similar elements can be found in the infrastructure provider's view in Bazaar.

3.2 Experimental Workbench

GridSpace2 Experiment Workbench [10] is a novel virtual laboratory framework, which enables researchers conducting virtual experiments on Grid-based resources and other HPC infrastructures.

Virtual laboratory is constructed based on a distributed layered architecture. Top layer constitutes a web portal, which is the main access point to all the mechanisms of the laboratory from each workstation equipped with a Web browser. The portal covers an Experiment Workbench layer, within which there are various tools used to create experiments, collaboration, communication and resource sharing. The following is an executive layer, which is responsible for interpreting the code introduced via the portal and its implementation in the context of user's individual experimental spaces on a special access machine called the experiments host.

3.3 Overview of Selected Tools and Middleware

Migrating Desktop. The Migrating Desktop (MD) platform [11] is a powerful and flexible graphical user interface to grid resources that hides the complexity of the grid middleware and supports advanced methods of applications results' visualization. The platform allows the user to run applications and tools, manage data files and store personal settings independently of the location or the terminal type; it gives a transparent user work environment and easy access to resources and network file systems independently of the system version and hardware. MD offers: scalability and portability, a set of tools, a single sign-on mechanism and support for multiple grid infrastructures. The key feature of MD is flexible personalized working environment.

Vine Toolkit. Vine Toolkit [12] is a set of the portal tools, which allow to create web applications enabling remote running and controlling the computer simulations. It also allows to integrate these applications with HPC resources, grid services and various existing large-scale computing infrastructures managed by grid middleware.

Vine, together with a set of built-in modular components, is an excellent solution to establish web gateways for advanced scientific and engineering applications with grid-enabled resources in the backend. Moreover, the heterogeneity of grid services and HPC resources can be unified thanks to Vine APIs and built-in capabilities for remote job submission, monitoring and control as well as data and workflow management, security and user management. Thus, integrating existing Vine modules and adding application-specific extensions it is possible to create a sophisticated Science Gateways to support collaborative scientific research.

FiVO/QStorMan Toolkit. FiVO/QStorMan [13] is a toolkit developed to provide requested quality of access to storage resources for data-intensive grid applications. QStorMan constitutes a part of the Framework for Intelligent Virtual Organization (FiVO).

The driving goal of FiVO/QStorMan toolkit is to provide access to user data at a certain level of quality (e.g. transfer, availability, security). To achieve the goal the system uses the Virtual Organizations feature of grid systems. The user of the system is able to: define non-functional requirements for accessing his data explicitly, gain a certain level of data access by selecting a grid VO and improve his grid computations speed.

Novel Grid Middleware (QCG). The QosCosGrid (QCG) [14] middleware is an integrated e-infrastructure offering advanced job and resource management capabilities to deliver to end-users supercomputer-like performance and structure.

By connecting many computing clusters together, QosCosGrid offers easy-to-use mapping, execution and monitoring capabilities for variety of applications, such as parameter sweep, workflows, MPI or hybrid MPI-OpenMP. Thanks to QosCosGrid, large-scale and complex computing models written in Fortran, C, C++ or Java can be automatically distributed over a network of computing resources with guaranteed Quality of Service. Consequently, the applications can be run at given periods of time, their execution time and waiting times can be reduced, and thus bigger problem instances can be considered.

4 Users, Software Packages and Their Applications

Attractive computational resources and seamless access to them, described in previous sections, are not the only important factors attracting users to grid platform. Another two, which are crucial for utilising the existing resources, are availability of a set of software packages as well as effective (real time) support and training for existing and new users. Below we present a short overview of available software packages, Helpdesk System and training activities. The final paragraph sums up users' applications studied on the infrastructure.

4.1 Software Packages

To fully support various scientific communities, each infrastructure has to provide variety of scientific software packages for their users. Until now, the most commonly used scientific software packages have been available only at High Performance Computers in several of Polish computing centers. Now, with the high increase of grid users, their expectations have raised and number of available software packages had to be extended. At present, PL-Grid users can choose one of two major middlewares to run their computations through. Majority of newly ported and installed software is provided under gLite middleware. To get easy access to scientific packages, a tool "Modules" [15] has been introduced on all sites supporting PL-Grid. Access to specific software package is realized via *"module add <program_module>"* command. Deployment of "Modules" has had two main benefits. On the one hand, for system administrators, it allows keeping the library of the software packages in administrator's favourite location, while for users – it provides unified, easy and comfortable way of accessing required program application. Recently the following list of commercial and freely available scientific packages is available: ADF, ANSYS FLUENT, AMBER, Abinit, Blender, CFOUR, CPMD, Dalton, GAMESS, Gaussian, Gromacs, Mathematica, Maple, Matlab, NAMD, NWCHEM, SIESTA, Quantum Espresso and Turbomole. This is of course a subject to change as new requests are constantly appearing. To ensure correctness of software functioning, an automated monitoring has been deployed, based on NAGIOS tool. Software tests are executed several times a day in each of the centers supporting PL-Grid infrastructure. Resulting pages of NAGIOS tests are available for any user with a valid certificate.

4.2 Helpdesk System and Training Courses

Constantly growing number of scientists performing computations in PL-Grid infrastructure requires professional help and support in solving various problems concerning access and computations on the infrastructure. For users' satisfaction, system Helpdesk has been introduced – a novel support system, which involves technical services and organization of experts. All the (incidental) issues with the infrastructure, problems, even requests (concerning new software for example) can be reported via `helpdesk@plgrid.pl` e-mail address. The registered users can also benefit from an on-line tool operated via a browser. We have found Helpdesk system highly valuable tool for users; currently a knowledge base is built based on analysis of past tickets. In addition to knowledge base (in form of FAQs), several training courses have been organized to introduce the infrastructure for majority of users. Special attention has been devoted to new users of the PL-Grid platform. For their satisfaction we provide training explaining both basic and advanced functionality of grid computing. Those come in a form of both – traditionally organized courses as well as the on-line ones available via Blackboard e-Learning system. Both ways of training increase overall users' satisfaction and benefits from grid computing.

4.3 Applications

Currently there are over 700 registered users with about two third being the active ones. They come from several of science domains including both 'grid-traditional' ones like physics – especially HEP, chemistry, astronomy, biology and medicine and newly adopted like material science, nanotechnology or even linguistics. Number of the scientific topics, for which computations are performed, reaches almost one thousand (over a period of a year). Biology, chemistry and High Energy Physics are the most active ones. The most time consuming computations in biology concerned anti-fungus antibiotics (165 CPU years over a period of a month). Chemical computations concern mainly electronic structure of molecules while HEP jobs analyse(d) data coming out of ATLAS experiment. All three above mentioned science domains are responsible for over 90% of the resource utilisation in PL-Grid infrastructure.

5 PL-Grid Platform as Part of European Infrastructure

After first few years of studies of Grid technology and applications, IT and Science communities, as well as the EU IST directorate, realized that one cannot base the future of European computing infrastructure on the projects – as they have limited life-time, and that a more stable organization is needed. In Europe a special project, the European Grid Initiative Design Study (EGI_DS), has been launched, with a goal to develop conceptual and logistical frameworks for a permanent organization, which would oversee the operation and development of the pan-European Grid infrastructure. About 40 European National Grid Initiatives (NGI), 3 European Research International Organizations (EIRO) and several Grid organizations from outside of Europe supported this concept and in March 2010 a European Grid Initiative [7] consortium has been established, with its location in Amsterdam. In this context PL-Grid platform is one of NGI organizations.

Today EGI.eu consortium involves 33 European NGI partners and 2 EIROs. One of the main goals of EGI.eu is to bring European distributed computing initiatives into an integrated e-Infrastructure that is able to seamlessly peer with equivalent e-Infrastructures around the world. EGI.eu collaborates with international policy bodies (OGF, e-IRG, EUGridPMA), several EU projects (e.g. EMI, IGE, SAGA, SIENA, GISELA, StratusLab), and non-European organizations (e.g. Asia-Pacific Grid, ROC Latin America).

As today EGI.eu integrates and supervises more than 200 thousand logical CPU's (cores) and about 100 PB of disk and 80 PB of tape storage; more than 13 thousand users are organized in more than 180 VOs, out of which about 30 are active – about 1 million jobs are performed daily, mainly related to particle physics but also from archeology, astronomy, astrophysics, civil protection, computational chemistry, earth sciences, finance, fusion, geophysics, life sciences, multimedia, material sciences, etc.

6 Conclusions

Several goals are being achieved during the platform development. These are computational and storage offerings to the international scientific community, novel middleware solutions and tools enabling easy usage of the resources and ability to achieve original scientific results by the users. In the future the platform is planned to be extended toward fulfilling requirements from specific groups of scientists.

Acknowledgements. This work was made possible thanks to the PL-Grid project: contract number: POIG.02.03.00-00-007/08-00, website: www.plgrid.pl. The Project is co-funded by the European Regional Development Fund as part of the Innovative Economy program. Acknowledgements are due to the members of the PL-Grid consortium for assistance during preparation of the paper.

References

1. Innovative Economy Operational Programme, http://www.poig.gov.pl/english/
2. MONARC, http://monarc.web.cern.ch/MONARC/
3. The CrossGrid project, http://www.cyf-kr.edu.pl/crossgrid/
4. The GridLab project, http://www.gridlab.org/about.html
5. Rudd, C.: Service Design. Office of Government Commerce (ITIL). The Stationery Office Ltd., London (2007)
6. International Organization of Standardization (ISO), Information technology - Service Management - Part 1: Specification (ISO/IEC 20000-1:2005)
7. EGI, http://www.egi.eu/
8. Szepieniec, T., Kocot, J., Schaaf, T., Appleton, O., Heikkurinen, M., Belloum, A.S.Z., Serrat-Fernández, J., Metzker, M.: On Importance of Service Level Management in Grids. In: Alexander, M., et al. (eds.) Euro-Par 2011 Workshops, Part II. LNCS, vol. 7156, pp. 64–75. Springer, Heidelberg (2012)
9. Szepieniec, T., Tomanek, M., Twaróg, T.: Grid Resource Bazaar: Efficient SLA Management. In: Cracow Grid Workshop 2009 Proceedings, pp. 314–319. ACK Cyfronet AGH, Kraków (2010)
10. GridSpace2 Experiment Workbench, https://gs2.cyfronet.pl/
11. Migrating Desktop Platform, http://desktop.psnc.pl/
12. Vine Toolkit, http://vinetoolkit.org/
13. FiVO, http://fivo.cyf-kr.edu.pl/trac/fivo/wiki/FIVO/
14. QosCosGrid Middleware, http://www.qoscosgrid.org/
15. Modules, http://modules.sourceforge.net

Request Distribution Toolkit for Virtual Resources Allocation

Jan Kwiatkowski and Mariusz Fras

Institute of Informatics,
Wroclaw University of Technology,
Wybrzeze Wyspianskiego 27, 50-370 Wroclaw, Poland
{jan.kwiatkowski,mariusz.fras}@pwr.wroc.pl

Abstract. The Service Oriented Architecture (SOA) concept facilitates building flexible services that can be deployed in distributed environment, and executed on different hardware and software platforms. On the other hand SOA paradigm rises many challenges in the area of Quality of Service and resources utilization. In the paper Resources Distribution Manager (RDM) that manages resources and service delivery in order to satisfy user requirements and service provider's needs is presented. Requests for services are distributed to selected virtualized instances of services or optionally new instances are created for handling the requests. The allocation decisions are based on the knowledge about the communication resources and the current load of computational resources which are dynamically monitored by special RDM module. The decision on the resources allocation is then compared with its utilization during service execution and causes changes in allocation strategy.

Keywords: Service Oriented Architecture, request distribution, virtualization management.

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 complex services. Adopting the concept of services SOA takes IT to another level, one that's more suited for interoperability and heterogeneous environments. A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services.

The information systems designed with SOA paradigm are working both in local and wide area networks, particularly built with Internet resources. For SOA service client, there is no need to carry about the localization and execution meanings of the service it want to use. However, the services are well described in the system and their localizations are known. It can dynamically change, however this process is not frequent, and usually the set of localizations of given service is constant for some period of time [5].

The very important issue for contemporary computer network service providers is quality of network services delivery. Users want to complete requested service without inconvenience and as fast as possible. Thus, the new software architectures are driven by quality requirements of designed solutions. The computer network systems designed with SOA paradigm enable flexible delivery of new services as new needs come into being. Service sharing, reusability, service composition, standardized service description and discovery are only some features that make SOA solutions so usable in present business. In SOA-based systems such a quality attributes as interoperability, extensibility and modifiability are secured by known solutions based on universally accepted standards. For such quality attributes as reliability, availability or scalability there are known solutions which, however, need some work on improving its usefulness. The most work is needed for two attributes security and performance [8].

On the other hand the Service Oriented Architecture and virtualization come closer to each other, then the need to combine them in an efficient way becomes one of the key challenges for designers of systems based on SOA paradigm. Protocols and languages describing the base framework for services have been already well described, standardise and established. Similarly virtualization which is often pointed as an SOA enabling technology has been quite matured. Moreover virtualization improves the overall effectiveness of the use of available resources. The number of existing solutions is not small and as reported in [12] it is hard to find one which outstands all the others.

In the paper an architecture of the Request Distribution Manager (RDM) that supports control of values of non-functional parameters of delivered services and distribution of service requests guaranteeing services quality, especially efficient allocating communication and computational resources to services is presented. The proposed architecture can be applied in the wide range of SOA-based systems. The paper is organized as follows. Section 2 briefly presents similar tools as RDM and pointed the main differences between them and RDM. Detailed description of the RDM is presented in section 3. It covers general structure of the RDM with short presentation of each module and its role. In section 4 results of some experiments performed using RDM are presented. Finally, section 5 outlines the work and discusses the further works.

2 Design Issues and Related Works

In distributed environment guaranteeing quality of delivered services is usually explored in the context of Web services. In [10] the general concept of guaranteeing SLA for Web services is presented. The need for monitoring of service processing at server side and for measurement of an concrete Web Service at client side is formulated. The work [13] presents a platform for composition of Web services so that quality criteria are satisfied. Quality of service delivery is guaranteed by service selection with use of utility functions over QoS attributes such as price, duration, reputation and availability. It proposes two approaches, the first one is an individual service selection (without taking into account the other services involved in the complex service), and the second one is service selection by global planning (taking into account whole complex service). In general, proposed solutions assume rather static values of parameters of services and don't consider ad hoc change of set of available services. On the other hand the method that guaranties quality for services which values of parameters can change very dynamically is proposed in [3]. The base of above approach is developing and building models of services with use of fuzzy-neural networks, similarly as in [1].

Apart from commercial software offered by the virtualization solutions vendors such as Microsofts System Center Virtual Machine Manager 2008 R2 or VMware's vSphere there is a number of open solutions for virtualization management which will be described shortly here. These are Nimbus, OpenNebula, OpenStack and Eucalyptus. Open solutions are in most cases hypervisor agnostic and offer wide range of features to ease the deployment and management of private, public or hybrid clouds. Their architecture is modular which gives high customization opportunities. According to [11] each of those open solutions has slightly different target thus provides different capabilities. That makes them more suitable either for private, public or scientific purposes.

Nimbus project has a very clear area of interest which is the scientific community [6]. It concentrates on capacity allocation and capacity overflow [11]. The project places a strong emphasis on the research and future cloud technology development. OpenNebula with centralized management is the solution for private clouds [11]. The project was started in 2005 and tries to deliver an efficient and scalable management of virtual machines on large-scale distributed infrastructures [7]. The next interesting proposition is the OpenStack software which is the result of joint effort of NASA and RackSpace with number of other parties involved in the cloud development. Currently OpenStack is often pointed as the best open solution to create public cloud [9].

2.1 The Differences between RDM and Other Approaches

The largest difference between RDM and aforementioned solutions is coming from another targets standing behind the projects. While most of other solutions are strictly devoted to manage the infrastructure, RDM is devoted to properly dispatch the requests placing the virtualization management on the second place. Nonetheless one can point a number of similarities starting from common modular architecture with possibilities to customize the software easily. Furthermore just like other solutions *libvirt* is used to overcome the problem with communication with various hypervisors.

The role of RDM as a dispatcher means that some of the functionalities are redundant. Under such situation one may put offering Amazon compatible API, billing integration, number of control panels and so on. On the other hand the functionality is extended to understand the SOAP messages, identify which services are capable of performing them and finally running those services and dispatching the requests. Analogical software is found as an addition on top of e.g. OpenNebula and offers service orchestration and deployment (SVMSched) or service management as a whole (Claudia).

3 Request Distribution Manager

Delivery of software services (computational, information, etc.) in the traditional form is characterized by the fact that applications are not open enough to follow rapidly changing needs of business. This involves functional requirements as well as assuring changing needs of quality of processing. The architecture of Request Distribution Manager (RDM) resolves the problem with traditional software lack of flexibility. RDM is built taking into account two needs: proper utilization of processing environment and support for services in accordance with Service Oriented Architecture paradigm.

3.1 The General Architecture of Request Distribution Manager

The architecture of the RDM 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. The main components are:

– RDM-Broker - which handles user requests and distribute them to proper instances of virtualized computational resources. Decision making is based on specified criteria of request distribution, in turn, based on non-functional requirements. It also performs internal requests to coordinate the operation of the system components as well as obtain some necessary information.
– RDM-Facade - which separates the rest of components, supports communication with them, 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).
– RDM-Controller - which 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 (capsules) independently, taking into account computational resource utilization and performs decision to start/stop another instance of service.
– RDM-Virtualizer - which offers the access to hypervisor commands. Uses *libvirt* to execute commands what gives the project independence from particular hypervisor.
– RDM-Monitor - which collects information about particular physical servers as well as virtual instances running.

The instances of given atomic service are functionally the same and differ only in the values of non-functional parameters such as completion time of execution or availability.

RDM modules interact using two interfaces. Internal communication is XML-RPC based, for components behind the Facade, and uses SOAP messages for

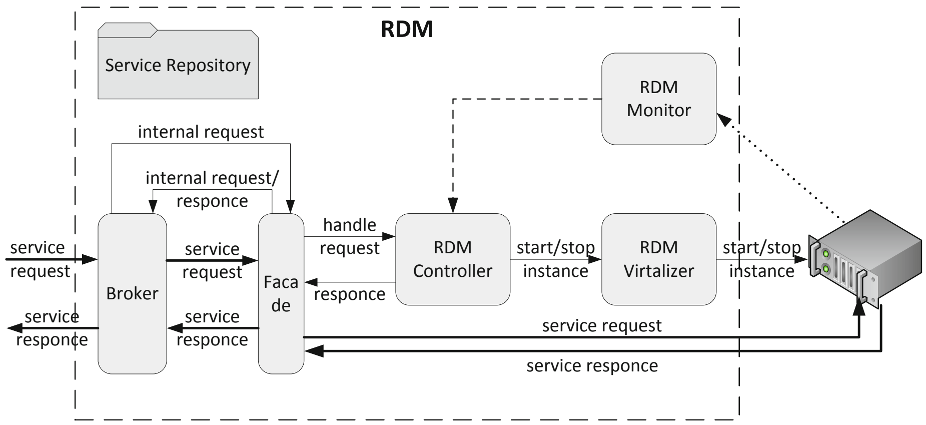


Fig. 1. Request Distribution Manager Architecture

communication between RDM-Broker and RDM-Facade. This allows flexibly manage distributed computational resources as well. Interaction with external components is based on RDM-Broker services with SOAP messages.

RDM delivers for clients access to network services hidden behind it. Clients see network services at Broker localization and don't know 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. However broker distinguishes individual processing resources, simultaneously coordinates activities with RDM-Controller, and passes requests on. Each and every request is redirected to proper 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.

3.2 Request Distribution Manager Components

The RDM-Broker acts as service delivery component, while in fact distributes requests for services to known service processing resources. It maintains repository of all known services and components that support them (Fig. 2), i.e. a set of atomic services $AS = \{as(1), \ldots, as(j), \ldots, as(J)\}$ available for clients, a set of service instances $IS = \{IS_1, \ldots, IS_j, \ldots, IS_J\} = \{is(1,1), \ldots, is(j,1), is(j,2), \ldots, is(j,m_j), \ldots, is(j,M_j), \ldots, is(j,M_J)\}$, where $is(j,m)$ is m-th instance of j-th atomic service (localized at given address (server)), and $m_j \in \{1, M_j\}$, where M_j is a number of existing instances of $as(j)$. The broker controls and maintains also information about complex services CS composed from atomic ones. It also recognizes and supports multiple execution systems (set of service providers SP). These two issues are not exploited here.

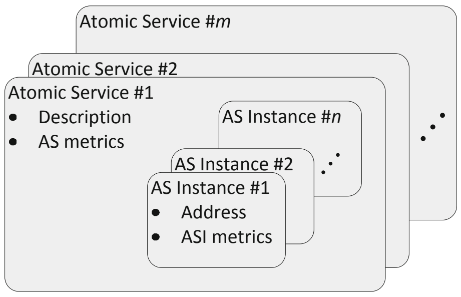


Fig. 2. The general structure of Broker Repository

The RDM-Broker collects data about instances of atomic services $is(j, m)$ based on measured or calculated values of non-functional parameters $\psi\ (is(j, m)) = \{\psi^1_{j,m}, \ldots, \psi^k_{j,m}, \ldots, \psi^K_{j,m}\}$, where $\psi^k_{j,m}$ is k-th non-functional parameter of m-th instance of j-th service. To make allocation decision which takes into account dynamic parameters (i.e. varying in time), values of these parameters must be obtained using any of estimation methods. The most important dynamic parameters, completion time of service execution and data (request/response) transfer time, RDM-Broker estimates with use of adaptive models of execution systems and communication links, built as a fuzzy-neural controllers. The input of the communication link model is the vector of measured values of communication link parameters, derived from the network monitoring unit. The input of the execution system model is the vector of system state parameters derived from the Facade. The more detailed description of used adaptive models of service instances and communication links is presented in [3].

The Facade supports communication between Broker and components inside the execution system. The Facade collects and delivers some essential information (e.g. load of the system) 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.

Virtualization management is based on open source *libvirt* toolkit. It offers the virtualization API supporting number of the most popular hypervisors. From the point of view of this paper it is of a less important how technically the management is performed. It is more important to note what are the capabilities of the management and how it is understood here.

3.3 Request Handling and Resource Management

Process of request handling starts with SOAP message coming from the client of the RDM (it can be end user or other component of SOA-based service delivery system). As mentioned before, the primary address of requested service is the Broker, which in fact hides all the infrastructure of the execution environment.

Next, Broker performs resource allocation decision. The decision making is based on the actual values of parameters of service instances, distribution strategy and optionally non-functional requirements SLA_{nf} formulated in the service request. If we consider fulfilling SLA_{nf} for each request separately, the problem of service request distribution system can be expressed with criterion Q on function h of parameters $\psi(is(j,m))$: $is(j,m^*) \leftarrow \arg\min Q(h(\psi^1_{j,m},\ldots,\psi^k_{j,m},\ldots,\psi^K_{j,m}))$ i.e. the task to select such instance (the optimal one) that criterion Q is minimized. A simple example of distribution criterion is $is(j,m^*) \leftarrow \arg\min (tt_{j,m}+te_{j,m})$, where $tt_{j,m}$ is transfer time for instance m of atomic service $as(j)$, and $te_{j,m}$ is completion time of execution for instance m of atomic service $as(j)$. It is simply single request response time minimization. For more general cases the problem is formulated more complex.

The decision of the choice of proper service instance $is(j,m^*)$ is performed on the basis of estimated values of service instance non-functional parameters. Dynamic parameters (such as actual completion time of execution, which may depend on system load) are derived from adaptive models of service instances and actual execution system states coming from the RDM-Facade.

Next, the readdressed request is passed to the Facade, which acts as the gateway to the system and hides all of the heavy lifting from outside world (Fig. 3 and Fig. 4). The Facade delivers the values of a number of parameters online monitored by RDM-Monitor (step A and B in figure 3). The further processing will be described for different scenarios of request handling. To assure quality of service delivery several scenarios may be realized.

In the first scenario: there is a running service (capsule) which satisfies requirements of request processing (Fig. 3). The instance is chosen using online actualized Service Repository. Then, the request is just passed to the Facade and further to chosen service instance (capsule).

The second scenario is when no running service instance can satisfy requests (all are overloaded) but there is an image of the service which can be instantiated and the new instance can perform the request (Fig. 4). In such a case the Broker sends "create new instance" internal request. It is passed to the RDM-Controller which selects localization of the new capsule using matching strategy. The RDM-Controler sends the request to RDM-Virtualizer and next the address of the new instance ID returned to the Facade and new instance is registered in the Broker repository (henceforth visible for it). Finally, the client SOAP request is marked with address of the new service instance and passed to it.

The third scenario is similar to the second one with the difference that creation of the new instance is triggered by RDM-Controller automatically, without Broker request, when value of some monitored parameter will exceed given threshold value.

The last case is the lack of proper service instance and/or possibility to instantiate service image. In this case the SOAP fault message will be returned to the client.

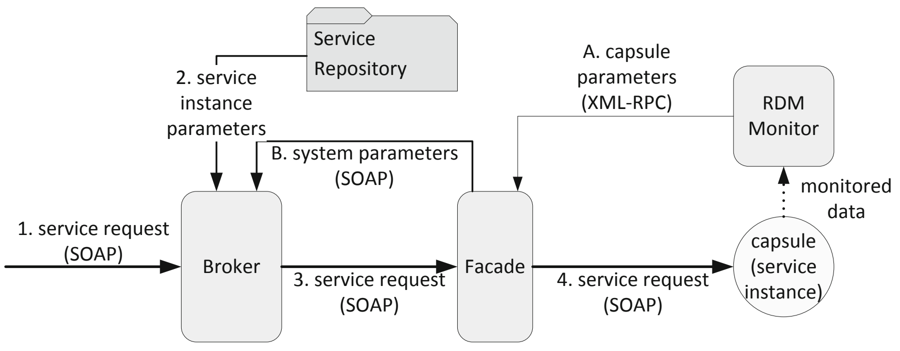

Service Repository
2. service instance parameters
A. capsule parameters (XML-RPC)
RDM Monitor
B. system parameters (SOAP)
monitored data
1. service request (SOAP)
Broker
3. service request (SOAP)
Facade
4. service request (SOAP)
capsule (service instance)

Fig. 3. Client request handling - scenario 1

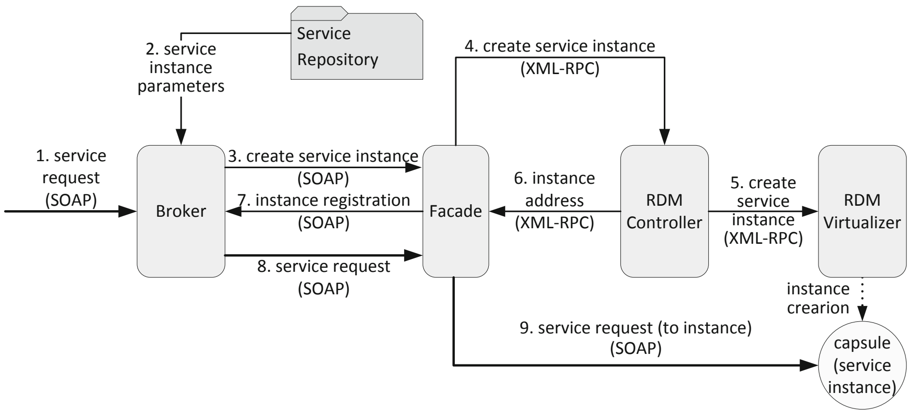

Service Repository
2. service instance parameters
4. create service instance (XML-RPC)
1. service request (SOAP)
Broker
3. create service instance (SOAP)
7. instance registration (SOAP)
8. service request (SOAP)
Facade
6. instance address (XML-RPC)
RDM Controller
5. create service instance (XML-RPC)
RDM Virtualizer
instance crearion
9. service request (to instance) (SOAP)
capsule (service instance)

Fig. 4. Client request handling - scenario 2

4 Preliminary Tests

The RDM has been implemented for Linux system as a set of Python and Java modules. The Broker serves the distribution with use of several algorithms, among the others: Round-Robin (RR) algorithm and Best-Predict (BP) algorithm. The last one accomplishes best effort strategy of requests distribution according to predicted response time estimated with use of fuzzy-neural controller. The preliminary experiments have concerned with testing behaviour of the Broker i.e. request distribution with use of selected distribution algorithms.

The aim of the experiment was to compare the effectiveness of Round-Robin and Best-Predict distribution algorithms. The last one was used to choose the fastest service for every request separately. During experiments six atomic services have been created and installed on four servers at different locations. The services have the following characteristics:

– each service generated different data transfer: 50kB, 100kB, 200kB, 500kB, 1MB, and 2MB,

– each service differed in initial completion time of execution,
– after exceeding the individual threshold, completion time of execution increase exponentially,
– the completion times of execution of instances of the same atomic service on different servers have been different.

There were defined four different complex services (service CS1, CS2, CS3 and CS4) aggregated from the atomic ones. The broker and clients were located at Wroclaw University of Technology campus, in Poland (pwr.wroc.pl), and servers running instances of test services were located in Spain (ait05.us.es), Finland (planetlab3.hiit.fi), Germany (onelab-1.fhi-fokus.de) and Italy (onelab6.iet.unipi.it). Many clients have generated multiple requests for services at the same time during 2 hours. Every half an hour the number of clients were increased i.e. the intense (the number) of service requests was increased in four intervals. The result of the experiment for three intervals (the last one is omitted due to overload of the servers) is shown in figure 5.

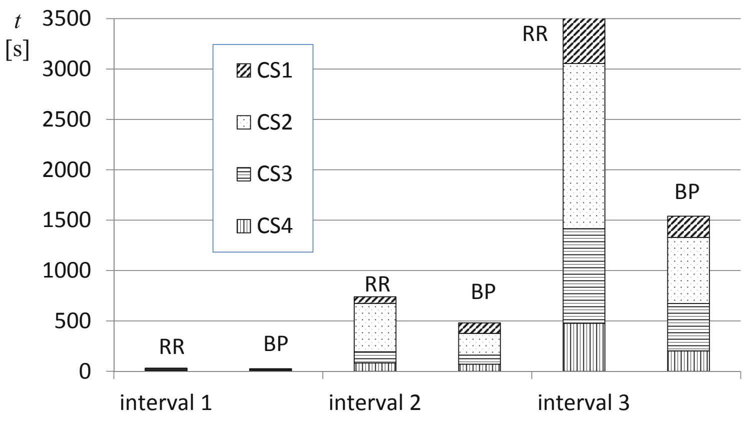


Fig. 5. Service time for five load intervals for RR and BP algorithm

In the figure there are two columns, which show the sum of response times t of four complex services. The response time of one service is 90th percentile of all measured response times in given interval. For each interval there is the column for RR algorithm (left one) and the column for BP algorithm (right one). Received results show that using Broker and proper distribution algorithm, the quality of service delivery (the response time in tested case) can be improved significantly. The Best-Predict algorithm works clearly better than simple reference Round-Robin algorithm.

5 Conclusions and Future Work

The project is in current study. The results of first experiments are really promising. The preliminary experiments concerning service requests distribution, based on the real measurements in Internet network show that proposed approach improve delivering of services in SOA-based systems. However the effect of using proposed methods can be still improved. Prediction of the values of service parameters is crucial for it. Deeper analysis of the results suggests that this issue can be decisive. Apart from improving the estimation of measured parameters the further studies will take into consideration more sophisticated requirements for service quality, as well as development a strategy for automatic creation of the new instance of an atomic service.

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. Borzemski, L., Zatwarnicka, A., Zatwarnicki, K.: Global Distribution of HTTP Requests Using the Fuzzy-Neural Decision-Making Mechanism. In: Nguyen, N.T., Kowalczyk, R., Chen, S.-M. (eds.) ICCCI 2009. LNCS (LNAI), vol. 5796, pp. 752–763. Springer, Heidelberg (2009)
2. Brawn, P.C.: Implementing SOA. Pearson Education (2008)
3. Fras, M., Zatwarnicka, A., Zatwarnicki, K.: Fuzzy-Neural Controller in Service Requests Distribution Broker for SOA-Based Systems. In: Kwiecień, A., Gaj, P., Stera, P. (eds.) CN 2010. CCIS, vol. 79, pp. 121–130. Springer, Heidelberg (2010)
4. 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. LNCS, vol. 6067, pp. 246–255. Springer, Heidelberg (2010)
5. Mabrouk, M.: SOA fundamentals in a nutshell, IBM Corp. (2008)
6. Nimbus Home Page, `http://www.nimbusproject.org/`
7. OpenNebula Home Page, `http://www.opennebula.org//`
8. O'vrien, L., Merson, P., Bass, L.: Quality Attributes for Service-Oriented Architectures. In: Proc. of the Int. Workshop on Systems Development in SOA Environments. IEEE Computer Society, Washington DC (2007)
9. Sumayan, A.: Behind the scenes of IaaS implementations, `http://salsahpc.indiana.edu/`
10. Schmietendorf, A., Dumke, R., Reitz, D.: SLA Management - Challenges in the Context of Web-Service-Based Infrastructures. In: Proc. of the IEEE International Conference on Web Services, San Diego, California (2004)
11. Sempolinski, P., Thain, D.: A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus. In: Proc. of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, pp. 417–426 (2010)
12. Venezia, P., Witkowski, M.: The duel of virtualization platforms. In: Networld (June 2011)
13. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for Web services composition. IEEE Transactions on Software Engineering 30(5), 311–327 (2004)

Vitrall: Web-Based Distributed Visualization System for Creation of Collaborative Working Environments

Piotr Śniegowski, Marek Błażewicz, Grzegorz Grzelachowski,
Tomasz Kuczyński, Krzysztof Kurowski, and Bogdan Ludwiczak

Poznan Supercomputer and Networking Center,
Z. Noskowskiego 12/14, 61-704 Poznań, Poland
{psnieg,marqs,ggrzel,tomasz.kuczynski,
krzysztof.kurowski,bogdanl}@man.poznan.pl

Abstract. Advanced parallel computing solutions using GPU for general purpose processing are becoming more and more popular. Applications like CFD or weather modeling take an extensive speed up using GPU-based clusters. Still, original purpose of graphic processing units – visualization, is not exploited as much as it could be in such powerful processing centres. Main reason of that situation is their fundamental difference from classic desktop configurations: being a structure remote and hidden from the actual viewer. Already existing GPU-based architectures consisting of many processing units may be used for visualization of complex issues from many points of view and in resolutions that are not accessible for single GPUs in real-time. Visualization is a very efficient way of collaboration, especially when collaborators can interact with presented content in a natural way, for example using multi-touch devices. Vitrall embraces these methods by introducing possibility of linkage between modern interfaces and complex visualizations. This paper will begin with summary of several researches that where conducted to establish actual concept of the Vitrall system. Next, proposed architecture of Vitrall will be introduced, following main and secondary usage scenarios. Finally, some of Vitrall's specific configurations will be shown, like real-time stereoscopic visualization.

Keywords: Vitrall, visualization, stereoscopic visualization, collaborative environments, multi-GPU, WebSockets.

1 Introduction

The main goal of the Vitrall system is a real-time remote visualization for scientific communities. To fulfill this purpose, Vitrall is capable of utilizing multi-GPU architectures, working in a distributed environment, communicating with various remote data sources and displaying of the generated content on tiled displays or publishing it on the Web. Flexibility, extensibility and support for thin clients are also very important in course of Vitrall system development.

The name was inspired by appearance of the system configurations containing multiple LCD displays. Frames of the LCD displays resembles lead frames used in a stained glass – Vitrall in Catalan language.

2 Evolution of the Vitrall System

Vitrall project started over a year ago and is since then under continuous development at the Applications Department of PSNC.

2.1 Initial Approach

As a first approach to the problem of remote visualization, a prototype HTTP based visualization server has been implemented. At that time, Vitrall was capable of using only a single OpenGL rendering context. Management of scene parameters (angle of view, etc.) was a responsibility of web browser client, which generated HTTP requests for subsequent frames. All scene parameters were sent to the server via query string of the image request (embedded in the *src* attribute of an image tag). Upon receiving request, server was setting up the scene (loading additional 3D models when necessary) and rendering to an off-screen buffer. When rendering was complete, resulting frame buffer was copied from GPU to the host memory, where JPEG compression was applied (using IJG [7] implementation). Compressed image was sent back to the web browser.

One of the first issues explored during Vitrall project was utilization of modern user interfaces. Windows 7 platform exposes an API to multi-touch enabled devices (e.g. HP TouchSmart tx2), so a dedicated client have been implemented. At the same time another important feature of Vitrall was introduced: rendering content for many clients concurrently.

The setup was successfully demonstrated at Supercomputing 2010 [16]. While content was rendered at PSNC, 4 displaying laptops were located at SC10 venue in New Orleans. The cross-continental connection was established with the use and support from GLIF [19] architecture. To the best of our knowledge, it was the first experiment where both example data and compute demanding applications used innovative bandwidth on-demand (BoD) lambda networking services to establish guaranteed trans-Atlantic connections over multiple domains located in Europe and the United States.

2.2 Second Iteration

The first prototype of Vitrall proved, that real-time remote visualization could be brought to the user over the Web. Nonetheless, number of drawbacks was identified in the described approach. Purpose of the next steps in development of the system was to refine previous solution by increasing its performance and shifting control logic from client-side to the server-side. Although resulting solution was in a fundamental way different from the initial idea, it was still a web-centric approach. For the sake of performance, communication was split into two channels, first responsible for the image serving, and the second one for control (e.g. sending user inputs). Thanks to moving of the control logic to the server-side, achievement of compatibility with web browsers coming from different vendors was possible. In addition, concept of the *Vitrall session* could be introduced. At the moment, the session consists of the following elements: state of the communication with clients, state of the rendered scene and running *working sessions.*

Vitrall session significantly differs from a conventional server-side session in that it is capable of taking actions asynchronously and completely independently of the communication with clients. If configured to do so, a Vitrall session may run and even render with no clients attached.

Vitrall session typically consists of several working sessions. Each working session is an independent thread of execution running at the specified Vitrall server instance. Vitrall working sessions are divided into two main types: management session (usually one per Vitrall session), rendering session (multiple but optional).

Another difference is the fact that multiple clients can connect to a single Vitrall session (through a management session). As a result, state of the scene may be shared, thus creating fundamentals for a collaborative environment. Thanks to specially designed XML-based protocol, each client keeps its own identity and can supply server with its individual capabilities (maximum screen resolution, available sensors – for mobile devices, etc.).

The Vitrall session decides when and what to render (if not necessary, there is no rendering at all, so GPU/CPU time, network bandwidth and energy are spared), then informs client about results of visualization using WebSockets (see next section). Information about generated content is pushed to the client at the same time when render request is submitted to the rendering session – this way, at the time client's HTTP request reaches its destination, the generated image will be potentially ready to download, and latency will be limited.

Introduction of the session concept enabled Vitrall with multi-GPU rendering. This fact opened the way for the system to offer new capabilities like e.g. stereoscopic rendering. New, unique features were demonstrated at the TERENA Networking Conference [18] 2011 in Prague, in a 4K resolution real-time rendering scenario involving two projectors equipped with polarized filters.

WebSockets. WebSockets is a technology of a crucial importance for the Vitrall ability to push information to the client. WebSockets is a standard designed to allow bi-directional full-duplex communication between clients and servers over the Web. It consists of two parts: WebSocket API available in JavaScript and WebSocket protocol, which has been recently described in [15]. In contrast to traditional HTTP, apart from a handshake, WebSockets has almost no protocol overhead (in a sense of data).

Because WebSockets was an emerging standard during Vitrall development, support had to be implemented twice, as the final specification is very different from the initial drafts. Instability of the protocol was the price paid for using most innovative solutions available in the field of Web communication.

Apache HTTP Server. One of key concepts of Vitrall system is distribution of Vitrall session among several working sessions (both management and rendering). The first implementation of the idea of working sessions was separating management and rendering sessions not only in a logical and physical way: whilst the rendering instance hosting rendering sessions was based on the first prototype of Vitrall project (C++ server with basic HTTP connectivity), the part hosting management session and keeping WebSockets connections with clients has been based on Apache HTTP Server [1].

Motivation behind that decision was possibility of utilization of Apache capabilities related to authentication, access control and DSO (Dynamic Shared Object) support. During the project, two Apache modules were developed: generic module providing WebSockets connectivity and Vitrall management module.

3 Current Approach

Apache based solution proposed above, although successfully demonstrated at TNC 2011, was rather impractical to configure and maintain. Code duplication between management part written in C and rendering part written in C++ was another considerable problem. Still, the most important disadvantage was the fact, that Apache server typically works only on a direct request from the client. Apache API does not facilitate any straight-forward mechanisms to create on the server-side an entity, that will take actions (nor send information to clients) independently from client input.

Above justifies a decision to abandon Apache based solution and shift all needed functionality into C++ server. As a side effect, a far more flexible approach to the idea of session is now possible: currently the notions of management and rendering sessions derive from a generic working session, so if required for a specific configuration of Vitrall, another types of session can be easily introduced – this is one of the most extensible points of Vitrall. In addition, functionalities can be moved freely between various kinds of sessions.

An important advantage of Apache HTTP Server i.e. its dynamic module loading capabilities, has been also implemented in Vitrall.

To be fully extensible and flexible, Vitrall defines abstraction layers in key fields such as scene description, communication interface and user input handling. Current Vitrall architecture is illustrated in the Fig. 1.

3.1 Management and Execution of Rendering

Vitrall adopted a scene graph paradigm for the management of the visualization process. Scene graph is a common approach for 3D rendering, it is typically a tree of nodes representing various elements of the scene (primitives, models, transformations, effects).

OpenSceneGraph (OSG) [10] provides an implementation of the scene graph idea in C++. Since the second iteration of Vitrall project OSG toolkit serves as a primary rendering back-end. Because it is built upon the OpenGL, many OSG classes are only object-oriented proxies to the underlying OpenGL concepts

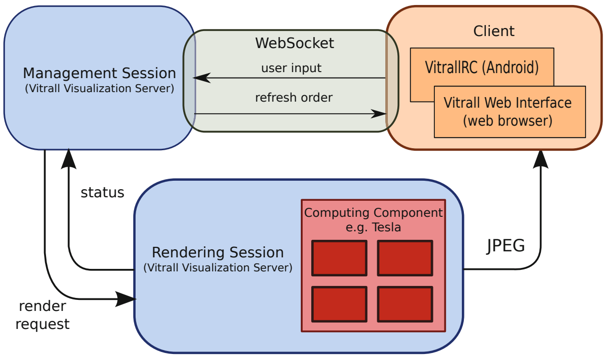


Fig. 1. Architecture of the Vitrall system

(buffers, light sources); other classes allow an easy creation of sophisticated visualization scenarios. One of key features of OSG is its extensibility: one can prepare plugins to read or write a specific 3D file formats or implement a class providing some special visual effect. In scope of the Vitrall project the, ability to load models from COLLADA [3] file format is especially utilized.

Because of the specific features of Vitrall project, i.e. distributed rendering, OSG could not be used straight-forward. Although OSG has a built-in support for rendering a single scene using multi-GPU, it does not support a distributed configuration, i.e. defining the scene at one host, whereas actual rendering is taking place at other host (and possibly more than one). Due to this limitation of OSG and keeping in mind planned utilization of OptiX [9] ray-tracing library – which also uses an object-oriented scene graph approach to scene definition – an abstraction layer over the scene graph had to be implemented.

This layer is a relatively simple scene graph representation, that is not capable of rendering by itself. Prepared nodes include groups, selectors, transformations, etc. but no node holds any actual data (like 3D meshes). Instead of this, special proxy nodes representing e.g. 3D models are provided. Such proxy holds only an URL to the resource – the model is loaded only at the Vitrall instance hosting rendering session. For a particular usage scenario, one may create his own proxy nodes representing other 3D resources (like a simulation done in real-time) as long as interpretation of these nodes at the rendering side is also defined.

All graph nodes of the Vitrall scene are exposed to the programmer in an object-oriented fashion, just like in OSG or OptiX. Performed operations are serialized and sent to rendering sessions, where they are being directly mapped to the operations on an actual scene graph of a given technology (OSG or OptiX) – abstract Vitrall scene graph structure does not exist in the rendering session.

One of secondary advantages of this solution is lack of necessity of installing libraries like OSG (as well as dependencies) on machines running Vitrall instances hosting management sessions.

3.2 Communication Abstraction

Communication abstract layer consists of two parts: resource serving and instant messaging. Both are generalizations of already prepared approach of sending images through HTTP and communicating input/orders through WebSockets connections. HTTP and WebSockets respectively are currently the only implementations of that layer.

3.3 Vitrall Clients

Vitrall can be currently accessed using two different client applications: Vitrall Web Interface accessible through an ordinary web browser (no plug-ins required) and the VitrallRC – a dedicated application designed for Android [6] based devices. Natural user interfaces offered by these clients, enables user with intuitive interaction with a remote visualization (see section 3.4). By natural interfaces, we mean here interfaces based on context acquisition process (exploitation of accelerometers, magnetic field sensors, gyroscopes, multi-touch screens, etc.).

Both clients are fully generic, their functionalities and behavior depend only on the current Vitrall configuration. They are exchangeable from the Vitrall server point of view. The only significant computation power needed by Vitrall clients is for image decompression, so they can be run on almost any device. Interpretation of user input is done completely server-side and is described in section 3.4.

An important part of client-server communication is the user forms mechanism. Forms provide the access to all Vitrall functionalities that cannot be manipulated using natural interfaces. The server may provide multiple forms at the same time and their list is accessible to the user on demand. Elements of the form include text boxes, seek bars, item lists, etc. in an abstract form, which is interpreted by client using corresponding HTML5 (Vitrall Web Interface) or UI controls (VitrallRC). Thanks to the forms mechanism it is possible to modify menus at runtime, dynamically adjusting the client to the current scenario and user's privileges (see section 5 for example).

Vitrall Web Interface. Vitrall Web Interface is a client prepared for desktop machines. It is a regular web page accessible through any HTML5 enabled web browser (nowadays nearly any browsers on the market, including browsers for mobile devices e.g. for Apple iPad [2]). The latest version of Vitrall Web Interface is based on HTML5 Canvas object, which provides a better performance than in previous approaches. User input events are based mostly on mouse motion events and keyboard, although there is a support for multi-touch screens (at the time of writing of this article only Firefox of version 4+ for Window 7 offers this functionality). Currently JPEG is utilized for image transfers. Though decompression of screen resolution JPEG images by a web browser is fast enough for a completely smooth animation effect, support for other formats is foreseen in the near future.

VitrallRC. VitrallRC is a dedicated client developed for Android based (version 2.1 or later) devices (smartphones, tablets, etc.). It is published on Android Market and is free for download. This client, implemented in Java, is composed of four main functional parts responsible for: connectivity, context acquisition (collecting data from sensors), image display and user forms management, QR code readings.

As in case of the Vitrall server, WebSockets connectivity had to be implemented twice, as the standard was emerging at the time.

The second module of mobile application is responsible for context acquisition. It is possible to obtain device orientation in world coordinate space, with utilization of accelerometer and magnetic field sensor or gyroscope. That information is sent to the server, where it is interpreted and translated to the appropriate actions (accordingly to the server state) e.g. rendered model or the whole scene can be rotated in the 3D space, reflecting orientation of the smartphone. Another source of user input events is multi-touch screen. Thanks to moving the interpretation to the server-side, customization of interpretation of gestures is very simple. Moreover interpretation of gestures, as well as sensor data, can be adjusted in runtime. As a result e.g. the pinch zoom gesture can mean an usual zoom and a few seconds later it can be used for moving two objects indicated with one finger each independently.

The first VitrallRC prototype consisted only of first two parts. It was used for conduction of tests checking whether Android can be used to send sensor data to the server over WebSockets with satisfactory frequency (at least 25 Hz). Results were much higher (by an order of magnitude) than demanded.

The third component of Android client is designed to display image from server and manage forms menus. Drawing to the screen is accomplished by a custom View class, this is due to the fact of gaining unsatisfactory frames per second (fps) rate when using standard Android components. Depending on image resolution and quality, our approach allows to reach 15-18 fps (tests were taken on Samsung Galaxy Tab P1000 running under Android 2.2). The bottleneck that did not allow to go over 18 fps is CPU, which is responsible for JPEG image decoding. The solution for this problem may be a hardware acceleration, however it was not yet available by the time of performing tests.

Vitrall mobile application has been equipped with QR code scanner (using ZXing library [22]). It may be used not only as a simple way of informing the client of the server's URL, but also as a form of authentication. In the second case, the QR code containing authentication information is generated dynamically by the server and displayed as a part of visualization, allowing at the same time to associate mobile device with another larger (e.g. tiled) screen.

VitrallRC debuted in September 2011. Demonstration for scientists took place during the PPAM 2011 conference. Two weeks later, during Researchers' Night [14] in Poznań, much broader community was able to test it on their own mobile devices. Presented 3D puzzle scenario involved mentioned before stereoscopic features of the Vitrall.

To the best of our knowledge, VitrallRC is the first mobile based, interactive, remote visualization client.

3.4 User Input Processing and Handling

As the interpretation of user inputs, actions or gestures is done completely on the server-side, abstractions of events and their processors were introduced, to allow flexibility and extensibility in this field. Typical event processing scenario is a flow net, of which internal vertices represent processors, net sources are connections with users and sinks are usually transformation nodes of the scene graph.

Events may be routed through the net basing on their type, user origin, scenario state or other conditions specific to the usage scenario. Processors are typically state-full objects, which are responsible of translating one kind of events into another, e.g. multi-touch processor converts points of contact into transformation events based on how much fingers are currently active and what are their current and previous positions. Transformation events are not represented by matrices, but more semantically as translations, rotations, and scaling – this allows filtering, selective amplification or routing to different nodes of scene.

This way, the designer of a specific usage scenario can freely combine and mix existing processors and events to deliver a satisfying user experience, tailored exactly to the user needs. If the already existing processors or events are not sufficient, one can with ease create his own types of events or processors.

3.5 Data Providers

Capabilities of the Vitrall are not limited to visualization of a static content. Series of mechanisms allowing for connectivity with external services and applications have been developed. A good example of scenario involving third-party service was demonstrated during the Future Internet Week in Poznań [5]. With the VitrallRC, visitor could enter a SMILES format based query for the molecule. The query was sent to the Protein Data Bank RESTful Web Service [13]. Response was transformed to a list of similar molecules, and then presented in VitrallRC as a dynamic form (described in Sect. 3.3). In the next step, user was able to choose a molecule for transformation from the pdb format to the 3D model. Result was presented on a high resolution hexagon prism shaped tower, consisting of 18 HD displays. Each side of the tower presented view from a different angle (shifted by 60 degrees around the tower axis).

Exploration of dynamically created semantic graphs out of digital libraries was another dynamic scenario demonstrated during the FIW. Also in this case, concurrent observation of the molecule from six different angles, as well as manipulation powered by smartphone sensors were possible.

4 Competitive Solutions

It must be noted, that Vitrall system doesn't exist in a vacuum. Several visualization systems with capabilities partially overlapping with those of Vitrall already exists – two are briefly described below. However, Vitrall manages to distinguish itself thanks to its Web character and support for mobile devices and natural user interfaces.

ParaView. ParaView [11] is an open source program for parallel and interactive scientific visualization. It builds upon VTK [20] libraries to allow visualization of extremely large data sets using distributed memory computing resources. ParaView works in a client-server architecture, where data processing and visualization can be done using remote resources.

VTK is the project related to ParaView. It provides a variety of visualization algorithms including: scalar, vector, tensor, texture, and volumetric methods.

VisIt. VisIt [21] is a free interactive parallel visualization and graphical analysis tool for viewing scientific data of terascale sizes. VisIt utilizes VTK library and adds many features like data distribution and advanced plotting. When using VisIt, rendering is typically done at the local machine, while data processing is done in parallel using remote resources.

5 Current and Future Work

Vitrall project is or will be in a near future a part of the following projects:

PL-Grid. Analysis made in the scope of the PL-Grid project [12] confirms that Vitrall can be integrated into modern science gateways. In this case, capability of rendering of the PDB format proved to be useful for preliminary check of the molecule, before executing of the time consuming computations, submitted to the grid by the NAMD application running under the Vine Toolkit [8]. In addition, not only input, but also output of the mentioned application can be rendered in order to enable scientists to better analyse results of their work. Future steps towards integration of Vitrall with the Vine Toolkit are considered in the scope of the PL-Grid successor named PL-Grid Plus.

TEFIS. As a part of the TEFIS [17] project, an experiment called TEFPOL will be conducted. The scenario concerns an e-learning lesson in video-conferencing environment with augmented reality and touch-less interfaces. For this experiment, Vitrall will be prepared to run in a grid environment of PacaGRID; the interface part will be based on research already conducted at PSNC.

CoolEmAll. Today, energy efficiency is an important issue – CoolEmAll [4] addresses this problem in scope of data centres. Main goal of this project – which is coordinated by PSNC – is to provide a simulation, decision and visualization toolkit to support building and maintain energy efficient data centres environments. Vitrall will be used as one of visualization platforms; for this task VTK system is foreseen to be integrated into Vitrall.

Acknowledgments. This work has been funded by the PL-Grid project: contract number: POIG.02.03.00-00-007/08-00, website: www.plgrid.pl. The PL-Grid project is co-funded by the European Regional Development Fund as part of the Innovative Economy program.

References

1. Apache HTTP Server, http://httpd.apache.org/
2. Apple iPad, http://www.apple.com/ipad/
3. COLLADA – Digital Asset and FX Exchange Schema, http://www.collada.org
4. CoolEmAll, http://www.coolemall.eu/
5. Future Internet Week, Poznań (2011), http://www.fi-poznan.eu/
6. Google Android, http://www.android.com/
7. Independent JPEG Group, http://ijg.org/
8. Kurowski, K., Dziubecki, P., Grabowski, P., Krysinski, M., Kuczynski, T., Szejnfeld, D.: Modern Portal Tools And Solutions with Vine Toolkit for Science Gateways. In: Proceedings of the International Workshop on Science Gateways, IWSG (2011) (to appear)
9. NVIDIA OptiX ray tracing engine, http://www.nvidia.com/object/optix.html
10. OpenSceneGraph – 3D graphics toolkit, http://www.openscenegraph.org
11. ParaView, http://www.paraview.org/
12. Kitowski, J., Turala, M., Wiatr, K., Dutka, L., Bubak, M., Szepieniec, T., Radecki, M., Sterzel, M., Mosurska, Z., Pajak, R., Slota, R., Palak, B., Kurowski, K., Balcerek, B., Bala, P., Filocha, M.: Polish Computational Research Space for International Scientific Collaborations. In: Proceedings of Parallel Processing and Applied Mathematics Conference (2011) (to appear)
13. Protein Data Bank RESTful Web Service, http://www.pdb.org/pdb/software/rest.do
14. Researchers' Night, http://ec.europa.eu/research/researchersnight/
15. RFC6455 – The WebSocket Protocol, http://tools.ietf.org/html/rfc6455
16. Supercomputing, New Orleans (2010), http://sc10.supercomputing.org/
17. TEFIS, http://www.tefisproject.eu/
18. TERENA Networking Conference (2011), https://tnc2011.terena.org/
19. Leigh, J., et al.: The global lambda visualization facility: An international ultra-high-definition wide-area visualization collaboratory. Future Generation Computer Systems 22(8), 964–971 (2006)
20. The Visualization Tookit, http://www.vtk.org/
21. VisIt, https://wci.llnl.gov/codes/visit/
22. ZXing barcode library, http://code.google.com/p/zxing/

CUDA Accelerated Blobby Molecular Surface Generation

Daniele D'Agostino[1], Sergio Decherchi[2], Antonella Galizia[1], José Colmenares[2], Alfonso Quarati[1], Walter Rocchia[2], and Andrea Clematis[1]

[1] Institute for Applied Mathematics and Information Technologies, National Research Council of Italy, Genoa, Italy
[2] Department of Drug Discovery and Development, Italian Institute of Technology Genoa, Italy

Abstract. A proper and efficient representation of molecular surfaces is an important issue in biophysics from several view points. Molecular surfaces indeed are used for different aims, in particular for visualization, as support tools for biologists, computation, in electrostatics problems involving implicit solvents (e.g. while solving the Poisson-Boltzmann equation) or for molecular dynamics simulations. This problem has been recognized in the literature, resulting in a multitude of algorithms that differ on the basis of the adopted representation and the approach/ technology used. Among several molecular surface definitions, the Blobby surface is particularly appealing from the computational and the graphics point of view. In the paper we describe an efficient software component able to produce high-resolution Blobby surfaces for very large molecules using the CUDA architecture. Experimental results show a speedup of 35.4 considering a molecule of 90,898 atoms and a resulting mesh of 168 million triangles.

Keywords: Blobby Molecular Surface, GPU Computing, Parallel Molecular Surface Generation.

1 Introduction

Molecular surface computation is a key issue from at least two perspectives, the visualization and the biophysical computation. In the first case a user is interested in the overall rendering quality of the molecular model: classical paradigms triangulate the surface and then visualize the mesh [1]. Furthermore, the use of present Graphic Processing Units (GPU) capabilities allows the direct rendering of quadrics patches by ray tracing [2].

From the biophysical stand point a user is interested in a molecular surface that is able to capture the physical problem at hand and that is computationally efficient. An example of such use case is the Poisson-Boltzmann equation (PB), where the electrostatic potential of a molecule (solute) in water (solvent) is sought. In particular in PB it is usually accepted that the Van Der Waals surface (VDW) should not be directly used: to solve this problem the Solvent

Excluded Surface (SES) (or Connolly surface) was introduced [3] (an algorithm for calculation was given in [4]) which expands the VDW surface by smoothing its concave parts with spherical patches that represent the rolling of a water molecule (the *probe*) over the surface. The Connolly model has been largely used in the solution of PB [5], however it still presents some limits [6]: among them, the fact that it leads to a non differentiable surface is of particular importance.

This issue calls for alternative surface models that retain a physically sound definition and overcome SES limitations. Whatever is the final goal, either visualization or biophysical computations, some properties of the molecular surface are desirable. Among them the computational efficiency, the differentiability everywhere and a good performance scalability for large molecular models. To this aim the Blobby surface [7][8] represents an interesting alternative to the SES surface.

In this paper we describe an efficient algorithm, based on the isosurface extraction, able to produce high-resolution Blobby surfaces for very large molecules using the CUDA architecture. Our aim was to design the algorithm able to act as a software component producing an output suitable for both the direct visualization and the efficient storage of the surface. In fact the possibility of the direct use of the produced mesh without any preprocessing step is of particular importance for the performance view points if the surface have to be further processed in a biophysical workflow.

The following Sections respectively present: related work, the definition of the Blobby surface, the CUDA accelerated algorithm for the generation of the Blobby surface and experimental results. At the end some conclusions are drawn.

2 Related Work

Molecular surface computation is a long standing problem. Among the various proposals, we mention the most used ones: the Connolly surface [4], the Skin surface [9], and the Blobby surface [7].

From the computational point of view, the recent emerging availability of relatively inexpensive GPU systems stimulated the research on the usability of GPU devices to accelerate the molecular surface processing and visualization.

In [2], it is shown how GPU can be used to ray trace the Skin surface [9]. The Skin surface in fact is composed by a set of quadric patches (i.e. spheres and hyperboloids), each of them bounded by a solid of the mixed complex [9]. This allows to use OpenGL shading language for the real time rendering.

In [10] it is shown how both the SES and the Skin surfaces can be built in parallel on CPU and effectively rendered on the GPU by ray tracing, obtaining high frame rates; analogous considerations hold for the Connolly [10].

However, these works aim exclusively at improving the rendering phase, while we are interested in exploiting the GPU computing capabilities also in the generation of the surface in a format that is able to suit both the visualization and further processing steps in a biophysical analysis workflow.

A similar approach is followed in [11], where a parallel workflow for the extraction of SES surfaces is described. It is based on the construction of the volumetric dataset starting from the atomic coordinates of the atoms that form the molecule, which is then processed using the isosurface extraction operation to produce the SES as a mesh of triangles. The main drawback of this work is that it do not considers the use of GPU devices.

Such aspect is addressed in [12], where an approach similar to the previous one is implemented for the CUDA architecture and extended to build smooth molecular surfaces. In particular this work considers also the creation of the Blobby surfaces but, as the previous ones, disregards the aspect related to the effective storing of the produced meshes.

3 Blobby Surface Definition

The Blobby surface S [7] [8] is defined as:

$$S := \{\mathbf{x} \in R^3 : G(\mathbf{x}) = 1\}, \;\; G(\mathbf{x}) = \sum_{i=1}^{n_a} e^{B\left(\frac{\|\mathbf{x}-\mathbf{c}_i\|^2}{r_i^2}-1\right)} \tag{1}$$

where r_i are the radii of atoms, n_a is the number of atoms, $\mathbf{x}$ is the query point, c_i is the $i-$th atom center and B is a negative parameter (the *blobbyness*) that plays the role of the probe radius when compared to the Connolly surface [4].

Blobby surface has some salient pros and cons from both the computational and physically soundness point of view. At first the surface is easy to implement because the central computation is simply an evaluation of a kernel function. The surface is also tangent continuous and is self-intersections and singularities free. From the physical model point of view it is not completely clear if this surface is superior to the SES when solving, for instance, electrostatics problems. Indeed it can be argued that the right setting of the blobbyness value B is a key parameter in order to obtain reliable energy estimations of molecular systems. Another point is that the surface it is not partitioned in analytical patches as in the Skin or in the SES. Additionally some values of B can modify the size of the atoms leading to non physically acceptable surfaces as observed in [13]. Despite these drawbacks, the implicit models are becoming rather used [14] when dealing with biophysical problems, because this surface is smooth, continuous and differentiable everywhere and because Gaussian functions mimic model electronic density functions.

The high resolution representation of molecules is a key aspect for their satisfactory visualization and also for the effectiveness of analysis operations. However the computation of this surface for molecules composed by dozen thousands atoms is a costly process that may require several minutes. This is the reason why we designed a parallel algorithm for the generation of the Blobby surface. In particular we made use of the CUDA architecture, that represents a cost-effective solution for many compute-intensive applications on regular domains as this one.

4 CUDA Accelerated Blobby Surface Generation

Following the CUDA naming convention, we define *host* the workstation and *device* the NVIDIA card providing the GPU of which CUDA is the computing engine. The algorithm we propose is based on two main operations, the *Scalar Field Generation* and the *Isosurface Extraction*, both performed on the device with a minimal amount of data transfer with the host.

Usually a molecule is represented through the set of the 3D atomic coordinates of the atoms that form it. This is for example the format adopted by the Protein Data Bank (PDB) [15], one of the most important repositories. A PDB file and a sampling step are the inputs of the algorithm, while the resulting isosurface is the output.

In particular, a triangle mesh may be represented by its vertex data and by its connectivity. In its simplest form, a Vertex table contains the coordinates of all the vertices, while connectivity can be represented by a Triangle-Vertex incidence table, which associates with each triangle the references to its three bounding vertices. Such representation suits well both the direct visualization and the successive processing steps, because these two tables allow to efficiently reconstruct all the other incidence relations. For this reason we adopted such format for the output.

4.1 Scalar Field Generation

The first operation of the algorithm is the generation of the three-dimensional grid containing the volumetric data representing the molecule. The size of the grid is determined on the basis of the coordinates of the atomic centers and on the required sampling step. Typical step values are chosen between 0.7 and 0.1 Å, according to the desired level of resolution. Smaller step values correspond to dense grids and high resolution surfaces, and vice versa. Within the grid, atoms are modeled as spheres having different radii.

The grid is usually considered as a set of XY planes, called *slices*. As the amount of memory of a device is limited, and the isosurface extraction operation requires to process a pair of slices at a time, we implemented this and the following operation in an iterative way for increasing values of the Z coordinate. This means that one slice is created at each iteration (except for the first one, where the slices for Z=0 and 1 are created) in order to replace the slice having the lowest Z value. In this way we are able to process very large data sets if the size of a pair of slices does not exceed the device memory.

The value of a grid point is the result of the influence of all the atoms on it. For large molecules (e.g. 10^5 atoms) this translates to considering several million points. The present CUDA architecture limits the number of threads (i.e. up to 1024 threads for a block and up to a grid of 65535x65535x1 blocks), and this means that is not possible to generate a thread for each pair atom-point. Therefore we have to group this large number of operations on the points or on the atoms.

We experimented that, even if the partitioning on the number of points allows a greater scalability and parallelism degree, the achieved performance is lower than with the alternative strategy due to the large number of non-local memory access. In fact even if we store the coordinates of the atomic centers and the radii in the constant memory, each of these values has to be accessed a number of times equal to the number of threads.

Even the association of one thread for each atom has the drawback that it needs to perform the updates of each point with atomic operations, because in principle the value of a point is the sum of the influence of all the atoms. This means that each point update has to be performed without race conditions, resulting in possible overhead due to the update serialization. However, as noted in [11] and [12], each atom influences in a significative way only the points within a limited bounding box surrounding it. This consideration has two important consequences. The first is to reduce the number of operation to be performed, since it is useless to consider all the atom-point pairs. Moreover, the concurrent updates are very limited, in the order of hundreds of atoms for each points, and therefore the impact of the serializations is negligible.

4.2 Isosurface Extraction

The *Marching Cubes* algorithm [16] is the most popular method used to extract triangulated isosurfaces from volumetric datasets. In the Marching Cubes algorithm the triangular mesh representing the isosurface is defined piecewise over the cells in which the grid is partitioned. A cell is intersected by the isosurface represented by the *isovalue* if the isovalue is between the minimum and the maximum of the values assumed by the eight points of the grid that defines each cell. This kind of cells is called *active cells*. An active cell contributes to approximate the isosurface for a patch made of triangles, and the union of all the patches represents the isosurface. The algorithm consists of two main operations, the *Cell Classification* and the *Active Cell Triangulation*.

The *Cell Classification* determines if a cell is intersected by the isosurface or not. This is done using a bit vector of 8 fields of one bit, each of them corresponding to one point of the cell. Points with values greater or equal to the isovalue are marked with 1, otherwise with 0: therefore a cell is an *active cell* if the bit vector has a value different from 0 (all points values lower than the isovalue) and 255 (all points values greater than or equal to the isovalue).

In these cases the *Active Cell Triangulation* operation is performed, consisting in the approximation of the intersection with the isosurface, using a triangular patch. Considering that a surface may intersect a cell in 254 ways, that is all the values of the bit vector except 0 and 255, a *look-up table* is used to enumerate all the possible connectivity schema. The coordinates of the vertices of the triangles are computed as a linear *interpolation* of the values of intersected edges.

The parallelization of the original algorithm for the CUDA architecture is a quite straightforward task, because it is achieved by assigning one cell for each thread, and it is provided as a C code example in the NVIDIA CUDA SDK[1].

[1] `http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html`

Obviously this application is only an example and it has many limitations, as for instance the small size of the volume that is able to process. Other proposals were published, with the main aim to speed up the processing of extracting and visualizing very large isosurfaces (see [17] for a survey). The main issue with these algorithms is the fact that they are designed for a direct visualization of the produced isosurfaces, and not for storing them. This means that they do not consider one major issue of the algorithm, that is the duplications of the vertices. Each active cell is in fact processed independently, and this means that a vertex may be recalculated up to four times in adjacent cells (see Figure 1(a)). The duplicated vertexes are useless and they may have a considerable impact on the computing time and on the size of the resulting mesh for further processing operations if these algorithms are used in a workflow. Obviously the vertices can be merged using a post-processing step, but this limits the achievable speed up.

In [17] we proposed a novel algorithm that is able to produce an isosurface equivalent to that produced by the sequential algorithm in an efficient way using the solution proposed in [18], that makes use of five auxiliary array data structures. The coordinates of a vertex are computed only the first time the corresponding edge intersected by the isosurface is considered. These coordinates are inserted in the Vertex table and the index corresponding to the vertex position in the table is stored in the correct position of one of the five auxiliary array data structures shown in Figure 1(b). As indicated in the Figure, in a generic cell (i.e., a cell which is not on the border of the volume) nine edges were previously considered in the processing of adjacent cells, therefore it is possible to produce at most three new vertices. The values in the auxiliary data structures are updated during the subsequent processing of all cells. For example, considering Figure 1(b), the black vertex is computed by the bottom left cell and its index is inserted in the corresponding position of the Ledge array structure. The next cell being processed is the bottom right one. This cell uses the stored index and moves it to the proper position in the Yedge array. When the next pair of slices is considered, the top left cell uses the index without needing to modify Yedge. Finally, the top right cell uses the index for the last time.

The CUDA-based version of the algorithm is composed by the following four kernels: *VerticesCalc*, where the coordinates of the vertices are computed; *VerticesCompact*, where vertices are associated with labels to be used to represent triangles and they are grouped to reduce transfer time; *TrianglesCalc*, where the triangles are computed as three labels of vertices; *TrianglesCompact*, to group the resulting triangles. The data transfer operations represent a considerable part of the time spent in performing the isosurface extraction operation on a device. Therefore we overlapped the data transfers and the kernel executions. In particular we overlapped: a) VerticesCalc with the transfer of the triangles found considering the previous pair of slices; b) TrianglesCalc with the transfer of the vertices; c) TrianglesCompact and the transfer of the next slice. This last overlap does not apply in this case, because the slice are created by the previous operation directly in the memory of the device. More details on this CUDA-based version of the algorithm are provided in [17].

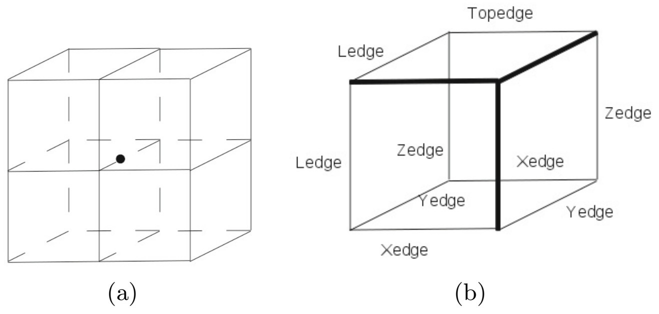


Fig. 1. The original algorithm computes the black vertex four times, one for each cell. The use of auxiliary data structures allows to avoid it. Thick lines represent edges not previously considered.

5 Experimental Results

Experimental results were collected considering two implementations of the Blobby surface generation, one sequential and one parallel, for the CUDA architecture[2]. The two programs were executed on a workstation equipped with an Intel i5-750 CPU and an NVIDIA GTX480 device. In particular the sequential implementation makes use of one core of the processor to perform the whole computation.

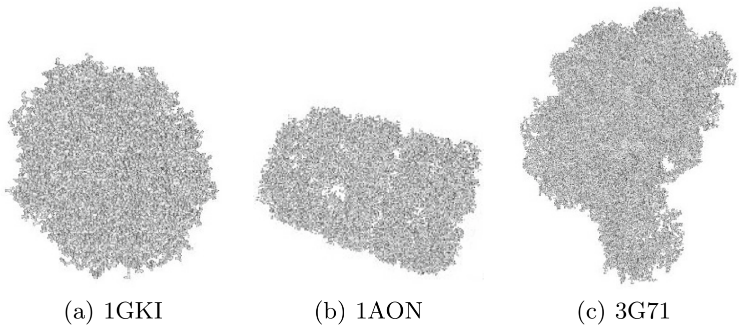


Fig. 2. The Blobby surfaces associated to the three molecules selected for the tests. We obtained them considering the volume with the resolution of 0.5 Å.

Three molecules of the Protein Data Bank repository, chosen on the basis of their size, were considered. The smallest one is the Plasmid coupling protein TrwB, identified as *1GKI* and made up by 19,536 atoms, followed by the crystal structure of the asymmetric GroEL-GroES-(ADP)7 chaperonin complex, identified as *1AON* and made up by 58,674 atoms, and one of the largest structure

[2] Our software is available upon request, and will be made publicly available in case the paper is accepted.

Table 1. This table summarizes the characteristics of the three considered molecules

Molecule	Atoms	Resolution	Grid	Triangles
1GKI	19,536	0.5	226x235x237	1,410,816
		0.1	1132x1177x1185	36,583,252
1AON	58,674	0.5	312x477x469	4,256,936
		0.1	1563x2385x2346	110,392,108
3G71	90,898	0.5	379x474x491	6,485,168
		0.1	1894x2367x2458	167,548,496

in the PDB repository, the co-crystal structure of Bruceantin bound to the large ribosomal subunit, identified as *3G71* and made up by 90,898 atoms. They are presented in Figure 2.

The B parameter was set to -2.3 and two steps, 0.5 and 0.1 Å ,were considered for the Scalar Field Generation operation. They represent, respectively, a medium and a high detailed resolution. The characteristics of the molecules, of the resulting volumetric datasets and of the Blobby surfaces are shown in Table 1, while Table 2 shows the performance of the sequential and the parallel implementations.

Table 2. This table presents the times, in seconds, for executing the sequential and the parallel implementations of the Blobby Surface Generation. In brackets the achieved speedups. It is worth noting that, in the total time for the CUDA version, we do not considered the initialization time, that is of about 6.5 seconds in all the cases.

		Scalar Field Generation		Isosurface Extraction		Total	
		Seq	CUDA	Seq	CUDA	Seq	CUDA
1GKI	0.5	2.78	0.11 (25.3)	1.38	0.07 (19.2)	4.16	0.18 (23.1)
	0.1	294.08	7.95 (37.0)	89.76	2.60 (34.5)	383.84	10.55 (36.4)
1AON	0.5	8.53	0.28 (30.9)	6.63	0.20 (32.8)	15.16	0.48 (31.6)
	0.1	832.14	20.70 (40.2)	472.03	12.90 (36.6)	1304.17	33.60 (38.8)
3G71	0.5	13.20	0.42 (31.4)	8.68	0.25 (34.7)	21.88	0.67 (32.7)
	0.1	1057.10	30.78 (34.3)	625.71	16.73 (37.4)	1682.81	47.51 (35.4)

We can see that, except in the smallest case, the speedups achieved vary between 30 and 40. This is an encouraging result considering the issues related to the implementation of both the Scalar Field Generation and the Isosurface Extraction in CUDA. As regards the Scalar Field Generation we can see that the fixed parallelism degree do not allow to scale in proportion to the volume size, but this limit neither involves a degradation. Each CUDA thread in fact is responsible to assign the value to a few points for each slices, whose number varies from 2 to 60, therefore we are able to achieve good performance. This is also due to the fact that no data movement are required: each slice is created on the device memory, used for the isosurface extraction and then replaced with a new one without the need to involve the host memory.

The data movement instead is the factor that limits the performance of the Isosurface Extraction operation. We have to consider in fact that it requires the transfer of the triangular mesh representing the Blobby surface: in the largest case this mesh is composed by about 168 million triangles and 84 million vertices, resulting in about 9 GB of data to transfer. However the overlaps between data transfers and kernel executions permit to achieve high performance figures also in this case.

A final issue, common to all the CUDA programs, is represented by the time required to initialize the CUDA device, that it is performed in correspondence of the first call to a CUDA function within a program. In our case this time is equal to about 6.5 seconds, therefore the use of the CUDA version is unfeasible for small datasets as the 1GKI with a step of 0.5. It is however to consider that the Blobby surface generation is an operation that can be inserted in a workflow where other CUDA-based operation are executed: in these case the impact of the initialization time on the whole processing time is limited.

6 Conclusions and Future Works

This work presented a CUDA-based efficient algorithm for the Blobby molecular surface generation. In particular, the algorithm is able to achieve a speedup of 35.4 considering a molecule of 90,898 atoms and a resulting mesh of 167 million triangles. We experimented that a parallelization on the atoms, even if involves a lower degree of parallelism, is able to provide higher performance figures than a parallelization on the points of the grid containing the scalar field representing the molecule, due to the lower number of device memory accesses.

Two future works are forecasted. The first one is a further improvement of the performance of the algorithm, in particular by an in depth analysis of the role of the B parameter on the performance. The second one is the adoption of the algorithm in tools for molecular surface construction [5], in order to use the produced meshes to solve the Poisson-Boltzmann equation and/or visualization purposes.

Acknowledgments. This work was partially supported by NIGMS, NIH, grant number, 1R0GM093937-01.

References

1. Yu, Z., Holst, M.J., Cheng, Y., McCammon, J.A.: Feature-preserving adaptive mesh generation for molecular shape modeling and simulation. Journal of Molecular Graphics and Modelling 26(8), 1370–1380 (2008)
2. Chavent, M., Levy, B., Maigret, B.: MetaMol: High-quality visualization of molecular skin surface. Journal of Molecular Graphics and Modelling 27(2), 209–216 (2008)
3. Richards, F.M.: Areas, volumes, packing and protein structure. Annu. Rev. Biophys. Bioeng. 6, 151–176 (1977)

4. Connolly, M.L.: Analytical molecular surface calculation. J. Appl. Cryst. 16(5), 548–558 (1983)
5. Rocchia, W., Sridharan, S., Nicholls, A., Alexov, E., Chiabrera, A., Honig, B.: Rapid Grid-Based Construction of the Molecular Surface and the Use of Induced Surface Charge to Calculate Reaction Field Energies: Applications to the Molecular Systems and Geometric Objects. Journal of Computational Chemistry 23(1), 128–137 (2001)
6. Vorobjev, Y.N., Hermans, J.: SIMS: Computation of a Smooth Invariant Molecular Surface. Biophysical Journal 73, 722–732 (1997)
7. Blinn, J.: A generalization of algebraic surface drawing. ACM Transactions on Graphics 1(3), 235–256 (1982)
8. Zhang, Y., Xu, G., Bajaj, C.: Quality meshing of implicit solvation models of biomolecular structures. Journal Computer Aided Geometric Design - Special Issue: Applications of Geometric Modeling in the Life Sciences 23(6) (2006)
9. Edelsbrunner, H.: Deformable Smooth Surface Design. Discrete and Computational Geometry 21(1), 87–115 (1999)
10. Lindow, N., Baum, D., Prohaska, S., Hege, H.C.: Accelerated Visualization of Dynamic Molecular Surfaces. In: Eurographics/ IEEE-VGTC Symposium on Visualization, vol. 29(3), pp. 943–952 (2010)
11. D'Agostino, D., Merelli, I., Clematis, A., Milanesi, L., Orro., A.: A parallel workflow for the reconstruction of molecular surfaces. Parallel Computing: Architectures, Algorithms and Applications, Advances in Parallel Computing 15, 147–154 (2008)
12. Dias, S., Bora, K., Gomes, A.: CUDA-based triangulations of convolution molecular surfaces. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010), pp. 531–540 (2010)
13. Lu, Q., Luo, R.: A Poisson Boltzmann dynamics method with nonperiodic boundary condition. J. Chem. Phys. 119, 11035–11047 (2003)
14. Im, W., Beglov, D., Roux, B.: Continuum solvation model: Electrostatic forces from numerical solutions to the Poisson-Bolztmann equation. Comp. Phys. Comm. 111, 59–75 (1998)
15. Berman, H.M., Bhat, T.N., Bourne, P.E., Feng, Z., Gilliland, G., Weissig, H., Westbrook, J.: The Protein Data Bank and the challenge of structural genomics. Nature Structural Biology 7(11), 957–959 (2000)
16. Lorensen, W.E., Cline, H.E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. Computer Graphics (Proceedings of SIGGRAPH 1987) 21(4), 163–169 (1987)
17. D'Agostino, D., Seinstra, F.J.: An Efficient Isosurface Extraction Component for Visualization Pipelines based on the CUDA Architecture. Technical Report IR-CS-64-2010, Vrije Universiteit, Amsterdam, The Netherlands. An extended version was submitted to the Special Issue on Accelerators for High-Performance Computing of the Journal of Parallel and Distributed Computing
18. Watt, A., Watt, M.: Advanced Animation and Rendering Techniques Theory and Practice. Addison-Wesley/ACM Press (1992)

GPU Accelerated Image Processing for Lip Segmentation

Lukasz Adrjanowicz, Mariusz Kubanek, and Adam Tomas

Czestochowa University of Technology,
Dabrowskiego 69, 42-201 Czestochowa, Poland
{lukasz.adrjanowicz,mariusz.kubanek,bkslash}@icis.pcz.pl

Abstract. This paper presents the problem of lip segmentation in parallel environment using computational capabilities of GPUs and CUDA. The presented implementation of lip segmentation is based on image processing methods using the most popular transformations such as morphological operations and convolution filters. The obtained experimental results for the parallel implementation on GPU indicate significant speedup in comparison to its sequential counterpart. Consequently, the use of popular graphics cards provides a very promising possibility of quick lips segmentation.

Keywords: image processing, lip segmentation, parallel processing, CUDA.

1 Introduction

In today's world image processing plays an important role in everyday life. Numerous algorithms are used very widely in computer graphic [17,13] and computer vision [15,5]. They allow us to complete simple tasks like photo enhancement, but also more advanced operations such as mobile robot steering [3] or object recognition [25]. A particularly interesting field is the audio-visual speech recognition (AVSR), where image processing techniques are used to aid the speech recognition process [11,2]. In most cases AVSR is applied in environments with high levels of noise, in which information from the audio channel is often insufficient [16]. Therefore, the visual observation of lip movement is applied to reduce the recognition error [14,26].

Before examining the lip area, segmentation techniques are used to extract the desired region of interest. Various image processing methods are utilized in this process [24,7,23,12]. Many of them apply different transformations to reduce noise, expose the lip area, etc. From the segmented lip area, specific feature points can be designated to determine the position and shape of upper and lower lip. This operation delivers valuable information to the classification process, resulting in speech recognition or supporting its operation.

Numerous capturing devices offer a high resolution and high frame rate, that can provide a better image quality and improve the recognition process. Unfortunately, processing large-size data requires the adequate computational power.

The complexity of some algorithms can often exhaust available system resources on a PC-based platform, where the lack of sufficient CPU power makes it difficult to run intensive real time applications. There is no doubt that GPUs provide better possibilities in fast image processing, since most algorithms in this area can be easily ported into parallel environment.

This paper proposes a parallel version of lips segmentation based on thresholding and image processing techniques. Furthermore, it shows the benefits of exploiting computer unified device architecture (CUDA) [1] in accelerated image processing. In particular, we present the use of CUDA memory model for storing partial results of different transformations on the device instead of performing intermediate data transfers. All the calculations are carried out in a single kernel invocation by reducing memory transfers.

2 Previous Work

Development of GPU has significantly contributed to the emergence of many parallel computer vision applications. Strzodka et al. [19] presented a motion estimation algorithm, where the optical flow was used for real-time analysis of 320x240 images. He also presented a framework for computing generalized distance transforms and skeletons of two-dimensional objects [20]. A graphical hardware implementation of generalized Hough transform for fast object recognition was also discussed [21]. Hopf and Ertl described the accelerated 3D convolution [8] and morphological analysis [9]. In the problem of object detection, a faster alternative version of circle Hough transform was described by Ujaldn et al. [22]. The feature tracking with SIFT feature extraction algorithms was presented in [18].

Image processing libraries and their applications were also described in literature. Probably the most popular are OpenVidia [6] and GPUCV [4]. They provide a wide comprehensive set of tools for image processing and allow users to use many popular transformations and techniques.

3 Lip Segmentation

Lips can be distinguished from skin area as they have different pixel components in the RGB color space. In images, the skin mixture contains more green and red values rather than blue one. On the other hand, the lip area combines green and blue channels with almost equal ratio. Based on this information, the pseudo-hue transformation can be computed with the use of the following equation:

$$h(x, y) = \frac{R(x, y)}{G(x, y) + R(x, y)} \tag{1}$$

This one-point transformation [10] converts a RGB image to the gray scale, where bright pixels represent the lip area. It facilitates the further processing process, as only one channel is left, and still delivers valuable information. Sample results are shown in Fig. 1.

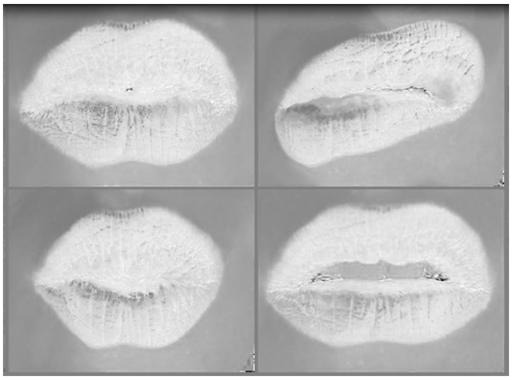


Fig. 1. Results of pseudo-hue transformation for selected test images

In image processing, digital filters are more comprehensive tools than the one-point transformation. In most cases, they assume the use of contextual operation. This means that to designate one pixel of the output image we need to perform calculations on many pixels from the surrounding of the considerate pixel. From a mathematical point of view, a digital filter is a certain multi-argument function transforming one image into another pixel by pixel. The properties of the filter result from analytical characteristics of the given function. It is convenient to use the convolution function as it provides a wide amount of useful transformations. In two-dimensional and discrete domain, it can be written as follows:

$$J_w(x, y) = \sum_{i,j \in K} J(x - i, y - j) w(i, j) \tag{2}$$

Depending on the weights that are used, different transformations can be achieved. For reducing noise and softening the image, the Gaussian operator can be applied. This helps in the further processing of data.

Many segmentation techniques are based on thresholding to achieve a mask of specified area [12,7]. This operation does not always result in one unified region. Therefore, morphological transformations need to be applied. They consist of moving a structuring element on the image and analyzing its overlapping points. If a match is found, a specific operation is executed that determines the type of transformation. Basic morphological operations are dilation and erosion. For the first one, the whole structuring element is reproduced on the image when the central point is included in the set. This can be expressed mathematically as a vector addition:

$$A + B = \left\{ z : (\hat{B})_z \cap A \neq 0 \right\} \tag{3}$$

For the erosion operation, only one pixel is reproduced if the whole central element is included in the set. Otherwise, the pixel laying under the central point of the structuring element is removed. This can be written as:

$$A - B = \{z : (B)_z \subseteq A\} \quad (4)$$

The erosion and dilation are used as a basis for the operation of closing and opening. In the first one, the erosion is applied after the dilation effecting in closing small gaps and softening the edges. This leads to linking objects in the image. By applying the dilation after the erosion the amount of image details is reduced. Small noise is removed and bigger objects are exposed more. After applying all of those transformations, the mask image can be achieved similar to the one presented in Fig. 2.

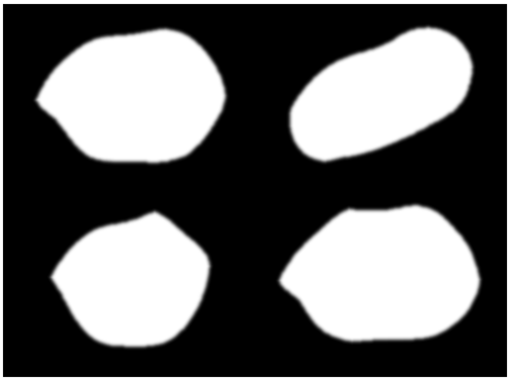


Fig. 2. Mask of segmented lip area

4 CUDA Implementation

All the operations required in the process of lips segmentation are performed in a predetermined sequence, one after another, using Eqns. (1-4) described in Section 3. In order to reduce the data size and speed up future computations, the pseudo-hue image is converted from the RGB color space into the grayscale on the CPU host. Next, we transfer the data to the GPU device, where it is being stored in the texture memory. The process of transferring image data to and from the GPU device is performed only once, without any intermediate transfers between consecutive transformations. Such a practice is common, as communication between the CPU host and the GPU device is a well known bottleneck. Therefore, by reducing the number of transfers we can observe an increase in performance.

Before executing the kernel, we calculate the number of blocks and threads by segmenting the image into numerous sections of rectangular shape. Finally, individual parts are divided between corresponding threads of the thread block and stored in the fast on-chip shared memory. Then consecutive image processing transformations are applied to the pixel block, and the output image is written back to the shared memory for use in the next step, as illustrated in Fig. 3. Consequently, all the threads within the thread block can fetch the data faster when applying new image processing operations. This way we can reduce the number of intermediate image transfers. However, this solution produces some restrictions, as the portions of image data stored in the shared memory can not be synchronized between the thread blocks without the necessity of accessing the global memory. Considering this fact we can utilize the available shared memory space to perform the fixed number of iterations of a filter, set in a given order.

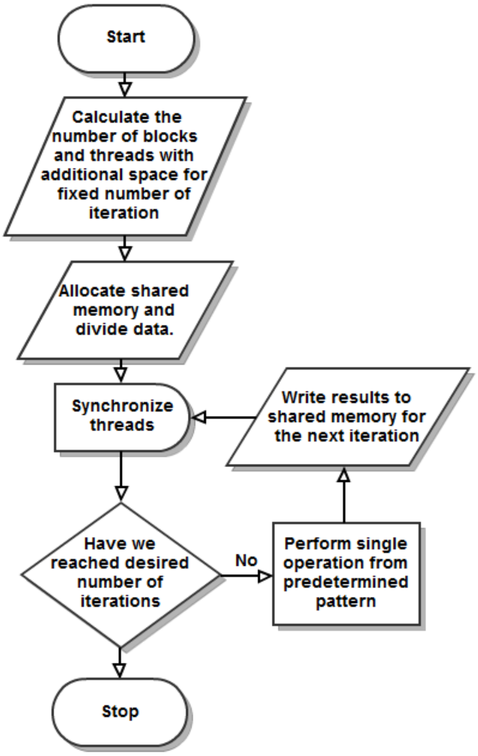

Start
Calculate the number of blocks and threads with additional space for fixed number of iteration
Allocate shared memory and divide data.
Synchronize threads
Write results to shared memory for the next iteration
Have we reached desired number of iterations
No
Perform single operation from predetermined pattern
Stop

Fig. 3. CUDA implementation flow chart with the fixed number of iterations executed on a predetermined pattern of transformations

The CUDA implementation of filters used for this lip segmentation algorithm is fairly straightforward. At first, we assume a constant dimension of both the convolution matrix and the morphological structuring element, which we defined as 3×3. For a single iteration, we need to allocate two additional rows and columns to ensure data accuracy, as every output value is calculated by a single thread by analyzing its neighboring pixels. To illustrate this, let us consider an example where we calculate a single morphological operation. Each thread has to load 9 values in order to designate a single output pixel. But when processing pixels on the boundary of the image block, we need to examine three missing values, as they have not been mapped into the memory. This can be seen in Fig. 4. Given this facts, a shared memory block is allocated with the dimension

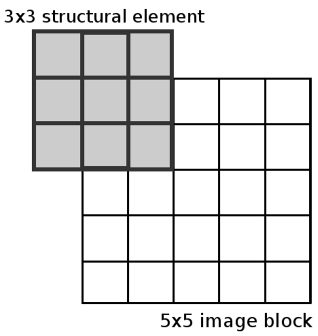


Fig. 4. 3x3 structuring element applied to a 5x5 image block: when processing boundary pixels, we require two additional rows and columns to ensure accuracy of output values

of $(blockDim.x + iterations * 2, blockDim.y + iterations * 2)$, where the total amount of allocated memory space can not exceed 16 KB per block.

In our implementation, we focus on the minimal number of fixed iterations that is required to efficiently create the mask image by means of morphological operation of closing. Because a single transformation consists of one erosion and two dilations, three iterations are required to complete a single closing operation. Additionally the pre-processing and post-processing filters are also applied, which as a result gives us the total number of 20 iterations. After performing the initial load and ensuring accuracy of the transformations, a synchronization function is executed. Then filtering operations are applied to the pixel block, and the output image is written back to the shared memory for the use in the next iteration. After finishing all calculations, the final image is offloaded to the host memory.

In the implementation process, three key issues were taken into account. First, we reduced the number of kernels and their launches. As a result, consecutive image processing transformations in the presented lip segmentation algorithm could be applied in a single kernel call. Second, the image data was processed in the fast shared memory when adding new transformations. Consequently, we decreased the amount of intermediate transfers to the CPU host or the GPU device global memory, as they provide a much slower memory throughput. Third, we reduced the data size, because it affects the general transfer time. Furthermore, we do not need to use high-precision calculations, since in image processing it is not a major factor, and it will not have any significant impact on the final results.

5 Performance Results

For the lip segmentation problem the sequential and parallel implementations were investigated. Visually the GPU results exhibit no differences in relation to the CPU implementation. Sample output images of the lip segmentation are presented in Fig. 5. As testing platforms Tesla C1060 and Intel Core 2 Duo E6550 were chosen. For various resolutions, the lip segmentation was carried out on pre-selected test images. The achieved performance results are included in Table 1.

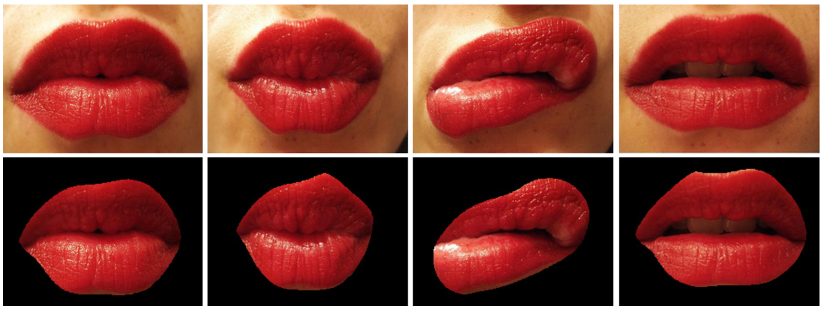


Fig. 5. Results of lip segmentation for selected test images

Table 1. Processing time for GPU and CPU implementation, with achieved speedup for various resolutions

Resolution	GPU [ms]	CPU [ms]	Speedup
640x480	15	35	2.3
800x600	16	56	3.5
1024x768	18	92	5.1
1280x1024	24	155	6.5
1600x1200	31	228	7.4
1920x1200	34	274	8.1

It is worth mentioning that the GPU processing times are considerably lower than results achieved on CPU, without using SSE or MMX instructions. The maximum speedup attained in relation to the sequential implementation reaches 8 times. For 640x480 resolution a single transformation on GPU, excluding the transfer time, takes under 0.20 milliseconds. Therefore, by applying more filtering operations the total GPU processing time does not increase significantly. As shown in Table 1, the speedup increases with every resolution, which indicates that by processing more data on GPU better results can be achieved. It is worth to note that by applying even higher resolution greater speedups can be achieved. Unfortunately, in practice most of capturing devices do not provide higher resolutions than those presented in Table 1.

In real-time image processing, it is necessary to calculate transformation results faster than the sampling frequency. Assuming standard frame rate of 30 frames per second, the processing time should not be higher than 33 milliseconds. In our implementation, such performance requirements were met with Tesla C1060. It confirms that CUDA-enabled graphic cards can be successfully used for the advanced image processing in real time.

6 Conclusions

In this paper we discussed the lip segmentation problem with the use of image processing, which was implemented in a parallel environment using the CUDA architecture. The number of kernels, as well as their launches were reduced, in order to process consecutive image transformations faster. As a result, a single kernel function was created, that used a predetermined filter set in a fixed number of iterations. This approach decreased the amount of intermediate data transfers to the global memory at the expense of additional space in the shared memory. In conclusion, partial results could be fetch faster for filtering operations that used a small structuring element.

Our results have shown a significant speedup of the parallel implementation in relation to the CPU counterpart. By applying consecutive transformations to the same image, the total time of calculations increased slightly. Moreover, the speedup obtained in this way did not decrease. This implies that GPUs can be successfully used in real-time image processing.

References

1. Parallel programming and computing platform CUDA NVIDIA, http://www.nvidia.com/object/cuda_home_new.html
2. Aleksic, P.S., Williams, J.J., Wu, Z., Katsaggelos, A.K.: Audio-Visual speech recognition using MPEG-4 compliant visual features. EURASIP Journal on Advances in Signal Processing 2002, 1213–1227 (2002)
3. Desouza, G.N., Kak, A.C.: Vision for mobile robot navigation: a survey. IEEE Transactions on Pattern Analysis and Machine Intelligence 24, 237–267 (2002)
4. Farrugia, J.P., Horain, P., Guehenneux, E., Alusse, Y.: GPUCV: a framework for image processing acceleration with graphics processors. In: 2006 IEEE International Conference on Multimedia and Expo., pp. 585–588 (2006)
5. Forsyth, D.A., Ponce, J.: Computer Vision: A Modern Approach. Prentice Hall (2002)
6. Fung, J., Mann, S.: OpenVIDIA: parallel GPU computer vision. In: MULTIMEDIA 2005: Proceedings of the 13th Annual ACM International Conference on Multimedia, pp. 849–852 (2005)
7. Guan, Y.: Automatic extraction of lips based on multi-scale wavelet edge detection. IET Computer Vision 2, 23 (2008)
8. Hopf, M., Ertl, T.: Accelerating 3D convolution using graphics hardware. In: Proceedings of the Visualization 1999, pp. 471–564 (1999)
9. Hopf, M., Ertl, T.: Accelerating morphological analysis with graphics hardware. In: Workshop on Vision, Modeling, and Visualization, VMV 2000, vol. 337, pp. 337–345 (2000)
10. Hurlbert, A., Poggio, T.: Synthesizing a color algorithm from examples. Science 239, 482–485 (1998)
11. Liew, A.W., Wang, S.: Visual speech recognition: lip segmentation and mapping. Idea Group Inc. (IGI) (2009)
12. Lucey, S., Sridharan, S., Chandran, V.: Adaptive mouth segmentation using chromatic features. Pattern Recognition Letters (2002)
13. McConnell, J.: Computer Graphics: Theory Into Practice. Jones & Bartlett Pub. (2005)
14. Neti, C., Potamianos, G., Luettin, J., Matthews, I., Glotin, H., Vergyri, D., Sison, J., Mashari, A., Zhou, J.: Audio-visual speech recognition. In: Final Workshop 2000 Report, vol. 764 (2000)

15. Shapiro, L.G., Stockman, G.C.: Computer Vision. Prentice Hall (2001)
16. Shin, J., Lee, J., Kim, D.: Real-time lip reading system for isolated korean word recognition. Pattern Recognition 44, 559–571 (2011)
17. Shirley, P.: Fundamentals of Computer Graphics. CRC Press (2002)
18. Sinha, S.N., Frahm, J., Pollefeys, M., Genc, Y.: Feature tracking and matching in video using programmable graphics hardware. Machine Vision and Applications 22, 207–217 (2007)
19. Strzodka, R., Garbe, C.: Real-time motion estimation and visualization on graphics cards. In: IEEE Visualization 2004, pp. 545–552 (2004)
20. Strzodka, R., Telea, A.: Generalized distance transforms and skeletons in graphics hardware. In: Proceedings of EG/IEEE TCVG Symposium on Visualization (VisSym 2004), pp. 221–230 (2004)
21. Strzodka, R., Ihrke, I., Magnor, M.: A graphics hardware implementation of the generalized hough transform for fast object recognition, scale, and 3D pose detection. In: Proceedings of IEEE International Conference on Image Analysis and Processing (ICIAP 2003), pp. 188–193 (2003)
22. Ujaldon, M., Ruiz, A., Guil, N.: On the computation of the circle hough transform by a GPU rasterizer. Pattern Recogn. Lett. 29, 309–318 (2008)
23. Wang, S., Lau, W., Liew, A.W., Leung, S.: Robust lip region segmentation for lip images with complex background. Pattern Recognition 40, 3481–3491 (2007)
24. Yang, M., You, Z., Shih, Y.: Lip contour extraction for language learning in VEC3D. Machine Vision and Applications 21, 33–41 (2008)
25. Zhao, W., Chellappa, R., Phillips, P.J., Rosenfeld, A.: Face recognition: A literature survey. ACM Comput. Surv. 35, 399–458 (2003)
26. Zhi, Q., Kaynak, M.N.N., Sengupta, K., Cheok, A.D., Ko, C.C.: A study of the modeling aspects in bimodal speech recognition. In: IEEE International Conference on Multimedia and Expo., ICME 2001 (2001)

Material Parameter Identification with Parallel Processing and Geo-applications

Radim Blaheta, Rostislav Hrtus, Roman Kohut,
Owe Axelsson, and Ondřej Jakl

Institute of Geonics AS CR,
Studentska 1768, Ostrava, Czech Republic
owea@it.uu.se, {blaheta,hrtus,kohut,jakl}@ugn.cas.cz

Abstract. The paper describes numerical solution of material parameter identification problems, which arise in geo-applications and many other fields. We describe approach based on nonlinear least squares minimization using different optimization techniques (Nelder-Mead, gradient methods, genetic algorithms) as well as experience with OpenMP+MPI parallelization of the solution methods.

Keywords: Parameter identification, Nelder-Mead, gradient methods, genetic algorithms, parallelization.

1 Introduction, Framework for Parameter Identification

Parameter identification problems deserve nowadays increasing interest. We describe some solution methods with a special interest in their parallelization and two applications – identification of heat conduction coefficients of rocks for analysis of in-situ experiment in underground rock laboratory and identification of local mechanical parameters for modelling of elastic behaviour of geocomposites.

To describe the parameter identification problem, we consider a real process (heat conduction, elastic deformation, etc.), which state is characterized by a *state variable* $u_{real} \in \Phi$ (temperature, displacement, etc.) and depends on a *control variable* f (heat sources, mechanical loads, etc.).

Let us assume that the real process is simulated by an initial–boundary value problem for PDE. Then the input involves some material (and/or control) parameters $p \in \mathcal{P}$ and the control variable f. The simulated state variable $u = u(p)$ fulfils the *state problem*, e.g. the following boundary/initial value problems of heat flow or elasticity:

$$\left.\begin{aligned} c(p)\tfrac{\partial u}{\partial t} - \operatorname{div}\phi &= f \\ \phi &= K(p)\delta \\ \delta &= \nabla u \end{aligned}\right\} \text{ in } \Omega_T \quad \text{or} \quad \left.\begin{aligned} -\operatorname{div}\sigma &= f \\ \sigma &= C(p)\varepsilon \\ \varepsilon &= \tfrac{1}{2}(\nabla + \nabla^T)u \end{aligned}\right\} \text{ in } \Omega \qquad (1)$$
$$+ \text{ initial/boundary conditions} \qquad\qquad + \text{ boundary conditions}$$

Above, for the parabolic heat flow problem (1 left), $\Omega_T = (0, T) \times \Omega$ is the time-space problem domain and u, δ, ϕ, c, K, f denote temperature (the state variable), its gradient, heat flux, heat capacity, heat conductivity (in isotropic case $K = kI$, where k is a positive parameter) and density of heat sources, respectively. For the elliptic elasticity problem (1 right), u, ε, σ, C, f denote displacement, strain, stress, elasticity tensor and density of volume force, respectively.

Assuming that the solution of the state problem is unique for given p, the solution can be expressed via the *parameter–to-solution operator* $S : \mathcal{P} \to \Phi$,

$$u(p) = S(p) = S(p, f)\,. \tag{2}$$

Further, assume that we are given a vector d of data arising from measurement of u_{real} (e.g. values in selected points and time moments, average values over some subdomains, etc.) and also an *observation operator* $D(u)$, which provides a computed counterpart to d. Note that $D(u_{real}) = d - \eta$, where η is a measurement error (noise). Then the *parameter identification problem* can be written in the nonlinear least squares form

$$F(p) = \| DSu(p) - d \|^2 \to \min_{p \in \mathcal{P}}, \tag{3}$$

Note that an implementation of (3) use u as a solution of discretized state problem. Moreover, the objective function F is frequently extended by adding a regularization term. The choice of regularization can be very important as it is discussed e.g. in [15].

In this paper, the solution of the parameter identification problem (3) will be done by application of some optimization technique as Nelder-Mead or other direct search methods, gradient type methods or genetic algorithms. All these methods, briefly described in Section 2, need repeated computations of the state variable $u(p) = S(p)$, or in other words, need to solve the discretized state PDE problem. We shall assume that this is done via some finite element discretization and the parallel iterative solution of the arising FE systems as briefly described in Section 4. We shall use unconstrained optimization methods, as some possible constrains, as positivity or box constraints on parameters, can be handled otherwise, e.g. by a suitable transformation of parameters.

2 Optimization Techniques

In this Section, we briefly describe three types of optimization techniques, which can be used for the nonlinear least squares optimization (3).

2.1 Nelder–Mead Method

The Nelder–Mead method is a typical representant of direct search methods, cf. [9], [13], [10]. The algorithm maintains a simplex $S^{(k)}$ in the space of parameter vectors p. This simplex locally approximates the objective function F and serves for getting information about its behaviour and getting approximation to the optimal point. We start with an initial simplex $S^{(0)}$, which is gradually changed.

The k-th step simplex $S^{(k)}$ is determined by $m+1$ vectors, $m = \dim(p)$, of parameters $p^{(k,\,1)}, \dots, p^{(k,\,m+1)}$ ordered according to the objective function values

$$F(p^{(k,\,1)}) \leq F(p^{(k,\,2)}) \leq \dots \leq F(p^{(k,\,m+1)}).$$

If a stopping criterion (see later) is not fulfilled, then the algorithm continues by finding new vertex in the form

$$p(\mu) = (1+\mu)\bar{p} - \mu p^{(k,\,m+1)},$$

where $\bar{p} = \big((p^{(k,\,1)} + \dots + p^{(k,\,m+1)})/m\big)$ and μ is typically equal to $\mu_r = 1$ for reflection, $\mu_e = 2$ for extension, $\mu_{oc} = 1/2$ for outer contraction and $\mu_{ic} = -1/2$ for inner contraction.

In the standard sequential algorithm, see [9], the procedure starts with evaluation of $F_r = F(p(\mu_r)$. If $F(p^{(k,\,1)}) \leq F_r < F(p^{(k,\,m)})$ then we take $p(\mu_r)$ as a new vertex. Otherwise, we gradually test for the expansion, outside and inside contraction and take the case fulfilling prescribed conditions [9]. It means that forming of a new simplex contains one or more evaluations of the objective function. We can also decide for shrinking the whole simplex, which costs m evaluation of the objective function. The details can be found in [9].

The optimization is stopped when both decrease of the objective function F is small (below ε_F) and changes of parameters are small (below ε_p) or if some limit number of the objective function evaluation is exceeded.

2.2 Gradient Gauss–Newton Type Methods

The objective function $F(p) = \frac{1}{2}F_1(p)$ from (3) has the nonlinear least squares structure, i.e. without regularization, we have

$$F(p) = \frac{1}{2}R(p)^T R(p), \tag{4}$$

where $R(p)$ is the residual defined by $R(p) = DS(p) - d$.

We shall consider methods exploiting Jacobian of the mapping $p \to R(p)$,

$$J(p) = D_p R(p) = (J_{ij}(p))\,,\ J_{ij}(p) = \frac{\partial R_i(p)}{\partial p_j},$$

The knowledge of J enables to express the gradient $\mathrm{grad} F(p) = J(p)^T R(p)$ and introduce gradient methods of the type

$$p^0 \text{ given };\ p^{i+1} = p^i - \alpha_i z^i,\ i \geq 0, \tag{5}$$

where

$$z^i = g^i \text{ or } z^i = H_i^{-1} g^i, \quad g^i = J(p^i)^T R(p^i), \tag{6}$$

$$H_i = \nu_i I + J(p^i)^T J(p^i). \tag{7}$$

Note that the choice $z^i = g^i$ provides the steepest descent method, $z^i = H_i^{-1} g^i$ provides the Gauss–Newton method for $\nu_i = 0$ and Levenberg-Marquardt method for $\nu_i > 0$. We can use $\alpha_i = 1$, but more robust is to take α_i by a line search or backtracking, see [9], [6], [13]. Positive values of ν_i guarantee positive definiteness of H_i and regularize the problem but can destroy the local quadratic convergence. Therefore, a suitable strategy for the choice of ν_i is needed, as e.g. $\nu_i = \min\{1;\ c \parallel J(p^i)^T R(p^i) \parallel\}$, where c is a positive constant, cf. [6].

2.3 Genetic Algorithms

Another class of methods, which need only evaluation of the objective function F, is created by genetic methods. These methods use the following framework,

GA with $N = N_{GA}$ sample population

(1) generate N random vectors $p^{(i)},\ i = 1, \dots, N$
(2) for a given generation, evaluate $F_i = F(p^{(i)})$, if F_i is not known yet
(3) select τN parameter vectors $p^{(i)}$ with smallest values F_i, so called parents. Then create $(1-\tau)N$ new vectors (children) by crossing randomly selected parents and evaluate their objective function
(4) create a new population $p^{(i)}_{new}$ by taking the selected parents and created children. Mutate all $p^{(i)}_{new}$ to $p^{(i)}_{mut}$ and evaluate $F(p^{(i)}_{mut})$. If $F(p^{(i)}_{mut}) < F(p^{(i)}_{new})$ then $p^{(i)} = p^{(i)}_{mut}$ else $p^{(i)} = p^{(i)}_{new}$
(5) evaluate stopping test and GOTO (3) if results are still not satisfactory.

In our case, the crossing and mutation acts on parameter vectors and can be described as algebraic (not binary) rule, see e.g. [8], [12] for the details.

3 Geo-applications

This section presents two applications, which motivate our interest. Both examples are from the field of geoengineering, which comprises also many other interesting applications, e.g. the use of identification procedures in tunnel engineering, soil mechanics, subsurface water flow and remediation, finding parameters for description of nonlinear behaviour of rocks etc.

3.1 APSE – Heat Conduction Problem

The in-situ Äspö Pillar Stability Experiment (APSE) was performed with the aid of investigation of granite mass damage due to mechanical and thermal loading. APSE used electrical heaters to increase temperatures and induce stresses in a rock pillar between two holes (Fig. 1) until its partial failure. To determine accurately the temperature changes, a heat flow model is formulated and monitored temperatures are used for model calibration via obtaining parameters taking into account water content and water flow in the rock. More details and another approach to the model calibration can be found in [1].

Monitoring of the temperatures during two month heating phase of APSE provides a vector d containing 168 data items – temperatures at 14 monitoring

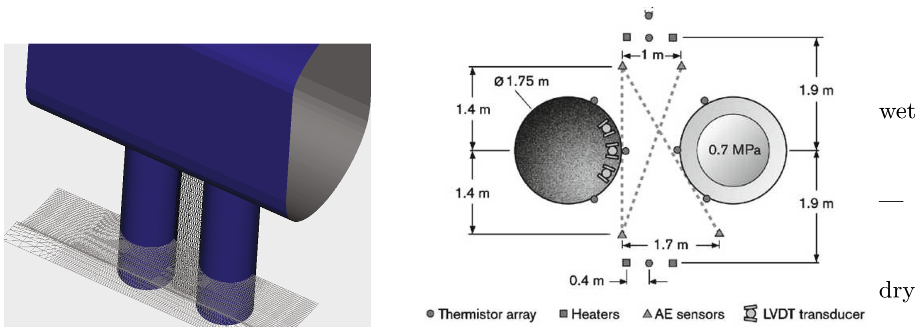


Fig. 1. The APSE model – detail of the FE grid around the pillar (GEM software [4]) and ground view on the pillar, holes, location of heaters and points of temperature measurement

positions in 12 time moments. The observation operator D picks–up the same data (temperatures) from the computed solution of the state problem.

The material parameters are variable conductivity k and heat capacity c, which are assumed to be constant in two subdomains – the dry and wet sides according to Fig. 1. It gives optimization with 4 parameters. Let us mention that some variants with more parameters will be tested in a near future.

The state problem is solved by the GEM software [4] and considers a computational domain of $105 \times 125 \times 118$ m and tetrahedral 3D mesh with $99 \times 105 \times 59$ nodes. The grid is refined around the pillar, see Fig. 1. The heaters are producing heat which varies in time. The model assumes original temperature $14.5°C$ on the outer boundaries, zero flux onto the tunnel and nonzero flux given the convection onto the holes. The initial condition is given again by the original temperature $14.5°C$. The computational results are discussed in Section 4.3.

3.2 Geocomposites – Local Elastic Properties

The second application concerns coal geocomposites, which arise from polyurethane resin grouting to the coal seam for improving its mechanical properties. To assess the influence of various resins, various levels of fill–in of the fractures etc., the properties of smaller $7\times7\times7$ cm cubic samples are tested both experimentally and analytically. The analytical tests use finite element analysis on dense voxel grids with information about the material microstructure obtained from CT scanning and assigning the local material properties. The local properties are obtained from extra experimental testing in the case of resin with different level of foaming.

The local properties of coal (parameters) are identified with the aid of effective elastic moduli E_1^{exp} and Poisson ratio ν^{exp} obtained from experimental testing on the cubic samples. In this case, the vector of data d contains values of stress and strain tensors corresponding to the behaviour of homogeneous sample with given elastic moduli E_1^{exp} and Poisson ratio ν^{exp}. Correspondingly, the observation operator D is taken as

$$Du(p) = D(p)u(p) = \begin{bmatrix} \langle (\, C(p)\varepsilon(u)\,)_{ij} \rangle \\ \langle \varepsilon_{ij}(u) \rangle \end{bmatrix}, \quad i,j = 1,2,3.$$

Note that $\langle \cdot \rangle$ is the operator of volume averaging over $\Omega_{coal} \subset \Omega_{sample}$. Both stress and strain fields are taken into account because one of them can be fully independent from the Poisson's ratio ν^{exp}, but generally the dependence on the Poisson's ratio ν is weak. The elasticity tensor $C(p)$ is assumed to be either isotropic or orthotropic.

4 Parallel Processing

We shall distinguish two levels of parallelization of the parameter identification methods:

1. first level of parallelization concerns the solution of the state (forward) PDE problems for evaluation of the objective function,
2. second level of parallelization concerns the adopted optimization method.

Note that we decided to use OpenMP for the first and MPI parallelization for the second level as it seems to be natural for parallelization on different levels and fitting the structure of the present multiprocessor/multicore computers.

4.1 Parallelization of the Forward Problem Solution

The solution of the (forward) state problem is the most expensive part of the parameter identification. It assumes discretization of the PDE problems, which can be done by the finite element (FE) method in space and by the finite differences in time. The discretization leads to the solution of large scale linear algebraic systems, which can be solved iteratively by the preconditioned conjugate gradient or some other methods, cf. [2]. Various domain decomposition techniques can be used for parallelization of both assembling of matrices and solution of the linear systems, see [14]. Particularly, we are using Schwarz type domain decomposition methods as described in [3], [4].

Figure 2 outlines our implementation of the solution of time dependent heat flow problem described in (1) and used in application in Subsection 3.1. It uses discretization by linear finite elements in space and time discretization by implicit Euler method with constant time step Δt (this assumption is not significant, but allows to perform incomplete factorization outside of the time stepping loop).

The full solution procedure (Fig. 2) has sequential and parallelizable parts. At present implementation, we skip parallelization of the assembly of FE matrices and concentrate on parallelization of the preconditioned CG iterations in Step 5. In Steps 5(a) and 5(b)i, we use decomposition of the matrix by rows for parallel matrix by vector multiplication, number of processors is equal to number of subdomains. Note that this decomposition by rows can be a bit better balanced comparing to the domain decomposition used in Step 5(b)ii. Parallel do loops are also used at Step 5(b)iii.

Heat flow evolution - computation of $\underline{u}_h^k$ - vectors of temperatures for time levels.

1. assembly the capacity and conductivity matrices $M_h = M_h(p)$ and $A_h = A_h(p)$,
2. decompose matrices M_h, A_h according to overlapping DD (domain is divided into subdomains (slices) in one coordinate direction). Let $M_{h,j}^{\delta}$, $A_{h,j}^{\delta}$ are the arising subdomain matrices, $j = 1, \ldots, ns$, where ns is the number of subdomains.
3. perform incomplete factorization $L_j L_j^T$ of the matrices $M_{h,j}^{\delta,L} + \Delta t A_{h,j}^{\delta}$, where $M_{h,j}^{\delta,L}$ is the diagonal matrix having rowsums of $M_{h,j}^{\delta}$ on the diagonal (lumped matrix). Lumping ensures that the incomplete factorization is stable even for small Δt.
4. set the initial temperature u_h^0
5. perform loop over time steps $k = 1, \ldots, nts$
   (a) compute rhs as $\Delta f^k = \Delta(b^k - A_h \underline{u}_h^{k-1})$,
   (b) perform iterative solution of $(M_h + \Delta t A_h)\Delta \underline{u}_h^k = \Delta f^k$ by the preconditioned CG method up to prescribed accuracy ε with $\Delta u_h^{k,0} = 0$ as the initial guess. Each iteration contains of the following parts:
      i. matrix by vector multiplication using $M_h + \Delta t A_h$,
      ii. application of additive Schwarz preconditioner $\sum R_j^T (L_j L_j^T)^{-1} R_j$, where R_j is the proper restriction,
      iii. two inner product and three vector updates.
   (c) $\underline{u}_h^k = \underline{u}_h^{k-1} + \Delta u_h^k$

Fig. 2. Solution procedure

4.2 Straightforward Parallelization of the Optimization Techniques

The standard *Nelder-Mead method* represents a sequential algorithm. But if we concentrate only on the wall clock time (ignoring the overall computing effort) then some look ahead strategy can be used. For example, in each iteration, we can perform simultaneously evaluation of the reflexion, extension and contraction phases. Then, we select the most suitable case and leave others without use in the next progress. Some more elaborated parallel variants of the algorithm [7], [11] use deeper look ahead strategies. This can additionally speed up the iterations and make the optimization more robust.

The *gradient methods* can compute the Jacobian $J(p)$ from differences

$$J_{ij}(p) = \frac{1}{\delta}\left[R_i(p + \delta e_j) - R_i(p)\right] \text{ or } J_{ij}(p) = \frac{1}{2\delta}\left[R_i(p + \delta e_j) - R_i(p - \delta e_j)\right].$$

This approach is used especially for smaller dimension m, otherwise a semianalytic differentiation can be used, which reduce the cost and increase accuracy of gradient computation, see [15]. But for differences, the computer time for $m+1$ or $2m+1$ evaluations of the residual $R(p)$ can be straightforwardly reduced by parallelization.

The *genetic* algorithms are a natural candidate for parallelization, which concerns concurrent evaluation of the objective functions for the whole population. We can also use either single population or multiple populations, see [5].

Table 1. Computing times [sec] and numbers of iterations for steps 4 and 5 of the full solution procedure (Fig. 2) for 560 timesteps and 613 305 DOF's. The computing time for sequential execution of steps 1, 2, 3 is 12 sec.

# proc.	# all CG iter.	total time	time para. part	speedup para. part	speedup 1 iter.	time for precond.	mat-vec mult. time
1	1008	360.9	357.2	-	-	195.5	86.7
2	973	203.4	200.0	1.786	1.723	116.3	48.2
4	981	129.4	125.9	2.837	2.761	71.3	29.3
8	1153	104.8	101.0	3.537	4.045	50.6	28.6

(a) $\varepsilon = 0.01$

# proc.	# all CG iter.	total time	time para. part	speedup para. part	speedup 1 iter.	time for precond.	mat-vec mult. time
1	3162	774.6	770.7	-	-	464.7	207.5
2	3417	515.8	512.3	1.504	1.626	309.7	140.9
4	3464	335.9	332.1	2.321	2.542	206.4	88.9
8	4014	252.2	248.2	3.105	3.942	155.2	64.0

(b) $\varepsilon = 0.0001$

4.3 Numerical Experiments

All numerical experiments are carried out in the GEM code environment [4]. In the framwork of GEM, all three types of optimization techniques have been realized and we describe test results for solving the heat flow problem described in Subsection 3.1.

First (see Table 1), we tested the OpenMP parallelization of the forward heat evolution problems. The test runs are performed at Hubert - a shared-memory machine with eight four-core AMD Opteron 8380 processors (32 cores in total), HyperTransport communication technology, 128GB of RAM, SLES 10 operating system, MPI implementation and Fortran compiler with OpenMP support through the Intel Cluster Toolkit with Compilers (ICT). From Tables 1(a,b), we can see that (1) the number of iterations depends weakly on the number of subdomains (cores) and is about 2 and 6 iterations per time step (depending on the adopted accuracy ε), (2) the computing time for the parallelized part of the whole algorithm is dominant - more than 90%, (3) parallel speedup, even if it is related to one iteration, is reasonable only for 2 and 4 cores, then deteriorates, (4) within the CG iterations, the application of the preconditioner is about twice more expensive than matrix by vector multiplication.

Second (see Table 2), we tested the whole parameter identification procedure for the same application using different optimization methods. All these methods started the iterative process from the same initial values, generated by random, and stopped as soon as the objective function value became less than a prescribed constant. Although it is not fair to compare the methods in this way, the times can provide some idea about their efficiency. The tests were performed on the Kalkyl cluster at UPPMAX computing centre at Uppsala University, which

Table 2. Real time of solution of the parameter identification APSE problem with increasing number of cores employed in the forward solution

# threads	NM solution [s]	GM solution [s]
1	22333	5783
2	17113	4216
4	14912	3620
8	14973	3224

(a) Nelder-Mead (NM) and gradient (GM) methods, only OpenMP parallelization.

	population size N	#processes x #threads	GA solution [s]
1	20	1 x 8	51900
2	20	20 x 8	4478
3	40	40 x 8	2653

(b) Genetic algorithm with OpenMP + MPI parallelization.

has 348 HP SL170h G6 compute nodes (each consisting of two quad-core Intel Xeon 5520 processors) interconnected with a 4:1 oversubscribed DDR Infiniband fabric. All computations used OpenMP parallelization of the forward problem solution.

From Tables 2(a,b), we can see that (1) efficiency of using more cores is again low, the speedup is below two even for eight cores, (2) straightforward parallelization of the GA is close to the theoretical speedup (remember that only one half of the population is changed), which is expected, but not so evident as some computer resources are still shared.

5 Conclusions

The paper describes some methods suitable for solving parameter identification problems and their OpenMP+MPI parallelization. We could see that

– gradient methods seem to be most efficient, but other methods can be useful for solving problems with more parameters and complicated materials,
– our OpenMP parallelization provided a bit low speedup, which deserves attempt for improvement by considering problems of data locality and memory access. Note that our past experience with MPI parallelization of the forward problem was considerably better [3],
– the straightforward parallelization of some optimization methods is efficient, but not scalable,
– further experiments show that gradual increase of accuracy during optimization can provide important saving of computational work. In this respect PDE-constrained optimization approach seems to be promising.

Acknowledgment. This paper has been supported by the project GA CR 105/09/1830 of the Grant Agency CR and has been elaborated in the framework of the IT4Innovations Centre of Excellence project, reg. no. CZ.1.05/1.1.00/02.0070 supported by Operational Programme 'Research and Development for Innovations' funded by Structural Funds of the European Union and state budget of the Czech Republic. Some computations were done within UPPMAX p2011076 project.

References

1. Andersson, J.C., Fälth, B., Kristensson, O.: Äspö pillar stability experiment TM back calculation. In: Advances on Coupled Thermo-Hydro-Mechanical-Chemical Processes in Geosystems and Engineering, pp. 675–680. HoHai University, Nanjing, China (2006)
2. Axelsson, O.: Iterative Solution Methods. Cambridge University Press, Cambridge (1994)
3. Blaheta, R., Kohut, R., Neytcheva, M., Starý, J.: Schwarz methods for discrete elliptic and parabolic problems with an application to nuclear waste repository modelling. Mathematics and Computers in Simulation 76, 18–27 (2007)
4. 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. LNCS, vol. 6067, pp. 266–275. Springer, Heidelberg (2010)
5. Cantu-Paz, E.: A Survey of Parallel Genetic Algorithms. Calculateurs Paralleles, Reseaux et Systems Repartis 10(2), 141–171 (1998)
6. Dennis, J.E., Schnabel, R.B.: Numerical Methods for Unconstrained Optimization and Nonlinear Equations. SIAM, Philadelphia (1996)
7. Dennis, J.E., Torczon, V.: Direct search methods on parallel machines. SIAM J. Optimization 1, 448–474 (1991)
8. Haslinger, J., Jedelsky, D., Kozubek, T., Tvrdik, J.: Genetic and Random Search Methods in Optimal Shape Design Problems. Journal of Global Optimization 16, 109–131 (2000)
9. Kelley, C.T.: Iterative Methods for Optimization. SIAM, Philadelphia (1999)
10. Kolda, T.G., Lewis, R.M., Torczon, V.: Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods. SIAM Review 45, 385–482 (2003)
11. Lee, D., Wiswall, M.: A Parallel Implementation of the Simplex Function Minimization Routine. Comput. Econ. 30, 171–187 (2007)
12. Mhlenbein, H., Schlierkamp-Voosen, D.: Predictive models for the breeder genetic algorithm, I. Continuous parameter optimization. Evolutionary Computation 1(1), 25–49 (1993)
13. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer (2006)
14. Toselli, A., Widlund, O.: Domain Decomposition Methods - Algorithms and Theory. Springer, Berlin (2005)
15. Vogel, C.R.: Computational Methods for Inverse Problems. Frontiers in Applied Mathematics, vol. (23). SIAM, Philadelphia (2002)

Hierarchical Parallel Approach in Vascular Network Modeling – Hybrid MPI+OpenMP Implementation

Krzysztof Jurczuk[1], Marek Kretowski[1], and Johanne Bezy-Wendling[2,3]

[1] Faculty of Computer Science, Bialystok University of Technology, Wiejska 45a, 15-351 Białystok, Poland
[2] INSERM, U642, Rennes, F-35000, France
[3] University of Rennes 1, LTSI, Rennes, F-35000, France
{k.jurczuk,m.kretowski}@pb.edu.pl

Abstract. This paper presents a two-level parallel algorithm of vascular network development. At the outer level, tasks (newly appeared parts of tissue) are spread over processing nodes. Each node attempts to connect/disconnect its assigned parts of tissue in all vascular trees. Communication between nodes is accomplished by a message passing paradigm. At the inner level, subtasks concerning different vascular trees (e.g. arterial and venous) within each task are parallelized by a shared address space paradigm. The solution was implemented on a computing cluster of multi-core nodes with mixed MPI+OpenMP support. The experimental results show that the algorithm provides a significant improvement in computational performance compared with a pure MPI implementation.

1 Introduction

The continuously increasing need for computing power and physical and economic limitations of processor frequency scaling (i.e. significant increase of costs and energy usage) caused that parallel machines have become the only known alternative to improve computing performance [1]. Firstly, the need of high performance arises from the necessity to solve problems of ever-rising size at the limits of available computing resources. Moreover, parallel computing seems to be more suitable to mimic natural world processes that may happen in the same time and are quite often interrelated with each other. Thus, parallel processing appears to be one of the most relevant issues in modern scientific computing [2].

In our previous studies, we developed a physiological model reflecting both morphology and functions of vascular networks in clinical images [3]. The solution consists of a macroscopic model (vascular network and pathological anomalies) and a microvascular model (blood flow simulation in capillaries and contrast agent diffusion processes [4]). Moreover, we coupled this two-level vascular model with imaging CT and MR simulators. Such a model-based approach represents a non-invasive way to control physiological parameters, what would be difficult or even impossible to do in real experiments. The whole solution constitutes a good way to improve the interpretation of dynamic medical images by linking image descriptors with morphological and functional perturbations, thus offering the potential to reveal early image indicators of pathologies.

In the model, the structure of vascular networks is obtained in the process of vascular development. Initially, we proposed a sequential algorithm [3]. Although we applied many algorithm and code optimizations, the simulation was still a time expensive process. Later, we introduced the distributed memory algorithm that parallelized the vascular development [5] (message passing interface (MPI) [6] implementation on computing cluster). Moreover, we proposed an advanced modeling framework [7] able to efficiently simulate vascular development on a computing cluster (distributed memory approach) as well as on low-cost multi-core desktop machines (shared memory approach - OpenMP [8] implementation).

Although in all our previous distributed memory algorithms we were able to gain substantial speedups on computing clusters of nodes with single-core chips, nowadays, the multi-core chips in clusters seem to be an industrial trend. Moreover, a combination of shared memory and message passing paradigms in one application may provide a better efficiency than e.g. pure MPI version [9]. Therefore, in this paper, we propose a two-level hybrid parallel algorithm of vascular development, that employs both shared address space and message passing paradigms on a cluster of nodes with multi-core chips (mixed MPI+OpenMP implementation). The main aim of this work is to further accelerate the simulation process, which will increase the possibility to create more elaborate and precise vascular models. Furthermore, our intention is to bring the model closer to reality in which processes of vascular development are performed inherently in a parallel way [10]. Many other vascular system have been proposed (e.g. [11], [12]), however, as far as we know, all the previous solutions, published by other authors, have been using only sequential approaches of vascular development. On the other hand, one can find at least several other applications of hybrid parallel modeling in computational biology and medicine, e.g. in PET image reconstruction algorithms [13] or in cardiac simulations [14].

In the next section, the vascular model and sequential algorithm of vascular development are recalled. In Sect. 3 the hybrid parallel algorithm of the vascular development is explained. An experimental validation is performed in Sect. 4. Conclusion and some plans for future research are sketched in the last section.

2 Vascular Model Description

In this paper, we focus on vascular development algorithms on the macroscopic level. Therefore, this section describes basic features of the macroscopic model. Then the sequential algorithm of vascular development is recalled.

In the macroscopic model we can distinguish two main components: the tissue and the vascular network [3]. The tissue is represented by a set of Macroscopic Functional Units (MFU) that are distributed inside the specified organ shape. An MFU is a small, fixed size part of the tissue and is characterized by a class that determines most of its structural/functional properties (e.g. size, probability of mitosis and necrosis) and also physiological features (e.g. blood pressures, blood flow rate). Several classes of MFUs can be defined to differentiate functional or pathological regions of tissue (e.g. normal, tumoral). Moreover, the class of MFU can be changed over time, which makes it possible to simulate the evolution of a disease (e.g. from benign nodule to malignant tumor).

2.1 Vascular Network

The model expresses the specificity of the liver, although most of its features are not linked with any specific organ. The liver stands out from other organs by the unique organization of its vascular network that consists of three vessel trees. Hepatic arteries and portal veins deliver blood to tissue, whereas, the hepatic venous tree is responsible for blood transport back to the heart.

In the model, each vascular tree is composed of vessels that can divide creating bifurcations. A vessel segment (part of vessel between two consecutive bifurcations) is represented by an ideal, rigid tube with fixed radius, wall thickness and position. The model distinguishes vessels larger than capillaries, while the capillaries themselves are hidden in the MFUs (micromodel [4]). Blood is transferred from hepatic arteries and portal veins to hepatic veins through MFUs. Vessel intersections (anastomosis), which occur particularly among vessels with very small radii or in pathological situations, are not taken into account. As a result, each vascular tree forms a binary tree.

In the model, the blood is treated as a Newtonian fluid (with constant viscosity) and its flow is governed by Poiseuille's law. Moreover, the vessels' parameters (e.g. radius, blood flow) are calculated according to two following physical laws. At each bifurcation the law of matter conservation is observed, i.e. the quantity of blood entering and leaving a bifurcation is the same. Second constraint deals with the decreasing vessel radii in the vascular tree, creating a morphological dependency between the radius of a vessel and radii of its two descendants.

2.2 Sequential Algorithm of Vascular Network Development

The algorithm begins with the model initialization [3]. Few vessels are placed in the 3D shape of an organ whose size is a fraction of the adult one. Afterwards, in discrete time moments (called cycles), the organ enlarges its size (growth phase) until it reaches its full, mature form. Additionally, each cycle consists of subcycles during which each MFU can divide and give birth to a new MFU of the same class (mitosis process) or die (necrosis process). The processes of mitosis and necrosis are repeated in consecutive subcycles until spaces appearing during the growth phase are filled by new MFUs.

The appearance and development of new vessels are induced by new MFUs that are initially not perfused by the existing vascular system. As a result, for each new MFU a fixed number of the closest/candidate vessels (in each tree) is found. Then each candidate vessel creates a new bifurcation that temporarily perfuses the MFU, i.e. new temporary vessels are sprouted. The spatial position of the bifurcation point is controlled by local minimization of additional blood volume needed for the MFU perfusion.

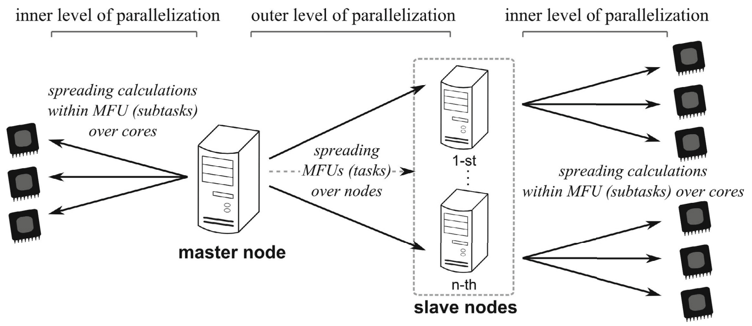

inner level of parallelization
outer level of parallelization
inner level of parallelization
spreading calculations within MFU (subtasks) over cores
spreading MFUs (tasks) over nodes
master node
1-st
n-th
slave nodes
spreading calculations within MFU (subtasks) over cores

Fig. 1. The two-level parallel perfusion process. At the outer level, new MFUs are spread over nodes and each node is responsible for searching optimal bifurcation points. At the inner level, within each MFU calculations concerning different vascular trees are delegated to different cores.

Only one candidate vessel from each tree can finally be designated to perfuse the new MFU. Therefore, the algorithm tests all possible combinations of candidate vessels (a single combination consists of one vessel from each tree). Since only non-crossing vessels can be accepted, the algorithm firstly detects intersections between vessels coming from the same tree or from two different trees (e.g. between arteries and veins) and rejects theses vessels. Finally, the combination with the lowest sum of volumes is chosen to permanently perfuse the MFU and a recalculation of vessels' characteristics in all vascular trees is performed.

In each subcycle, after the reproduction (mitosis and perfusion processes), the algorithm goes to the degeneration phase. Based on the necrosis probabilities of individual MFUs, some of them can die (necrosis process). Next, all the vessels supplying these MFUs retract and disappear (retraction process), and the algorithm goes back to the reproduction phase again.

3 Hybrid Parallel Algorithm of Vascular Development

At the beginning, the general idea of the algorithm is described. Then we present in more detail parallel perfusion and retraction algorithms. The main attention is focused on the perfusion phase since it is the most time consuming part of the algorithm (from 70% to 95% of the total simulation time).

The hybrid algorithm applies a hierarchical (two-level) parallelism. At the outer level, tasks are carried out by pool of processes running on different processing nodes. Interactions between the nodes are accomplished by the message passing paradigm with the master-slave model [2]. At the inner level, parallel subtasks are spread over threads running on different cores and the shared memory space paradigm is exploited. For the sake of explanation clarity, we neglect the mapping of threads/processes to cores/processing nodes and assume that one node is identified with one process and one core is identified with one thread.

During the whole simulation, each processing node has its own copy of the organ. Therefore, at the beginning, the master node broadcasts the initial vascular system and tissue providing the same starting information for all slave nodes. After that, cycle and subcycles are iterated until the adult organ is obtained.

Each subcycle starts from a sequential mitosis performed at the master node. Next, the two-level parallel perfusion is performed (see Fig. 1). The new MFUs that are created during the mitosis are spread over slave nodes (outer level of parallelization). Moreover, calculations within each single MFU are divided between cores (inner level of parallelization). After the perfusion process, the degeneration phase follows. At the beginning, at the master node the sequential necrosis is carried out. Next, the two-level parallel retraction is performed.

3.1 Two-Level Hybrid Parallel Perfusion Algorithm

The outer level parallelization of the perfusion is based on the distributed memory approach of vascular development [5]. After the sequential mitosis, the master node spreads new MFUs (tasks) over slave nodes (see Fig. 1) and then this node is responsible for managing the perfusion process. When it receives a message with an optimal bifurcation of one of the new MFUs, it takes a decision about permanent perfusion. Communication latency and independent work of slave nodes cause that vascular networks at individuals nodes can be slightly different (tree nonuniformity). Therefore, the master node searches, in its vasculature, the vessels related with the proposed optimal bifurcation (vessels to form the optimal bifurcation). If at least one of these vessels cannot be found, then the MFU is rejected. But in the other case, the new MFU is permanently perfused and all organ changes related with the new MFU are broadcasted over slaves.

As regards the slave nodes, each of them is responsible for searching optimal bifurcation points to perfuse the received new MFUs. Each time, when the search ends successfully, the optimal bifurcation parameters are sent to the master node. Next, if there are any queued messages with permanent organ changes broadcasted by the master node, the slave node applies these changes and continues to perform its remaining tasks. Moreover, when the master node is under-loaded (e.g. as a result of small number of slave nodes), it can also perform calculations to find parameters of optimal bifurcation points [7].

The inner level parallelization of the perfusion, both at the master and slave nodes, introduces a possibility to divide calculations concerning single MFU (see Fig. 1). Individual cores are responsible for the calculation concerning different vascular trees (i.e. hepatic arteries, portal veins or hepatic veins in liver).

In the case of the master node, there are two algorithm blocks (i.e. making decision about permanent perfusion and permanent perfusion) during which individual cores are responsible for calculations in different vascular trees. When the master node receives a message with an optimal bifurcation to perfuse a new MFU, each core tries to find, in its assigned tree, the vessel that may create the proposed optimal bifurcation. If all cores find such vessels, the new MFU is permanently perfused in a parallel way in all vascular trees based on the information from the optimal bifurcation. However, if at least one of the cores cannot find the vessel, the other cores abandon their jobs and the MFU is rejected.

In the case of the slave nodes, the inner parallelization is applied to spread calculations within each new MFU (see Fig. 1) during following subtasks: searching of optimal bifurcations, choice of one optimal bifurcation and permanent perfusion. At first, each core of a slave node searches the candidate vessels in its vascular tree, then it creates one optimal temporary bifurcation to each found candidate vessel and recalculates tree characteristics, taking into account the new vascular structures for these temporary bifurcations. Afterwards, all possible combinations of candidate vessels (a single combination consists of one vessel from each tree) are created, sorted according to increasing volume and then tested. Each time, to determine the volume of a verified combination, calculations in different vascular trees are divided between cores. Moreover, the inner parallelization is applied when slave nodes perform permanent perfusions as a result of organ changes broadcasted by the master node.

3.2 Two-Level Hybrid Parallel Retraction Algorithm

In comparison to our previous solutions [5], [7], in the hybrid one, we decided to pay more attention to the retraction phase. The profiling results showed that the time needed for that part of the algorithm is too short to apply a sophisticated message passing strategy. However, in order to create a possibility to gain higher performance (especially in the context of Amdahl's law of a maximum attainable speedup [15]), we decided to apply here a naive message passing strategy at outer level and shared address space paradigm at inner level.

After the sequential necrosis, the master node broadcasts to all other nodes identifiers of the MFUs that have to be removed. Then, the entire algorithm of retraction is performed at each node. As regards the inner parallelization, both the master and slave nodes spread calculations concerning different vascular trees over cores. Hence, the MFUs are disconnected concurrently in all vascular trees.

4 Experimental Results

This section focuses on performance analysis. The presented mean results come from experiments on the vascular development algorithm starting from small size configurations (about 1000 MFUs) and ending on large configurations (about 50000 MFUs and consequently 300000 vessel segments). Figure 2 shows a visualization of one of the simulated vascular networks.

The solution was implemented in C++ with support MPI [6] and OpenMP [8] interfaces. At the outer level of parallelization, the MVAPICH2 version 1.6.1 as MPI2 implementation over infiniband networks was used. The OpenMP was exploited at the inner level of parallelization. The Intel C++ Compiler 10.1 was used. For performance measuring we made use of the Multi-Processing Environment (MPE) library with the graphical visualization tool Jumpshot-4 [6].

Two computing clusters were used. The first one consisted of sixteen SMP nodes running on Linux and connected by an Infiniband network. Each node was equipped with two single-core chips (two 64-bit Xeon 3.2GHz CPUs) with

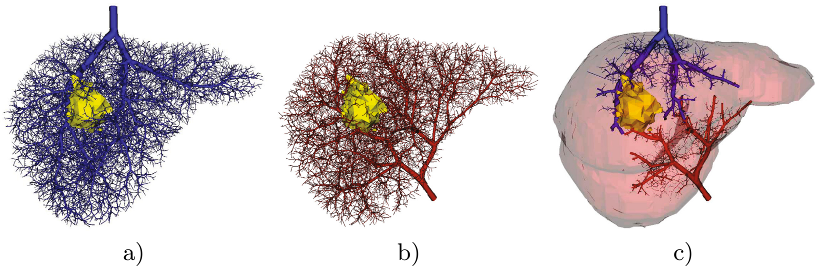


Fig. 2. Visualization of an adult liver (about 50000 MFUs and 300000 vessel segments): a) hepatic veins with a tumor shape, b) portal veins with a tumor shape, c) main hepatic arteries, portal veins and hepatic veins with liver and tumor shapes

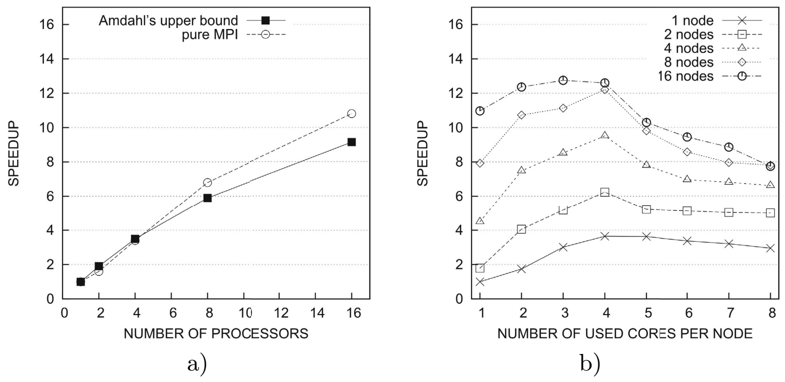


Fig. 3. Speedup of the pure MPI version on the cluster of: a) single-core dual-socket SMP nodes, b) multi-core dual-socket SMP nodes

2MB L2 cache, 2GB of RAM and an Infiniband 10GB/s HCA connected to a PCI-Express port. The second cluster was also built with sixteen SMP nodes but each node was equipped with two multi-core chips (two 64-bit quad-core Xeon 2.66 GHz CPUs) with 2MB L2 cache and 8GB of RAM.

Figure 3a shows the mean speedup of the distributed memory algorithm [5] (pure MPI implementation) on the computing cluster of single-core nodes. It is clearly visible that good performance was obtained. Moreover, we were able to obtain the speedup better than its upper bound value based on Amdahl's law [15]. It results from the introduced periodical memory reallocation mechanisms improving memory cache usage. However, the same algorithm running on the computing cluster of multi-core nodes tends to decrease its performance if too many cores inside each node are involved in computations (see Fig. 3b). Profiling results showed that it is caused by intra-node memory traffic that increases the mean time of message passing and the time of searching optimal bifurcations.

The performance of the proposed two-level hybrid solution is presented in Fig. 4. The outer level parallelization is accomplished by MPI processes, while the inner level one by OpenMP threads. We verify two cases: i) one MPI process (three

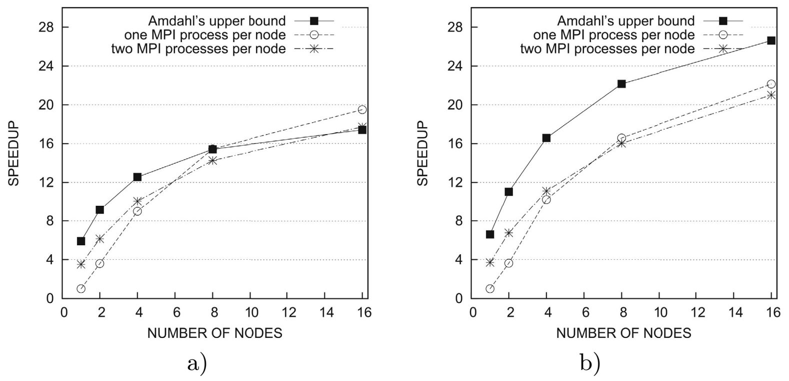


Fig. 4. Speedup of the two-level hybrid version on a cluster of multi-core SMP nodes: a) without and b) with two-level parallelization during retraction

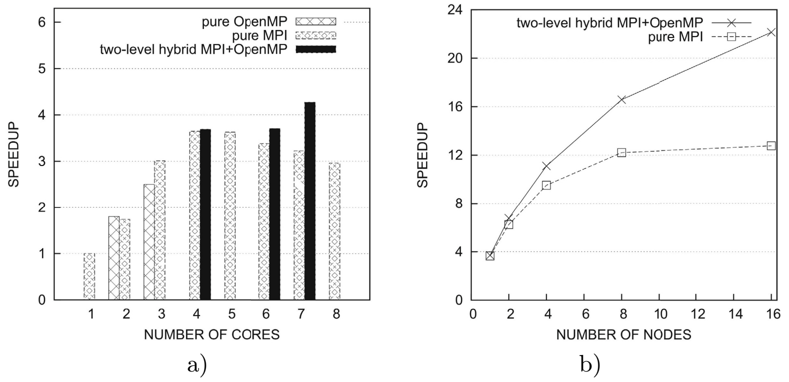


Fig. 5. Performance comparison of pure MPI, pure OpenMP [7] and hybrid MPI-OpenMP versions on the cluster of multi-core SMP nodes: a) speedup within one node, b) speedup across nodes. Different configurations in the hybrid version within one node: four cores (master process - one thread, slave process - three threads), six cores (master process - three threads, slave process - three threads), seven cores (master process - one thread, two slave processes - each three threads)

OpenMP threads) per node and ii) two MPI processes (six OpenMP threads) per node. Figure 4a shows the mean speedup if the two-level hybrid parallelization is applied only during the perfusion process. It is clearly visible that the obtained efficiency is better than in the pure MPI version. If also the retraction process exploits the two-level hybrid parallelization, the gained speedup still remarkably increases. Although in the presented results the retraction process takes approximately only 2% of total CPU time during sequential algorithm execution (in contrast to 95% for the perfusion), it is enough to further accelerate the solution, especially in terms of Amdahl's law.

It can be also observed that the number of MPI processes per node has an influence on the performance (see Fig. 4). When the number of cluster nodes increases (i.e. eight and more), only one MPI process and consequently three OpenMP threads should be run per node even though each node is equipped with eight cores. The performance reduction, in the case of two MPI processes per node, comes from the higher load of a master process having more slaves to manage. Profiling results indicated that an overloaded master process can be inefficient in broadcasting permanent changes, i.e. time between sending the message with an optimal bifurcation by a slave and making a decision about permanent perfusions till broadcasting related changes by a master is lengthened excessively. Such a situation increases the tree's nonuniformity between slave nodes, which causes more MFUs rejections and finally more algorithm iterations.

In Fig. 5a, the performance of our all parallel solutions running within one cluster node is summarized. The best speedup can be gained with the two-level hybrid algorithm, especially in the case of seven cores and three MPI processes (master process - one thread, two slave processes - each three threads).

On the other hand, Fig. 5b shows the summary comparison between the best results of pure MPI version and hybrid one across nodes. It is clearly visible that the proposed hybrid solution provides better speedup than the pure MPI version. The improvement rises with the increase in the number of cluster nodes.

The hybrid solution was tested on the cluster of nodes with two quad-core chips. Hence, it may seem a waste of computational power since only three or six cores in each node can be arranged in computations. However, due to limited memory bandwidth, it can be even advantageous (e.g. in terms of power consumption) to use fewer treads than available cores [16]. On the other hand, there also exist six-cores processors (e.g. AMD Phenom II X6) that could be used with better efficiency. Moreover, most of internal human organs are supplied by two vascular trees (i.e. arterial and venous) and then the proposed approach would be more suitable for the most widespread two and quad-core processors.

5 Conclusion

In the paper a two-level parallel algorithm of vascular development is presented. The algorithm employs shared memory and message passing paradigms (mixed MPI+OpenMP implementation). Experimental results on a multi-core cluster show that a significant improvement in computational efficiency has been obtained. As a result, it helps us to extend the vascular model and to test multiply sets of parameters in reasonable period of time.

In the future, we will continue to work on the hybrid approach, among others, to investigate the influence of communication and calculations overlapping model, e.g. splitting one OpenMP thread off only to handle communication and the others to perform useful calculations.

Acknowledgments. This work was supported by the grant W/WI/3/11 from Bialystok University of Technology.

References

1. Gebali, F.: Algorithms and Parallel Computing. Wiley, NJ (2011)
2. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Boca Raton (2010)
3. Kretowski, M., Rolland, Y., Bezy-Wendling, J., Coatrieux, J.-L.: Physiologically Based Modeling for Medical Image Analysis: Application to 3D Vascular Networks and CT Scan Angiography. IEEE Trans. Med. Imaging 22(2), 248–257 (2003)
4. Mescam, M., Kretowski, M., Bezy-Wendling, J.: Multiscale Model of Liver DCE-MRI Towards a Better Understanding of Tumor Complexity. IEEE Trans. Med. Imaging 29(3), 699–707 (2010)
5. Jurczuk, K., Krętowski, M., Bézy-Wendling, J.: Vascular Network Modeling - Improved Parallel Implementation on Computing Cluster. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 289–298. Springer, Heidelberg (2010)
6. Pacheco, P.: Parallel Programming with MPI. Morgan Kaufmann Publishers, San Francisco (1997)
7. Jurczuk, K., Kretowski, M., Bezy-Wendling, J.: Vascular System Modeling in Parallel Environment - Distributed and Shared Memory Approaches. IEEE Trans. Inf. Technol. Biomed. 15(4), 668–672 (2011)
8. Chapman, B., Jost, B.G., van der Pas, R., Kuck, D.J.: Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, Cambridge (2007)
9. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: 17th Euromicro Int. Conf. on Parallel, Distributed & Network-based Processing, pp. 427–436. IEEE Press, Weimar (2009)
10. Shima, D.T., Ruhrberg, C.: Angiogenesis. In: Pelengaris, S., Khan, M. (eds.) The Molecular Biology of Cancer, pp. 411–423. Blackwell, Oxford (2006)
11. Zamir, M.: Arterial Branching Within the Confines of Fractal L-system Formalism. Journal of General Physiology 118, 267–275 (2001)
12. Schreiner, W., et al.: Optimized Arterial Trees Supplying Hollow Organs. Medical Engineering & Physics 28(5), 416–429 (2006)
13. Jones, M.D., Yao, R., Bhole, C.P.: Hybrid MPI-OpenMP Programming for Parallel OSEM PET Reconstruction. IEEE Trans. Nucl. Sci. 53(5), 2752–2758 (2006)
14. Pope, B., et al.: Performance of Hybrid Programming Models for Multiscale Cardiac Simulations: Preparing for Petascale Computation. IEEE Trans. on Biomed. Eng. 58(10), 2965–2969 (2011)
15. Amdahl, G.M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: Proc. AFIPS, Atlantic City, vol. 30, pp. 483–485 (1967)
16. Curtis-Maury, M., et al.: Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores. In: Proc. 17th Int. Conf. on Parallel Architectures & Compilation Techniques, pp. 250–259. IEEE Press, Toronto (2008)

Runtime Optimisation Approaches for a Real-Time Evacuation Assistant

Armel Ulrich Kemloh Wagoum, Bernhard Steffen, and Armin Seyfried

Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH,
Leo-Brandt-Strasse, 52428, Jülich, Germany
{u.kemloh,a.seyfried,b.steffen}@fz-juelich.de
http://www.fz-juelich.de/jsc/ped

Abstract. This paper presents runtime optimisation approaches for a real-time evacuation assistant. The pedestrian model used for the forecast is a modification of the centrifugal force model which operates in continuous space. It is combined with an event driven route choice algorithm which encompasses the local shortest path, the global shortest path and a combination with the quickest path. A naive implementation of this model has the complexity of $O(N^2)$, N being the number of pedestrians. In the first step of the optimisation the complexity is reduced to $O(N)$ using special neighbourhood lists like Verlet-List or Linked-Cell commonly used in molecular dynamics. The next step in this optimisation process is parallelisation on a multicore system. The Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) application programming interfaces are used to this extend. The simulation is performed on the Juropa cluster installed at the Jülich Supercomputing Centre. The speedup factors obtained are ~ 10 for the linked-cells, ~ 4 for 8 threads and ~ 3 for the parallelisation on 5 nodes using a static domain decomposition.

Keywords: pedestrian dynamics, high performance computing, evacuation, route choice.

1 Introduction

The Hermes project [12] which is funded by the German Federal Ministry of Education and Research, aims at developing an evacuation assistant for complex facilities to support decision makers and securities services in case of emergency. The target of the project is to forecast the evacuation of 50,000 pedestrians in a stadium for the next 15 minutes within 2 minutes of computation, thus by a factor of 7 faster than real-time. The test venue for the system is the ESPRIT arena in Düsseldorf, North Rhine-Westphalia, Germany. The layout of the assistant is presented in Fig. 1. The input data for the simulation is divided in three streams. The first stream is the geometry. There are two different configurations for the geometry for different events: seats only and mostly seats with some standing areas. The second stream comes from the safety and security management system. The information about the states of the escape routes, which is important in an hazardous situation, are made available. This information includes which doors are still usable (not blocked for instance) and which areas of the stadium are smoke-filled. The third stream comes from an automatic persons counting system. The output is the number of pedestrians in each block. Other input for the simulation includes the type of the event, concert or football game for instance and the type of the evacuation that should be simulated. This is important for choosing the appropriate routing strategy and reproducing the correct behaviour. The practical tests of the system will hopefully only sees routine clearings, no emergency evacuations. In this case, the route choice is not necessarily the local shortest path out of the building, but a lengthy way along the promenade towards the parking lots and the train station. After a real-time simulation, the results are analysed, visualised and discussed by the decisions makers. Majors challenges for this assistant include the proper development of microscopic models to accurately reproduce individual pedestrian motions, the proper development of a route choice model and the efficient implementation of these with respect to the runtime. In this paper we focus on the runtime optimisation.

The second part of this work presents the modelling approaches used at the strategic and at the operational level of the pedestrian motion. The third part presents the results of the runtime optimisation approaches with a focus on the parallelisation. This contribution is thereafter closed with some concluding remarks and outlooks.

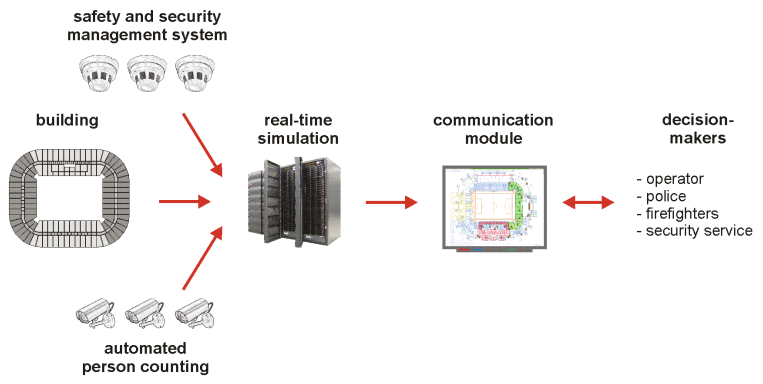


Fig. 1. Layout of the evacuation assistant

2 Modelling

There are mainly three different classes of models for pedestrian dynamic: cellular automata models [4,15], rule based models [26,6] and force based models [11,28]. Cellular Automata have the advantage of being computationally efficient, but the resolution of the simulated geometry is limited by the size of the cells. Force based models usually operate on a continuous geometry. They need more computations. For more about the advantages and disadvantages of the individual models we refer to [22]. Cellular automaton and force based models are used in the evacuation assistant but the focus is set on the force based models in this contribution.

2.1 The Generalized Centrifugal Force Model

The force based model used in the evacuation assistant is the generalized centrifugal force model (GCFM) [5]. In the GCFM at the operational level pedestrians are described with ellipses with velocity dependent semi-axes. The motion is ruled by the social forces [11,16]. At each simulation step the forces between the pedestrians and the obstacles (e.g. walls) are computed. Given a pedestrian i with coordinates $\overrightarrow{R_i}$, the equation of motion is:

$$m_i \ddot{\overrightarrow{R_i}} = \overrightarrow{F_i} = \overrightarrow{F_i^{\text{drv}}} + \sum_{j \in \mathcal{N}_i} \overrightarrow{F_{ij}^{\text{rep}}} + \sum_{w \in \mathcal{W}_i} \overrightarrow{F_{iw}^{\text{rep}}}\,, \tag{1}$$

where $\overrightarrow{F_{ij}^{\text{rep}}}$ denotes the repulsive force from pedestrian j acting on pedestrian i, $\overrightarrow{F_{iw}^{\text{rep}}}$ is the repulsive force emerging from the obstacle w and $\overrightarrow{F_i^{\text{drv}}}$ is a driving force. m_i is the mass of pedestrian i. $\mathcal{N}_i$ is the set of all pedestrians that influences pedestrian i and $\mathcal{W}_i$ the set of walls or borders that acts on pedestrian i. They are within a certain cut-off radius $r_c = 2m$. One should note here that it is not a hart cut-off as the repulsive forces are Hermite interpolated, so that they smoothly reach the values 0 at the distance r_c.

2.2 Event Driven Routing

The routing approach used in the assistant is a quickest path approach which operates on a graph-based structure. The visibility and the internal state of the pedestrians is used to (re)direct the pedestrians. Pedestrians minimise their travel time by systematically avoiding jams. They also try to escape from an already existing jam situation whenever possible. The process of systematically optimising the travel time is shown in Fig. 2. When a pedestrian enters a new location, he/she senses the environment and chooses the quickest path to reach the final destination. The same applies if the pedestrian is caught in a jam and there is an escape possibility, not in the middle of the jam for instance. The approximation is done by selecting a reference pedestrian in the sight range and evaluation his/her evolution over an observation time. Detailed information is found in [14].

3 Runtime Optimisation

The evacuation assistant should perform a real-time computation. This is a particular challenge for the force models. For instance a naive implementation of the GCFM requires for stability issues a time step $dt = 0.001s$ whereas $dt = 0.5s$ are usually enough for CA. A simulation of 15 minutes therefore corresponds to

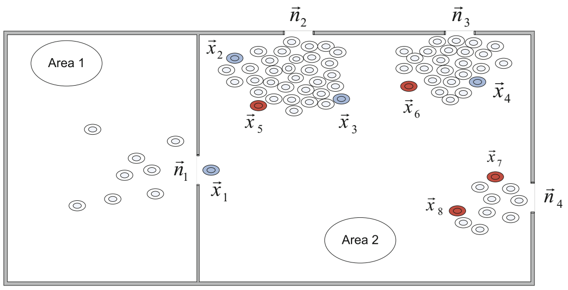


Fig. 2. Process of selecting a reference pedestrian prior to a route change. Pedestrians are denoted with their positions. $\boldsymbol{x_1}$ will select $\boldsymbol{x_5}$, $\boldsymbol{x_6}$ and $\boldsymbol{x_8}$. $\boldsymbol{x_2}$ has no clearance of the current situation and will not select any. $\boldsymbol{x_3}$ selects $\boldsymbol{x_6}$ and $\boldsymbol{x_8}$. $\boldsymbol{x_4}$ will only select $\boldsymbol{x_7}$.

900000 update steps. Considering the fact that at least 10000 pedestrians are updated in each simulation step, we end up having 9.10^9 update steps. Each step includes arithmetic operations like computing the forces between the pedestrians, the forces to the walls, the new velocities and positions. Each of which comprises several trigonometric and square root functions. Detailed information about the operations and performance are found in [24].

The optimisation is performed at three levels. The serial code is implemented using the linked-cells neighbourhood list to consider the short range character of the repulsive forces. The code is executed in a multi-threaded environment using OpenMP. Finally MPI is used to run the code across many computer nodes.

3.1 Simulation Area

The simulation domain (part of the arena) is presented in Fig. 3a and Fig. 3b. It is logically subdivided in 15 sections, which are mapped into the detection areas of the automatic person counting systems. This division will also be used as domain decomposition technique for the parallelisation as shown in the next sections.

3.2 Linked-Cells

A naive implementation of the GCFM has the complexity of $O(N^2)$, N being the number of simulated pedestrians. This is due to the fact that at each simulation step, the neighbourhood of each pedestrian has to be determined. The complexity can be reduced to $O(N)$ using special neighbourhood lists like verlet-list or linked-cell commonly used in molecular dynamics [2,25]. The performance comparison of the two list types and some of the results achieved are presented in [23]. The linked-cells have provided better results than the verlet-list in terms of memory requirements. Both perform equally in terms of speedup. The cell size is $2.2m$. Recall that the cell size should be at least equal to the cut-off radius.

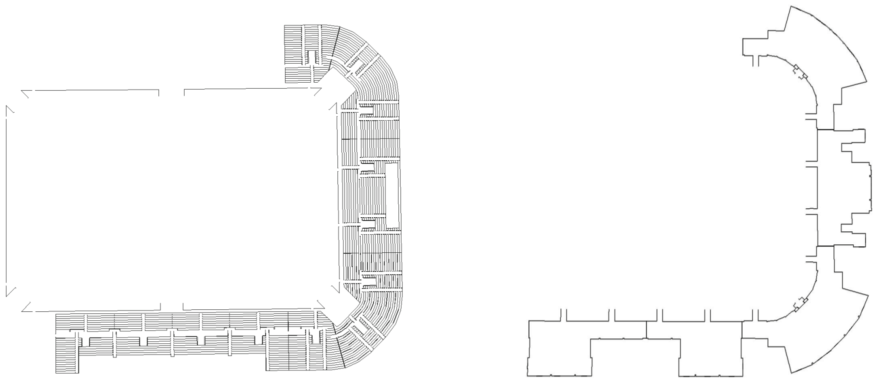

(a) Tribune (b) Promenade

Fig. 3. Simulation area subdivided in 15 sections

3.3 Parallelisation

The next step in this optimisation process is the use of parallelisation on a multicore system. The Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) application programming interfaces are used to this extend.

The most suitable parallelisation strategy for an application is usually coupled to the underlying hardware architecture. Some techniques on GPUs are presented in [17], and results on the cell-broadband engine are presented in [21]. More general techniques for particles in molecular dynamics are found in [8] and especially for pedestrian dynamics in [20,18].

Independently of the underlying hardware architecture, there are several parallelisation techniques with different complexities in the computation, the memory required and the communication resp. the data exchange between the processors.

One technique is the replicated data approach [13]. Each processor keeps a copy of all data in its memory, but works only on the portion it is responsible of. This method has a large communication complexity. At each step, all processors have to actualize their data. For P processors and N pedestrians this algorithm achieves a parallel complexity of the order $O(N/P)$ (N thanks to the linked-cells) for the CPU time, but its communication and memory complexities are of the order $O(N)$. The replicated data approach does not scale with P and the total runtime is dominated by the communication.

Another approach is parallelization by data partitioning. Here, each processor only stores the data of the pedestrians required during the computation. These are the N/P pedestrians that are assigned to the processors in the parallel computation, and the other pedestrians that interact with those pedestrians, i.e the pedestrians in the neighborhood (see Eq. 1). The data partitioning method scales as $O(N/P)$ for computation and communication as long as P is so small that latency is negligible.

A similar approach to the data partitioning is achieved by static domain decomposition [9,19]. The main goal here is to limit the communication between the processors. For that purpose the simulation domain is decomposed into subdomains and each processor is assigned a subdomain. The data are partitioned and distributed to processors in such a way that as little communication as possible is needed. In each communication step only the pedestrians from neighboring subdomains have to be communicated. Using the linked-cells, this number of neighboring pedestrians can be further reduced using the so-called ghost areas. In this way, the number of pedestrians for which data to be received or sent decreases to $O(\sqrt{N/P})$. The complexity of the entire computation is of the order $O(N/P)$. One should note that this applied only when the pedestrians are uniformly distributed. In the case where the pedestrians are not uniformly distributed, dynamic decompositions [10,3,27] might be required to, ensure an almost uniform distribution of the N pedestrians on the P processors.

We end up choosing a static decomposition on the geometry presented in Fig. 3 not only because of its relatively low communication requirements but also due to the natural partitions given by the persons counting system. This partitioning was done to achieve as less interactions as possible between the domains. This is done by choosing the boundaries between the domains, which are also the counting lines for the system, as small as possible, thus mainly at doors. We also assume an initial uniform distribution of the pedestrians.

3.4 Load Balancing

Load balancing is important to parallel programs for performance reasons. All processors should have approximately the same amount of work to do. This is not always possible in the case of a static partitioned simulation area. One solution is to use a dynamic partitioning as presented earlier. This is done at the cost of a higher communication between the processors. In the case of the evacuation assistant, the initial pedestrians number is received via sensors (see Fig. 1) of an automatic persons counting system. The received pedestrians are then homogeneously distributed in their respective areas. The simulation area is initially partitioned in 15 sections well suited for a load balancing at the start time. A major difference to general particles simulation in molecular dynamics or in N-Body systems in general is that the pedestrians stream and direction of movement is predictable. Predictable in that sense that at a certain time in the simulation, they will gather at exits. Therefore the idea of "manually" splitting the simulation area into static area for the processor is well suited. Another reason is the production machine dedicated to the application. 15 nodes are available for the space continuous model in the evacuation assistant. In addition the results of the simulation, i.e. the trajectories of the pedestrians are written with respect to those areas, which means that each of the processors can perform IO operations without any need of synchronisation with others.

3.5 Results

The presented results have been obtained on JUROPA [1], an Intel based cluster installed at the Jülich Supercomputing Centre from the Forschungszentrum

Jülich GmbH. For the results presented here, only the promenade has been simulated and the pedestrians are always equally and homogeneously distributed in the 5 sections. Two common values to measure the performance of a parallel application are the speedup and the efficiency. The speedup is defined by $S(P) = T/T(P)$ where T is the time needed by the serial application and $T(P)$ the time needed by the parallel (optimised) application. The efficiency is the defined as $E(P) = S(P)/P$. We only evaluate using the speedup.

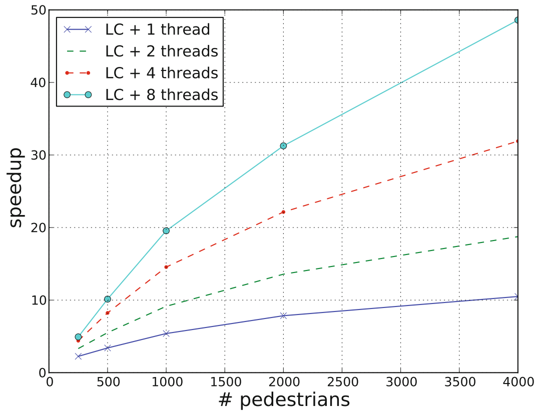


Fig. 4. Speedup of the linked-cells with different number of threads over the brute force method

Fig. 4 shows a comparison of the brute force method and the linked-cells on a single node using 1, 2 , 4 and 8 threads. One should note here that the brute force is only in the neighbourhood detection, not on the force computation between the pedestrians. The runtime applies to a complete simulation i.e. all pedestrians have left the facility. The overall speedup obtained for 5 X 800 Pedestrians in this case is 10.50 for 1 thread and 48.61 for 8 threads.

Fig. 5 shows the results obtained using an hybrid MPI+OpenMP parallelisation approach on 5 computing nodes. The overall speedup obtained in this case for 5 X 800 pedestrians is 131.44 over the brute force serial program and 12.52 over the code optimised with the linked cells. The individual runtimes in seconds are presented in Table 1.

In the real case scenario the awaited initial load for each simulation area is around 1000 pedestrians. A simulation with this configuration is actually performed in $\sim$ 543 seconds. This is still four time higher than the required 2 minutes computation time. The simulation on the production machine however is by a factor ~ 5 faster than on JUROPA. The production machine is a 12 cores Nehalem processors machine, with 15 nodes dedicated to the force GCFM model. Its nodes have a larger memory and a higher CPU frequency. Under these conditions the real-time requirement will be met. Still, a closer look at the simulation with appropriate debugging tools of the Scalasca toolset [7], shows that the communication between the processors offers further possibilities of optimisation.

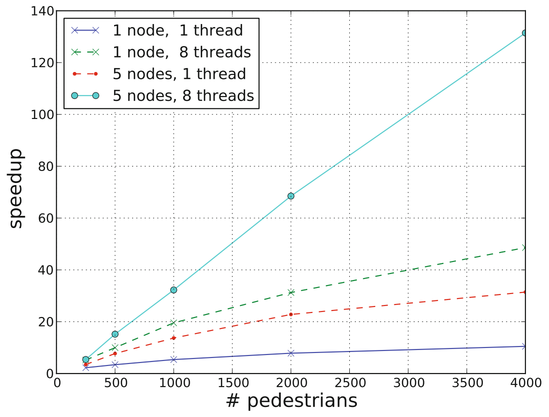


Fig. 5. Speedup of the hybrid program (MPI + OpenMP) over the serial brute force method

Table 1. Runtime in seconds using the brute force (BF) the linked cells (LC) and a hybrid parallel implementation (5 nodes + 8 threads + LC)

#pedestrians	BF	LC	Hybrid
250	186	82	34
500	608	178	40
1000	2427	450	75
2000	11490	1464	167
4000	71422	6803	543

4 Conclusion

The Hermes project has been presented in this paper, together with the new challenges it is involved with. A special focus has been set on the runtime requirements. The goal of performing a real-time simulation has been almost reached with help of different optimisation techniques including neighbourhood lists and parallelisation on a super computer. The real-time requirement will be met with the actual program on the production machine which has a different configuration. Nevertheless other optimisation steps will be undertaken. For instance the implementation of solvers which allow a larger step size like the Verlet or the Leapfrog Algorithms.

Acknowledgments. This work has been performed within the program "Research for Civil Security" in the field Protecting and Saving Human Life" funded by the German Government, Federal Ministry of Education and Research (BMBF). The project is granted under the Grant-Nr.: 13N9952. We also thank the John von Neumann-Institut for Computing (NIC) at the Forschungszentrum Jülich GmbH for providing the computation time on JUROPA.

References

1. Juropa-JSC - HPC-FF (August 2009), http://www.fz-juelich.de/portal/EN/Research/InformationTechnology/Supercomputer/JUROPA.html
2. Allen, M.P., Tildesley, D.J.: Computer simulation of liquids, vol. 18. Oxford University Press (1989)
3. Baiardi, F., Bonotti, A., Ferrucci, L., Ricci, L., Mori, P.: Load balancing by domain decomposition: the bounded neighbour approach. In: Proc. of 17th European Simulation Multiconference, pp. 9–11 (2003)
4. Blue, V.J., Adler, J.L.: Cellular automata microsimulation for modeling bidirectional pedestrian walkways. Transportation Research Part B 35, 293–312 (2001)
5. Chraibi, M., Seyfried, A., Schadschneider, A.: Generalized centrifugal force model for pedestrian dynamics. Physical Review E 82, 046111 (2010)
6. Galea, E.R., Gwynne, S., Lawrence, P., Filippidis, L., Blackspields, D., Cooney, D.: buildingEXODUS V 4.0 - User Guide and Technical Manual (2004)
7. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
8. Griebel, M., Knapek, S., Zumbusch, G.: Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications, 1st edn. Springer Publishing Company, Incorporated (2007)
9. Hanxleden, R.V., Clark, T.W., Clark, T.W., Hanxleden, R., Mccammon, J.A., Scott, L.R.: Parallelizing molecular dynamics using spatial decomposition. In: Scalable High Performance Computing Conference, pp. 95–102. IEEE Computer Society Press (1993)
10. Hegarty, D., Kechadi, M., Dawson, K.: Dynamic Domain Decomposition and Load Balancing for Parallel Simulations of Long-Chained Molecules. In: Waśniewski, J., Madsen, K., Dongarra, J. (eds.) PARA 1995. LNCS, vol. 1041, pp. 303–312. Springer, Heidelberg (1996)
11. Helbing, D., Molnár, P.: Social force model for pedestrian dynamics. Phys. Rev. E 51, 4282–4286 (1995)
12. Holl, S., Seyfried, A.: Hermes - an Evacuation Assistant for Mass Events. inSiDe 7(1), 60–61 (2009), http://inside.hlrs.de/pdfs/inSiDE_spring2009.pdf
13. Janak, J., Pattnaik, P.: Protein calculations on parallel processors. ii. parallel algorithm for the forces and molecular dynamics. Journal of Computational Chemistry 13(9), 1098–1102 (1992)
14. Kemloh Wagoum, A.U., Seyfried, A., Holl, S.: Modelling dynamic route choice of pedestrians to assess the criticality of building evacuation. Advances in Complex Systems 15(3) (2012)
15. Kirchner, A., Schadschneider, A.: Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics. Physica A 312, 260–276 (2002)
16. Molnár, P.: Modellierung und Simulation der Dynamik von Fußgängerströmen. Dissertation, Universität Stuttgart (1995)
17. Richmond, P., Romano, D.: A High Performance Framework For Agent Based Pedestrian Dynamics On GPU Hardware. In: Proceedings of EUROSIS ESM 2008 (European Simulation and Modelling) (October 2008)
18. Pettré, J., De Heras Ciechomski, P., Maïm, J., Yersin, B., Laumond, J.P., Thalmann, D.: Real-time navigating crowds: scalable simulation and rendering: Research articles. Comput. Animat. Virtual Worlds 17, 445–455 (2006)

19. Plimpton, S., Hendrickson, B.: Parallel molecular dynamics algorithms for simulation of molecular systems. In: Mattson, T.G. (ed.) Parallel Computing in Computational Chemistry, pp. 114–136 (1995)
20. Quinn, M.J., Metoyer, R.A., Hunter-zaworski, K.: Parallel implementation of the social forces model. In: Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics, pp. 63–74 (2003)
21. Reynolds, C.: Big fast crowds on PS3. In: Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames (2006)
22. Schadschneider, A., Klingsch, W., Klüpfel, H., Kretz, T., Rogsch, C., Seyfried, A.: Evacuation Dynamics: Empirical Results, Modeling and Applications. In: Encyclopedia of Complexity and System Science, vol. 5, pp. 3142–3176. Springer, Heidelberg (2009)
23. Seyfried, A., Chraibi, M., Mehlich, J., Schadschneider, A.: Runtime Optimization of Force Based Models within the Hermes Project. In: Pedestrian and Evacuation Dynamics 2010(2010)
24. Steffen, B., Kemloh Wagoum, A.U., Chraibi, M., Seyfried, A.: Parallel real time computation of large scale pedestrian evacuations. In: Ivanyi, P., Topping, B.H.V. (eds.) The Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering, p. 95. Civil-Comp Press, S (2011) 978-1-905088-44-7
25. Sutmann, G., Stegailov, V.: Optimization of neighbor list techniques in liquid matter simulations. Journal of Molecular Liquids 125(2-3), 197–203 (2006)
26. Thompson, P.A.: Developing new techniques for modelling crowd movement. Phd thesis, University of Edinburgh (1994)
27. Wang, S., Armstrong, M.P.: A quadtree approach to domain decomposition for spatial interpolation in grid computing environments. Parallel Comput. 29, 1481–1504 (2003)
28. Yu, W.J., Chen, R., Dong, L., Dai, S.: Centrifugal force model for pedestrian dynamics. Phys. Rev. E 72(2), 026112 (2005)

A Parallel Genetic Algorithm Based on Global Program State Monitoring

Adam Smyk[1] and Marek Tudruj[1,2]

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

Abstract. A new approach to the design of parallel genetic algorithms (GA) for execution in distributed systems is presented. It is based on the use of global parallel program control functions and asynchronous process/thread internal execution control based on global application states monitoring. A control design graphical infrastructure is provided for a programmer based on generalized synchronization processes called synchronizers. They collect local states of program elements, compute global control predicates and send control signals to program computational elements. It enables an easy construction and management of global program states for the purpose of the program execution control at both thread and process level. At each level we create a hierarchical control/synchronization infrastructure which is used to optimize the control of computations in programs. As an example we present the design of a parallel genetic algorithm used to partition a macro data flow graph for FDTD (Finite Difference Time Domain method) computations.

Keywords: distributed program design paradigms, mesh partitioning, global application states monitoring, graphical program design tools, FDTD.

1 Introduction

Optimization of many parallel applications which solve numerical simulation needs global synchronization primitives in programs. It is because program code decomposition and distribution for such applications in parallel systems is not simple. The first problem is to ensure a proper processor load balancing, which can be especially difficult if the application is based on irregular data structures [9]. A volume of data supplied and processed in each computational element must correspond to its real computational power. Another problem concerns a transfer of data between each computational element in a given distributed system. A total communication volume should be adjusted to the available network performance. To obtain such optimal graph partitioning (it is a NP-complete problem) we can use some direct (based on the cut-min optimization) [8] or iterative techniques [6–8]. In this paper, we present a solution for heuristical program partitioning algorithm which is based on a parallel genetic algorithm [12]. Convergence of genetic algorithms depends on definitions of genetic operators (selection, crossover and mutation) and also on the evaluation of the fitness function.

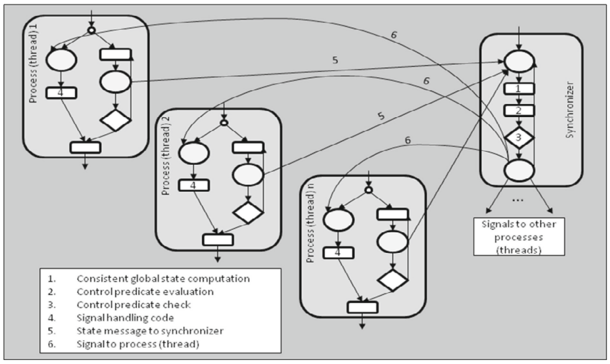

Process (thread) 1
Process (thread) 2
Process (thread) n
Synchronizer
Signals to other processes (threads)
1. Consistent global state computation
2. Control predicate evaluation
3. Control predicate check
4. Signal handling code
5. State message to synchronizer
6. Signal to process (thread)

Fig. 1. Synchronizer co-operation with processes or threads

We consider a parallel implementation of the genetic algorithm [5, 12], so the populations of individuals (solutions) will be distributed among computational processors. Fitness functions for individuals can be computed independently on each processor. But, genetic operations in many cases must be done globally. It means that for genetic operations we can select two or more individuals located on different processors. To perform it efficiently, we should answer several questions. On which processors, the best individuals are located? Which individuals should be distributed? Will they be distributed to all processors, or only to some chosen ones? What kind of communication should be used: broadcast, multicast, point to point? How often it should be done? How to avoid a network contention? In fact, the answers to these questions are not obvious. Distribution of the best individuals should be controlled globally [11]. It must depend on the global state of the application which should be dynamically monitored [16]. To perform such monitoring, a special control synchronization infrastructure can be created. Such an asynchronous program execution control mechanism is illustrated in Fig.1 [2, 4].

We have assumed that an application consists of computational processes or threads that can be executed in parallel in a system with shared-distributed memory. All these processing elements can send and receive data by using standard communication mechanisms like sockets, MPI or shared memory. This kind of communication is usually implemented manually by the programmer. In order to support control communication for synchronization operations, a special control infrastructure will be delivered.

The core of the control infrastructure is an element called a synchronizer. It collects local state messages from all computational elements. It determines, if the application has reached a strongly consistent global state SCGS [1]. SCGS is a set of fully concurrent local states detected without doubt by a synchronizer. The construction of strongly consistent global states is based on projecting the local states of processes or threads on a common time axis and finding time intervals which are covered by local states in all participating processes or threads [14]. Next, the synchronizer evaluates control predicates on global states and undertakes predefined control actions. If some predicates are met, then the synchronizer sends control signals to selected processes or threads, see Fig.2. The signals must be handled by these processes and some desired actions should be performed. In the code of a process (thread), we can distinguish regions sensitive to incoming signals. They are marked by special delimiters. When a computational process enters a sensitive region, it will start receiving signals from a synchronizer and performs reactions defined in its code. If not, the signal is ignored and the reaction will not appear. To increase the control performance, all control messages are physically separated from messages used by computational elements for data communication by using separate communication networks. The use of such dual communication network to support global application states monitoring was discussed in [3].

The described above control/synchronization environment has currently been under development as a PEGASUS (Program Execution Governed by Asynchronous Supervision of States) system [15]. It is assumed for the implementation of the genetic algorithms discussed in this paper.

The paper is composed of 3 parts. In the first part an overview of the FDTD method is presented. In the second part FDTD computation optimisation is discussed basd on genetic algorithms. In the last part parallelization of the genetic algorithms with the use of the PEGASUS system infrastructure has been outlined.

2 FDTD Method Overview

Using the FDTD computational method we can simulate high frequency electromagnetic wave propagation by solving Maxwell equations (1).

We have assumed that a simulated area is represented by a two dimensional, irregular shape, see Fig.3. Before the simulation starts, a computational mesh (CM) must be created. CM structure depends on the FDTD theory and for a two dimensional problem it is defined according to differential equations (2). CM consists of a given number of points (mesh density depends on the frequency of simulation) each of which contains alternately electric component Ez of electromagnetic field and one from two magnetic field components Hx or Hy (depending on coordinates).

Simulation is an iterative process which is divided into a given number of iterations. Each iteration consists of two steps: computation of the values of all Ez components in the first step and computation of the values of Hx and Hy components in the second step. For regular computational areas, FDTD computations can be easily parallelized (e.g. by stripe partitioning) but when

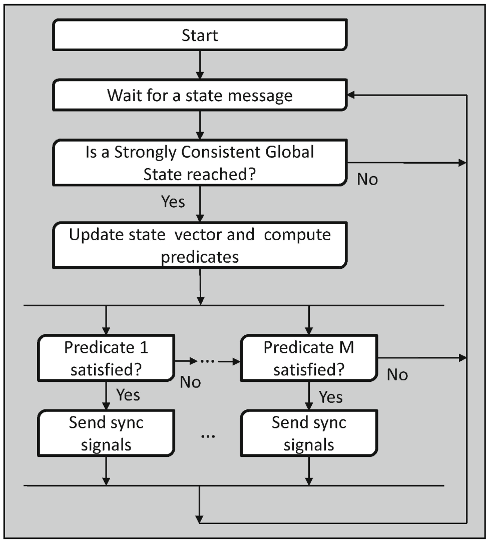


Fig. 2. Control flow diagram of a synchronizer

an irregular shape of the computational area is considered the decomposition is much more complicated. It is because a special partitioning algorithm must be used before parallel computations can be started. This algorithm must take into consideration both a proper load of all processing nodes and the minimal number of data transmissions. To solve this problem, a macro data flow graph for the FDTD computations mapped to executive resources must be created. It is done in three steps: 1) creation of a data flow graph showing basic data dependencies between FDTD operations; 2) merging data flow graph nodes into separate macro data flow nodes; 3) mapping each macro data flow graph node to an executive processor. All these steps are described in details in [13].

All computations in each sub-area are represented by one macro node. A macro node can be fired for execution only if all input data have been delivered to the physical processor on which this macro node has been mapped. The data dependencies are described by edges between macro nodes. Edges can be attributed with weights which give the amount of data, sent from one macro node to another. The weight depends on the length of the boundary line between two adjacent sub-areas.

3 Computation Partitioning Using a Genetic Algorithm

In this section, we describe main assumptions on a genetic algorithm which partitions the FDTD data flow graph to find the heuristical partition of a computational mesh, to assure time-balanced simulation execution in processors, see Fig.4. Before the genetic algorithm will start, we need to create: an computational area with an initial partitioning (*CAIP*) and an initial population (*IP*).

$$\nabla \times H = \gamma E + \varepsilon \frac{\partial E}{\partial t}, \qquad \nabla \times E = -\mu \frac{\partial H}{\partial t} \tag{1}$$

$$\begin{cases} \overline{H}_y^n(i,j) = \overline{H}_y^{n-1}(i,j) + RC \cdot [\overline{E}_z^{n-0.5}(i,j-1) - \overline{E}_z^{n-0.5}(i,j+1)], \\ \overline{H}_x^n(i,j) = \overline{H}_x^{n-1}(i,j) + RC \cdot [\overline{E}_z^{n-0.5}(i-1,j) - \overline{E}_z^{n-0.5}(i+1,j)], \\ \overline{E}_z^n(i,j) = CA_z(i,j) \cdot \overline{E}_z^{n-1}(i,j) + CB_z(i,j) \cdot [\overline{H}_y^{n-0.5}(i+1,j) - \overline{H}_y^{n-0.5}(i-1,j) + \overline{H}_x^{n-0.5}(i,j-1) - \overline{H}_x^{n-0.5}(i,j+1)] \end{cases} \tag{2}$$

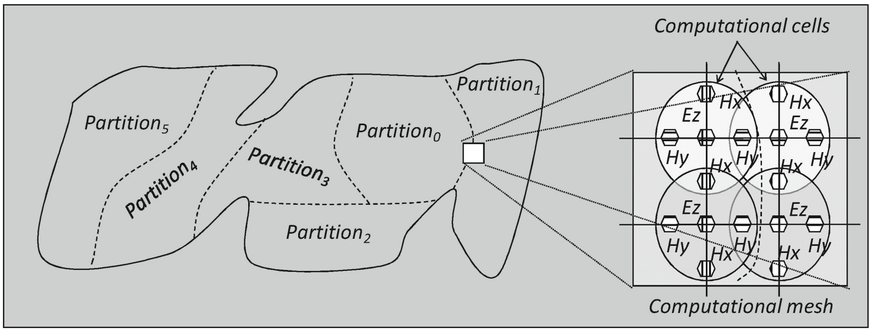


Fig. 3. Irregular computational area with FDTD computational mesh

CAIP is obtained from the data flow graph which represents computations and communication pattern in given computational area. It is done in two steps: 1) leader data flow nodes identification (each leader node represents a further single initial macro node) and 2) initial macro nodes creation, by simple assignment of nodes to the macro node which is determined by the closest leader. This process is described in details in [13]. The number of leaders is usually much bigger than the assumed number of processors in a given computational system. To create an initial population containing an assumed number of individuals we need to provide a definition of a chromosome. As we have mentioned above, our genetic algorithm produces a program which performs a sequence of merging operations. Each merging operation has two parameters: which are the identifiers of macro data flow nodes to be merged together. These identifiers must be found by the genetic algorithm *GA*. To do this, we have introduced a set of rules (see Fig.5) that indicate objectives of each merging operation. We can design a *GA*, which focuses only on e.g. communication optimization and it will define priorities to "edge cut reduction" rules. In Fig.5 we present a definition of a chromosome.

As we can see, a chromosome is an array of integers. Each integer is an index to a chosen merging rule which will be executed in each step by our genetic algorithm. The length L of the array, determines how many merging operations must be executed to obtain a given number of partitions and it is equal to:

$$L = InitialNumberOfParttionsInIMDF – GivenNumberOfProcesors$$

When all (L) merging rules are defined, we have to evaluate the fitness function for all individuals. The value of the fitness function indicates the best candidates for reproduction. In our experiments we have used two fitness functions: the execution time of the partitioned macro data flow graph and the number of the cut edges in the obtained DFG graph partition. When the fitness function is evaluated we can select the best candidates for reproduction and we can perform crossover and mutation operations. This part of the algorithm is presented in details in [13].

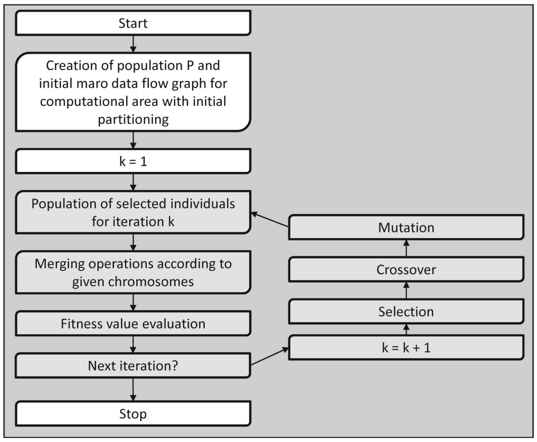

Start
Creation of population P and initial maro data flow graph for computational area with initial partitioning
k = 1
Population of selected individuals for iteration k
Merging operations according to given chromosomes
Fitness value evaluation
Next iteration?
Stop
Mutation
Crossover
Selection
k = k + 1

Fig. 4. General overview of the *GA* partitioning method

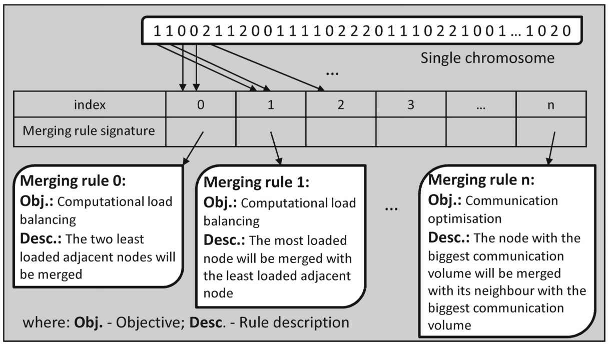

1 1 0 0 2 1 1 2 0 0 1 1 1 1 0 2 2 2 0 1 1 1 0 2 2 1 0 0 1 ... 1 0 2 0
Single chromosome
...
index
0
1
2
3
...
n
Merging rule signature
Merging rule 0:
Obj.: Computational load balancing
Desc.: The two least loaded adjacent nodes will be merged
Merging rule 1:
Obj.: Computational load balancing
Desc.: The most loaded node will be merged with the least loaded adjacent node
...
Merging rule n:
Obj.: Communication optimisation
Desc.: The node with the biggest communication volume will be merged with its neighbour with the biggest communication volume
where: Obj. - Objective; Desc. - Rule description

Fig. 5. Single chromosome structure

4 Parallelization of a Genetic Algorithm Based on Global Program State Monitoring

Genetic algorithms, can be easily implemented in parallel way [10]. Theoretically, all operations shown in dark gray boxes (Fig.4) are data independent and they can be executed on separated processors. The problem appears, when we want to perform a crossover operation. One instance of the *GA* working on one processor, can create an isolated population without any interference of individuals from populations created on other processors. So, if any *GA* instance creates some promising individuals, they should be broadcast among other instances of *GA*.

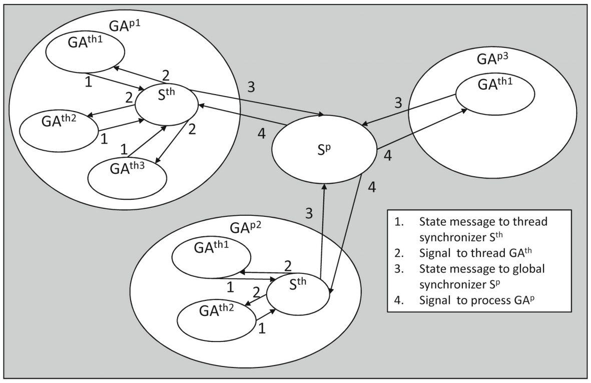

GA^p1
GA^th1
GA^th2
GA^th3
S^th
GA^p2
GA^th1
GA^th2
S^th
S^p
GA^p3
GA^th1
1
2
3
4
1. State message to thread synchronizer S^th
2. Signal to thread GA^th
3. State message to global synchronizer S^p
4. Signal to process GA^p

Fig. 6. Co-operation between instances of genetic algorithm (processes and threads) and synchronizers

However, no known *GA* parallelization examples have not been supported yet by a convenient control infrastructure provided by the runtime system. In Fig.6 we show how to parallelize *GA* based on the infrastructure of PEGASUS synchronizers which enable convenient monitoring and management of global application states for proper control purposes. We can assume that each instance of *GA* can be executed as a process (e.g. $GA^{\mathrm{p}3}$) or as a thread (e.g. $GA^{\mathrm{th}2}$).

The state of a single individual is described by its chromosome, a total execution time for current partitioning and by cut min value. The state of the whole local population is an array containing states for all local individuals. To share state information during the optimization we have used a hierarchical infrastructure of synchronizers and GA instances. We distinguish synchronizers for threads (S^{th}) and for processes (S^{p}). Each thread synchronizer is a special thread that co-operates with group of computational threads created inside one process. Each process synchronizer is a special process that co-operates with group of computational process. A process synchronizer doesn't directly co-operate with group of computational threads. It is because, there is no technical support for communication between threads belonging to different processes, so some intermediate level of communication has been introduced. During the execution, all information concerning states of individuals and of the whole local population is stored in local structures of the synchronizers (Fig.7 left). Control flow diagram of a synchronizer for the parallel genetic algorithm implementation is presented in Fig.7 (right). The synchronizer waits for messages from all computation elements connected to it, containing reports on their local states. A synchronizer collects messages on all local states and store them into a global state array. All local states for all computations, are kept on the synchronizer. When a global state array, is updated, the synchronizer can start to compute a predicate. In the case of our parallel GA, we have specified the following predicates: the synchronizer has found definitely the best individual ($DTBS$), many better solutions have been found (BS) and there is no improvement observed (NIO).

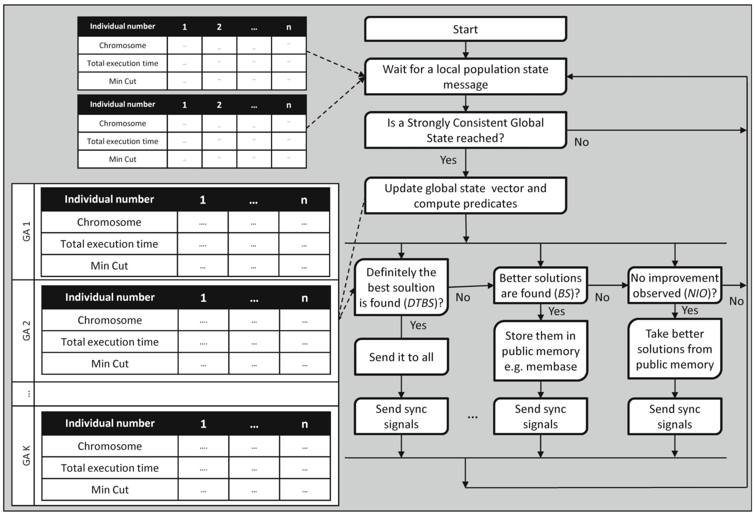

Individual number
Chromosome
Total execution time
Min Cut
GA 1
GA 2
GA K
Start
Wait for a local population state message
Is a Strongly Consistent Global State reached?
No
Yes
Update global state vector and compute predicates
Definitely the best soultion is found (DTBS)?
Better solutions are found (BS)?
No improvement observed (NIO)?
Send it to all
Store them in public memory e.g. membase
Take better solutions from public memory
Send sync signals

Fig. 7. (left) Global population state array on a synchronizer and (right) Control flow diagram of a synchronizer for the parallel genetic algorithm

If the first predicate ($DTBS$) is met, the best solution will be send immediately to all computational elements. If the second predicate (BS) is met, all better solutions, will be stored in a public memory. Public memory is a globally accessed memory pool, available for all remote processes and threads (e.g. membase) . In this case, there is no communication between any computational elements. If the third predicate (NIO) is met, it means that no improvement has been observed for whole global population and local populations must be supported by local solutions stored in a public memory. The synchronizer decides which local individuals will be used for global population improvement. After a given number of NIO actions, the synchronizer can decide to sends a kill signal to all managed computational elements. It means that for such merging rules and for such local configurations, no better global solution can be found. In Fig.8 a control flow diagram for the parallel genetic algorithm (PGA) for computational processes (threads) is shown. The general scheme of PGA is similar to this presented in Fig.4 for GA. There are some modifications which have been introduced. First, a computational process must record its local state in a special array (see. Fig.7). So in every iteration it sends to the closest synchronizer a full quality report for the whole local population. The synchronizer, receives such reports, stores them in its local memory, computes predicates ($DTBS$, *BS, NIO*), and sends signals with suggested actions for all computational elements. Each computational process must receive these messages. In response, it performs some predefined actions to increase the quality of its local population.

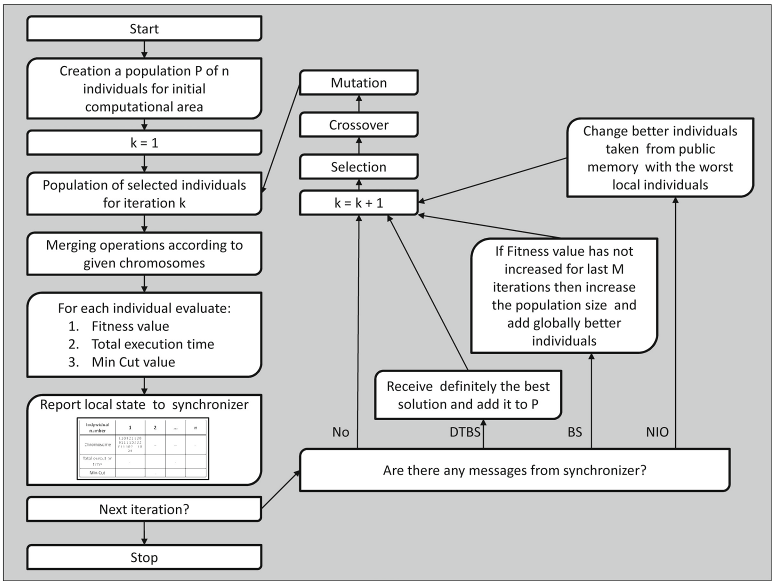

Start
Creation a population P of n individuals for initial computational area
k = 1
Population of selected individuals for iteration k
Merging operations according to given chromosomes
For each individual evaluate: 1. Fitness value 2. Total execution time 3. Min Cut value
Report local state to synchronizer
Next iteration?
Stop
Mutation
Crossover
Selection
k = k + 1
Change better individuals taken from public memory with the worst local individuals
If Fitness value has not increased for last M iterations then increase the population size and add globally better individuals
Receive definitely the best solution and add it to P
No
DTBS
BS
NIO
Are there any messages from synchronizer?

Fig. 8. Control flow diagram of a computational process (thread) in the genetic algorithm with synchronizers

5 Conclusions

In this paper we have presented a parallel implementation of a genetic algorithm which can be executed in distributed systems. The control of all genetic operators (selection, crossover and mutation) execution is based on the use of global parallel program control functions and asynchronous process/thread internal execution control based on global application states monitoring. The control infrastructure is based on processes (or threads) called synchronizers, which gather local states from distributed program elements, compute global predicates and send back control signals to simulate desired reactions in the program elements. We have shown, that such a control infrastructure provides a convenient support to design a parallel implementation of a genetic algorithm used to partition a computational mesh of the FDTD problem.

References

1. Babaoglu, O., Marzullo, K.: Consistent global states of distributed systems: fundamental concepts and mechanisms. In: Distributed Systems, Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. Addison-Wesley (1995)

2. Borkowski, J.: Interrupt and Cancellation as Synchronization Methods. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 3–8. Springer, Heidelberg (2002)
3. Borkowski, J., Tudruj, M.: Dual Communication Network in Program Control Based on Global Application State Monitoring. In: ISPDC 2007, pp. 37–44. IEEE CS, Hagenberg (2007)
4. Borkowski, J., Tudruj, M., Kopański, D.: Global predicate monitoring applied for control of parallel irregular computations. In: Euromicro PDP 2007, pp. 105–111. IEEE CS, Naples (2007)
5. Coley, D.A.: An Introduction to Genetic Algorithms for Scientists and Engineers (Hardcover), Har/Dsk edn. World Scientific Publishing Company (November 1997) ISBN-10: 9810236026
6. Dutt, S., Deng, W.: VLSI Circuit Partitioning by Cluster-Removal using Iterative Im-provement Techniques. In: Proc. IEEE International Conference on Computer-Aided Design, pp. 350–355 (1997)
7. Karypis, G., Kumar, V.: Unstructured Graph Partitioning and Sparse Matrix Ordering. Technical Report, Department of Computer Science, University of Minesota (1995), http://www.cs.umn.edu/~kumar
8. Khan, M.S., Li, K.F.: Fast Graph Partitioning Algorithms. In: Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing, Victoria, B.C., Canada, pp. 337–342 (May 1995)
9. Lin, H.X., van Gemund, A.J.C., Meijdam, J.: Scalability analysis and parallel execution of unstructured problems. In: Eurosim 1996 Conference (1996)
10. Nowastowski, M., Poli, R.: Parallel Genetic Algorithm Taxonomy. In: Third International Conference on Knowledge-Based Intelligent Information Engineering Systems, KES 1999, May 13 (1999)
11. Raynal, M., Helart, J.-M.: Synchronization and control of distributed systems and programs. John Wiley and Sons Ltd. (1990)
12. Smyk, A., Tudruj, M.: Optimization of Parallel FDTD Computations Using a Genetic Algorithm. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 559–569. Springer, Heidelberg (2008)
13. Smyk, A., Tudruj, M.: Parallel Implementation of FDTD Computations Based on Macro Data Flow Paradigm. In: PARELEC 2004, Dresden, Germany, September 7-10 (2004)
14. Stoller, S.D.: Detecting Global Predicates in Distributed Systems with Clocks. Distributed Computing 13(2), 85–98 (2000)
15. Tudruj, M., Borkowski, J., Smyk, A., Kopański, D., Laskowski, E., Maśko, Ł.: Program Design Environment for Multicore Processor Systems With Program Execution Controlled by Global States Monitoring. In: ISPDC 2011. IEEE CS, Clui-Napoca (2011)
16. Tudruj, M., Kacsuk, P.: Extending Grade Towards Explicit Process Synchronization in Parallel Programs. Computers and Artificial Intelligence 17 (1998)

Parallel Approach to the Functional Decomposition of Logical Functions Using Developmental Genetic Programming

Stanislaw Deniziak and Karol Wieczorek

Departament of Computer Science, Kielce University of Technology, Poland
S.Deniziak@computer.org
K.Wieczorek@tu.kielce.pl

Abstract. Functional decomposition is the main step in the FPGA-oriented logic synthesis, where a function is decomposed into a set of functions, each of which must be simple enough to be implementable in one logic cell. This paper presents a method of searching for the best decomposition strategy for logical functions specified by cubes. The strategy is represented by a decision tree, where each node corresponds to a single decomposition step. In that way the multistage decomposition of complex logical functions may be specified. The tree evolves using the parallel developmental genetic programming. The goal of the evolution is to find a decomposition strategy for which the cost of FPGA implementation of a given function is minimal. Experimental results show that our approach gives significantly better results than other existing methods.

Keywords: developmental genetic programming, parallel processing, functional decomposition,FPGA devices.

1 Introduction

Genetic algorithms (GAs) [1] are powerful search techniques that are used to solve complex optimization problems in different domains. But for some kind of problems GAs are very time-consuming or have large memory requirements. In such cases the only practical way is to the use of parallel processing. It was observed that parallel GAs (PGAs) often provide better efficiency than sequential approaches. Therefore, a lot of parallelization models for GAs were proposed. This paper concerns the parallel approach to the developmental genetic programming (DGP) [2], which is used to find the optimal decompositions of complex logical functions.

Decomposition is a process of splitting a complex function into a set of smaller sub-functions. It reduces the complexity of the problem of system analysis and synthesis by dividing it into smaller subsystems. Decomposition is used in machine learning, pattern analysis, data mining, knowledge discovery and logic synthesis of digital systems [3] [4]. In the case of digital circuits, objectives of optimisation are the minimal cost and the minimal latency of target system implementation.

Decomposition is a NP-complete problem, so using exhaustive method is very time consuming and impractical. There exist a lot of heuristic methods of decomposition dedicated to Boolean functions [5] [6]. As far as LUT-based FPGA (Look-Up Table Based Field Programmable Arrays) implementations are considered, the most effective method is the functional decomposition [6]. It splits a logical function into two smaller functions using a parallel or serial strategy. This step should be repeated recursively for each result function which is not implementable in one logic cell. The type of decompositions performed at the following stages are defined by the multilevel decomposition strategy. There are efficient methods of single-step functional decomposition, giving quite good results [7], but it seems that multilevel decomposition strategies are not studied enough. The only known method is the balanced decomposition [7], but it was not proved that this strategy is the optimal one.

This paper presents a new approach to the multilevel functional decomposition. For each circuit a dedicated strategy of decomposition is evaluated. The strategy defines the methods of decompositions which are applied during each step. In our approach the strategy of decomposition is optimized using the genetic programming. We observed that our method gives significantly better results for the most of the evaluated benchmark circuits. Moreover, using parallel procesing we obtained significant improvement in efficiency, thus our method may be applied to decompose large functions.

In the next section the functional decomposition is described. Section 3 presents the idea of the developmental genetic programming. In the section 4 our method is presented. Section 5 presents the parallelization model of our approach. Section 6 contains experimental results. The paper ends with conclusions.

2 Functional Decomposition

Let F be a multiple-input/multiple-output function. The function may be decomposed using parallel or serial strategy. The parallel decomposition expresses the function $F(X)$ through functions G and H with disjoint sets of output variables. The serial decomposition expresses the function $F(X)$ through functions G and H, such that $F{=}H(U,G(V))$, where $U \cup V{=}I$. If the number of inputs and outputs of the result function does not exceed the number of inputs and outputs in the LUT, then the function is implementable in one LUT cell.

To find the optimal result the following problems of defining the decomposition strategy should be resolved:

1. which decomposition method should be used;
2. which sets of separated inputs (in serial decomposition) or outputs (in parallel decomposition) should be chosen.

The only known solution for the first problem is the balanced decomposition [7]. In this strategy the parallel decomposition is chosen if the decomposed function has more outputs than inputs, otherwise the serial decomposition is applied. However, it was not proved that this strategy is the best one for the functional decomposition. Thus alternative approaches should be studied, and the strategy giving the best final results should be found.

For the variable partitioning problem a lot of heuristics were proposed [8]. In [9] the best variable partition is determined using the information relationship measures. Separated sets of inputs may be also optimised using evolutionary algorithms [10]. An efficient method of finding variable partitions, based on so called r-admissibility was proposed in [8]. A method, applying "divide-and-conquer" paradigm, was presented in [11]. The goal of all above methods is to find the input partitions providing the best serial decomposition of a given input function. It should be noticed that a decision giving the best results in a single step does not guarantee obtaining the optimal solution. Thus the local as well as the global optimisation methods should be studied, to find the best strategy of the multilevel decomposition.

3 Developmental Genetic Programming

Genetic programming (GP) [12] evolves a population of computer programs. The goal is to obtain a program that produce the expected results. This method was applied with success to optimization and development of computing programs, game strategies, control algorithms etc. Unlike classical GA, GP uses variable-length chromosomes which are represented by tree structures.

In the DGP, methods creating solutions evolve, instead of computer programs. In this approach the genotype and the phenotype are distinguished. The genotype is a procedure that constructs the solution of a problem. It is composed of genes representing elementary functions. Phenotype represents the target solution. During evolution only genotypes are evolved, while the genotype-to-phenotype mapping is used to create phenotypes. Next, all genotypes are rated according to the estimated quality of the corresponding phenotypes. The goal of the optimisation is to find the procedure constructing the best solution.

DGP is especially helpful in optimizing solutions of hard-constrained problems. In these cases most of randomly generated solutions are not valid. Thus in classical GAs some restrictions should be applied to enforce genetic operators to produce only legal individuals. But these restrictions may also create infeasible regions in a search space, eliminating sequences of genes which may lead to high quality solutions. This problem does not appear in the DPG, because genotypes are evolved without any restrictions and legal only phenotypes are guaranteed by appropriate genotype to phenotype mapping. DGP proves to be effective in such problems like synthesis of electronic circuits, synthesis of the control algorithms, image recognition, game playing and others [13].

The only constraint in the functional decomposition is that the original behaviour must be preserved. Since generation of networks of Boolean functions which are equivalent to the given function is a hard problem, there is no efficient GA approach for optimising functional decomposition. However, taking into consideration that DGP evolves a system construction procedure instead of the system itself, such an approach seems to be very promising for optimisation of the decomposition strategy for Boolean functions.

All parallel GP approaches may be classified using three main parallelization models: the global model, the coarse-grained (island) model and the fine-grained (grid, cellular) model [14]. Since in the GP the most time consuming step is the fitness evaluation, in the global model the master process manages the whole population by assigning subsets of individuals to slave processes. After computing the fitness the results are sent back to the master process. There is no communication between slave processes. The main problem in this model is a load imbalance caused by the unpredictable time of fitness computation for individuals represented with trees of different sizes. In the island model a population of M individuals is divided into N subpopulations (demes) of M/N individuals. All demes are evolved separately. The information between demes is exchanged periodically by migration of some individuals. In the grid model the population is represented as a network of interconnected individuals. Only neighbours may interact during evolution. Parallelism is achieved by mapping the grid onto a multiprocessor architecture.

It was showed that the island and grid models of parallel genetic algorithms are more efficient than the corresponding sequential implementations. Evolution of many subpopulations reduces the problem of premature convergence, such approach finds also the same solution quality in fewer generations. For GP approaches the cellular model that outperforms both the sequential and the island models was proposed [15]. But there are no any studies concerning the best parallel strategy for DGP approaches.

4 Evolution of the Multilevel Decomposition Strategy

In our method genotypes are represented by binary trees specifying the decomposition strategy. Each node (gene) specifies the decomposition of the function created by the parent node into 2 functions passed to offspring nodes for further processing. Functions created by tree leaves constitute the target solution. The goal of optimisation is to find the solution with minimal cost of implementation for the target FPGA technology.

4.1 Genotypes and Phenotypes

Each gene specifies a decomposition strategy used in a single step. The strategy is defined by the type of decomposition and the rules according to which the sets of separated inputs or outputs are determined. 16 different genes are defined.

The initial generation consists of individuals generated randomly, where all genes are selected with the same probability. The number of nodes in each genotype is calculated according to following formula:

$$G = \Theta\left(\frac{n * m}{I_{LUT}} * A_{0.8}^{1.2}\right) \tag{1}$$

where: n - is the number of inputs, m - is the number of outputs, I_{LUT} – is the number of LUT's inputs, $A_{0.8}^{1.2}$ is a random value from range $[0.8 \ldots 1.2]$, Θ – is a function rounding upward the argument to the nearest natural odd value.

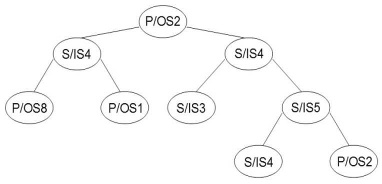


Fig. 1. Sample genotype

A sample genotype tree is shown in Fig.1. The genotype corresponds to a function with 8 inputs and 3 outputs. It is assumed, that LUT has 4 inputs and 1 output. Thus possible number of the nodes is equal to 5, 7 or 9. P means the parallel decomposition, S the serial one, IS3, IS4, IS5 and OS1, OS2, OS8 are different methods of separation of inputs and outputs, respectively.

Genotype to phenotype mapping is done by traversing the genotype in the depth-first order, for each node the decomposition is performed according to rules defined by the corresponding gene. Two exceptions are possible: first, the node has successors but further decomposition is not necessary, second, the node is a leaf but the function requires further decompositions. In the first case, the decomposition of the given subfunction is not continued and useless nodes are removed from the genotype tree immediately. This process is similar to withering of unused features in live organisms. In the second case, the decomposition result is estimated according to the expected cost of implementation (number of LUTs), defined as follows:

$$ECI = 2^{n-k} * m \tag{2}$$

where k is the number of inputs of the target LUT cell. Results estimated by this rule are enough pessimistic to be worse than most of the fully decomposed solutions. Thus such individuals usually became extinct very fast.

The phenotype is a network of functions which is functionally equivalent to the original function. According to the sample genotype shown in Fig.1, function F was decomposed into 5 smaller functions: *gg*, *gh*, *hg*, *hhg* and *hhh* (Fig.2).

4.2 Genetic Operators

Each generation contains the same number of individuals. To ensure that the examined solution space will be proportional to the function complexity, the size of the generation depends on the number of inputs and outputs of the original function. Thus the number of individuals is calculated according to the following formula:

$$N = (n + m) * \Omega \tag{3}$$

where Ω is a DGP parameter.

Genotypes are ranked according to the implementation cost of the corresponding phenotypes. Solutions which require less LUT cells for implementation have

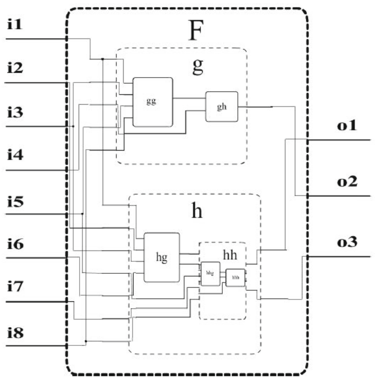


Fig. 2. The phenotype mapped from the genotype presented in Fig. 1

higher position in the ranking. All genotypes are evolved randomly with the probability P that depends on the quality of the solution as follows:

$$P = \frac{N - R}{N} \tag{4}$$

where R is the position in the ranking.

Reproduction copies the best individuals from the current generation to the next generation. Cross-over selects randomly 2 genotypes with the probability P. Next, both trees are pruned by removing randomly selected edge. Then subtrees are swapped between both parent genotype. In that way 2 new individuals are created and added to the next generation.

Mutation selects randomly one genotype with the probability P. Next, one of the following modifications is done for the chosen genotype:

– randomly selected gene is changed to another,
– randomly selected edge is pruned and the subtree is removed,
– two random nodes are created and added to the randomly selected leaf.

Each type of modification is selected with the same probability, but for single-node genotypes the subtree extraction can not be selected. Implementation of genetic operators ensures that the correct genotype-tree structure is always kept.

The new generation is created using the above operations in the following way: $r = \alpha * N$ individuals are reproduced, $c = \beta * N$ individuals are created using cross-over, and $m = \gamma * N$ individuals are created using mutation. N is calculated according to the formula (3), and α, β and γ are the DGP parameters. The following requirement must be fulfilled:

$$\alpha + \beta + \gamma = 1 \tag{5}$$

If the algorithm does not find any bettter solution in λ succeeding generations, the evolution stops.

5 Parallel DGP Model

In our approach we propose the island model of parallelization, where migration is controlled by additional process called the migration manager (MM). There is no direct communication between islands, each island communicate only with the migration manager. The size of each subpopulation may vary, the best demes grow up while the worst are diminished. This process is also controlled by the MM. Since worse individuals usually require more time to compute fitness (greater number of subfunctions requires more decomposition steps), thus variable size of demes leads also to better load balancing.

Migration is an interchange of group of individuals between different islands. The migration rate and the migration frequency are parameters of the algorithm. Candidates for migration are chosen using the selection operator (p.4.2). The migration is asynchronous, the migrated groups are buffered by the MM.

The main principles of the parallel DGP algorithm are the following:

1. For each island the initial population, with the same number of individuals, is generated.
2. Quality of each deme is evaluated after every 5 generations. The quality of the deme is the arithmetic mean of fitnesses computed for 20% of the best individuals. The MM increases the number of individuals in the best deme by 5%, while the size of the worst deme is decreased by 5%.
3. When the migration should be performed, the group of individuals is selected and sent to the MM, than the process is suspended waiting for the response.
4. The MM accepts the group of immigrants and stores it in the FIFO (first in first out) queue.
5. If there is any group of immigrants sent from other deme, the MM sends it. In other case an empty message is sent.
6. When the process will receive answer from the MM, it resumes computation. If group of immigrants is got, it will participate in further evolution.

Values of migration parameters have been attuned experimentally.

6 Experimental Results

The described method has been implemented and evaluated with the help of some MCNC benchmarks [16]. The following values of DGP parameters have been assumed: Ω=12, α=0.05, β=0.65, γ=0.3, λ=20. The same experiments were performed using other existing methods of logic synthesis for FPGAs: GUIDek [17], an automatic, academic tool which decomposes functions using deterministic approach, it also uses the functional decomposition, the ABC system [18], it is also an academic tool but it uses approach based on cofactoring, and commercially available tool Quartus II v.10.0 from Altera Corp. Experimental results are shown in Tab. 1. The following columns contain: the name of the benchmark, results obtained using our (DGP) method(the best results obtained in 20 trials), and results obtained using GUIDek, ABC and Quartus. The results represent the cost of the FPGA implementation (number of 4-input LUT cells).

Table 1. Decompositon results

Benchmark	**DGP**	**GUIDek**	**ABC**	**Quartus II**
5xp1	18	22	34	33
9sym	8	9	92	9
dk17	27	32	34	31
dk27	12	13	11	11
f51m	15	22	39	19
inc	27	35	39	42
m1	20	25	23	20
misex1	16	20	14	22
newcpla2	24	33	26	24
rd53	5	5	14	7
rd73	9	9	43	11
seq	18	23	22	20
sqn	21	24	42	33
sqrt8	11	13	17	14
squar5	13	14	17	16
t4	15	17	14	14
tms	60	73	87	80
$\sum$	**321**	**389**	**568**	**406**
Avg. gain	–	**18%**	**44%**	**21%**

Table 2. Computation time results $[s]$

Benchmark	**I core**	**II cores**	**III cores**	**IV cores**
5xp1	468,5	280,1	202,7	182,3
9sym	15,0	9,8	7,6	6,4
dk17	140,0	71,1	46,5	68,3
dk27	1,2	0,8	0,6	0,5
f51m	163,0	121,3	81,5	57,4
inc	578,0	350,5	241,2	165,6
m1	102,0	50,1	30,5	27,6
misex	63,0	45,7	35,8	19,7
newcpla2	146,0	104,0	75,3	60,4
rd53	0,6	0,43	0,31	0,28
rd73	7,5	5,1	3,7	3,3
seq	272,0	161,4	134,5	100,3
sqn	40,7	20,4	14,5	11,6
sqrt8	12,0	7,2	5,3	4,4
squar5	5,9	3,4	2,3	2,2
t4	249,0	120,0	92,8	58,5
$\sum$	**2264**	**1351**	**975**	**768**
Avg. speed up	–	**1,68**	**2,32**	**2,95**

The efficiency of the proposed DGP parallel model was evaluated using implementation dedicated to multicore architectures, with migration after every 10 generations. CPU runtimes for various number of utilized processor cores are shown in Tab.2. All experiments were run using AMD64 Phenom x4 2.8 GHz processor. Tab.2 presents preliminary results, the implementation is not fully optimized, nevertheless the average speedup is quite good, for some examples we obtained near linear speedup.

One of the most important factor deciding about the efficiency of the parallel approach is the CPU load. Table 3 presents the CPU load for each core. The average load is about 90%, but it may be improved by applying more advanced load balance methods.

Table 3. CPU cores' load [%]

Benchmark	**I*st* core**	**II*nd* core**	**III*rd* core**	**IV*th* core**	**Avg. CPU load**
dk17	97	100	92	88	**94**
dk27	95	82	100	95	**93**
f51m	100	93	86	92	**92**
m1	100	86	86	86	**89**
rd73	78	100	80	75	**83**
seq	93	97	100	96	**96**
Avg.	94	93	91	89	**91,2**

7 Conclusions

In this paper the parallel developmental genetic programming was applied to the problem of functional decomposition of Boolean functions. In our approach the multilevel decomposition strategy for given function evolves, instead of the solutions itself. In that way we use strategy optimized for the given system instead of the global decomposition strategy. To our best knowledge this is the first DGP approach targeting the multilevel decomposition problem and the first parallel DGP approach.

Preliminary results show, that the method is efficient; it gives significantly better results than existing methods. Future work will concentrate on analyzing other implementations of genetic operators. We will work also on optimization of the parallel DGP model by developing the load balancing and developing the model dedicated to cluster architectures.

References

1. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
2. Keller, R.E., Banzhaf, W.: The evolution of genetic code in genetic programming. In: Proc. of the Genetic and Evolutionary Computation Conf., pp. 1077–1082 (1999)

3. Ashenhurst, R.L.: The Decomposition of Switching Functions. In: Proc. of International Symposium on Theory of Switching Functions, pp. 74–116 (1957)
4. Ashar, P., Devadas, S., Newton, A.R.: Sequential Logic Synthesis. Kluwer Academic Publisher, Norwell (1992)
5. Scholl, C.: Functional Decomposition with Application to FPGA Synthesis. Kluwer Academic Publishers (2001)
6. Brzozowski, J., Luba, T.: Decomposition of Boolean Functions Specified by Cubes. Journal of Mult.-Valued Logic & Soft Computing 9, 377–417 (2003)
7. Nowicka, M., Luba, T., Rawski, M.: FPGA-Based Decomposition of Boolean Functions. Algorithms and Implementation. In: Proc. of the 6th International Conference on Advanced Computer Systems, Szczecin (1999)
8. Muthukumar, V., Bignall, R.J., Selvaraj, H.: An efficient variable partitioning approach for functional decomposition of circuits. Journal of Systems Architecture 53, 53–67 (2007)
9. Rawski, M., Jóźwiak, L., Łuba, T.: Functional decomposition with an efficient input support selection for sub-functions based on information relationship measures. Journal of Systems Architecture 47/2, 137–155 (2001)
10. Rawski, M.: Efficient Variable Partitioning Method for Functional Decomposition. Electronics and Telecommunications Quarterly 53(1), 63–81 (2007)
11. Morawiecki, P., Rawski, M., Selvaraj, H.: Input variable partitioning method for functional decomposition of functions specified by large truth tables. In: Proceedings of Int. Conf. on Comp. Intelligence and Multimedia Applications, pp. 164–168 (2007)
12. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
13. Koza, J.R.: Human-competitive results produced by genetic programming. In: Genetic Programming and Evolvable Machines, pp. 251–284 (2010)
14. Tomassini, M.: Parallel and distributed evolutionary algorithms: A review. In: Neittaanmki, P., Miettinen, K., Mkel, M., Periaux, J. (eds.) Evolutionary Algorithms in Engineering and Computer Science. J. Wiley and Sons, Chichester (1999)
15. Folino, G., Pizzuti, C., Spezzano, G.: A scalable cellular implementation of parallel genetic programming. IEEE Transactions on Evolutionary Computation 7(1), 37–53 (2003)
16. Yang, S.: Logic synthesis and optimization benchmarks. version 3.0. Microelectronics Center of North Carolina, Tech. Rep. (1991)
17. http://rawski.zpt.tele.pw.edu.pl/pl/node/161
18. http://www.eecs.berkeley.edu/~alanmi/abc

The Nine Neighbor Extrapolated Diffusion Method for Weighted Torus Graphs

Katerina A. Dimitrakopoulou and Michail N. Misyrlis

Department of Informatics and Telecommunications,
University of Athens,
Panepistimiopolis, 157 84, Athens, Greece
{kdim,misyrlee}@di.uoa.gr

Abstract. The convergence analysis of the Extrapolated Diffusion (EDF) was developed in [7] and [8] for the weighted torus and mesh graphs, respectively using the set $\mathcal{N}_1(i)$ of nearest neighbors of a node i in the graph. In the present work we propose a Diffusion scheme which employs the set $\mathcal{N}_1(i) \cup \mathcal{N}_2(i)$, where $\mathcal{N}_2(i)$ denotes the four neighbors of node i with path length two (see Figure 1) in order to increase the convergence rate. We study the convergence analysis of the new Diffusion scheme with nine neighbors (NEDF) for weighted torus graphs. In particular, we find closed form formulae for the optimum values of the edge weights and the extrapolation parameter. A 60% increase in the convergence rate of NEDF compared to the conventional EDF method is shown analytically and numerically.

Keywords: Laplacian matrix, load balancing, weighted torus, iterative diffusion, Fourier analysis.

1 Introduction

The performance of a balancing algorithm can be measured in terms of number of iterations it requires to reach a balanced state and in terms of the amount of load moved over the edges of the underlying processor graph. In the Diffusion (DF) method [3], [1] a processor simultaneously sends workload to its neighbors with lighter workload and receives from its neighbors with heavier workload. More specifically DF is given by the following iterative scheme

$$u_i^{(n+1)} = u_i^{(n)} - \sum_{j \in N(i)} c_{ij}(u_i^{(n)} - u_j^{(n)}), \quad n = 0, 1, 2, \dots, \tag{1}$$

where $c_{ij} > 0$ are the edge weights (or diffusion parameters), $N(i)$ is the set of the nearest neighbors of node i of the graph $G = (V, E)$ and $u_i^{(n)}, i = 0, 1, 2, \dots, |V|$ is the load after the nth iteration on node i. The main problem here is the determination of the parameters c_{ij} such that the rate of convergence of DF is maximized. Various attempts to solve this problem are presented in [3,6,4,5] and [9]. In [7] the following Extrapolated Diffusion (EDF) method was introduced

$$u_i^{(n+1)} = u_i^{(n)} - \tau \sum_{j \in \mathcal{N}_1(i)} c_{ij}(u_i^{(n)} - u_j^{(n)}), \quad n = 0, 1, 2, \ldots, \tag{2}$$

where $\tau \in \mathbb{R} \setminus \{0\}$ and $c_{ij} > 0$ are the extrapolation parameter and the edge weights, respectively, $\mathcal{N}_1(i)$ is the set of the nearest neighbors of the node i of the graph $G = (V, E)$ and $u_i^{(n)}, i = 0, 1, 2, \ldots, |V|$ is the load after the nth iteration on node i. In fact, we let the extrapolation parameter vary for each node i, so instead of τ in (2) we introduced a set of parameters $\tau_i, i = 0, 1, 2, \ldots, |V|$ and called (2) the local EDF method [7,8]. Note that if $\tau = 1$, (2) is the classical Diffusion (DF) method [3,1]. In [9] optimum values for the edge weights c_{ij} were determined under the hypothesis that they are all equal. The generalized problem of determining optimum values for the edge weights c_{ij} and τ_i was first solved in [7]. In particular, closed form formulae for the optimum values for τ_is and c_{ij}s were determined for the nD-torus [7] and nD-mesh [8] by applying local Fourier analysis [2]. As a result, it was found that for square tori and meshes EDF coincides with DF whereas for orthogonal tori and meshes it becomes twice as fast as DF. In addition, it was shown that EDF for orthogonal tori is four times faster than orthogonal meshes [8].

In the present work we consider the case of increasing the number of neighbors of node i for the computation of its load. In other words we consider the nine neighbor weighted Laplacian matrix formed not only by the five nearest neighbor nodes but also by their four neighbors with path length two from node i in an attempt to increase the convergence rate of the EDF method. As a consequence, we show that the rate of convergence of EDF is improved asymptotically by 60% for torus graphs.

The rest of the paper is organized as follows: In section 2 we introduce the local NEDF method using nine neighbors of node i. In section 3 we find closed form formulae for the optimum values of the parameters τ_i such that the convergence of the local NEDF method is maximized. The determination of the optimum values for the edge weights c_{ij} is presented in section 4 with a comparison of the NEDF and EDF methods with five and nine neighbors, respectively for torus graphs. Section 5 presents our numerical experiments and finally, in section 6 we state our conclusions.

2 The Nine Neighbor Extrapolated Diffusion (NEDF) Method

Let $G = (V, E)$ be a connected, weighted undirected graph with $|V|$ nodes and $|E|$ edges. Let $c_{ij} \in \mathbb{R}^+$ be the weight of edge $(i, j) \in E, u_i \in \mathbb{R}$ be the load of node $v_i \in V$ and $u \in \mathbb{R}^{|V|}$ be the vector of load values. Let us consider the iterative scheme that requires communication not only with adjacent nodes but with their four neighbors also,

$$u_i^{(n+1)} = u_i^{(n)} - \tau \sum_{j \in \mathcal{N}(i)} c_{ij}(u_i^{(n)} - u_j^{(n)}), \tag{3}$$

where $\tau \in \mathbb{R} \setminus \{0\}$ and $c_{ij} > 0$ for $i = 1, 2, \ldots, |V|$ and $j \in \mathcal{N}(i) = \mathcal{N}_1(i) \cup \mathcal{N}_2(i)$ are parameters that play an important role in the convergence of the whole system to the equilibrium state and $\mathcal{N}_2(i)$ is the set of the cross-shaped neighbors with path length two (see Figure 1). Then, the overall workload distribution at step n, denoted by $u^{(n)}$, is the transpose of the vector $(u_1^{(n)}, u_2^{(n)}, \ldots, u_{|V|}^{(n)})$ and $u^{(0)}$ is the initial workload distribution. The iterative scheme (3) will be referred to as the Nine neighbor Extrapolated Diffusion (NEDF) method and has the matrix form

$$u^{(n+1)} = Mu^{(n)}, \tag{4}$$

where $M \in \mathbb{R}^{|V| \times |V|}$ is called the *diffusion matrix*. M can be written as

$$M = I - \tau L, \qquad L = D - A, \tag{5}$$

where L is the nine neighbor Laplacian matrix, $D = diag(L)$, A is the nine neighbor weighted adjacency matrix of G. The elements of A, a_{ij}, are

$$a_{ij} = \begin{cases} c_{ij} & \text{if } j \in \mathcal{N}(i), \\ 0 & otherwise. \end{cases} \tag{6}$$

The diffusion matrix M must have the following properties [3], [1]: nonnegative, symmetric and stochastic. Before we close this section we consider the following version of NEDF, which involves a set of parameters $\tau_i, i = 1, 2, \ldots, |V|$

$$u_i^{(n+1)} = (1 - \tau_i \sum_{j \in \mathcal{N}(i)} c_{ij}) u_i^{(n)} + \tau_i \sum_{j \in \mathcal{N}(i)} c_{ij} u_j^{(n)}. \tag{7}$$

Note that if $\tau_i = \tau$ for $i = 1, 2, \ldots, |V|$, then (7) yields the NEDF method. The iterative scheme (7) will be referred to as the local NEDF method.

3 Quasi Optimum τ_i

In this section we determine quasi optimum values for the parameters τ_i in case of weighted torus graphs. We define M_{ij} as the local NEDF operator for the $N_1 \times N_2$ torus and apply Fourier analysis, in a similar way as in [7], to find its eigenvalues in terms of the edge weights. The local NEDF scheme at a node (i, j) can be written as:

$$u_{ij}^{(n+1)} = M_{ij} u_{ij}^{(n)}, \tag{8}$$

where

$$M_{ij} = 1 - \tau_{ij} L_{ij}. \tag{9}$$

Next, we define

$$\begin{aligned} L_{ij} = d_{ij} - (&c_{i,i+1}^j E_1 + c_{i,i-1}^j E_1^{-1} + c_{j,j+1}^i E_2 + c_{j,j-1}^i E_2^{-1} + \\ &c_{i+1,i+2}^j E_1^2 + c_{i-1,i-2}^j E_1^{-2} + c_{j+1,j+2}^i E_2^2 + c_{j-1,j-2} E_2^{-2}) \end{aligned} \tag{10}$$

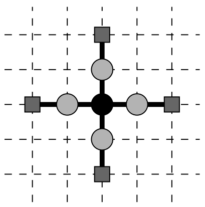


Fig. 1. ◯ denotes neighbors with path length one, □ denotes cross-shape neighbors with path length two

the local operator of the nine neighbor Laplacian matrix, with $d_{ij} = c^j_{i,i+1} + c^j_{i,i-1} + c^j_{i+1,1+2} + c^j_{i-1,i-2} + c^i_{j,j+1} + c^i_{j,j-1} + c^i_{j+1,j+2} + c^i_{j-1,j-2}$, where c^t_{rs} denotes the weight of the edge (r,s), in the t direction of the torus. The operators E_1, E_1^{-1}, E_2, E_2^{-1}, E_1^2, E_1^{-2}, E_2^2 and E_2^{-2} are defined as: $E_1 u_{ij}{=}u_{i+1,j}$, $E_1^{-1}u_{ij}{=}u_{i-1,j}$, $E_2 u_{ij}{=}u_{i,j+1}$, $E_2^{-1}u_{ij}{=}u_{i,j-1}$, $E_1^2 u_{ij}{=}u_{i+2,j}$, $E_1^{-2}u_{ij}{=}u_{i-2,j}$, $E_2^2 u_{ij}{=}u_{i,j+2}$, $E_2^{-2}u_{ij}{=}u_{i,j-2}$, which are the *forward-shift* and *backward-shift* operators in the x_1-direction, (x_2-direction), respectively with $u_{ij} = u(ih_1, jh_2) = u(x_1, x_2)$, where $x_1 = ih_1$, $x_2 = jh_2$, $h_1{=}\frac{1}{N_1}$ and $h_2{=}\frac{1}{N_2}$.
Since the Laplacian matrix L is symmetric, we impose the conditions $c^j_{i,i+1}{=}c^j_{i,i-1}$ and $c^i_{j,j+1}{=}c^i_{j,j-1}$. Next, we use the following notation for the edge weights:

$$c_i^{(1)} = c^j_{i,i+1} \quad i = 1, 2, \ldots, N_1, \text{ and } c_j^{(2)} = c^i_{j,j+1}, \; j = 1, 2, \ldots, N_2 \tag{11}$$

for the rows and columns, respectively. From (10) and (11), it follows that

$$L_{ij} = d_{ij} - [c_i^{(1)}(E_1 + E_1^{-1} + E_1^2 + E_1^{-2}) + c_j^{(2)}(E_2 + E_2^{-1} + E_2^2 + E_2^{-2})] \tag{12}$$

where $d_{ij} = 4(c_i^{(1)} + c_j^{(2)})$. The eigenvalues μ_{ij}, λ_{ij} of the local operators M_{ij}, L_{ij}, respectively, are related as follows:

$$\mu_{ij} = 1 - \tau_{ij}\lambda_{ij}. \tag{13}$$

Lemma 1. *The spectrum of the operator L_{ij} is given by*

$$\lambda_{ij}(k_1, k_2) = 2[c_i^{(1)}(2-\cos k_1h_1-\cos 2k_1h_1)+c_j^{(2)}(2-\cos k_2h_2-\cos 2k_2h_2)], \tag{14}$$

where $i = 1, 2, \ldots, N_1$, $j = 1, 2, \ldots, N_2$, $k_1 = 2\pi\ell_1$, $\ell_1 = 0, 1, 2, \ldots, N_1 - 1$, $k_2 = 2\pi\ell_2$ *and* $\ell_2 = 0, 1, 2, \ldots, N_2 - 1$.

Proof. If the input error function $e_{ij}^{(n)}$ is the complex sinusoid $e^{i(k_1x_1+k_2x_2)}$ we have

$$L_{ij}e^{i(k_1x_1+k_2x_2)} = \lambda_{ij}(k_1, k_2)e^{i(k_1x_1+k_2x_2)},$$

which, because of (12), yields

$$\begin{aligned}\lambda_{ij}(k_1, k_2) = d_{ij} - [&c_i^{(1)}(e^{ik_1h_1} + e^{-ik_1h_1} + e^{2ik_1h_1} + e^{-2ik_1h_1})+\\ &c_j^{(2)}(e^{ik_2h_2} + e^{-ik_2h_2} + e^{2ik_2h_2} + e^{-2ik_2h_2})].\end{aligned} \tag{15}$$

So we may view $e^{i(k_1x_1+k_2x_2)}$ as an eigenfunction of L_{ij} with eigenvalues $\lambda_{ij}(k_1, k_2)$ given by (15). It is easily verified that (15) yields (14). □

In our case γ_{ij} is a spatially varying function (see (14)) and generally is not equal to the convergence factor $\gamma(M)$ of the NEDF method. Nevertheless, if the edge weights are all equal to a constant value, then M_{ij} and hence γ_{ij} are space invariant in which case γ_{ij} is equal to $\gamma(M)$ [2]. Note that

$$\gamma_{ij}(M_{ij}) = \max_{k_1,k_2} |\mu_{ij}(k_1, k_2)|, \tag{16}$$

where not both k_1, k_2 can take the value zero. From (13) and (16) it follows that the minimum value of γ_{ij} with respect to τ_{ij} is attained at [10]

$$\tau_{ij}^{opt} = \frac{2}{\lambda_{i,j,2} + \lambda_{i,j,N}}, \tag{17}$$

where $\lambda_{i,j,2}, \lambda_{i,j,N}$ are the smallest and largest eigenvalues of the operator L_{ij}, respectively. Thus, the corresponding minimum value of $\gamma_{ij}(M_{ij})$ is given by

$$\gamma_{ij}^{opt} = \frac{P_{ij} - 1}{P_{ij} + 1}, \tag{18}$$

where

$$P_{ij} = \frac{\lambda_{i,j,N}}{\lambda_{i,j,2}} \tag{19}$$

is the P-condition number of L_{ij}. The last quantity plays an important role in the behavior of γ_{ij}^{opt}. Indeed, from (18) it follows that γ_{ij}^{opt} is a increasing function of P_{ij}. Therefore, minimization of P_{ij} has the effect of maximizing R(NEDF), the rate of convergence of the local NEDF method, defined by [10]

$$R(NEDF) = -\log \gamma_{ij}^{opt} \simeq \frac{2}{P_{ij}}. \tag{20}$$

Theorem 1. *The convergence factor $\gamma_{ij}(M_{ij})$ of the operator M_{ij} is minimized at*

$$\tau_{ij}^{opt} = \begin{cases} \dfrac{8}{25c_j^{(2)} + c_i^{(1)}(41 - 8\cos\frac{2\pi}{N_1} - 8\cos\frac{4\pi}{N_1})}, & \sigma_{ij} \geq \sigma \\[2ex] \dfrac{8}{25c_i^{(1)} + c_j^{(2)}(41 - 8\cos\frac{2\pi}{N_2} - 8\cos\frac{4\pi}{N_2})}, & \sigma_{ij} \leq \sigma, \end{cases} \tag{21}$$

and its corresponding minimum is

$$\gamma_{ij}^{opt} = \begin{cases} \dfrac{25c_j^{(2)} + c_i^{(1)}(9 + 8\cos\frac{2\pi}{N_1} + 8\cos\frac{4\pi}{N_1})}{25c_j^{(2)} + c_i^{(1)}(41 - 8\cos\frac{2\pi}{N_1} - 8\cos\frac{4\pi}{N_1})}, & \sigma_{ij} \geq \sigma \\[2ex] \dfrac{25c_i^{(1)} + c_j^{(2)}(9 + 8\cos\frac{2\pi}{N_2} + 8\cos\frac{4\pi}{N_2})}{25c_i^{(1)} + c_j^{(2)}(41 - 8\cos\frac{2\pi}{N_2} - 8\cos\frac{4\pi}{N_2})}, & \sigma_{ij} \leq \sigma, \end{cases} \tag{22}$$

where σ_{ij} and σ are given by (24).

Proof. The optimum value for τ_{ij} will be determined by (17), while the minimum value of γ_{ij}^{opt} by (18) and (19). It is therefore necessary to determine $\lambda_{i,j,2}$ and $\lambda_{i,j,N}$. For the determination of $\lambda_{i,j,2}$ we let ℓ_1=0 and ℓ_2=1, or ℓ_1=1 and ℓ_2=0 in (14) thus obtaining $\lambda_{i,j,2} = 2c_j^{(2)}(1 - \cos\frac{2\pi}{N_2} - \cos\frac{4\pi}{N_2})$ or $\lambda_{i,j,2} = 2c_i^{(1)}(2 - \cos\frac{2\pi}{N_1} - \cos\frac{4\pi}{N_1})$, for each of the above choices of ℓ_1, ℓ_2, respectively, which lead to the following

$$\lambda_{i,j,2} = \begin{cases} 2c_i^{(1)}(2 - \cos\frac{2\pi}{N_1} - \cos\frac{4\pi}{N_1}),\ \sigma_{ij} \geq \sigma \\ \\ 2c_j^{(2)}(1 - \cos\frac{2\pi}{N_2} - \cos\frac{4\pi}{N_2}),\ \sigma_{ij} \leq \sigma, \end{cases} \tag{23}$$

$$\text{where: } \sigma_{ij} = \frac{c_j^{(2)}}{c_i^{(1)}} \ \text{ and } \ \sigma = \frac{2 - \cos\frac{2\pi}{N_1} - \cos\frac{4\pi}{N_1}}{2 - \cos\frac{2\pi}{N_2} - \cos\frac{4\pi}{N_2}}. \tag{24}$$

By studying the behavior of (14) with respect to $\cos k_1 h_1$ and $\cos k_2 h_2$ it is readily verified that its maximum is attained at $\cos k_1 h_1$=$\cos k_2 h_2$=$-\frac{1}{4}$. Therefore, from (14) the maximum eigenvalue $\lambda_{i,j,N}$ is given by

$$\lambda_{i,j,N} = \frac{25}{4}(c_i^{(1)} + c_j^{(2)}). \tag{25}$$

Using the expressions of $\lambda_{i,j,2}$ and $\lambda_{i,j,N}$ given by (23) and (25), respectively in (17), (18) and (19), we easily verify (21) and (22). □

4 Determination of Optimum $\mathbf{c_i^{(1)}}$ and $\mathbf{c_j^{(2)}}$

We will determine the $c_i^{(1)}$'s and $c_j^{(2)}$'s such that P_{ij} (and hence γ_{ij}) is minimized.

Theorem 2. *The convergence factor $\gamma_{ij}(M_{ij})$ is minimized when*

$$c_j^{(2)} = \sigma c_i^{(1)} \tag{26}$$

$$\text{and } \tau_{ij}^{opt} = \tau^{opt}/c_i^{(1)}, \ \ c_i^{(1)} \text{ arbitrary} \tag{27}$$

for any $i = 1, 2, \ldots, N_1,\ j = 1, 2, \ldots, N_2$, where

$$\tau^{opt} = \frac{8}{25\sigma - 8(\cos\frac{2\pi}{N_1} + \cos\frac{4\pi}{N_1}) + 41} \tag{28}$$

and its corresponding minimum is

$$\gamma^{opt} = \frac{25\sigma + 8(\cos\frac{2\pi}{N_1} + \cos\frac{4\pi}{N_1}) + 9}{25\sigma - 8(\cos\frac{2\pi}{N_1} + \cos\frac{4\pi}{N_1}) + 41}. \tag{29}$$

Proof. The P-condition number of L_{ij} is, because of (25) and (23), given by

$$P_{ij}(L_{ij}) = \begin{cases} \dfrac{25(c_i^{(1)} + c_j^{(2)})}{8c_i^{(1)}(1 - \cos\frac{2\pi}{N_1} - \cos\frac{4\pi}{N_1})}, \ \sigma_{ij} \geq \sigma \\ \dfrac{25(c_i^{(1)} + c_j^{(2)})}{8c_j^{(2)}(1 - \cos\frac{2\pi}{N_2} - \cos\frac{4\pi}{N_2})}, \ \sigma_{ij} \leq \sigma. \end{cases} \tag{30}$$

Studying the behavior of the above expression with respect to σ_{ij} we can easily verify that it is minimized at σ. Therefore, (17), because of (25) and (23), yields the optimum value of τ_{ij}^{opt} given by (27). From (30) and (26) it follows that

$$P_{ij}(L_{ij}) = \frac{25(1+\sigma)}{8(1 - \cos\frac{2\pi}{N_1} - \cos\frac{4\pi}{N_1})}. \tag{31}$$

Finally, the optimum value for $\gamma_{ij}(M_{ij})$ is obtained from (18) using (31). □

Similar results hold in case $c_j^{(2)}$ is arbitrary.

Corollary 1. *If the edge weights in one dimension of a 2D-torus are all equal to the same constant value and (26) holds, then $\gamma(M)$, the convergence factor of the diffusion matrix M, is minimized at τ_{NEDF}^{opt}, given by (27) and its corresponding minimum is γ^{opt}, where τ^{opt}, γ^{opt} are given by (28) and (29), respectively.*

Corollary 2. *Under the hypothesis of corollary 1 and if $N{=}N_1{=}N_2$, then the convergence factor $\gamma(M)$ is minimized at τ^{opt} given by (27) with $\sigma = 1$,*

$$\tau^{opt} = \frac{4}{33 - 4(\cos\frac{2\pi}{N} + \cos\frac{4\pi}{N})} \tag{32}$$

and its corresponding minimum is given by

$$\gamma^{opt} = \frac{17 + 4(\cos\frac{2\pi}{N} + \cos\frac{4\pi}{N})}{33 - 4(\cos\frac{2\pi}{N} + \cos\frac{4\pi}{N})}. \tag{33}$$

Proof. If $N{=}N_1{=}N_2$, then $\sigma{=}1$ hence (32) and (33) are direct results of (28) and (29), respectively. □

Corollary 3. *Under the hypothesis of corollary 2 then*

$$\lim_{N\to\infty} \frac{R(NEDF)}{R(EDF)} = 3.2 \tag{34}$$

Proof. From (20) we have

$$R(NEDF) \simeq \frac{80\pi^2 h^2}{25}. \tag{35}$$

Moreover, it is known (see (35) of [7]) that for EDF

$$P_{ij}^{(EDF)}(L_{ij}) = \frac{4}{1 - \cos\frac{2\pi}{N}}. \tag{36}$$

From (20) and (36) we have

$$R(EDF) \simeq \pi^2 h^2. \tag{37}$$

Clearly, using (35) and (37) we obtain (34). □

Letting $c_i^{(1)} = 1$ for any $i = 1, 2, \ldots, N_1$ in (26) we obtain $c_j^{(2)} = \sigma$ for any $j = 1, 2, \ldots, N_2$, which is one of the infinite optimum values one can obtain by this relation. We will refer to this choice for the edge weights as the *normalized* one. From corollary 1 we have the following.

Corollary 4. *For the normalized edge weights the convergence factor $\gamma(M)$ is minimized at $\tau_{NEDF}^{opt} = \tau^{opt}$ and its corresponding minimum is given by γ^{opt}, where τ^{opt}, γ^{opt} are given by (28) and (29), respectively.*

4.1 The Stretched Torus

In order to be able to have a direct comparison of the convergence behavior of the NEDF and EDF methods we study the case, where one dimension of the torus is large compared to the other one (stretched torus).

Corollary 5. *For stretched torus under the hypothesis of corollary 1 we have*

$$\lim_{N\to\infty} \frac{R(NEDF)}{R(EDF)} = 3.2 \tag{38}$$

where $N = N_1$ or N_2.

Proof. Let $N_1 \gg N_2$ be both even, then from (20) and (31) we have then

$$R(NEDF) = -\log \gamma_{NEDF}^{opt} \simeq \frac{16\left(2 - \cos\frac{2\pi}{N_1} - \cos\frac{4\pi}{N_1}\right)}{25(1+\sigma)}. \tag{39}$$

From (39) it follows

$$\gamma_{NEDF}^{opt} = \frac{32\pi^2 h_1^2}{5(1+\sigma)} \tag{40}$$

since $\cos x \simeq 1 - \frac{x^2}{2}$. Similarly, for the EDF method we have

$$R(EDF) = -\log(\gamma_{EDF}^{opt}) \simeq \frac{2\pi^2 h_1^2}{1+\sigma_2}. \tag{41}$$

By dividing (39) by (41) we obtain

$$\frac{R(NEDF)}{R(EDF)} \simeq \frac{32(1+\sigma_2)}{10(1+\sigma)} \tag{42}$$

and noting that $\sigma \to 0$, $\sigma_2 \to 0$ for $N_1 \to \infty$ and N_2 fixed, (42) yields (38). If now $N_1 \ll N_2$, then

$$\gamma_{NEDF}^{opt} = \frac{160\pi^2 h_2^2}{25\left(1 + \frac{1}{\sigma}\right)} \tag{43}$$

$$\gamma_{EDF}^{opt} = \frac{2\pi^2 h_2^2}{1 + \frac{1}{\sigma_2}} \tag{44}$$

hence

$$\frac{R(NEDF)}{R(EDF)} \simeq \frac{32\left(1 + \frac{1}{\sigma_2}\right)}{10\left(1 + \frac{1}{\sigma}\right)}. \tag{45}$$

But now $\sigma \to \infty$, $\sigma_2 \to \infty$ for $N_2 \to \infty$ and N_1 fixed, hence (45) yields (38). Following a similar treatment for the other cases (both N_1, N_2 are odd, or $N_1(N_2)$ is even and $N_2(N_1)$ odd) we can easily verify that (38) holds also in these cases.

5 Numerical Experiments

In order to test our theoretical results obtained so far we applied NEDF for different sizes of tori. The initial load of the network was placed on a single node of the graph, while we normalized the balanced load $\overline{u}$=1. Hence, the total number of amount of load was equal to the total number of nodes in the graph. For purposes of comparison we considered the application of the NEDF and EDF methods with optimum parameters (*normalized edge weights*) and kept iterating until an almost evenly distributing flow was calculated. The iterations were terminated when the criterion

$$\|u^{(n)} - \overline{u}\|_2 / \|u^{(0)} - \overline{u}\|_2 < \epsilon$$

with $\epsilon = 10^{-7}$ was satisfied. A comparison of the number of iterations is presented in Tables 1 and 2 for both aforementioned methods. The fourth column shows the ratio of the number of iteration of EDF over NEDF. These results clearly show that the rate of convergence of NEDF becomes about three times as fast as the EDF method verifying corollaries 5 and 3.

Table 1. Orthogonal tori

$N_1 \times N_2$	n_{EDF}	n_{NEDF}	$\frac{n_{EDF}}{n_{NEDF}}$
6×6	60	22	2.72
6×10	107	37	2.89
6×20	325	105	3.09
6×50	1,812	570	3.17
6×100	6,982	2,196	3.18

Table 2. Square tori

$N \times N$	n_{EDF}	n_{NEDF}	$\frac{n_{EDF}}{n_{NEDF}}$
10×10	153	51	3.00
20×20	569	180	3.16
40×40	2,149	669	3.21
80×80	8,136	2,538	3.20
120×120	17,709	5,551	3.19

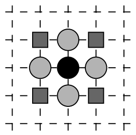


Fig. 2. ◯ denotes neighbors with path length one, □ denotes diagonal-shape neighbors with path length two

6 Conclusions and Future Work

In an attempt to improve the convergence rate of the EDF method, we considered additional nodes apart from the nearest neighbors of a node for the computation of its load. More specifically, we considered the four cross-shape nodes of path length two (see Fig. 1). In this case circulant matrix theory [9] cannot be applied since the diffusion matrix of the NEDF method is not circulant. To overcome this difficulty we used local Fourier Analysis to find the eigenvalues of the diffusion matrix in order to determine the optimum values of the edge weights and the extrapolation parameter of the NEDF method. Our results are presented by corollaries 3 and 5 which state that the rate of convergence of NEDF is 3.2 times faster than that of EDF asymptotically for torus graphs. However, since NEDF requires approximately twice as much computation and communication compared to EDF the aforementioned improvement is halved, thus it is expected to obtain in overall a 60% better performance using NEDF over EDF. Finally, it should be mentioned that instead of selecting the cross-shape neighbors of a node with path length two, one may select the four nodes lying in the four corners (see Fig. 2). Alternating paths for communicating the load will produce diffusion schemes with different convergence rates.

Acknowledgements. This research was partially funded by the University of Athens Special Account of Research Grants no 10812.

References

1. Boillat, J.E.: Load balancing and poisson equation in a graph. Concurrency: Practice and Experience 2, 289–313 (1990)
2. Brandt, A.: Multi-level adaptive solutions to boundary-value problems. Math. Comput. 31, 333–390 (1977)
3. Cybenko, G.: Dynamic load balancing for distributed memory multi-processors. J. Parallel and Distr. Comp. 7, 279–301 (1989)
4. Diekmann, R., Muthukrishnan, S., Nayakkankuppam, M.V.: Engineering Diffusive Load Balancing Algorithms Using Experiments. In: Lüling, R., Bilardi, G., Ferreira, A., Rolim, J.D.P. (eds.) IRREGULAR 1997. LNCS, vol. 1253, pp. 111–122. Springer, Heidelberg (1997)
5. Elsässer, R., Monien, B., Preis, R.: Optimal Diffusion schemes and Load Balancing on Product Graphs. Parallel Proc. Letters 14, 61–73 (2004)

6. Hong, J., Tau, X., Cheu, M.: From local to global: an analysis of nearest neighbor balancing on hypercube. In: Proc. ACM Symp. on SIGMETRICS, pp. 73–82 (1988)
7. Karagiorgos, G., Missirlis, N.: Convergence of the diffusion method for weighted torus graphs using Fourier analysis. J. Theor. Comp. Sc. 401(1-3), 1–16 (2008)
8. Markomanolis, G.S., Missirlis, N.M.: Optimum Diffusion for Load Balancing in Mesh Networks. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 230–241. Springer, Heidelberg (2010)
9. Xu, C., Lau, F.M.: Load balancing in parallel computers: Theory and Practice
10. Young, D.M.: Iterative Solution of Large Linear Systems. Academic (2003)

On the Weak Convergence of the Recursive Orthogonal Series-Type Kernel Probabilistic Neural Networks in a Time-Varying Environment

Piotr Duda[1] and Yoichi Hayashi[2]

[1] Department of Computer Engineering,
Czestochowa University of Technology, Czestochowa, Poland
pduda@kik.pcz.pl
[2] Department of Computer Science, Meiji University,
Tama-ku, Kawasaki 214-8571, Japan
hayashiy@cs.meiji.ac.jp

Abstract. In a paper a recursive version of general regression neural networks, based on the orthogonal series-type kernels, is presented. Sufficient conditions for convergence in probability are given assuming time-varying noise. Experimental results are provided and discussed.

1 Introduction

In this paper we consider the following model

$$Y_i = \phi(X_i) + Z_i, \qquad i = 1, \ldots, n, \tag{1}$$

where $X_1, \ldots, X_n$ are independent random variables with a probability density $f(\cdot)$, Z_i are random variables such that

$$E(Z_i) = 0 \quad E(Z_i^2) = d_i \quad i = 1, \ldots, n \tag{2}$$

and the input random variables $(X_1, \ldots, X_n)$ have the same probability density function $f(\cdot)$. To estimate function $\phi(\cdot)$ we propose the following formula

$$\hat{\phi}_n(x) = \frac{\hat{R}_n(x)}{\hat{f}_n(x)}, \tag{3}$$

where

$$\hat{R}_n(x) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=0}^{M(i)} Y_i g_j(X_i) g_j(x), \tag{4}$$

and

$$\hat{f}_n(x) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=0}^{N(i)} g_j(X_i) g_j(x), \tag{5}$$

where $\{g_j\}$ is a complete orthonormal system (see e.g. [1]) and

$$N(n) \xrightarrow{n} \infty \qquad M(n) \xrightarrow{n} \infty. \tag{6}$$

One can see that estimators (5) and (4) be expressed as follows

$$\hat{f}_n(x) = \hat{f}_{n-1}(x) + \frac{1}{n}[\sum_{j=0}^{N(n)} g_j(X_n)g_j(x) - \hat{f}_{n-1}(x)], \tag{7}$$

$$\hat{R}_n(x) = \hat{R}_{n-1}(x) + \frac{1}{n}[\sum_{j=0}^{M(n)} Y_n g_j(X_n)g_j(x) - \hat{R}_{n-1}(x)], \tag{8}$$

where $M(n), N(n)$ are the same as in (6).

The idea of general regression and probabilistic neural networks was proposed by Specht [35] by making use of the Parzen-type kernels. In this paper we use an alternative approach based on the orthogonal series. In literatur both approaches have been applied to solve various stationary (see [3], [5], [6],[8], [10], [12] - [15], [21] - [23], [26] - [29]) and non-stationary problems ([7], [16] -[20], [24], [25]). Figure 1 shows probabilistic neural network block diagram.

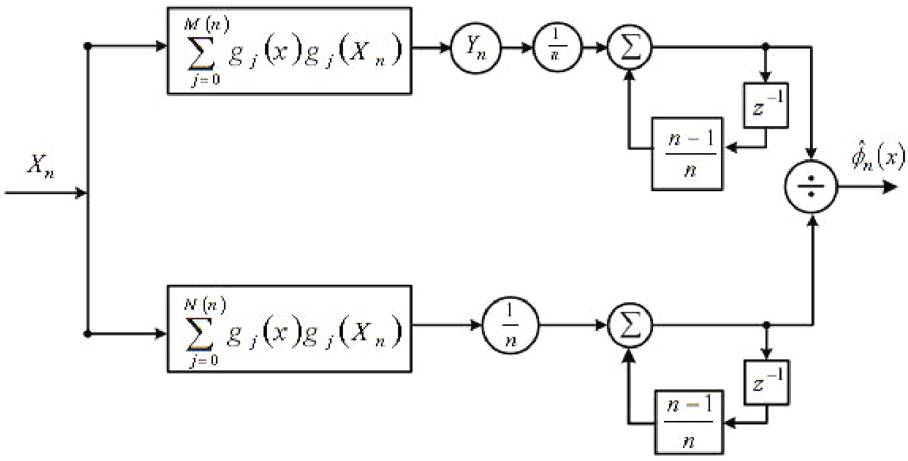


Fig. 1. General regression neural network

2 Main Result

Let us assume that

$$\max_x |g_j| \leq G_j, \tag{9}$$

It should be noted that $d = -\frac{1}{12}$ for the Hermite sytem, $d = -\frac{1}{4}$ for the Laguerre system, $d = 0$ for the Fourier system, $d = \frac{1}{2}$ for the Legendre and Haar systems (see [38]).

Theorem 1. *Let* $s_i = \int\limits_A \phi^2(u)f(u)du + d_i < \infty$. *If the following conditions hold*

$$\frac{1}{n^2}\sum_{i=1}^{n}(\sum_{j=0}^{M(i)} G_j^2)^2 s_i \xrightarrow{n} 0, \qquad M(n) \xrightarrow{n} \infty \tag{10}$$

$$\frac{1}{n^2}\sum_{i=1}^{n}(\sum_{j=0}^{N(i)} G_j^2)^2 \xrightarrow{n} 0, \qquad N(n) \xrightarrow{n} \infty \tag{11}$$

then

$$\hat{\phi}_n(x) \xrightarrow{n} \phi(x) \qquad \textit{in probability,} \tag{12}$$

at every point $x \in A$ *at which*

$$\sum_{j=0}^{N(n)} a_j g_j(x) \xrightarrow{n} f(x), \tag{13}$$

$$\sum_{j=0}^{M(n)} b_j g_j(x) \xrightarrow{n} R(x) \tag{14}$$

where

$$a_j = \int\limits_A f(x)g_j(x)dx = Eg_j(X_i), \tag{15}$$

$$b_j = \int\limits_A \phi(x)f(x)g_j(x)dx = E(Y_i g_j(X_i)). \tag{16}$$

Proof. The proof can be based on the arguments similar to those in [13].

3 Experimental Results

Let us assume that:

$$M(n) = [c_1 n^{q_M}], \quad N(n) = [c_2 n^{q_N}], \quad d_n = c_3 n^{\alpha}, \tag{17}$$

where q_M, q_N and α are positive numbers.

We consider the following regresion function

$$\phi(x) = \sin(3x)\sin(x+2). \tag{18}$$

Distribution of random variables X_i is uniform on the interval $[0, \pi]$, for $i = 1, \ldots, n$ and Z_i are realizations of random variables $N(0, d_i)$, $d_i = i^{\alpha}$, $\alpha > 0$. Parameters q_M and q_N in (17) are equal to $0,4$. All constans c_1, c_2, c_3 are equal to 1. The Hermite orthonormal system is chosen to perform calculations. Number of data set is taken from the interval $[500; 10000]$, and parameter α is tested in the interval $[\frac{1}{10}, \frac{12}{10}]$.

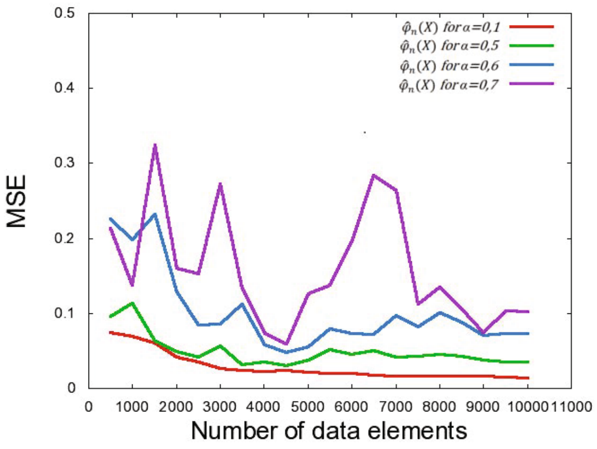


Fig. 2. The MSE as a function of n

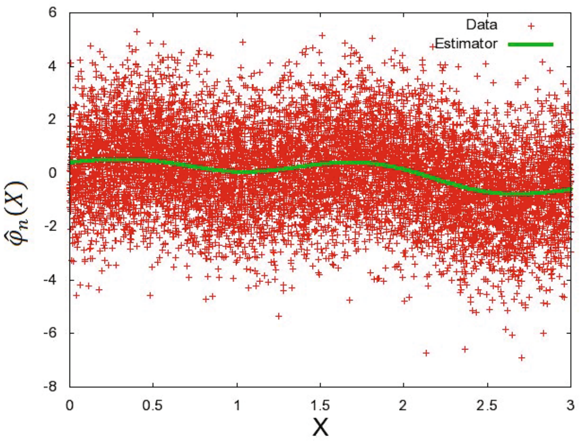


Fig. 3. Training set and obtained estimator

Figure 2 shows how the MSE (Mean Square Error) changes with number of data elements n for different values of parameter α. For parameter $\alpha \in [0,1;0,6]$ we can see that, when n goes to infinity, the MSE goes to 0. For $\alpha = 0,7$ this trend is not maintained. Experimental results show that for higher α the MSE is growing. For $\alpha = 1,2$ and $n = 10^4$, the MSE is equal to 6,34.

In Figure 3 input data and the results of estimation for $n = 10^4$ and $\alpha = 0,1$ are depicted. As we can see estimator found in the appropriate manner center of data and maintained its trend.

Figure 4 shows the course of the function given by (18), and estimators obtained for $n = 10^4$, with parameters α equal to $0,1$ and $0,8$.

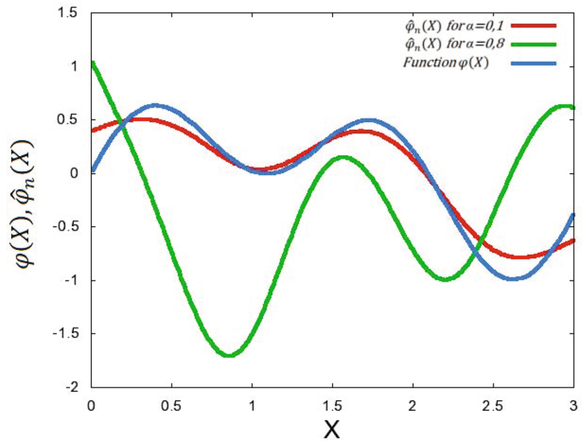


Fig. 4. Function $\phi(\cdot)$ and its estimators for different values of parameter α

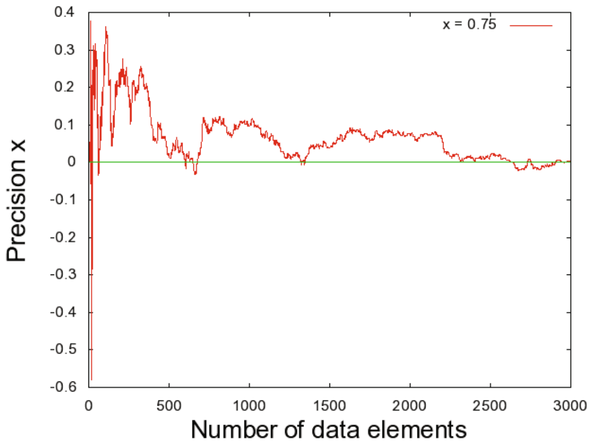


Fig. 5. The accuracy at the point x

In Figure 5 the differences between value of function (18) and estimator in step n in point $x = 0,75$ are depicted. The parameter α is set to 0,1 and $n = 1, \ldots, 3000$. One can see that differences tend to 0.

4 Conclusions

In this paper we studied a recurisive version of general regression neural networks, based on the orthogonal series-type kernel. We proved convergence in probability assuming time-varying noise. In future work it would be interesting to adopt unsupervised and unsupervised training algorithms for neural networks [2], [4], [11] and neuro-fuzzy systems [9], [30] - [32], [34], [36], [37] for handling time-varying noise in model (1).

Acknowledgments. The paper was prepared under project operated within the Foundation for Polish Science Team Programme co-financed by the EU European Regional Development Fund, Operational Program Innovative Economy 2007-2013, and also supported by the National Science Center NCN.

References

1. Alexits, G.: Convergence Problems of Orthogonal Series, Budapest, Academia and Kiado, pp. 261–264 (1961)
2. Bilski, J., Rutkowski, L.: A fast training algorithm for neural networks. IEEE Transactions on Circuits and Systems II 45, 749–753 (1998)
3. Cacoullos, P.: Estimation of a multivariate density. Annals of the Institute of Statistical Mathematics 18, 179–190 (1965)
4. Cierniak, R., Rutkowski, L.: On image compression by competitive neural networks and optimal linear predictors. Signal Processing: Image Communication - a Eurasip Journal 15(6), 559–565 (2000)
5. Gałkowski, T., Rutkowski, L.: Nonparametric recovery of multivariate functions with applications to system identification. Proceedings of the IEEE 73, 942–943 (1985)
6. Gałkowski, T., Rutkowski, L.: Nonparametric fitting of multivariable functions. IEEE Transactions on Automatic Control AC-31, 785–787 (1986)
7. Greblicki, W., Rutkowska, D., Rutkowski, L.: An orthogonal series estimate of time-varying regression. Annals of the Institute of Statistical Mathematics 35, Part A, 147–160 (1983)
8. Greblicki, W., Rutkowski, L.: Density-free Bayes risk consistency of nonparametric pattern recognition procedures. Proceedings of the IEEE 69(4), 482–483 (1981)
9. Nowicki, R.: Nonlinear modelling and classification based on the MICOG defuzzifications. Journal of Nonlinear Analysis, Series A: Theory, Methods and Applications 7(12), e1033–e1047 (2009)
10. Parzen, E.: On estimation of a probability density function and mode. Analysis of Mathematical Statistics 33(3), 1065–1076 (1962)
11. Patan, K., Patan, M.: Optimal training strategies for locally recurrent neural networks. Journal of Artificial Intelligence and Soft Computing Research 1(2), 103–114 (2011)
12. Rutkowski, L.: Sequential estimates of probability densities by orthogonal series and their application in pattern classification. IEEE Transactions on Systems, Man, and Cybernetics SMC-10(12), 918–920 (1980)
13. Rutkowski, L.: Sequential estimates of a regression function by orthogonal series with applications in discrimination, New York, Heidelberg, Berlin. Lectures Notes in Statistics, vol. 8, pp. 236–244 (1981)
14. Rutkowski, L.: On system identification by nonparametric function fitting. IEEE Transactions on Automatic Control AC-27, 225–227 (1982)

15. Rutkowski, L.: Orthogonal series estimates of a regression function with applications in system identification. In: Probability and Statistical Inference, pp. 343–347. D. Reidel Publishing Company, Dordrecht (1982)
16. Rutkowski, L.: On Bayes risk consistent pattern recognition procedures in a quasi-stationary environment. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4(1), 84–87 (1982)
17. Rutkowski, L.: On-line identification of time-varying systems by nonparametric techniques. IEEE Transactions on Automatic Control AC-27, 228–230 (1982)
18. Rutkowski, L.: On nonparametric identification with prediction of time-varying systems. IEEE Transactions on Automatic Control AC-29, 58–60 (1984)
19. Rutkowski, L.: Nonparametric identification of quasi-stationary systems. Systems and Control Letters 6, 33–35 (1985)
20. Rutkowski, L.: The real-time identification of time-varying systems by nonparametric algorithms based on the Parzen kernels. International Journal of Systems Science 16, 1123–1130 (1985)
21. Rutkowski, L.: A general approach for nonparametric fitting of functions and their derivatives with applications to linear circuits identification. IEEE Transactions Circuits Systems CAS-33, 812–818 (1986)
22. Rutkowski, L.: Sequential pattern recognition procedures derived from multiple Fourier series. Pattern Recognition Letters 8, 213–216 (1988)
23. Rutkowski, L.: Nonparametric procedures for identification and control of linear dynamic systems. In: Proceedings of 1988 American Control Conference, June 15-17, pp. 1325–1326 (1988)
24. Rutkowski, L.: An application of multiple Fourier series to identification of multivariable nonstationary systems. International Journal of Systems Science 20(10), 1993–2002 (1989)
25. Rutkowski, L.: Nonparametric learning algorithms in the time-varying environments. Signal Processing 18, 129–137 (1989)
26. Rutkowski, L., Rafajłowicz, E.: On global rate of convergence of some nonparametric identification procedures. IEEE Transaction on Automatic Control AC-34(10), 1089–1091 (1989)
27. Rutkowski, L.: Identification of MISO nonlinear regressions in the presence of a wide class of disturbances. IEEE Transactions on Information Theory IT-37, 214–216 (1991)
28. Rutkowski, L.: Multiple Fourier series procedures for extraction of nonlinear regressions from noisy data. IEEE Transactions on Signal Processing 41(10), 3062–3065 (1993)
29. Rutkowski, L., Gałkowski, T.: On pattern classification and system identification by probabilistic neural networks. Applied Mathematics and Computer Science 4(3), 413–422 (1994)
30. Rutkowski, L.: A New Method for System Modelling and Pattern Classification. Bulletin of the Polish Academy of Sciences 52(1), 11–24 (2004)
31. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, Melbourne, December 2-5, vol. 3, pp. 1428–1431 (2001)
32. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)

33. Sansone, G.: Orthogonal functions, vol. 9. Interscience Publishers In., New York (1959)
34. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)
35. Specht, D.F.: Probabilistic neural networks. Neural Networks 3, 109–118 (1990)
36. Starczewski, J., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
37. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)
38. Szegö, G.: Orthogonal Polynomial, vol. 23. American Mathematical Society Coll. Publ. (1959)

On the Cesaro Orthogonal Series-Type Kernel Probabilistic Neural Networks Handling Non-stationary Noise

Piotr Duda[1] and Jacek M. Zurada[2]

[1] Department of Computer Engineering,
Czestochowa University of Technology, Czestochowa, Poland
pduda@kik.pcz.pl

[2] Department of Electrical and Computer Engineering,
University of Louisville, Louisville, KY, USA
jacek.zurada@louisville.edu

Abstract. The Cesaro means of orthogonal series are applied to construct general regression neural networks. Sufficient conditions for convergence in probability are given assuming nonstationary noise. An experiment with syntetic data is described.

1 Introduction

In literature various soft computing techniques (see e.g. [2], [4], [7]) have been applied to solve problems characterized by stationary noise. In this paper we consider the following model

$$Y_i = \phi(X_i) + Z_i, \qquad i = 1, \ldots, n, \tag{1}$$

where $X_1, \ldots, X_n$ are independent random variables with a probability density $f(\cdot)$, Z_i are random variables, such that

$$E(Z_i) = 0 \quad E(Z_i^2) = d_i \quad i = 1, \ldots, n \tag{2}$$

and the input random variables $(X_1, \ldots, X_n)$ have the same probability density function $f(\cdot)$. Let $g_j(\cdot)$, $j = 0, 1, 2, \ldots$ be a complete orthonormal system in $L_2(A)$, $A \subset R$. Let us define the so-called Cesaro kernels

$$K_n^1(x, u) = \sum_{j=1}^{N(n)} \left(1 - \frac{j}{N(n)+1}\right) g_j(x) g_j(u) \tag{3}$$

$$K_n^2(x, u) = \sum_{j=1}^{M(n)} \left(1 - \frac{j}{M(n)+1}\right) g_j(x) g_j(u) \tag{4}$$

where $N(n) \xrightarrow{n} \infty$ and $M(n) \xrightarrow{n} \infty$. Let

$$R(x) = f(x)\phi(x). \tag{5}$$

Then as estimator of $\phi(x)$ we propose:

$$\hat{\phi}_n(x) = \frac{\hat{R}_n(x)}{\hat{f}_n(x)}, \tag{6}$$

where

$$\hat{R}_n(x) = \frac{1}{n}\sum_{i=1}^{n} Y_i K_n^2(x, X_i), \tag{7}$$

and

$$\hat{f}_n(x) = \frac{1}{n}\sum_{i=1}^{n} K_n^1(x, X_i). \tag{8}$$

It should be noted that kernels (3) and (4) are alternatives to Parzen-type kernels proposed by Spocht to design general regression neural networks. In literature both type of kernels have been applied to solve stationary ([5], [9], [11] - [14], [20] - [22], [25] - [28]) and nonstationary ([6], [15] - [19], [23], [24]) problem. The block diagram of regression neural network is depiced in Fig. 1.

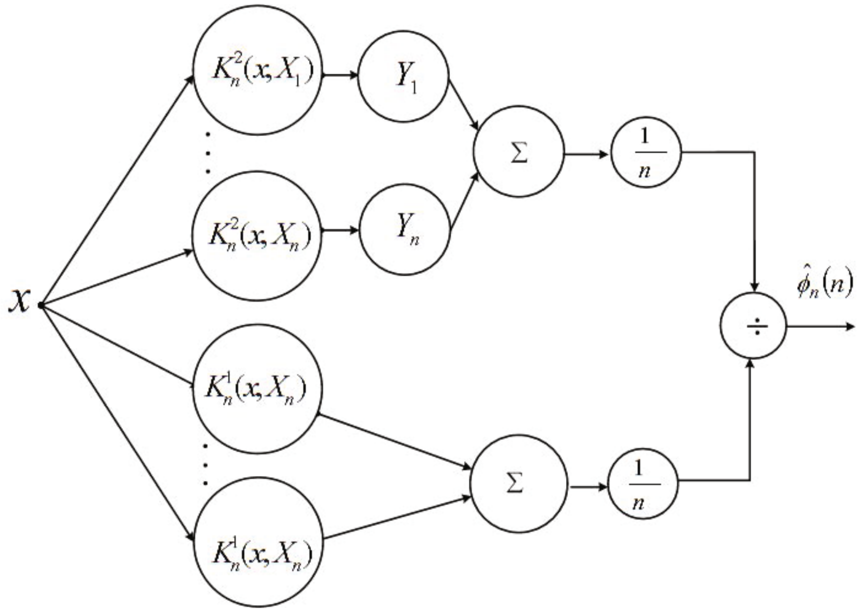


Fig. 1. Regression neural network

2 Main Result

Let us assume that

$$\max_x |g_j| \le G_j, \tag{9}$$

It should be noted that $d = -\frac{1}{12}$ for the Hermite sytem, $d = -\frac{1}{4}$ for the Laguerre system, $d = 0$ for the Fourier system, $d = \frac{1}{2}$ for the Legendre and Haar systems (see [35]). Let us denote

$$s_i = d_i + \int\limits_A \phi^2(u) f(u) du \tag{10}$$

Theorem 1. *If $s_i < \infty$, for all $i \geq 1$, and the following conditions hold*

$$\frac{1}{n^2}(\sum_{j=0}^{M(n)} G_j^2)^2 \sum_{i=1}^{n} s_i \xrightarrow{n} 0, \quad M(n) \xrightarrow{n} \infty \tag{11}$$

$$\frac{1}{n}(\sum_{j=0}^{N(n)} G_j^2)^2 \xrightarrow{n} 0, \quad N(n) \xrightarrow{n} \infty \tag{12}$$

then

$$\hat{\phi}_n(x) \xrightarrow{n} \phi(x) \qquad \textit{in probability,} \tag{13}$$

at every point $x \in A$ at which

$$\sum_{j=0}^{N(n)} (1 - \frac{j}{N(n)+1}) a_j g_j(x) \xrightarrow{n} f(x), \tag{14}$$

$$\sum_{j=0}^{M(n)} (1 - \frac{j}{M(n)+1}) b_j g_j(x) \xrightarrow{n} R(x) \tag{15}$$

where

$$a_j = \int\limits_A f(x) g_j(x) dx = E g_j(X_i), \tag{16}$$

$$b_j = \int\limits_A \phi(x) f(x) g_j(x) dx = E(Y_i g_j(X_i)). \tag{17}$$

Proof. The proof can be based on the arguments similar to those in [12].

3 Experimental Results

Let us assume that

$$M(n) = [c_1 n^{q_M}], \quad N(n) = [c_2 n^{q_N}], \quad d_n = c_3 n^{\alpha}, \quad G_j = c_4 j^d, \tag{18}$$

where q_M, q_N and α are positive numbers.

We consider the following regresion function

$$\phi(x) = 4\sin(x)\cos(x)(x^2 - 1). \tag{19}$$

Distribution of random variables X_i is uniform on the interwal $[-\pi, \pi]$, for $i = 1, \ldots, n$ and Z_i are realizations of random variables $N(0, i^\alpha)$, where $\alpha > 0$. Parameters q_M and q_N are both equal to $0,4$. The constans c_1 and c_2 are equal to 4 and c_3 is equal to 1. The Fourier orthonormal system is chosen to perform calculations. Number of data set is taken from the interval $[500; 10000]$, and parameter α is tested in the interval $[\frac{1}{10}, \frac{12}{10}]$.

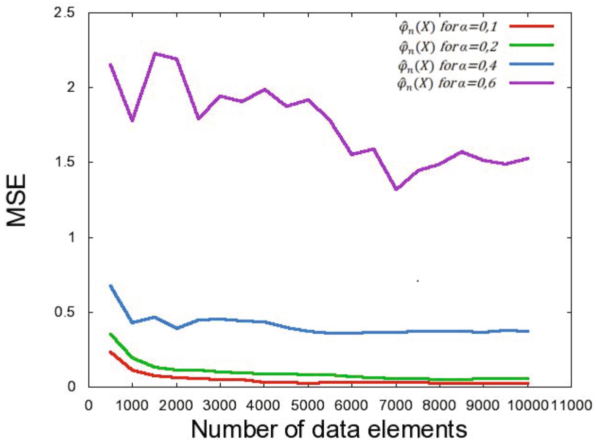


Fig. 2. MSE as a function of n

Figure 2 shows how the MSE (Mean Square Error) changes with number of data elements n for different values of parameter α. For parameter $\alpha \in [0,1; 0,2]$ we can see that, when n goes to infinity, the MSE goes to 0. For $\alpha \geq 0,6$ value of the MSE is much bigger than for lowers values of parameter α. For $\alpha = 1,2$ and $n = 10^4$, the MSE is equal to 281,66.

In Figure 3 input data and the results of estimation for $n = 10^4$ and $\alpha = 0,1$ are depicted. As we can see estimator found in the appropriate manner center of data and maintained its trend.

Figure 3 shows the course of the function given by (19) and estimators obtained for $n = 10^4$ data, with parameters α equal to $0,1$ and $0,6$. Figure 4 shows this course on the interval $[-1, 1]$.

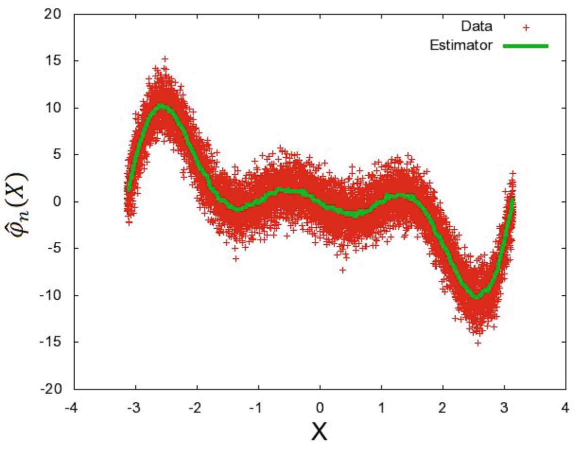


Fig. 3. Training set and obtained estimator

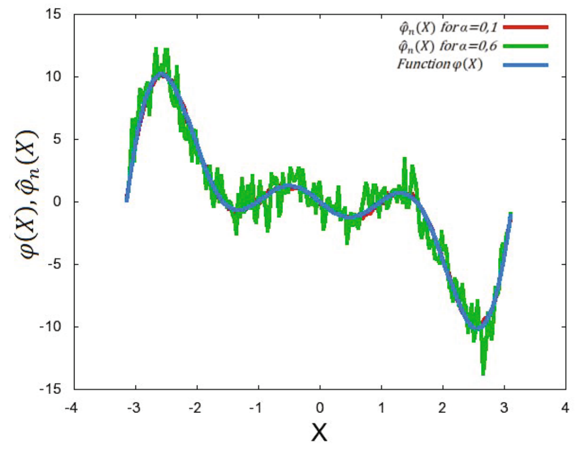


Fig. 4. Function $\phi(\cdot)$ and its estimators for different values of parameter α

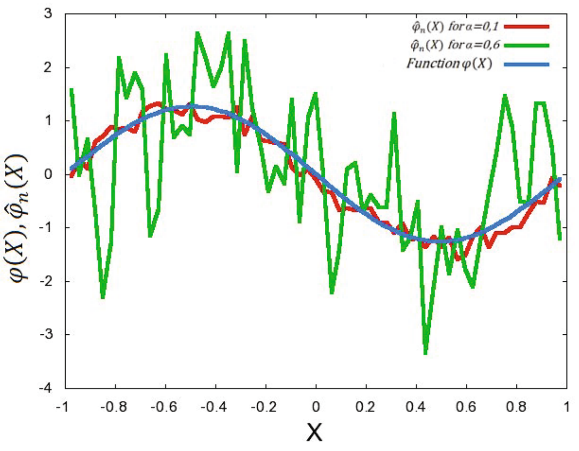


Fig. 5. Function $\phi(\cdot)$ and its estimators for different values of parameter α

4 Conclusions

In this paper the Cesaro means of orthogonal series were applied to construct general regresion neural networks. We established conditions for covergence in probability assuming nonstationary noise. Our on-going project is devoted to adapting neurofuzzy structures [8], [29] - [31], [32], [33], [34] and supervised and unsupervised neural netwoerks [1], [3], [10] for learning in nonstationary environment.

Acknowledgments. The paper was prepared under project operated within the Foundation for Polish Science Team Programme co-financed by the EU European Regional Development Fund, Operational Program Innovative Economy 2007-2013, and also supported by the National Science Center NCN.

References

1. Bilski, J., Rutkowski, L.: A fast training algorithm for neural networks. IEEE Transactions on Circuits and Systems II 45, 749–753 (1998)
2. Cacoullos, P.: Estimation of a multivariate density. Annals of the Institute of Statistical Mathematics 18, 179–190 (1965)
3. Cierniak, R., Rutkowski, L.: On image compression by competitive neural networks and optimal linear predictors. Signal Processing: Image Communication - a Eurasip Journal 15(6), 559–565 (2000)
4. Gałkowski, T., Rutkowski, L.: Nonparametric recovery of multivariate functions with applications to system identification. Proceedings of the IEEE 73, 942–943 (1985)
5. Gałkowski, T., Rutkowski, L.: Nonparametric fitting of multivariable functions. IEEE Transactions on Automatic Control AC-31, 785–787 (1986)

6. Greblicki, W., Rutkowska, D., Rutkowski, L.: An orthogonal series estimate of time-varying regression. Annals of the Institute of Statistical Mathematics 35, Part A, 147–160 (1983)
7. Greblicki, W., Rutkowski, L.: Density-free Bayes risk consistency of nonparametric pattern recognition procedures. Proceedings of the IEEE 69(4), 482–483 (1981)
8. Nowicki, R., Pokropińska, A.: Information Criterions Applied to Neuro-Fuzzy Architectures Design. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 332–337. Springer, Heidelberg (2004)
9. Parzen, E.: On estimation of a probability density function and mode. Analysis of Mathematical Statistics 33(3), 1065–1076 (1962)
10. Patan, K., Patan, M.: Optimal training strategies for locally recurrent neural networks. Journal of Artificial Intelligence and Soft Computing Research 1(2), 103–114 (2011)
11. Rutkowski, L.: Sequential estimates of probability densities by orthogonal series and their application in pattern classification. IEEE Transactions on Systems, Man, and Cybernetics SMC-10(12), 918–920 (1980)
12. Rutkowski, L.: Sequential estimates of a regression function by orthogonal series with applications in discrimination, New York, Heidelberg, Berlin. Lectures Notes in Statistics, vol. 8, pp. 236–244 (1981)
13. Rutkowski, L.: On system identification by nonparametric function fitting. IEEE Transactions on Automatic Control AC-27, 225–227 (1982)
14. Rutkowski, L.: Orthogonal series estimates of a regression function with applications in system identification. In: Probability and Statistical Inference, pp. 343–347. D. Reidel Publishing Company, Dordrecht (1982)
15. Rutkowski, L.: On Bayes risk consistent pattern recognition procedures in a quasi-stationary environment. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4(1), 84–87 (1982)
16. Rutkowski, L.: On-line identification of time-varying systems by nonparametric techniques. IEEE Transactions on Automatic Control AC-27, 228–230 (1982)
17. Rutkowski, L.: On nonparametric identification with prediction of time-varying systems. IEEE Transactions on Automatic Control AC-29, 58–60 (1984)
18. Rutkowski, L.: Nonparametric identification of quasi-stationary systems. Systems and Control Letters 6, 33–35 (1985)
19. Rutkowski, L.: The real-time identification of time-varying systems by nonparametric algorithms based on the Parzen kernels. International Journal of Systems Science 16, 1123–1130 (1985)
20. Rutkowski, L.: A general approach for nonparametric fitting of functions and their derivatives with applications to linear circuits identification. IEEE Transactions Circuits Systems CAS-33, 812–818 (1986)
21. Rutkowski, L.: Sequential pattern recognition procedures derived from multiple Fourier series. Pattern Recognition Letters 8, 213–216 (1988)
22. Rutkowski, L.: Nonparametric procedures for identification and control of linear dynamic systems. In: Proceedings of 1988 American Control Conference, June 15-17, pp. 1325–1326 (1988)
23. Rutkowski, L.: An application of multiple Fourier series to identification of multivariable nonstationary systems. International Journal of Systems Science 20(10), 1993–2002 (1989)
24. Rutkowski, L.: Nonparametric learning algorithms in the time-varying environments. Signal Processing 18, 129–137 (1989)

25. Rutkowski, L., Rafajłowicz, E.: On global rate of convergence of some nonparametric identification procedures. IEEE Transaction on Automatic Control AC-34(10), 1089–1091 (1989)
26. Rutkowski, L.: Identification of MISO nonlinear regressions in the presence of a wide class of disturbances. IEEE Transactions on Information Theory IT-37, 214–216 (1991)
27. Rutkowski, L.: Multiple Fourier series procedures for extraction of nonlinear regressions from noisy data. IEEE Transactions on Signal Processing 41(10), 3062–3065 (1993)
28. Rutkowski, L., Gałkowski, T.: On pattern classification and system identification by probabilistic neural networks. Applied Mathematics and Computer Science 4(3), 413–422 (1994)
29. Rutkowski, L.: A New Method for System Modelling and Pattern Classification. Bulletin of the Polish Academy of Sciences 52(1), 11–24 (2004)
30. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, Melbourne, December 2-5, vol. 3, pp. 1428–1431 (2001)
31. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)
32. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)
33. Starczewski, J., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
34. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)
35. Szegö, G.: Orthogonal Polynomials, vol. 23. American Mathematical Society Coll. Publ. (1959)

On the Weak Convergence of the Orthogonal Series-Type Kernel Regresion Neural Networks in a Non-stationary Environment

Meng Joo Er[1] and Piotr Duda[2]

[1] Nanyang Technological University,
School of Electrical and Electronic Engineering, Singapore
[2] Department of Computer Engineering,
Czestochowa University of Technology, Czestochowa, Poland
`emjer@ntu.edu.sg, pduda@kik.pcz.pl`

Abstract. In the paper general regression neural networks, based on the orthogonal series-type kernel, is studied. Convergence in probability is proved assuming non-stationary noise. The performance is investigated using syntetic data.

1 Introduction

Let $X_1, \ldots, X_n$ be a sequence of independent random variables with a common desity function f. Consider the following model

$$Y_i = \phi(X_i) + Z_i, \qquad i = 1, \ldots, n, \tag{1}$$

where Z_i are random variables such that

$$E(Z_i) = 0, \qquad EZ_i^2 = d_i, \qquad i = 1, \ldots, n, \tag{2}$$

and $\phi(\cdot)$ is an unknown function.

$$f(x) \sim \sum_{j=0}^{\infty} a_j g_j(x), \tag{3}$$

where

$$a_j = \int_A f(x) g_j(x) dx = E g_j(X_j). \tag{4}$$

and $\{g_j(\cdot)\}$, $j = 0, 1, 2, \ldots$ is a complete orthonormal set defined on $A \subset R^p$. Then the estimator of density $f(\cdot)$ takes the form

$$\hat{f}_n(x) = \sum_{j=0}^{N(n)} \hat{a}_j g_j(x), \tag{5}$$

where $N(n) \xrightarrow{n} \infty$ and

$$\hat{a}_j = \frac{1}{n} \sum_{k=0}^{n} g_j(X_k) \tag{6}$$

Let us define

$$R(x) = f(x)\phi(x). \tag{7}$$

We assume that function $R(\cdot)$ has the representation

$$R(x) \sim \sum_{j=0}^{\infty} b_j g_j(x), \tag{8}$$

where

$$b_j = \int\limits_A \phi(x) f(x) g_j(x) dx = E(Y_k g_j(X_k)) \tag{9}$$

We estimate function $R(\cdot)$ using

$$\hat{R}_n(x) = \sum_{j=0}^{M(n)} \hat{b}_j g_j(x), \tag{10}$$

where $M(n) \xrightarrow{n} \infty$ and

$$\hat{b}_j = \frac{1}{n} \sum_{k=0}^{n} Y_k g_j(X_k). \tag{11}$$

Then the estimator of the regression function is of the following form

$$\hat{\phi}_n(x) = \frac{\hat{R}_n(x)}{\hat{f}_n(x)} = \frac{\sum\limits_{i=1}^{n} \sum\limits_{j=0}^{M(n)} Y_i g_j(X_i) g_j(x)}{\sum\limits_{i=1}^{n} \sum\limits_{j=0}^{N(n)} g_j(X_i) g_j(x)} \tag{12}$$

It should be noted estimate (12) can be presented in the form of general regression neural networks [35] with appropriately selected kernels $K_n^1(x,u) = \sum\limits_{j=0}^{N(n)} g_j(x) g_j(u)$ and $K_n^2(x,u) = \sum\limits_{j=0}^{M(n)} g_j(x) g_j(u)$. In literature nonparametric estiames have been widely studied both in stationary (see e.g. [3], [5], [6],[8], [10], [12] - [15], [21] - [23], [26] - [29]) and time-varying case (see [7], [16] -[20], [24], [25]). In this paper it will be shown that procedure (12) is applicable even if variance of noise diverges to infinity. Block diagram of general regression neural network is shown in Fig. 1.

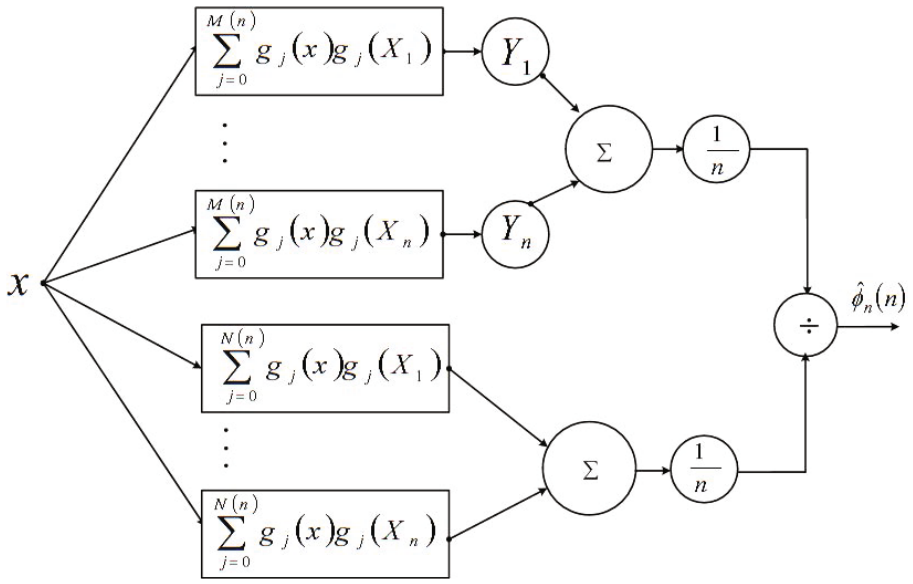

x
$\sum_{j=0}^{M(n)} g_j(x)g_j(X_1)$
$\sum_{j=0}^{M(n)} g_j(x)g_j(X_n)$
$\sum_{j=0}^{N(n)} g_j(x)g_j(X_1)$
$\sum_{j=0}^{N(n)} g_j(x)g_j(X_n)$
Y_1
Y_n
Σ
$\frac{1}{n}$
Σ
$\frac{1}{n}$
$\div$
$\hat{\phi}_n(n)$

Fig. 1. General regression neural network

2 Main Result

Let us assume that

$$\max_x |g_j| < G_j. \tag{13}$$

Let us denote

$$s_i = d_i + \int_A \phi^2(u)f(u)du \tag{14}$$

Theorem 1. *If $s_i < \infty$, for all $i \geq 1$, and the following conditions hold*

$$\frac{1}{n^2}(\sum_{j=0}^{M(n)} G_j^2)^2 \sum_{i=1}^{n} s_i \xrightarrow{n} 0, \quad M(n) \xrightarrow{n} \infty \tag{15}$$

$$\frac{1}{n}(\sum_{j=0}^{N(n)} G_j^2)^2 \xrightarrow{n} 0, \quad N(n) \xrightarrow{n} \infty \tag{16}$$

then

$$\hat{\Phi}_n(x) \xrightarrow{n} \Phi(x) \qquad \textit{in probability,} \tag{17}$$

at every point $x \in A$ at which series (3) and (8) converge to $f(x)$ and $R(x)$, respectively.

Proof. It is sufficient to show that:

$$E[\hat{R}_n(x) - R(x)]^2 \xrightarrow{n} 0 \tag{18}$$
$$E[\hat{f}_n(x) - f(x)]^2 \xrightarrow{n} 0, \tag{19}$$

in probability, at every point $x \in A$, at which series (3) and (8) are convergent. Observe that

$$E(\hat{R}_n(x) - R(x))^2 \leq \sum_{j=0}^{M(n)} G_j^2 \sum_{j=0}^{M(n)} E(\hat{b}_j - b_j)^2 + (\sum_{j=0}^{M(n)} b_j g_j(x) - R(x))^2. \tag{20}$$

One can see that the mean square error $E(\hat{b_j} - b_j)^2$ is bounded by

$$E(\hat{b}_j - b_j)^2 \leq \frac{G_j^2}{n^2} \sum_{i=1}^{n} (\int_A \phi^2(u) f(u) du + d_i). \tag{21}$$

Then

$$E[\hat{R}_n(x) - R(x)]^2 \leq \frac{1}{n^2} (\sum_{j=0}^{M(n)} G_j^2)^2 \sum_{i=1}^{n} (\int_A \phi^2(u) f(u) du + d_i) + (\sum_{j=0}^{M(n)} b_j g_j(x) - R(x))^2. \tag{22}$$

In view of assumption (15), convergence (18) is established. Convergence (19) can be proved in a similar way. This concludes the proof.

Remark 1. Conditions for convergence of series (3) and (8) can be found in [1],[33],[38].

Example. Let assume that

$$M(n) = [c_1 n^{q_M}] \quad N(n) = [c_2 n^{q_N}] \quad d_n = c_3 n^{\alpha} \quad G_j = c_4 j^d, \tag{23}$$

where q_M, q_N and α are positive numbers. It is easily seen that if

$$4dq_M + 2q_M + \alpha < 1, \qquad 4dq_N + 2q_N < 1, \tag{24}$$

then Theorem 1 holds. It should be noted that $d = -\frac{1}{12}$ for the Hermite sytem, $d = -\frac{1}{4}$ for the Laguerre system, $d = 0$ for the Fourier system, $d = \frac{1}{2}$ for the Legendre and Haar systems (see [33]).

3 Experimental Results

For computer simulations we use synthetic data. Distribution of random variables X_i is uniform on the interval $[0; \pi]$, for $i = 1, \ldots, n$. Consider the following model

$$\phi(x) = \exp(\sin(2x)), \tag{25}$$

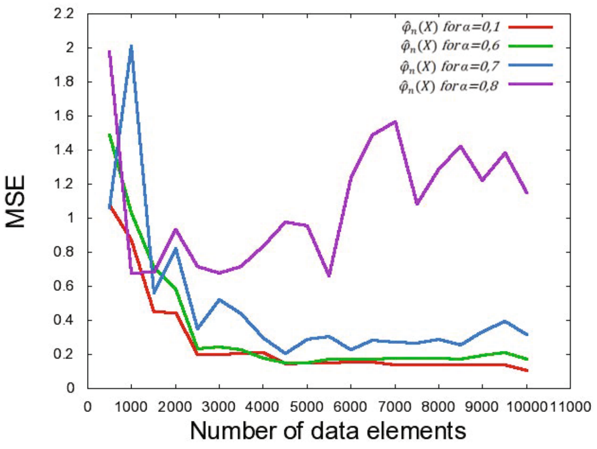


Fig. 2. The MSE as a function of n

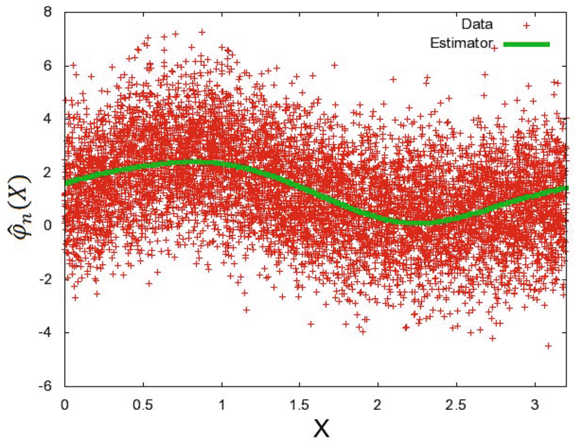


Fig. 3. Training set and obtained estimator

with Z_i which are realizations of random variables $N(0, d_i)$, $d_i = i^\alpha$, $\alpha > 0$. All constants (c_1, c_2, c_3) in (23) are equal to 1. Parameters q_M and q_N are both equal to $0, 25$. The Hermite orthonormal system is chosen to perform calculations. Number of data set is taken from the interval $[500; 10000]$ and parameter α is tested in the interval $[\frac{1}{10}, \frac{12}{10}]$.

Figure 2 shows how the MSE (Mean Square Error) changes with the number of data elements n for different values of parameter α. For parameter $\alpha \in [0, 1; 0, 7]$ we can see that, when n goes to infinity, the MSE goes to 0. For $\alpha = 0, 8$ this trend is not maintained. Moreover, for $\alpha = 0, 8$, value of the MSE is much bigger than for lower values of parameter α. Experimental results show that for higher values of α the MSE is growing. For $\alpha = 1, 2$ and $n = 10^4$, the MSE is equal to 7,37.

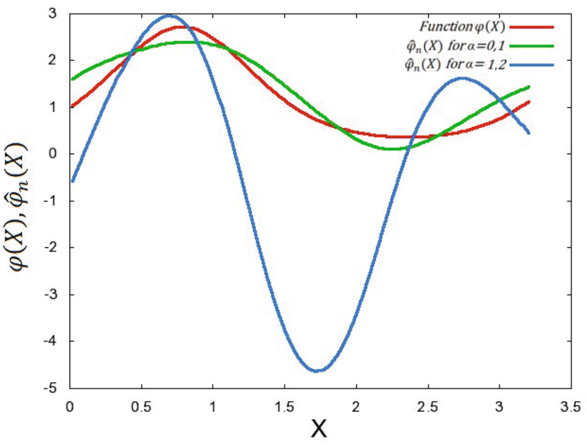


Fig. 4. Function $\phi(\cdot)$ and its estimators for different values of parameter α

In Figure 3 input data and the result of estimation for $n = 10^4$ and $\alpha = 0,1$ is indicated. As we can see the estimator found in the appropriate manner center of data and maintained its trend.

Figure 3 shows the course of the function given by (25) and estimators obtained for $n = 10^4$, with parameters α equal to $0,1$ and $1,2$.

4 Conclusions

In this paper we studied general regression neural networks, based on the orthogonal series-type kernel. We proved convergence in probability assuming nonstationary noise. In future works alternative methods, based on neural networks [2], [4], [11] and neuro-fuzzy structures [9], [30] - [32], [34], [36], [37], will be adopted to handle nonstationary noise.

Acknowledgments. The paper was prepared under project operated within the Foundation for Polish Science Team Programme co-financed by the EU European Regional Development Fund, Operational Program Innovative Economy 2007-2013, and also supported by the National Science Center NCN.

References

1. Alexits, G.: Convergence Problems of Orthogonal Series, Budapest, Academia and Kiado, pp. 261–264 (1961)
2. Bilski, J., Rutkowski, L.: A fast training algorithm for neural networks. IEEE Transactions on Circuits and Systems II 45, 749–753 (1998)

3. Cacoullos, P.: Estimation of a multivariate density. Annals of the Institute of Statistical Mathematics 18, 179–190 (1965)
4. Cierniak, R., Rutkowski, L.: On image compression by competitive neural networks and optimal linear predictors. Signal Processing: Image Communication - a Eurasip Journal 15(6), 559–565 (2000)
5. Gałkowski, T., Rutkowski, L.: Nonparametric recovery of multivariate functions with applications to system identification. Proceedings of the IEEE 73, 942–943 (1985)
6. Gałkowski, T., Rutkowski, L.: Nonparametric fitting of multivariable functions. IEEE Transactions on Automatic Control AC-31, 785–787 (1986)
7. Greblicki, W., Rutkowska, D., Rutkowski, L.: An orthogonal series estimate of time-varying regression. Annals of the Institute of Statistical Mathematics 35, Part A, 147–160 (1983)
8. Greblicki, W., Rutkowski, L.: Density-free Bayes risk consistency of nonparametric pattern recognition procedures. Proceedings of the IEEE 69(4), 482–483 (1981)
9. Nowicki, R.: Rough Sets in the Neuro-Fuzzy Architectures Based on Monotonic Fuzzy Implications. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 510–517. Springer, Heidelberg (2004)
10. Parzen, E.: On estimation of a probability density function and mode. Analysis of Mathematical Statistics 33(3), 1065–1076 (1962)
11. Patan, K., Patan, M.: Optimal training strategies for locally recurrent neural networks. Journal of Artificial Intelligence and Soft Computing Research 1(2), 103–114 (2011)
12. Rutkowski, L.: Sequential estimates of probability densities by orthogonal series and their application in pattern classification. IEEE Transactions on Systems, Man, and Cybernetics SMC-10(12), 918–920 (1980)
13. Rutkowski, L.: Sequential estimates of a regression function by orthogonal series with applications in discrimination, New York, Heidelberg, Berlin. Lectures Notes in Statistics, vol. 8, pp. 236–244 (1981)
14. Rutkowski, L.: On system identification by nonparametric function fitting. IEEE Transactions on Automatic Control AC-27, 225–227 (1982)
15. Rutkowski, L.: Orthogonal series estimates of a regression function with applications in system identification. In: Probability and Statistical Inference, pp. 343–347. D. Reidel Publishing Company, Dordrecht (1982)
16. Rutkowski, L.: On Bayes risk consistent pattern recognition procedures in a quasi-stationary environment. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4(1), 84–87 (1982)
17. Rutkowski, L.: On-line identification of time-varying systems by nonparametric techniques. IEEE Transactions on Automatic Control AC-27, 228–230 (1982)
18. Rutkowski, L.: On nonparametric identification with prediction of time-varying systems. IEEE Transactions on Automatic Control AC-29, 58–60 (1984)
19. Rutkowski, L.: Nonparametric identification of quasi-stationary systems. Systems and Control Letters 6, 33–35 (1985)
20. Rutkowski, L.: The real-time identification of time-varying systems by nonparametric algorithms based on the Parzen kernels. International Journal of Systems Science 16, 1123–1130 (1985)
21. Rutkowski, L.: A general approach for nonparametric fitting of functions and their derivatives with applications to linear circuits identification. IEEE Transactions Circuits Systems CAS-33, 812–818 (1986)

22. Rutkowski, L.: Sequential pattern recognition procedures derived from multiple Fourier series. Pattern Recognition Letters 8, 213–216 (1988)
23. Rutkowski, L.: Nonparametric procedures for identification and control of linear dynamic systems. In: Proceedings of 1988 American Control Conference, June 15-17, pp. 1325–1326 (1988)
24. Rutkowski, L.: An application of multiple Fourier series to identification of multivariable nonstationary systems. International Journal of Systems Science 20(10), 1993–2002 (1989)
25. Rutkowski, L.: Nonparametric learning algorithms in the time-varying environments. Signal Processing 18, 129–137 (1989)
26. Rutkowski, L., Rafajłowicz, E.: On global rate of convergence of some nonparametric identification procedures. IEEE Transaction on Automatic Control AC-34(10), 1089–1091 (1989)
27. Rutkowski, L.: Identification of MISO nonlinear regressions in the presence of a wide class of disturbances. IEEE Transactions on Information Theory IT-37, 214–216 (1991)
28. Rutkowski, L.: Multiple Fourier series procedures for extraction of nonlinear regressions from noisy data. IEEE Transactions on Signal Processing 41(10), 3062–3065 (1993)
29. Rutkowski, L., Gałkowski, T.: On pattern classification and system identification by probabilistic neural networks. Applied Mathematics and Computer Science 4(3), 413–422 (1994)
30. Rutkowski, L.: A New Method for System Modelling and Pattern Classification. Bulletin of the Polish Academy of Sciences 52(1), 11–24 (2004)
31. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, Melbourne, December 2-5, vol. 3, pp. 1428–1431 (2001)
32. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)
33. Sansone, G.: Orthogonal functions, vol. 9. Interscience Publishers In., New York (1959)
34. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)
35. Specht, D.F.: Probabilistic neural networks. Neural Networks 3, 109–118 (1990)
36. Starczewski, J., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
37. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)
38. Zygmund, A.: Trigonometric Series. Cambridge University Press, Cambridge (1959)

A Graph-Based Generation of Virtual Grids

Ewa Grabska, Wojciech Palacz, Barbara Strug, and Grażyna Ślusarczyk

Faculty of Physics, Astronomy and Applied Computer Science,
Jagiellonian University, Reymonta 4, Krakow, Poland
{ewa.grabska,wojciech.palacz,barbara.strug,gslusarc}@uj.edu.pl

Abstract. This paper presents a unified formal model to be used in the grid representation based on hierarchical graph structures. It aims at both helping in a better understanding of grid generation and in grid simulation problems. A new graph structure called layered graphs is proposed. This approach enables one to use attributed graph grammars as a tool to generate both a grid structure and its parameters at the same time. To illustrate the method an example of a grid generated by means of graph grammar rules is presented.

1 Introduction

Grid computing is based on the distributed computing concept [2]. The grid is usually considered to be composed of several layers: Fabric, Connectivity, Resources, Collective and Application [5]. As the grid can be used for many different applications we propose each of the grid layers to be represented as a subgraph of a hierarchical graph, called layered graph. The graph layers are connected by inter-layer edges which represent how the communication between different layers of a grid is carried out. Such a representation allows us to investigate properties of a grid in a more general way and to better understand its working and behavior.

As the grid is a dynamic structure, new resources can be added and removed from the overall structure at any time. These changes can be modelled with the use of attributed graph grammars that are capable of generating both a grid structure and its parameters at the same time. As in the grid changes may occur at different layers at the same time there is a need to use grammar rules operating on several layers. To make sure the functionality of a grid is preserved some rules have also to be able to invoke other rules responsible for maintaining the stability of affected elements. Moreover, using graph grammars enables us to quickly generate many alternative structures obeying the specified constraints.

Grid generation, discussed by Lu and Dinda [1], separates topology generation and annotations, while the method proposed here, which uses graph grammars, makes it possible to generate the structure with the annotations (attributes) [4]. Our earlier work on layered graphs is presented in [8].

2 Hierarchical Graphs

Hierarchical Graphs (HGs) consist of edges and nodes which can contain internal nodes. These nodes can be connected to other nodes with the exception of their ancestors. This property makes hierarchical graphs [7] well suited to represent the structure of the grid.

Let us assume that if $f : A \to A$ is a function, then $f^0(x) = x$, $f^1(x) = f(x)$, $f^2(x) = f(f(x))$, $f^+(x) = \bigcup_{i=1}^{\infty} f^i(x)$, $f^*(x) = \bigcup_{i=0}^{\infty} f^i(x)$.

Formally a hierarchical graph is defined in the following way:

Definition 1 (hierarchical graph). *$G = (V_G, E_G, s_G, t_G, ch_G)$ is a hierarchical graph if:*

1. *$V_G \cap E_G = \emptyset$, where V_G and E_G are finite sets of nodes and edges,*
2. *$s_G : E_G \to V_G$ and $t_G : E_G \to V_G$ are functions specifying edge source and target,*
3. *$ch_G : V_G \to \mathcal{P}(V_G)$ is a function specifying children nodes,*
4. *$\forall e \in E_G : s_G(e) \neq t_G(e)$, i.e. there are no loops,*
5. *$\forall v \in V_G : v \notin ch_G^+(v)$, i.e. a node cannot be its own descendant,*
6. *$\forall v, w \in V_G : v \neq w \Rightarrow ch_G(v) \cap ch_G(w) = \emptyset$, i.e. a node cannot have two parents, and*
7. *$\forall e \in E_G : s_G(e) \notin ch_G^*(t_G(e)) \wedge t_G(e) \notin ch_G^*(s_G(e))$, i.e. there are no edges between hierarchically related nodes.*

The universe of all graph nodes is denoted by $\mathcal{V}$, the universe of edges by $\mathcal{E}$. Let $\perp$ be a fixed value, not equal to any graph node: $\perp \notin \mathcal{V}$.

Definition 2 (parent and ancestors of a node). *Let G be a hierarchical graph.*

$par_G : V_G \to V_G \cup \{\perp\}$ defined as $par_G(v) = \begin{cases} w \text{ if } \exists\, w \in V_G : v \in ch_G(w) \\ \perp \text{ otherwise} \end{cases}$

is called the parent function of G.

Let $anc_G : V_G \cup \{\perp\} \to V_G \cup \{\perp\}$ be defined as $anc_G(x) = \begin{cases} par_G(x) & \text{if } x \in V_G \\ \perp & \text{if } x = \perp \end{cases}$

For a given node v, $anc_G^1(v)$ is its direct ancestor (parent), $anc_G^2(v)$ is its level 2 ancestor (grandparent), and so on.

A set of all ancestors of v can be specified as $anc_G^+(v)$. It should be noticed that this set will always include $\perp$ – this must be taken into account when checking if two graph nodes share common ancestors.

Definition 3 (root nodes, leaf nodes). *Let G be a hierarchical graph. Let us define $roots(G) = \{v \in V_G : par_G(v) = \perp\}$, i.e. the set of root nodes of G, and $leaves(G) = \{v \in V_G : ch_G(v) = \emptyset\}$, i.e. the set of leaf nodes of G.*

Nodes and edges in hierarchical graphs can be labelled and attributed. Attributes represent properties of components and relations represented by graph nodes and edges, respectively. For the rest of this paper, let R_V and R_E be the sets of node and edge labels, respectively; let A_V and A_E be the sets of node and edge attributes. Every attribute $a \in A_V \cup A_E$ is associated with set D_a, called the domain of a. Let D be the union of all attribute domains.

Let us also assume that a special symbol, ε, is included in all attribute domains. This symbol will be used if a value of a given attribute is undefined.

Definition 4 (labelled attributed graph). *A 5-tuple $G = (g, vlab_G, elab_G, vatr_G, eatr_G)$ is a labelled, attributed hierarchical graph if*

1. *$g = (V_g, \dots, ch_g)$ is a hierarchical graph,*
2. *$vlab_G : V_g \to R_V$ and $elab_G : E_g \to R_E$ are functions specifying node and edge labels,*
3. *$vatr_G : V_g \to \mathcal{P}(A_V)$ and $eatr_G : E_g \to \mathcal{P}(A_E)$ are functions specifying node and edge attributes.*

The labelling and attributing of a subgraph H of a labelled attributed hierarchical graph G is defined by restricting respective functions in G. The given hierarchical graph G can represent a potentially infinite subset of grids having the same structure. To represent an actual grid we must define an instance of a graph. An instance of a hierarchical graph is a hierarchical labelled attributed graph in which to each attribute a a value from the set of possible values of this attribute has been assigned.

Definition 5 (instance graph). *A triple $I_G = (G, vval_G, eval_G)$ is an instance of a labelled, attributed, hierarchical graph if*

- *$G = (g, \dots, eatr_G)$ is a labelled, attributed, hierarchical graph,*
- *$vval_G = \{(v, a, x) : v \in V_G, a \in vatr_G(v), x \in D_a\}$ is a set of node attribute values which contains exactly one triple for every possible combination of v and a, and*
- *$eval_G = \{(e, a, x) : e \in E_G, a \in eatr_G(e), x \in D_a\}$ is a set with analogous condition (for edge attribute values).*

In the paper notation $a(v) = x$, $b(e) = y$ will be used instead of $(v, a, x) \in vval_G$, $(e, b, y) \in eval_G$.

Definition 6 (instance subgraph). *Instance graph I_H is a subgraph of I_G if*

- *$H = (h, \dots, eatr_H)$ is a subgraph of $G = (g, \dots, eatr_G)$,*
- *$vval_H = vval_G|_{V_H \times A_V \times D}$, and $eval_H = eval_G|_{E_H \times A_E \times D}$.*

A computational grid contains different types of elements: physical nodes (computers, computational elements), virtual ones (software, operating systems, applications) and storage elements which can be treated both as physical elements (designated hard drives) or virtual (residing on other physical elements). Moreover, some virtual elements of a grid are responsible for performing computational tasks sent to the grid, while other elements (services) are only responsible for managing the grid structure data, behaviour and the flow of the tasks.

In this paper we introduce a notion of a layer composed of subgraphs of hierarchical graphs as a formal description of one part of a grid. For example at the physical layer a graph may represent a network located at one place. As such parts of a grid are independent of each other they are represented by disjoint subgraphs. On the other hand, each such graph consists of elements that can communicate with each other, so they are represented by connected graphs.

The type of element represented by a given node is defined by a node label. Thus we can partition the set of node labels into four subsets, each one containing labels representing objects from different grid layers: computational, managing, resources and user/tasks. On the basis of this division a graph can also be partitioned provided there are no hierarchically related nodes with labels from different layers. Thus, formally a layer can be defined as:

Definition 7 (graph layers). *Let G be an instance graph. Let $L = \{L_1, L_2, \ldots L_n\}$ be a partition of R_V such that $\forall v \in V : vlab(v) \in L_i \Rightarrow vlab(ch^+(v)) \subseteq L_i$. Let $\{G_1, \ldots G_n\}$ be a family of subgraphs of G such that every G_i is induced by $\{v \in V : vlab(v) \in L_i\}$. Members of this family are called layers of G.*

The layers are denoted by numbers, but to make the given example clear they are named CL, ML, RL and UTL. The graph layers are connected by interlayer edges which represent how the communication between different layers of a grid is carried out. While the layered graph represents the structure (topology) of a grid, a semantics is needed to define the meaning of its elements. Such information may include an operation system installed on a given computational element, specific software installed, size of memory or storage available, type of a resource represented by a given element etc. This information is usually encoded in attributes assigned to elements of a graph. Graph elements representing grid nodes can have several attributes assigned.

3 Grid Representation

The layered graph defined in the previous section can contain any number of layers. Three layers are used to describe the structure of the grid.

Let $R_V = \{C, CE, RT, ST, CM, index, services, broker, jobscheduling, monitoring\}$ (where C stands for a computer, CE – for a computational element, CM – for a managing unit, RT – for a router and ST – for storage), be a set of node labels used in a grid representation. Let R_V be partitioned into three sets: L_1, L_2 and L_3, such that $L_1 = \{C, CE, RT\}$, $L_2 = \{CM\}$, and $L_3 = \{index, services, broker, jobscheduling, monitoring\}$.

An example of a layered graph representing a simple grid is depicted in Fig. 1b. The top layer of this graph, layer RL, represents the main resources/services responsible for task distributing/assigning/allocating. The second layer represents the elements responsible for the grid management. Each node labelled CM represents a management system for a part of a grid, for example for a

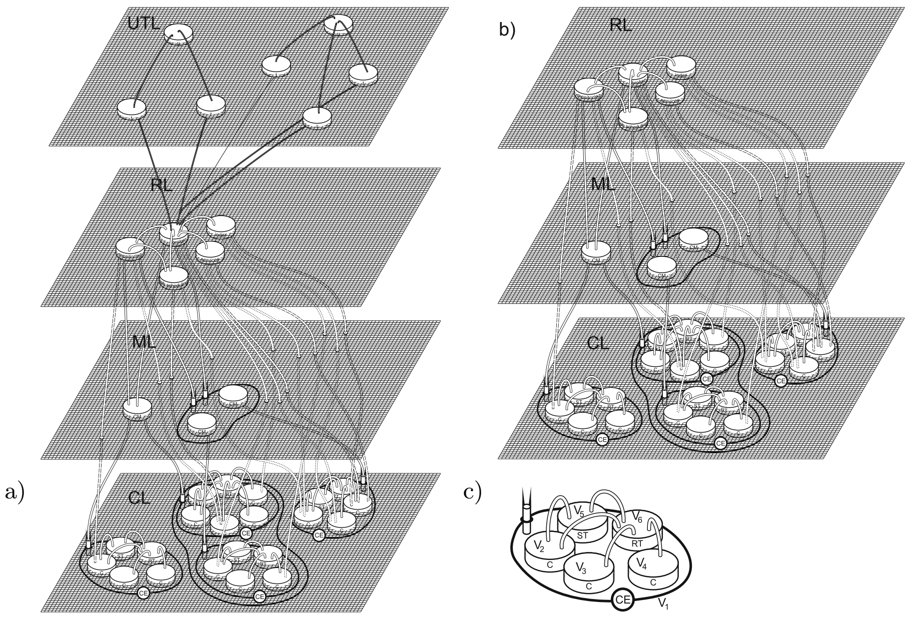


Fig. 1. A layered graph representing a grid and its part representing a single computational element

given subnetwork/computing element. The management elements CM can be hierarchical. Such a hierarchy represents a situation in which data received from the grid services is distributed internally to lower-level managing components and each of them in turn is responsible for some computational units. At the same time each CM element can be responsible for managing one or more computational elements.

Each node label describes the type of a grid element represented by a given node. Grid elements have some additional properties represented by attributes. Let attributes of nodes be defined on the basis of the node label. We also assume that attributes are defined for low-level nodes only. The attributes for their ancestors are computed on the basis of the children attributes.

Thus, let the set of node attributes be $A = \{capacity, RAM, OS, apps, CPU, class\}$. Let $vatr$ be a function assigning sets of attribute names to nodes of a layered graph depicted in Fig. 1b.

The sets of attributes are determined according to two following rules. For nodes with children (i.e. $\forall v : ch(v) \neq \emptyset$) $vatr(v) = \bigcup_{w \in ch(v)} vatr(w)$, while for leaf nodes (i.e. $\forall v : ch(v) = \emptyset$) labels determine attributes, according to the following table.

vlab	*vatr*
C	$RAM, OS, CPU, apps$
ST	$capacity, type$
RT	$class$
CM	$load$
$broker$	$load$
$index$	$size$

In Fig. 1c a part of the graph from Fig. 1b is shown. It represents one computational element, which is a part of a computational layer. For this graph the sets of attributes are described in the following way:

$$vatr(v_i) = \begin{cases} \{RAM, OS, CPU, apps\} & i = 2, 3, 4 \\ \{capacity, type\} & i = 5 \\ \{class\} & i = 6 \end{cases}$$

For node v_1, which is a higher level node, its attributes are computed on the basis of children attributes. Thus $vatr(v_1) = \{RAM$, OS, CPU, $apps$, $capacity$, $type$, $class\}$.

To node attributes the appropriate values have to be assigned. Firstly, domains for all attributes have to be defined. In this example let $D_{OS} = \{Win$, Lin, $MacOS$, $GenUnix\}$, $D_{apps} = \{app1$, $app2$, $app3$, $app4\}$, $D_{CPU} = \{cpu1$, $cpu2$, $cpu3\}$, $D_{class} = \{c1$, $c2$, $c3$, $c4\}$, $D_{RAM} = \{n \in \mathcal{N} : 0 \leq n \leq 64\}$, $D_{capacity} = \{m \in \mathcal{N} : 0 \leq m \leq 1000\}$, $D_{type} = \{FAT$, $FAT32$, $NTFS$, $ext3$, $ext4$, $xfs\}$. In the example the values of attributes for node v_2 are as follows: $RAM(v_2) = 4$, $CPU(v_2) = cpu1$, $OS(v_2) = Win$, $apps(v_2) = app1$. For the hierarchical node v_1 the values for the properties CPU, OS and $apps$ are a collection containing all the values of attributes of the children nodes of v_1. In case of numerical properties it is a sum of the values of its children nodes.

The above approach allows us to represent a static grid structure. To represent its dynamics we will have to introduce users and tasks. Let a label u represent a user and t a task. Let $R_V^d = R_V \cup \{u, v\}$ be the total set of node labels for the dynamic grid representation. Let $L_4 = \{u, v\}$ be a set determining a layer representing users and tasks. This layer is placed above the layer RL (Fig. 1a). The set of attributes for nodes labelled by t contains RAM, OS, CPU and $apps$, while for the ones labelled by u - $slots$ and $tasks$.

4 Graph Generation

Graphs can be generated by graph grammars, which are systems of graph transformations called productions. Different types of graph grammars have been investigated and their ability to generate different structures has been proved [7,3].

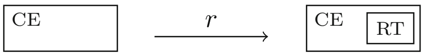


Fig. 2. A transformation rule (attributes omitted)

To be able to find in a current graph a subgraph that can be replaced by a right side of the transformation rule, a morphism of graphs has to be defined. On the basis of the morphisms we can formally define a transformation rule.

Definition 8 (graph morphism). *Let I_G and I_H be two instance graphs. Let $f = (f_V, f_E)$ be a pair of functions, $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$. V_G and E_G are disjoint, thus instead of $f_V(v)$ and $f_E(e)$ we will simply write $f(v)$ and $f(e)$, as well as $f : I_G \to I_H$.*

f is a graph morphism if

- *$f(s_G(e)) = s_H(f(e))$ for all $e \in E_G$,*
- *$f(t_G(e)) = t_H(f(e))$ for all $e \in E_G$,*
- *$f(v) \in ch_H(f(p))$ for all $v \in V_G$ such that $\exists\, p \in V_G : v \in ch_G(p)$, i.e. parent-child relations are preserved,*
- *$f(v) \notin ch_H^+(f(w))$ and $f(w) \notin ch_H^+(f(v))$ for all $v, w \in V_G$ such that $v \notin ch_G^+(w)$ and $w \notin ch_G^+(v)$, i.e. unrelated nodes stay unrelated,*
- *$vlab_G(v) = vlab_H(f(v))$ for all $v \in V_G$,*
- *$elab_G(e) = elab_H(f(e))$ for all $e \in E_G$,*
- *$vatr_G(v) \subseteq vatr_H(f(v))$ for all $v \in V_G$, and*
- *$eatr_G(e) \subseteq eatr_H(f(e))$ for all $e \in E_G$.*

It should be noted that in this definition it is required that all attributes from I_G have to be present in I_H, but it is not required that their values are identical. Node v, labelled “C” and having attributes $ram(v) = 8$, $cpu(v) = cpu1$, can be mapped to the other node u with the same label, and attributes $ram(u) = 16$, $cpu(u) = cpu2$, $apps(u) = app2$.

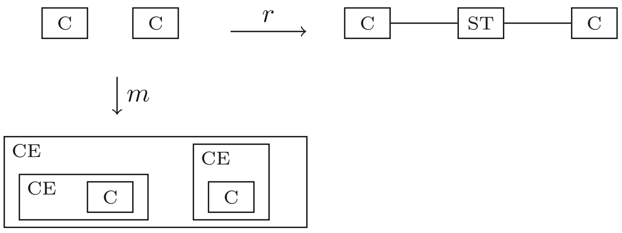


Fig. 3. A rule matched to a graph

Definition 9 (root-level and nesting morphisms). *Let $f : I_G \to I_H$ be a graph morphism. f is root-level if $f(root(G)) \subseteq root(H)$. Otherwise, f is nesting.*

Definition 10 (partial morphism). *A partial morphism f from I_G to I_H is a morphism from some instance subgraph $I_S \subseteq I_G$ to I_H.*

Each rule has two graphs, L and R, and a partial morphism $r : L \to R$. Finding a matching subgraph in G means finding a morphism $m : L \to G$ (the matching morphism). Let us denote the preimage of R, i.e. $r^{-1}(R)$, by $dom(r)$. These nodes and edges of L which are not in $dom(r)$ represent elements to be removed, and will be denoted by V_{DEL} and E_{DEL}. What will actually be removed, though, are nodes and edges of G: $m(V_{DEL})$ and $m(E_{DEL})$. Nodes and edges of R which are not in $r(L)$ will be denoted by V_{INS} and E_{INS}; their copies will be added to G.

The nodes and edges which are neither deleted nor inserted provide a context in which these operations take place. For example, there may be nodes labelled "CE" present in L and R (with r mapping one of them to the other). The right-hand side can then specify that a new node needs to be created as a child of the "CE" node (see Fig. 2).

We will use V_{RINS} to denote nodes from V_{INS} which are at the root level in R ($V_{RINS} = V_{INS} \cap root(R)$). When their copies are added to G it has to be remembered that their siblings, context nodes from R, can be mapped to nodes in G which are not root-level. Moreover even if two nodes are siblings in L and R, their images in G do not have to be such (see Fig. 3).

Therefore a rule must contain additional instructions, specifying which nodes exactly should be assigned as parents. A copy of root-level node from R can be inserted either at the root level of G, or as a brother of a node corresponding to one of its siblings from R. This will be specified by a function, which assigns to each node from V_{RINS} either $\perp$, or some context node from R (in which case the rule diagram will have a dotted rectangle drawn around the considered nodes).

Definition 11 (graph transformation rule). *Let L and R be instance graphs, $r : L \to R$ be an injective root-level partial morphism, $PI : V_{RINS} \to \{\perp\} \cup (root(R) - V_{INS})$ be a function (parent instructions), and AI be a sequence of attribute assignment instructions on R. $p = (L, R, r, PI, AI)$ is a transformation rule.*

In a grid generation process we use several categories of productions [8]. Some productions operate on only one layer, other productions - on several layers. Moreover, we need productions that can both add and remove elements from the grid represented by the layered graph. To make sure the functionality of a grid is preserved the productions removing elements have also to be able to invoke rules responsible for maintaining the stability of affected elements. The production can also be required to start some actions. For example if a production removes a node representing a computational element containing a virtual resource, all other elements using this resource must be redirected to its copy. If such a copy does not exist it must be generated on other element before it is removed from the grid.

During the application of the production the attributes must also be properly assigned. To make it possible the attribute assignment equations are defined as a

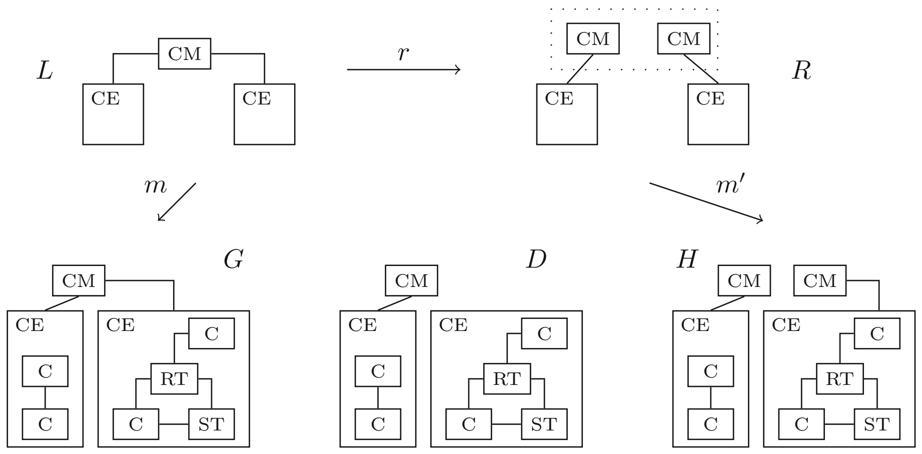


Fig. 4. Stages of the rule application process (attributes omitted)

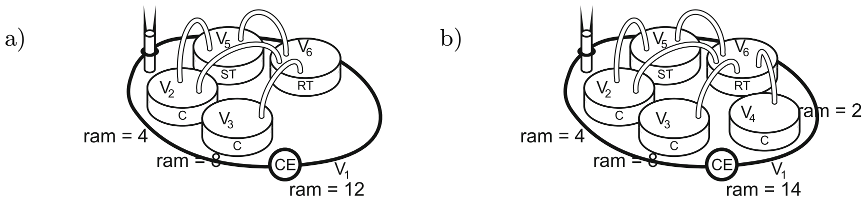


Fig. 5. A part of the graph from Fig. 1b representing a single computational element a) before b) after applying a production

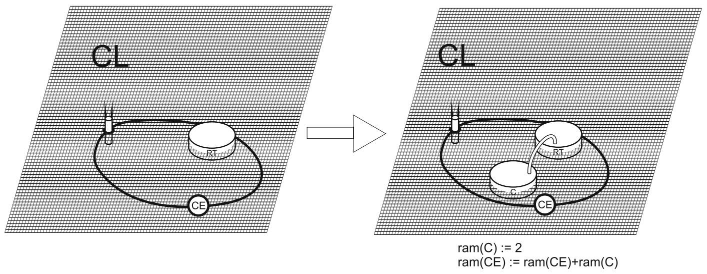


Fig. 6. The production responsible for adding a computer to the grid

part of each production. In Fig. 5 an instance of a part of the graph from Fig. 1b is depicted. In Fig. 6 a production responsible for adding a computer is shown. There are two attribute assignments associated with this production. When the left side of the production is matched to the current graph the matched elements are replaced by the right side of the production. Then the attribute assignments are applied. The first one assigns a value 2 to the attribute *ram* of the node labeled by C on the right side. The second assignment is more complex: it takes the current value of the attribute *ram* of the node matched to the one labelled by CE, then takes the value of the same attribute assigned to the node labelled by C, adds them and the sum is then used to replace the previous value of the attribute *ram* of the node labelled CE.

5 Conclusions

In this paper a new approach to grid representation, which uses graph structures, has been presented. Layered graphs has been defined and an example of such a graph as a grid representation has been shown. Using a graph based representation enables us to use graph grammars as a tool to generate a grid. In this paper a grid graph grammar has been described and some of its productions have been depicted and explained. As layered graphs are attributed, both the structure and the parameters of the grid can be generated at the same time. Productions presented in this paper are used to generate the grid. The next step will consist in adding productions that will be able to simulate the work and behaviour of the grid as well. Then a new grid simulator will be implemented. As we have the possibility of modeling the structure of the environment in a dynamic way, the design of the simulator focuses on building the topology and adopting it in the runtime. The basic requirement is to enable the generation of a wide range of different grid structures which would be described using the proposed grid grammar for the simulation purpose. It will enable one quick verification of suggested solutions and allow for testing different types of grids and different grid configurations before actually implementing them. The management layer defined explicitly gives us the opportunity to check some specific configuration or scheduling algorithms. In this paper we do not consider the problem of whether an existing grid structure belongs to the language generated be the given graph grammar, but it seems to be an interesting area for future research.

References

1. Lu, D., Dinda, P.A.: GridG: Generating Realistic Computational Grids. SIGMETRICS Performance Evaluation Review, 30(4), 33–40 (2003)
2. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Int. J. Supercomp. App., 200–220 (2001)
3. Grabska, E., Palacz, W.: Hierarchical graphs in creative design. MG&V 9(1/2), 115–123 (2000)
4. Grabska, E., Strug, B.: Applying Cooperating Distributed Graph Grammars in Computer Aided Design. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 567–574. Springer, Heidelberg (2006)
5. Joseph, J., Ernest, M., Fellenstein, C.: Evolution of grid computing architecture and grid adoption models (online)
6. Joseph, J., Fellenstein, C.: Grid Computing. IBM Press (2004)
7. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph. Transformations, vol. 1-3. World Scientific, London (1997-1999)
8. Strug, B., Ryszka, I., Grabska, E., Ślusarczyk, G.: Generating a virtual computational grid by graph transformations. In: INGRID 2010. Springer (2011)

On General Regression Neural Network in a Nonstationary Environment

Yoichi Hayashi[1] and Lena Pietruczuk[2]

[1] Department of Computer Science, Meiji University,
Tama-ku, Kawasaki 214-8571, Japan
hayashiy@cs.meiji.ac.jp

[2] Department of Computer Engineering, Czestochowa University of Technology,
ul. Armii Krajowej 36, 42-200 Czestochowa, Poland
lena.pietruczuk@kik.pcz.pl

Abstract. In this paper we present the method for estimation of unknown function in a time-varying environment. We study the probabilistic neural network based on the Parzen kernels combined with the recursive least square method. We present the conditions for convergence in probability and we discuss the experimental results.

1 Introduction

In literature there are many problems requiring finding the unknown regression function in a time-varying environment. This article will describe a method of solving this problem in the following case. Let (X_i, Y_i) for $i = 1, \ldots, n$ be a sequence of pairs of random variables, where $X_i \in \mathbb{R}^p$ and $Y_i \in \mathbb{R}$. We will study the system

$$Y_i = \phi(X_i) + ac_i + Z_i, \quad i = 1, \ldots, n, \tag{1}$$

where $\phi(x)$ is an unknown regression function, c_i in a known sequence of numbers, a is unknown constant, Z_i are random variables representing noise and X_i are equally distributed random variables with density function $f(x)$. We consider the case when the noise Z_i satisfies the following conditions

$$E[Z_i] = 0, \qquad Var[Z_i] = \sigma_i^2 < \sigma_Z^2, \qquad \text{for } i = 1 \ldots, n. \tag{2}$$

The problem is to find unknown value of parameter a and to estimate unknown function ϕ. It should be emphasized that such problem was never solved in literature. The method applied in this paper is based on the nonparametric estimates, named in the area of soft computing, probabilistic neural networks [40]. Nonparametric regression estimates in a stationary environment were studied in [6], [7], [10], [17]-[21], [27]-[29] and [32]-[35], whereas non-stationary environment was considered in [9], [22]-[26], [30] and [31], assuming stationary noise. For excelent overviews of those algorithms the reader is reffered to [8] and [11].

2 Algorithm

The estimation of regression function $\phi(x)$ requires two steps. First we estimate the unknown value of parameter a by estimate $\hat{a}_n$. Then we find the regression function of random variables (X_i, Y_i') for $i \in 1, \ldots, n$, where Y_i' is in the form

$$Y_i' = Y_i - \hat{a}_n c_i = \phi(X_i) + (a - \hat{a}_n)c_i + Z_i, \quad i = 1, \ldots, n. \tag{3}$$

To estimate the value of parameter a we use the recursive least squares error method [1]. Therefore we use the formula

$$\hat{a}_i = \hat{a}_{i-1} + \frac{c_i}{\sum_{j=1}^{i} c_i}(Y_i - \hat{a}_{i-1}c_i). \tag{4}$$

After computing the value of $\hat{a}_n$ we can estimate the regression function $\phi(x)$ under assumption $a := \hat{a}_n$. In literature there are known methods of finding the unknown regression function. We propose to use the Parzen kernel in the form

$$K_n'(x, u) = h_n'^{-p} K(\frac{x - u}{h_n'}), \tag{5}$$

$$K_n(x, u) = h_n^{-p} K(\frac{x - u}{h_n}), \tag{6}$$

where K is an appropriately selected function such that

$$||K||_\infty < \infty, \tag{7}$$

$$\int |K(y)|dy < \infty, \tag{8}$$

$$\lim_{y \longrightarrow \infty} |yK(y)| = 0, \tag{9}$$

$$\int_{R^p} K(y)dy = 1 \tag{10}$$

and h_n, h_n' are certain sequences of numbers. Let us denote the estimator of density function $f(x)$ by $\hat{f}(x)$ and the estimator of function $R(x) = f(x)\phi(x)$ by $\hat{R}(x)$. Then function

$$\phi(x) = \frac{R(x)}{f(x)} \tag{11}$$

can be estimated by

$$\hat{\phi}_n(x) = \frac{\hat{R}_n(x)}{\hat{f}_n(x)}. \tag{12}$$

By using the formulas 5 and 6 we obtain

$$\hat{f}_n(x) = \frac{1}{n}\sum_{i=1}^{n} K_n'(x, X_i) \tag{13}$$

and

$$\hat{R}_n(x) = \frac{1}{n}\sum_{i=1}^{n} Y_i' K_n(x, X_i). \tag{14}$$

Then the estimator of a regression function takes the form

$$\hat{\phi}_n(x) = \frac{\sum_{i=1}^{n} Y_i' K\left(\frac{x-X_i}{h_n}\right)}{\sum_{i=1}^{n} K\left(\frac{x-X_i}{h_n'}\right)} \tag{15}$$

which is known in the literature under the name probabilistic neural network [40]. The block diagram of generalized regression neural network is presented in Fig. 1.

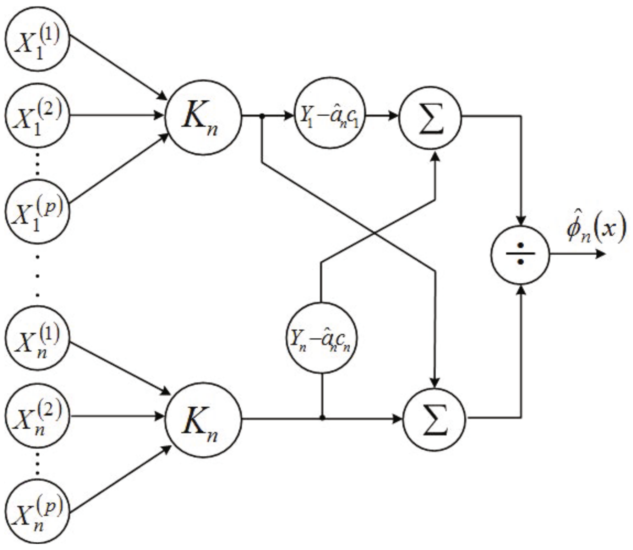


Fig. 1. Block diagram of generalized regression neural network

Theorem 1. *If the following conditions are satisfied*

$$\int_{\mathbb{R}^p} \phi^2(x) f(x) dx < \infty, \tag{16}$$

$$\lim_{n\longrightarrow\infty}\left(\frac{\sum_{i=1}^{n} c_i}{\sum_{i=1}^{n} c_i^2}\right) = 0 \tag{17}$$

and

$$\lim_{n\longrightarrow\infty}\left(\frac{\sum_{i=1}^{n} c_i^2}{\left(\sum_{i=1}^{n} c_i^2\right)^2}\right) = 0 \tag{18}$$

then

$$\hat{a}_n \xrightarrow{n} a \text{ in probability.} \tag{19}$$

Theorem 2. *Let us assume that*

$$\int_{\mathbb{R}^p} \phi^2(x) f(x) dx < \infty. \tag{20}$$

If conditions (7)-(10) are satisfy and

$$\hat{a}_n \xrightarrow{n} a \text{ in probability}, \tag{21}$$

$$n^{-1} h_n'^{-p} \to 0, \qquad n^{-1} h_n^{-p} \to 0, \tag{22}$$

$$h_n' \to 0, \qquad h_n \to 0, \tag{23}$$

then $\phi_n(x) \xrightarrow{n} \phi(x)$ in probability for each x where $\phi(x)$ is continuous.

Proofs of Theorems 1 and 2 can be based on arguments similar to those in [1] and [4], using theorem 4.3.8 in [43].

3 Experimental Results

In the following simulations we consider the system

$$Y_i = 10x^2 \sin(x) + 5n^t + Z_i, \tag{24}$$

where Z_i are random variables from normal distribution $\mathcal{N}(0, 1)$. Let us assume that input data come from the uniform distribution. First we select the kernel function. For the purpose of this paper we choose Epanecznikow kernel [12] and assume that $h_n' = h_n$.

By examine the relationship between the number of data elements and the value of the estimator $\hat{a}_n$ we obtained the results illustrated in Fig. 2. In this experiment $D = D' = 2.5$, $H = H' = 0.4$, $t = 0.27$ and the input data come from the uniform distribution $\mathcal{U}(-5.5; 5.5)$. As we can see with increasing number of input data the value of $\hat{a}_n$ converges to the real value of parameter a.

In Figure 3 we can observe how the number of data elements affects the value of the mean square error (MSE) between estimator $\hat{\phi}_n(x)$ and function $\phi(x)$. We assume that $D = D' = 2.2$, $H = H' = 0.38$, $t = 0.25$ and input data come from the uniform distribution $\mathcal{U}(-7; 7)$. We can observe that the value of the MSE is decreasing with increasing number of input data.

The next experiment shows the dependence between the value of parameter $H = H'$ and the value of the MSE (see Fig. 4). In this case $n = 5000$, $t = 0.25$, input data come from the uniform distribution $\mathcal{U}(-4.9; 4.9)$ and $D = D'$. As we can see with increasing value of H the error of the algorithm decreases however for big values of D and large H the MSE can still be small. We should notice that with increasing value of parameters D and D' the value of the MSE, for small value of H, can increas. Therefore these values should be properly chosen.

The last experiment shows how well the function $\hat{\phi}(x)$ is adjusted to the input-output data and to function $\phi(x)$. The input-output data come from the uniform distribution $\mathcal{U}(-5; 5)$, $D = D' = 1.7$, $H = H' = 0.4$ and $t = 0.3$.

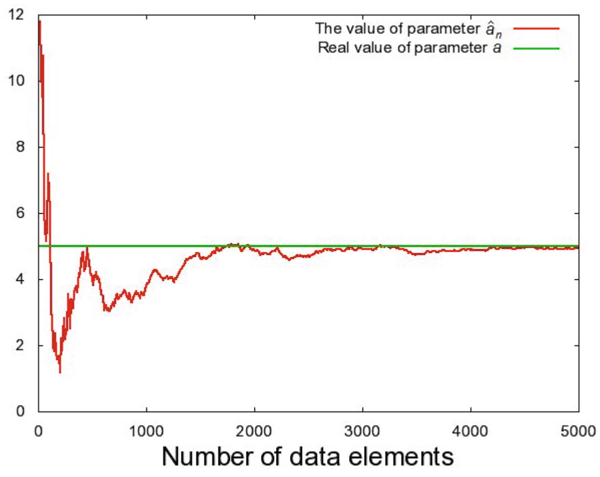


Fig. 2. The dependence between the number of data elements and the value of the estimator $\hat{a}_n$

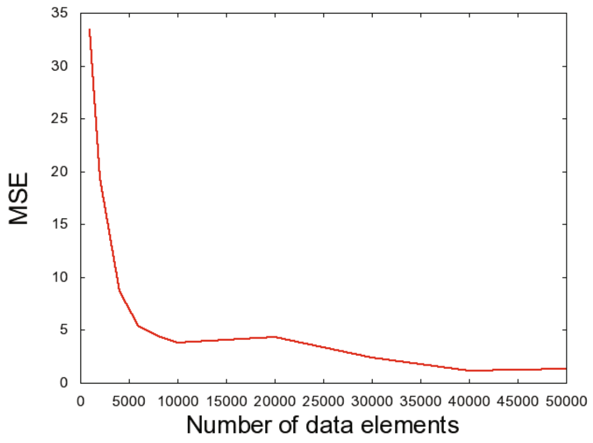


Fig. 3. The dependence between the number of data elements and the value of the MSE

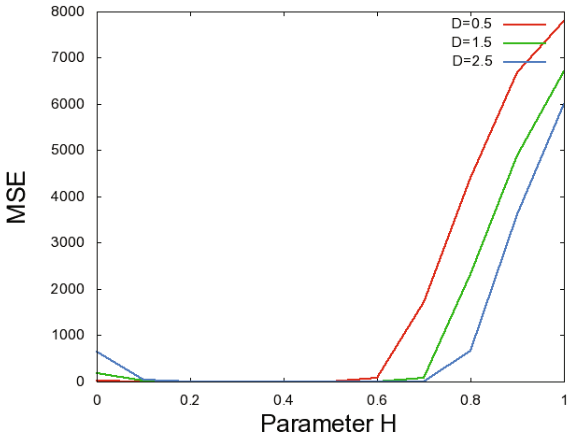


Fig. 4. The dependence between the value parameter H and the value of the MSE

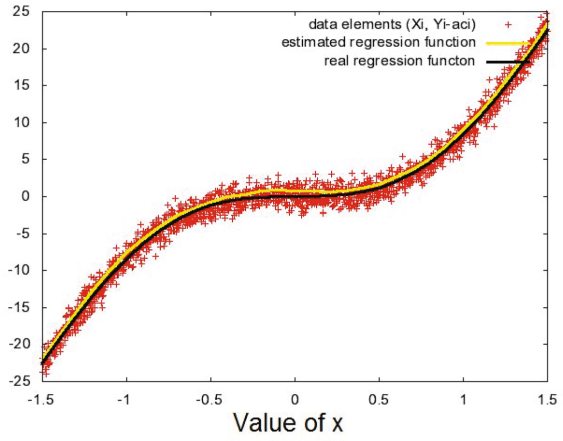


Fig. 5. The input-output data, the obtained estimator values and value of function $\phi(x)$

4 Conclusion and Future Work

In this paper we presented the probabilistic neural network, based on the Parzen kernels, combined with the recursive least square method as a method of estimation of unknown function in a time-varying environment. We presented the conditions for convergence and we discussed the experimental results. Currently we are adapting supervised and unsupervised neural networks [3], [5], [16] and neuro-fuzzy structures [13], [36]-[39], [41], [42] for learning in time-varying environment.

Acknowledgments. This paper was prepared under project operated within the Foundation for Polish Science Team Programme co-financed by the EU European Regional Development Fund, Operational Program Innovative Economy 2007-2013, and also supported by the National Science Center NCN.

References

1. Albert, A.E., Gardner, L.A.: Stochastic Approximation and Nonlinear Regression, vol. (42). MIT Press, Cambridge (1967)
2. Benedetti, J.: On the nonparametric estimation of regression function. Journal of Royal Statistical Society B 39, 248–253 (1977)
3. Bilski, J., Rutkowski, L.: A fast training algorithm for neural networks. IEEE Transactions on Circuits and Systems II 45, 749–753 (1998)
4. Cacoullos, P.: Estimation of a multivariate density. Annals of the Institute of Statistical Mathematics 18, 179–190 (1965)
5. Cierniak, R., Rutkowski, L.: On image compression by competitive neural networks and optimal linear predictors. Signal Processing: Image Communication - a Eurasip Journal 15(6), 559–565 (2000)
6. Gałkowski, T., Rutkowski, L.: Nonparametric recovery of multivariate functions with applications to system identification. Proceedings of the IEEE 73, 942–943 (1985)
7. Gałkowski, T., Rutkowski, L.: Nonparametric fitting of multivariable functions. IEEE Transactions on Automatic Control AC-31, 785–787 (1986)
8. Greblicki, W., Pawlak, M.: Nonparametric system indentification. Cambridge University Press (2008)
9. Greblicki, W., Rutkowska, D., Rutkowski, L.: An orthogonal series estimate of time-varying regression. Annals of the Institute of Statistical Mathematics 35, Part A, 147–160 (1983)
10. Greblicki, W., Rutkowski, L.: Density-free Bayes risk consistency of nonparametric pattern recognition procedures. Proceedings of the IEEE 69(4), 482–483 (1981)
11. Györfi, L., Kohler, M., Krzyżak, A., Walk, H.: A Distribution-Free Theory of Nonparametric Regression, USA. Springer Series in Statistics (2002)
12. Härdle, W.: Applied Nonparametric Regression. Cambridge University Press, Cambridge (1990)
13. Nowicki, R.: Rough Sets in the Neuro-Fuzzy Architectures Based on Non-monotonic Fuzzy Implications. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 518–525. Springer, Heidelberg (2004)

14. Nowicki, R., Pokropińska, A.: Information Criterions Applied to Neuro-Fuzzy Architectures Design. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 332–337. Springer, Heidelberg (2004)
15. Parzen, E.: On estimation of a probability density function and mode. Analysis of Mathematical Statistics 33(3), 1065–1076 (1962)
16. Patan, K., Patan, M.: Optimal training strategies for locally recurrent neural networks. Journal of Artificial Intelligence and Soft Computing Research 1(2), 103–114 (2011)
17. Rafajłowicz, E.: Nonparametric orthogonal series estimators of regression: A class attaining the optimal convergence rate in L_2. Statistics and Probability Letters 5, 219–224 (1987)
18. Rutkowski, L.: Sequential estimates of probability densities by orthogonal series and their application in pattern classification. IEEE Transactions on Systems, Man, and Cybernetics SMC-10(12), 918–920 (1980)
19. Rutkowski, L.: Sequential estimates of a regression function by orthogonal series with applications in discrimination, New York, Heidelberg, Berlin. Lectures Notes in Statistics, vol. 8, pp. 236–244 (1981)
20. Rutkowski, L.: On system identification by nonparametric function fitting. IEEE Transactions on Automatic Control AC-27, 225–227 (1982)
21. Rutkowski, L.: Orthogonal series estimates of a regression function with applications in system identification. In: Probability and Statistical Inference, pp. 343–347. D. Reidel Publishing Company, Dordrecht (1982)
22. Rutkowski, L.: On Bayes risk consistent pattern recognition procedures in a quasi-stationary environment. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4(1), 84–87 (1982)
23. Rutkowski, L.: On-line identification of time-varying systems by nonparametric techniques. IEEE Transactions on Automatic Control AC-27, 228–230 (1982)
24. Rutkowski, L.: On nonparametric identification with prediction of time-varying systems. IEEE Transactions on Automatic Control AC-29, 58–60 (1984)
25. Rutkowski, L.: Nonparametric identification of quasi-stationary systems. Systems and Control Letters 6, 33–35 (1985)
26. Rutkowski, L.: The real-time identification of time-varying systems by nonparametric algorithms based on the Parzen kernels. International Journal of Systems Science 16, 1123–1130 (1985)
27. Rutkowski, L.: A general approach for nonparametric fitting of functions and their derivatives with applications to linear circuits identification. IEEE Transactions Circuits Systems CAS-33, 812–818 (1986)
28. Rutkowski, L.: Sequential pattern recognition procedures derived from multiple Fourier series. Pattern Recognition Letters 8, 213–216 (1988)
29. Rutkowski, L.: Nonparametric procedures for identification and control of linear dynamic systems. In: Proceedings of 1988 American Control Conference, June 15-17, pp. 1325–1326 (1988)
30. Rutkowski, L.: An application of multiple Fourier series to identification of multivariable nonstationary systems. International Journal of Systems Science 20(10), 1993–2002 (1989)
31. Rutkowski, L.: Nonparametric learning algorithms in the time-varying environments. Signal Processing 18, 129–137 (1989)
32. Rutkowski, L., Rafajłowicz, E.: On global rate of convergence of some nonparametric identification procedures. IEEE Transaction on Automatic Control AC-34(10), 1089–1091 (1989)

33. Rutkowski, L.: Identification of MISO nonlinear regressions in the presence of a wide class of disturbances. IEEE Transactions on Information Theory IT-37, 214–216 (1991)
34. Rutkowski, L.: Multiple Fourier series procedures for extraction of nonlinear regressions from noisy data. IEEE Transactions on Signal Processing 41(10), 3062–3065 (1993)
35. Rutkowski, L., Gałkowski, T.: On pattern classification and system identification by probabilistic neural networks. Applied Mathematics and Computer Science 4(3), 413–422 (1994)
36. Rutkowski, L.: A New Method for System Modelling and Pattern Classification. Bulletin of the Polish Academy of Sciences 52(1), 11–24 (2004)
37. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, Melbourne, December 2-5, vol. 3, pp. 1428–1431 (2001)
38. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)
39. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)
40. Specht, D.F.: A general regression neural network. IEEE Transactions on Neural Networks 2, 568–576 (1991)
41. Starczewski, L., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
42. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)
43. Wilks, S.S.: Mathematical Statistics. John Wiley, New York (1962)

Determination of the Heat Transfer Coefficient by Using the Ant Colony Optimization Algorithm

Edyta Hetmaniok, Damian Słota, and Adam Zielonka

Institute of Mathematics,
Silesian University of Technology,
Kaszubska 23, 44-100 Gliwice, Poland
{edyta.hetmaniok,damian.slota,adam.zielonka}@polsl.pl

Abstract. The paper presents a numerical method of solving the inverse heat conduction problem based on the respectively new tool for combinational optimization named Ant Colony Optimization (ACO). ACO belongs to the group of swarm intelligence algorithms and is inspired by the technique of searching for the shortest way connecting the ant-hill with the source of food. In the proposed approach we use this algorithm for minimizing the proper functional appearing in the procedure of determining the value of heat transfer coefficient in the successive cooling zones.

Keywords: Swarm Intelligence, ACO algorithm, Inverse Heat Conduction Problem.

1 Introduction

In recent time there appeared many algorithms, like genetic algorithms, neural algorithms or immune algorithms, inspired by the mechanisms functioning successively in nature. This type of methods includes also the Swarm Intelligence (SI) algorithms [1, 2], which is a group of algorithms of artificial intelligence based on the collective behavior of decentralized, self-organized systems of objects. The idea was introduced by Gerardo Beni and Jing Wang in 1989, in the context of cellular robotic systems. Each object of this system is insignificant individually but together, by using some specific techniques, their community is able to solve complicated problems.

Examples of such systems of units are swarms of insects like ants or bees. During the centuries of evolution they elaborated special unique techniques of searching for the food, imitating of which resulted with the Ant Colony Optimization Algorithm (ACO) and Artificial Bee Colony Algorithm (ABC). In recent times the ACO algorithms were applied to a number of different combinatorial optimization problems, like for example traveling salesman problem [3–5], quadratic assignment problem [6], vehicle routing problem [7], connection-oriented network routing problem [8], connection-not-oriented network routing problem [9] and others, as well as to continuous optimization problems [10, 11]. In paper [12] the authors have applied the ACO algorithm for solving the inverse heat conduction problem, whereas in [13] the ABC algorithm was used for this purpose.

The inverse heat conduction problem is a problem of determining the temperature distribution in considered domain, usually together with reconstructing one of the boundary condition, having the incomplete information about the process. Therefore, finding an analytical solution of inverse problem is almost impossible, except the simplest cases, and searching for the new approximate methods is desirable. The bibliography dedicated to the inverse heat conduction problem is much less large than the bibliography about the direct problems. Examples of the analytical techniques for solving the direct and inverse problems concerning steady and unsteady heat flow are given for example in [14, 15]. In [16] the authors determine the heat flux with the aid of the momentary measurements of temperature, by using the Green function, method of iterative regularization and Tichonov regularization. The other methods appeared for solving the inverse problems are for example: the Monte Carlo method [17], the mollification method introduced by Mourio [18], methods based on the wavelets theory [19] and very popular in recent time genetic algorithms [20–22]. In the current paper, we propose to use the ACO algorithm for minimizing some functional, being a crucial part in the procedure of reconstructing the values of heat transfer coefficient appearing in boundary condition of the third kind of the inverse problem. Similar procedure applying the ACO algorithm, but leading to reconstruction of the boundary condition of the first and second kind, the authors have presented in [12].

2 Ant Colony Optimization Algorithm for Finding the Global Minimum

Ant Colony Optimization algorithm was invented by Marco Dorigo, who in 1992 in Doctoral Thesis [3] presented the first algorithm imitating the ant colony organization. This first algorithm, named as Ant System [5, 6], was inspired by the observation of ants community behavior and the question how these almost blind creatures can find the best way from the ant-hill to source of food, how they communicate and what makes them to follow one after another. Answer to these questions is: the pheromones. Pheromone is a chemical substance produced and recognized by the most ant species. Ants are not endowed with intelligence but they leave the pheromone trace in the ground which is the information for the other ants where to go and for the ant itself how to return to the ant-hill. The more ants traverse the trail, the stronger is the pheromone trace. The shorter is the way, the sooner can the ant reach the source of food and return to the ant-hill, which makes the pheromone trace stronger and force the other ants to choose this specific way.

Analogically like in nature phenomena, the role of ants is played by vectors $\mathbf{x}^k$, randomly dispersed in the searching region. In each step, one of the ants is selected as the best one, $\mathbf{x}^{best}$ – the one, for which the minimalized function $F(\mathbf{x})$ takes the lowest value. In the next step, to each vector $\mathbf{x}^k$ is applied a modification, based on the pheromone trail. Vector of each ant is updated at the beginning of each iteration, by using the following formula: $\mathbf{x}^k = \mathbf{x}^{best} + \mathbf{dx}$, where $\mathbf{dx}$ is a vector determining the length of jump, elements of which are randomly generated from the interval $[-\beta, \beta]$ (where $\beta = \beta_0$ is a parameter of narrowing, defined in the initialization of the algorithm). At the end of each iteration the range of ants dislocations is decreasing, according to the formula $\beta_{t+1} = 0.1\beta_t$. This procedure simulates the evaporation of the pheromone trail in nature. Pheromone trails located far from the food source, not attended by natural ants, evaporate. The role of food source in our simulation is played by the point of the lowest function value, that is why the presence of ants – vectors is condensing around this point. The described procedure is iterated until the number of maximum iteration.

We will proceed according the following algorithm.

Initialization of the algorithm.

1. Initial data:
   $F(\mathbf{x})$ – minimized function, $\mathbf{x} = (x_1, \ldots, x_n) \in D$;
   n^M – number of ants in one population;
   I – number of iterations;
   β – narrowing parameter.
2. Random selection of the initial ants localization: $\mathbf{x}^k = (x_1^k, \ldots, x_n^k)$, where $\mathbf{x}^k \in D$, $k = 1, 2, \ldots, n^M$.
3. Determination of the best located ant $\mathbf{x}^{best}$ in the initial ants population.

The main algorithm.

1. Updating of the ants locations:
   – random selection of the vector $\mathbf{dx}$ such that
   $$-\beta_i \leq dx_j \leq \beta_i;$$
   – generation of the new ants population:
   $$\mathbf{x}^k = \mathbf{x}^{best} + \mathbf{dx}, \qquad k = 1, 2, \ldots, n^M.$$
2. Determination of the best located ant $\mathbf{x}^{best}$ in the current ant population.
3. Points 1 and 2 are repeated I^2 times.
4. Narrowing of the ants dislocations range: $\beta_{i+1} = 0.1\beta_i$.
5. Points 1 – 4 are repeated I times.

There are two basic advantages of such approach – the algorithm is effective, even for the not differentiable functions with many local minimums, and time needed for finding the global minimum is respectively short, even for quite complicated functions. The only assumption needed by the algorithm is the existence of solution. If the solution of the optimized problem exists, it will be found, with some given precision of course. Additionally, it is worth to mention that solution received by using the algorithm should be treated as the best solution in the given moment. Running the algorithm again can give different solution, even better. But it does not decrease the effectiveness of the algorithm.

3 Formulation of the Problem

The considered problem is described by the following heat conduction equation

$$c\,\rho\frac{\partial u}{\partial t}(x,t)=\lambda\frac{\partial^2 u}{\partial x^2}(x,t),\qquad x\in[0,d],\ \ t\in[0,T] \tag{1}$$

with the initial and boundary conditions of the form

$$u(x,0)=u_0,\qquad\qquad x\in[0,d], \tag{2}$$

$$\frac{\partial u}{\partial x}(0,t)=0,\qquad\qquad t\in[0,T], \tag{3}$$

where c is the specific heat, ρ denotes the mass density, λ is the thermal conductivity and u, t and x refer to the temperature, time and spatial location. On the boundary for $x=d$ the boundary condition of the third kind is assumed

$$-\lambda\frac{\partial u}{\partial x}(d,t)=\alpha(t)\,(u(d,t)-u_\infty),\qquad t\in[0,T], \tag{4}$$

where u_∞ describes the temperature of environment and $\alpha(t)$ denotes the heat transfer coefficient, form of which we desire to find. We assume that the heat transfer coefficient takes three values α_i, $i=1,2,3$, in the successive cooling zones. Another sought element is the distribution of temperature $u(x,t)$ in the considered region. By setting the value α of heat transfer coefficient the problem, defined by equations (1)–(4), turns into the direct problem, solving of which gives the values of temperature $u_{ij}=u(x_i,t_j)$.

In the approach presented in this paper we propose to solve the direct heat conduction problem, described by equations (1)–(4), by taking the value of heat transfer coefficient as an unknown parameter α. Solution $\tilde{u}_{ij}=\tilde{u}(x_i,t_j)$, received in this way, depends on the parameter α. Next, we determine the value of α by minimizing the following functional:

$$P(\alpha)=\sqrt{\sum_{i=1}^{k}\sum_{j=1}^{m}\big(u(x_i,t_j)-\tilde{u}(x_i,t_j)\big)^2}, \tag{5}$$

representing the differences between the obtained results $\tilde{u}$ and given values u in the measurement points x_i. For minimizing the functional (5) we use the ACO algorithm.

4 Numerical Example

In this section we will illustrate the presented method by an example. We take the following values: $c=1000\,[\mathrm{J/(kg\ K)}]$, $\rho=2679\,[\mathrm{kg/m^3}]$, $\lambda=240\,[\mathrm{W/(m\ K)}]$, $T=1000\,[\mathrm{s}]$, $d=1\,[\mathrm{m}]$, $u_0=980\,[\mathrm{K}]$ and $u_\infty=298\,[\mathrm{K}]$. We need to reconstruct the values of three parameters α_i, $i=1,2,3$, which describe the value of heat transfer coefficient in the successive cooling zones. Exact values of the sought parameters are as follows:

$$\alpha(t) = \begin{cases} 250 & \text{for } t \in [0, 90], \\ 150 & \text{for } t \in [91, 250], \\ 28 & \text{for } t \in [251, 1000]. \end{cases}$$

In calculations we are using the measurement values of temperature located on the boundary for $x = 1$, read in five series: at every 1 s, 2 s, 5 s, 10 s and 20 s. For every series three values of the heat transfer coefficient are reconstructed, one for each cooling zone. We assume that, for every $j = 1, 2, 3$, the initial population of ants locating the best localization of the sought parameters is randomly selected from the range $[0, 500]$. We evaluate the experiment for number of ants $n^M = 2 \div 5$ and number of iterations $I = 1 \div 10$. Value of the narrowing parameter is $\beta = 0.1$ and the initial $\beta_0 = 300$. Because of the limited size of the paper we will present only the results for $n^M = 5$ and for two frontier series, at every 1 s, and 20 s. The approximate values of reconstructed parameters are received by running the algorithm 30 times and by averaging the obtained results.

Figures 1 and 2 present the relative error δ_{α_i} of reconstructed values of coefficients α_i, $i = 1, 2, 3$, in dependence on the number of iterations I. It can be seen that even for measurement values with the time step of 20 s error of reconstruction is satisfactorily small for 3 iterations.

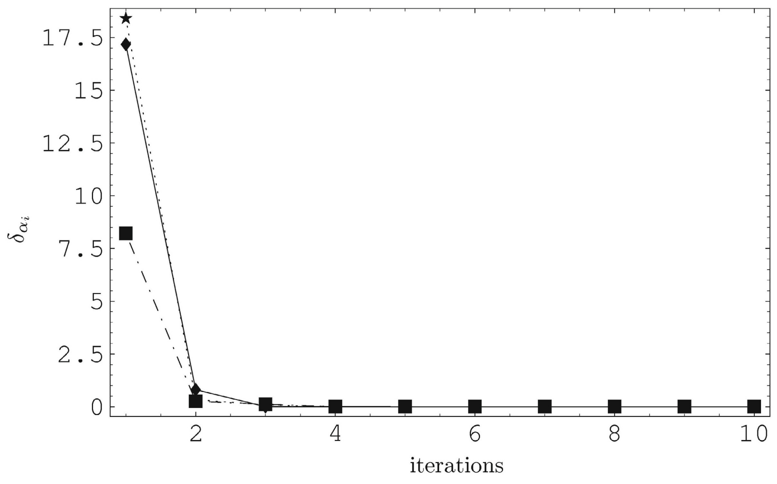


Fig. 1. Relative error of reconstruction of coefficients α_i for the successive iterations (♦ – for α_1, ★ – for α_2, ■ – for α_3) – series of the control points with the step 1 s, number of ants $n^M = 5$

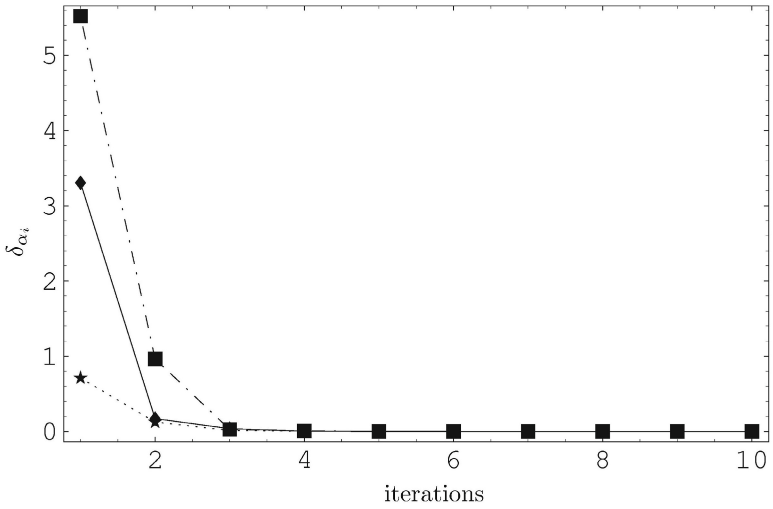


Fig. 2. Relative error of reconstruction of coefficients α_i for the successive iterations (♦ – for α_1, ★ – for α_2, ■ – for α_3) – series of the control points with the step 20 s, number of ants $n^M = 5$

Besides the heat transfer coefficients, another unknown element desired for determining is the distribution of temperature $u(x,t)$. For estimating the quality of reconstructed values of state function $\widetilde{u}(1,t_j)$ in points where the exact values $u(1,t)$ are known, we calculate the absolute and relative errors of this reconstruction. The relative errors δ_u for series of control points at every 1 s and at every 20 s, in dependence on the number of iterations I, are displayed in Figure 3. Again, we can notice that in both cases 3 iterations are enough to receive estimated values of temperature distribution with small errors.

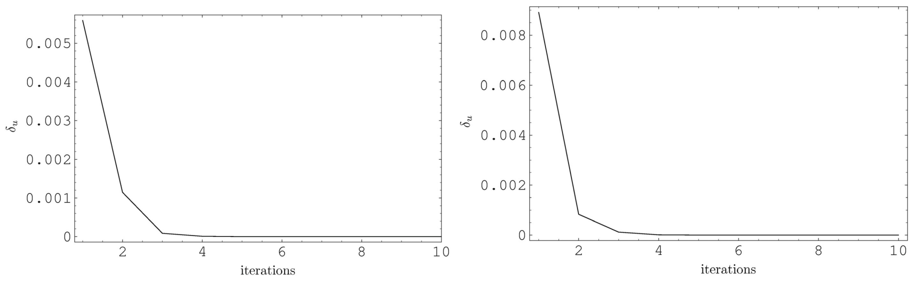


Fig. 3. Relative errors of reconstruction of the state function in control points for the successive cycles – number of ants $n^M = 5$, series with the step 1 s (left figure) and with the step 20 s (right figure)

Figure 4 shows the comparison of exact values of state function $u(x, t)$ on the boundary for $x = 1$ with the reconstructed values calculated for the series with time step 1 s and for 1 and 10 iterations, respectively. The same comparison, but for the time step 20 s is displayed in Figure 5.

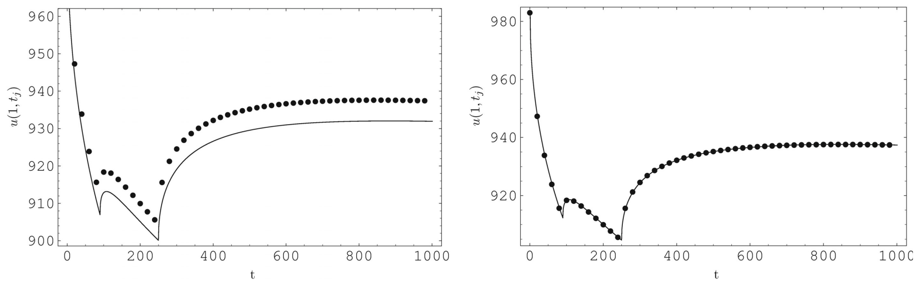


Fig. 4. Comparison of the given $u(1, t_j)$ (dashed line) and reconstructed (solid line) values of the state function received for 1 iteration (left figure) and for 10 iterations (right figure), for the series with the step 1 s and number of ants $n^M = 5$

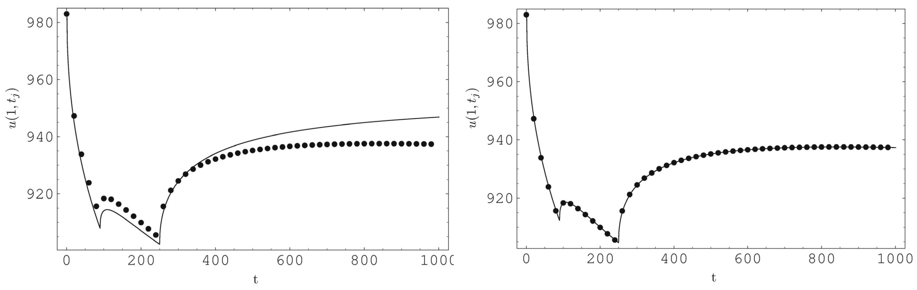


Fig. 5. Comparison of the given $u(1, t_j)$ (dashed line) and reconstructed (solid line) values of the state function received for 1 iteration (left figure) and for 10 iterations (right figure), for the series with the step 20 s and number of ants $n^M = 5$

Finally, in Tables 1 and 2 there are compiled the errors of reconstruction of the coefficients α_i and values of the state function in given control points received in the successive iterations for the number of ants $n^M = 5$ and for the series with step 1 s and 20 s.

Results compiled in tables and presented in figures indicate that the proposed procedure ensures the approximate solution rapidly convergent to exact solution. Moreover, for the exact input data we received in few iterations the results almost equal to the exact solution. Using the analytical method for that kind of problem is almost impossible, therefore the approximate method giving so good solution after so short time of working is very needed. There are also many other heuristic algorithms, like simulated annealing, tabu search, memetic algorithms, other swarm intelligence algorithms etc. Authors of the current paper have already applied the Artificial Bee Colony algorithm [23] and Harmony Search algorithm [24] in solving the inverse heat conduction problem with boundary condition of the third kind. Comparison of those methods shows that all the applied heuristic algorithms give satisfying results, however the ACO algorithm is better convergent, especially in comparison with HS algorithm.

Table 1. Errors of reconstruction of the coefficients α_i and values of the state function received in the successive iterations for $n^M = 5$ and for series with step 1 s

I	$\max \delta_{\alpha_i}\,[\%]$	$\Delta_u\,[\mathrm{K}]$	$\delta_u\,[\%]$
1	18.4100	170.0130	0.0056
2	0.8078	34.8640	0.0012
3	0.1187	2.6202	0.0001
4	0.0074	0.2968	$9.76 \cdot 10^{-6}$
5	0.0010	0.0113	$3.76 \cdot 10^{-7}$
6	$3.63 \cdot 10^{-5}$	0.0015	$5.03 \cdot 10^{-8}$
7	$6.85 \cdot 10^{-6}$	0.0004	$1.18 \cdot 10^{-8}$
8	$4.89 \cdot 10^{-7}$	$1.69 \cdot 10^{-5}$	$5.57 \cdot 10^{-10}$
9	$6.88 \cdot 10^{-8}$	$3.15 \cdot 10^{-6}$	$1.04 \cdot 10^{-10}$
10	$3.59 \cdot 10^{-9}$	$2.40 \cdot 10^{-7}$	$7.90 \cdot 10^{-12}$

Table 2. Errors of reconstruction of the coefficients α_i and values of the state function received in the successive iterations for $n^M = 5$ and for series with step 20 s

I	$\max \delta_{\alpha_i}\,[\%]$	$\Delta_u\,[\mathrm{K}]$	$\delta_u\,[\%]$
1	5.52192	167.7300	0.0055
2	0.9634	28.2307	0.0009
3	0.0362	1.0032	$3.30 \cdot 10^{-5}$
4	0.0061	0.2178	$7.17 \cdot 10^{-6}$
5	0.0004	0.0196	$6.44 \cdot 10^{-7}$
6	$6.90 \cdot 10^{-5}$	0.0016	$5.33 \cdot 10^{-8}$
7	$9.32 \cdot 10^{-6}$	0.0003	$9.73 \cdot 10^{-9}$
8	$2.92 \cdot 10^{-7}$	$6.50 \cdot 10^{-5}$	$2.13 \cdot 10^{-11}$
9	$6.77 \cdot 10^{-8}$	$1.51 \cdot 10^{-6}$	$4.97 \cdot 10^{-11}$
10	$2.75 \cdot 10^{-9}$	$2.83 \cdot 10^{-7}$	$9.41 \cdot 10^{-12}$

5 Conclusions

In the current paper we have proposed the method of determining the heat transfer coefficient appearing in the boundary condition of the third kind, a crucial part of which consists in minimization of the proper functional. For this part of the procedure we have applied the Ant Colony Optimization algorithm, which turned out to be an efficient idea. Results received for the example in which we reconstruct three values of the unknown coefficients in three successive cooling zones are satisfying for five series of control points with various step, for various number of ants in the algorithm and for various number of iterations. Obtained results show that the similarly good reconstructions of the unknown coefficients and of the state function values can be received for the dense series of measurement points (at every 1 s) as well as for the rare series (at every 20 s), for relatively small number of ants (not exceeding 5) and for relatively small number of iterations (not exceeding 10).

Additionally, it is worth to mention that an indisputable advantage of the ACO algorithm is, apart from the effectiveness and relative simplicity, its universality. The only assumption needed by this algorithm is the existence of solution.

In future we intend to parallelize the computations in ACO algorithm and to compare the efficiency of ACO algorithm with other, not investigated yet in this field, heuristic approaches.

References

1. Beni, G., Wang, J.: Swarm intelligence in cellular robotic systems. In: Proceed. NATO Advanced Workshop on Robots and Biological Syst., Tuscany (1989)
2. Eberhart, R.C., Shi, Y., Kennedy, J.: Swarm Intelligence. Morgan Kaufmann, San Francisco (2001)
3. Dorigo, M.: Optimization, Learning and Natural Algorithms (in Italian). PhD thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milan (1992)
4. Dorigo, M., Stützle, T.: Ant Colony Optimization. Massachusetts Institute of Technology Press, Cambridge (2004)
5. Dorigo, M., Maniezzo, V., Colorni, A.: The ant system: Optimization by a colony of cooperatin gagents. IEEE Transactionson Systems, Man and Cybernetics – Part B 26(1), 29–41 (1996)
6. Maniezzo, V., Colorni, A., Dorigo, M.: The ant system applied to the quadratic assignment problem. Technical Report IRIDIA, Universite Libre de Bruxelles, 94–128 (1994)
7. Gambardella, L.M., Taillard, E., Agazzi, G.: A Multiple Colony System for Vehicle Routing Problems with Time Windows. Technical Report IDSIA, IDSIA-06-99, Lugano (1999)
8. Schoonderwoerd, R., Holland, O., Bruten, J., Rothkrantz, L.: Ant-based load balancing in telecommunications networks. Adaptive Behavior 5(2), 169–207 (1996)
9. DiCaro, G., Dorigo, M.: AntNet: A mobile agents approach to adaptive routing. Technical Report, IRIDIA, Universite Libre de Bruxelles, 97–112 (1997)
10. Socha, K., Dorigo, M.: Ant colony optimization for continuous domains. Eur. J. Oper. Res. 185, 1155–1173 (2008)
11. Korošec, P., Šilc, J., Filipič, B.: The differential ant-stigmergy algorithm. Inform. Sciences 181 (2011) (to appear)
12. Hetmaniok, E., Zielonka, A.: Solving the inverse heat conduction problem by using the ant colony optimization algorithm. In: CMM 2009, pp. 205–206. University of Zielona Góra Press (2009)

13. Hetmaniok, E., Słota, D., Zielonka, A.: Solution of the Inverse Heat Conduction Problem by Using the ABC Algorithm. In: Szczuka, M., Kryszkiewicz, M., Ramanna, S., Jensen, R., Hu, Q. (eds.) RSCTC 2010. LNCS, vol. 6086, pp. 659–668. Springer, Heidelberg (2010)
14. Beck, J.V., Blackwell, B., St. Clair, C.R.: Inverse Heat Conduction: Ill Posed Problems. Wiley Intersc., New York (1985)
15. Beck, J.V., Blackwell, B.: Inverse Problems. Handbook of Numerical Heat Transfer. Wiley Intersc., New York (1988)
16. Beck, J.V., Cole, K.D., Haji-Sheikh, A., Litkouhi, B.: Heat Conduction Using Green's Functions. Hempisphere Publishing Corporation, Philadelphia (1992)
17. Haji-Sheikh, A., Buckingham, F.P.: Multidimensional inverse heat conduction using the Monte Carlo method. Trans. of the ASME, J. Heat Trans. 115, 26–33 (1993)
18. Mourio, D.A.: The Mollification Method and the Numerical Solution of Ill-posed Problems. Wiley and Sons, New York (1993)
19. Qiu, C.Y., Fu, C.L., Zhu, Y.B.: Wavelets and regularization of the sideways heat equation. Comput. Math. Appl. 46, 821–829 (2003)
20. Słota, D.: Identification of the Cooling Condition in 2-D and 3-D Continuous Casting Processes. Numer. Heat Transfer B 55, 155–176 (2009)
21. Słota, D.: Solving the inverse Stefan design problem using genetic algorithm. Inverse Probl. Sci. Eng. 16, 829–846 (2008)
22. Słota, D.: Restoring Boundary Conditions in the Solidification of Pure Metals. Comput. & Structures 89, 48–54 (2011)
23. Zielonka, A., Hetmaniok, E., Słota, D.: Using the Artificial Bee Colony Algorithm for Determining the Heat Transfer Coefficient. In: Czachórski, T., Kozielski, S., Stańczyk, U. (eds.) Man-Machine Interactions 2. AISC, vol. 103, pp. 369–376. Springer, Heidelberg (2011)
24. Hetmaniok, E., Jama, D., Słota, D., Zielonka, A.: Application of the Harmony Search algorithm in solving the inverse heat conduction problem, Zeszyty Nauk. Pol. Sl. Mat. Stos. 1, 99–108 (2011)

Learning in a Non-stationary Environment Using the Recursive Least Squares Method and Orthogonal-Series Type Regression Neural Network

Maciej Jaworski[1] and Meng Joo Er[2]

[1] Department of Computer Engineering, Czestochowa University of Technology, Armii Krajowej 36, 42-200 Czestochowa, Poland
maciej.jaworski@kik.pcz.pl

[2] School of Electrical and Electronic Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore
emjer@ntu.edu.sg

Abstract. In the paper the recursive least squares method, in combining with general regression neural network, is applied for learning in a non-stationary environment. The orthogonal series-type kernel is applied to design the general regression neural networks. Sufficient conditions for convergence in probability are given and simulation results are presented.

1 Introduction

A plenty of real-life systems has a time-varying nature. Examples can be found in geophysics or biomedicine. A group of such systems can be modeled with the use of nonlinear regression. Among the methods to perform nonlinear regression, nonparametric ones seem to be the most useful tools. They need no knowledge about the probability distribution of incoming data, henceforth they are applicable to the wide variety of problems. In literature, a variety of nonparametric techniques have been developed to solve stationary (see e.g. [4], [5], [6], [10]-[13], [19]-[21] and [24]-[27]) and non-stationary problem ([7], [14]-[18], [22] and [23]), with the noise assumed to be stationary. In this paper systems described by the following equation are investigated

$$Y_i = \phi(X_i) + ac_i + Z_i,\ i = 1, \dots, n, \tag{1}$$

where $\phi(x)$ is an unknown regression function, $X_i \in A \subset \mathbb{R}^{\prime}$ are i.i.d. input random variables with some unknown probability density function $f(x)$, Y_i are the output random variables, c_i is a known sequence and the constant a is unknown. Random variables Z_i introduce a statistic perturbation to the system and satisfy the following conditions

$$\forall_{i\in\{1,\dots,n\}},\ \ E[Z_i] = 0,\ E[Z_i^2] = d_i, \tag{2}$$

In the next section the algorithm for estimating the constant a and the function $\phi(x)$ will be presented. In section 3 the conditions will be formulated, under which the algorithm is convergent. Section 4 presents some simulation results. Conclusions are drawn in section 5.

2 Algorithm

In [1] the estimator $\hat{a}_i$ for parameter a was presented, based on the recursive least squares method

$$\hat{a}_i = \hat{a}_{i-1} + \frac{c_i}{\sum_{j=1}^{i} c_j^2} \left(Y_i - \hat{a}_{i-1} c_i\right),\ i \in \{1, \ldots, n\},\ \hat{a}_0 = 0. \tag{3}$$

This method can be generalized to the following form

$$\hat{a}_i^{(\omega)} = \hat{a}_{i-1}^{(\omega)} + \frac{c_i^{\omega-1}}{\sum_{j=1}^{i} c_j^{\omega}} \left(Y_i - \hat{a}_{i-1}^{(\omega)} c_i\right), \tag{4}$$

where ω is a nonnegative real number. For example, in this notation the estimator (3) is denoted by $\hat{a}_i^{(2)}$. Estimator $\hat{a}_n^{(\omega)}$ is calculated after n steps, with the use of variables $Y_1, \ldots, Y_n$.

In the next step, the regression function $\phi(x)$ is estimated, under assumption that $a = \hat{a}_n^{(\omega)}$. Function $\phi(x)$, in each point x at which $f(x) \neq 0$, can be expressed as follows

$$\phi(x) = \frac{\phi(x) f(x)}{f(x)}. \tag{5}$$

Nominator (further denoted by $R(x) = \phi(x) f(x)$) and denominator of the above formula are estimated separately. Therefore, the estimator $\hat{\phi}_n(x, \hat{a}_n^{(\omega)})$ of function $\phi(x)$ is given by

$$\hat{\phi}_n(x, \hat{a}_n^{(\omega)}) = \frac{\hat{R}_n\left(x, \hat{a}_n^{(\omega)}\right)}{\hat{f}_n(x)}. \tag{6}$$

The estimators can be established with the use of nonparametric method based on kernel functions $K_n : A \times A \to \mathbb{R}$ and $K_n' : A \times A \to \mathbb{R}$. Given n pairs of random variables $(X_1, Y_1), \ldots, (X_n, Y_n)$, kernel functions $K_n(x, u)$ and $K_n'(x, u)$, the following estimators for $R(x)$ and $f(x)$ are proposed

$$\hat{R}_n(x, \hat{a}_n^{(\omega)}) = \frac{1}{n} \sum_{i=1}^{n} \left(Y_i - \hat{a}_n^{(\omega)} c_i\right) K_n(x, X_i), \tag{7}$$

$$\hat{f}_n(x) = \frac{1}{n} \sum_{i=1}^{n} K_n'(x, X_i), \tag{8}$$

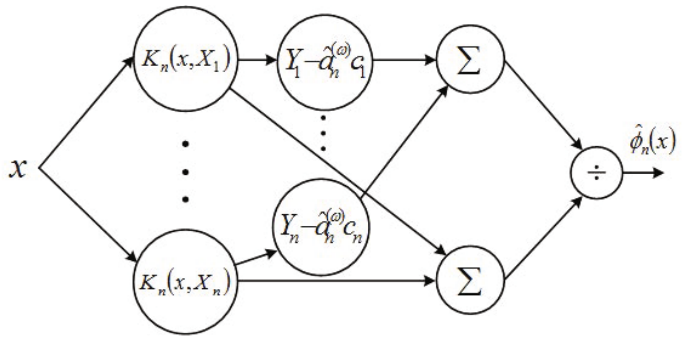


Fig. 1. General regression neural network for the estimator $\hat{\phi}_n(x, \hat{a}_n^{(\omega)})$ of the regression function $\phi(x)$

where in particular $K_n(x,u)$ and $K'_n(x,u)$ can be the same. The algorithm described by equations (6) - (8) is the general regression neural network [32], the scheme of which is presented in Fig. 1.

Let $g_j : A \to \mathbb{R}$, $j \in \mathbb{N}$, be the orthogonal series, satisfying $\sup_{w\in A} |g_j(w)| \leq G_j$, $\forall_{j\in\mathbb{N}}$. The kernel functions $K_n(x,u)$ and $K'_n(x,u)$ in estimators (7) and (8) can be proposed in the forms

$$K_n(x,u) = \sum_{j=0}^{M(n)} g_j(x)g_j(u), \tag{9}$$

$$K'_n(x,u) = \sum_{j=0}^{N(n)} g_j(x)g_j(u), \tag{10}$$

where $M(n)$ and $N(n)$ are some monotonically increasing sequences of n, satisfying $\lim_{n\to\infty} M(n) = \infty$ and $\lim_{n\to\infty} N(n) = \infty$, respectively.

3 Convergence of the Algorithm

To ensure the proper performance of the above algorithm, functions $M(n)$ and $N(n)$ and sequences c_i and d_i must satisfy the assumptions of the following theorem. Let us denote

$$s_i = \max\{E[\phi^2(X_i)]), d_i\}. \tag{11}$$

Theorem 1. *If the following conditions hold*

$$\lim_{n\to\infty} \left[\frac{\sum_{i=1}^{n} c_i^{\omega-1}}{\sum_{i=1}^{n} c_i^{\omega}}\right] = 0, \tag{12}$$

$$\lim_{n\to\infty} \left[\frac{\sum_{i=1}^{n} c_i^{2\omega-2} s_i}{\left(\sum_{i=1}^{n} c_i^{\omega}\right)^2}\right] = 0, \tag{13}$$

$$\lim_{n\to\infty}\left[\frac{1}{n^2}\left(\sum_{j=0}^{M(n)} G_j^2\right)^2 \sum_{i=1}^{n} s_i\right] = 0, \tag{14}$$

$$\lim_{n\to\infty}\left[\frac{1}{n}\left(\sum_{j=0}^{N(n)} G_j^2\right)^2\right] = 0, \tag{15}$$

then

$$\hat{a}_n^{(\omega)} \overset{n\to\infty}{\longrightarrow} a \quad \text{in probability} \tag{16}$$

and

$$\hat{\phi}_n(x, \hat{a}_n^{(\omega)}) \overset{n\to\infty}{\longrightarrow} \phi(x) \quad \text{in probability} \tag{17}$$

at each point x, at which the orthogonal expansions of functions $f(x)$ and $R(x)$ are convergent.

Proof. Convergence (16) can be proven under the simple analysis of the variance and bias of estmator $\hat{a}_n^{(\omega)}$. Convergence (17) arirsed from the convergence of estimator (6) [11] and from Theorem 4.3.8 in [36].

4 Upgrading Procedure for Estimator $\hat{a}_n^{(\omega)}$

The bias $b_n^{(\omega)}$ of estimator $\hat{a}_n^{(\omega)}$ is of the form

$$b_n^{(\omega)} = E[\phi(X_i)]\frac{\sum_{j=1}^{n} c_j^{\omega-1}}{\sum_{j=1}^{n} c_j^{\omega}}. \tag{18}$$

Since $\lim_{n\to\infty} c_n = \infty$, the bias $b_n^{(\omega)}$ tends to 0. In some special cases of sequence c_n, the speed of bias convergence is known. For example, if $c_i = i^t, i \in \mathbb{N}, t > 0$, the trend is given by

$$b_n^{(\omega)} \sim n^{-t} = \frac{1}{c_n}. \tag{19}$$

Therefore, the estimator $\hat{a}_n^{(\omega)}$ can be expressed in the form

$$\hat{a}_n^{(\omega)} = a + b_n^{\omega} = a + B\frac{1}{c_n}, \tag{20}$$

where B is some unknown constant, unnecessary for further considerations. Having the values of $(n - n_0 + 1)$ estimators $\hat{a}_{n_0}^{(\omega)}, \ldots, \hat{a}_n^{(\omega)}$ one can calculate the upgraded version of estimator for parameter a, using the linear regression procedure

$$\tilde{a}_n^{(\omega)} = \frac{\sum_{i=n_0}^{n} \frac{\hat{a}_i^{(\omega)}}{c_i} \sum_{i=n_0}^{n} \frac{1}{c_i} - \sum_{i=n_0}^{n} \hat{a}_i^{(\omega)} \sum_{i=n_0}^{n} \frac{1}{c_i^2}}{\sum_{i=n_0}^{n} \frac{1}{c_i} \sum_{i=n_0}^{n} \frac{1}{c_i} - (n - n_0 + 1) \sum_{i=n_0}^{n} \frac{1}{c_i^2}}. \quad (21)$$

The algorithm presented in section 2 can be applied with the upgraded estimator $\tilde{a}_n^{(\omega)}$. Then, in formulas (6) and (7), the estimator $\hat{a}_n^{(\omega)}$ has to be replaced by the estimator $\tilde{a}_n^{(\omega)}$.

5 Simulations

In the following simulations data X_i are generated from the exponential distribution with probability density function given by

$$f(x) = e^{-x},\ x \in (0; \infty). \quad (22)$$

Random variables Z_i come from the normal distribution $N(0, d_i)$. Sequence of variances d_i is given in the form

$$d_i = i^{\alpha},\ i \in \{1, \ldots, n\},\ \alpha > 0. \quad (23)$$

The output data Y_i are calculated using formula (1), where sequence c_i is given as follows

$$c_i = i^{t},\ i \in \{1, \ldots, n\},\ t > 0. \quad (24)$$

The Laguerre orthogonal series is taken for the $g_j(x)$ functions

$$g_j(x) = \begin{cases} \exp\left(-\frac{x}{2}\right), & j = 0, \\ g_0(x)(1 - x), & j = 1, \\ \frac{1}{j}\left[(2j - 1 - x)g_{j-1}(x) - (j - 1)g_{j-2}(x)\right], & j > 1 \end{cases}, x \in (0; \infty). \quad (25)$$

Each function $g_j(x)$ is bounded by (see [35])

$$\forall_{x \in (0;\infty)} \forall_{j \in \mathbb{N}} |g_j(x)| \leq G_j = Cj^{-1/4}. \quad (26)$$

It is additionally assumed that the sequences $N(n)$ and $M(n)$ take the following forms

$$M(n) = \lceil Dn^{Q} \rceil,\ N(n) = \lceil D'n^{Q'} \rceil,\ Q, Q' > 0,\ D, D' > 0. \quad (27)$$

To satisfy the assumptions (12), (13), (14) and (15) of Theorem 1, the exponents t, α, Q and Q' should obey the following inequalities

$$Q' < 1,\ Q + \alpha < 1,\ \alpha < 2t + 1. \quad (28)$$

In the presented simulations the parameters are set to $t = 0,25$ and $D = D' = 2$. Exponents Q and Q' are equal in all simulations. The parameter ω in estimator (4) (and (21)) is set to 4. The regression function $\phi(x)$ to be estimated is given in the form

$$\phi(x) = 5\cos(5x)\exp\left(-\frac{x^2}{2}\right) + 3. \qquad (29)$$

The value of constant a is set to 3.

In Figure 2 the estimators $\hat{a}_n^{(4)}$ and $\tilde{a}_n^{(4)}$ are presented as a function of number of data elements n (for $\alpha = 0,2$).

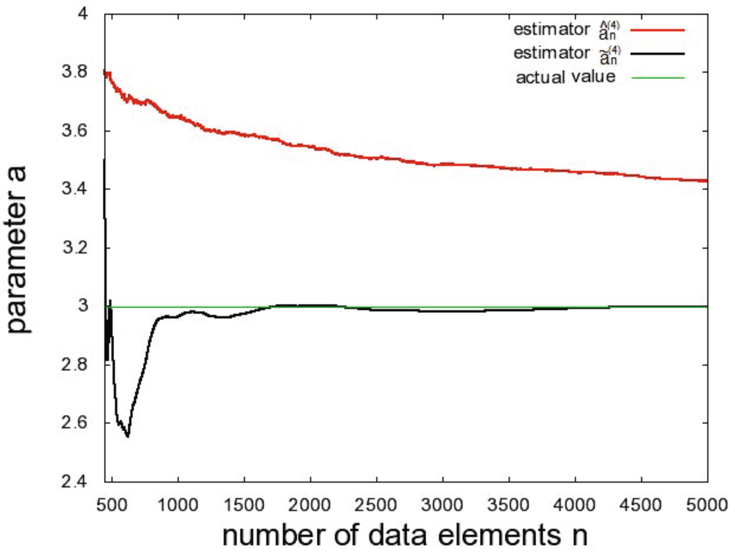


Fig. 2. The estimators $\hat{a}_n^{(4)}$ and $\tilde{a}_n^{(4)}$ as a function of number of data elements n, $\alpha = 0,2$

In considered range of number n, the estimator $\hat{a}_n^{(4)}$ is far away from the real value of parameter a. The upgraded estimator $\tilde{a}_n^{(4)}$ converges dramatically faster. The influence of the estimators on the estimation of the regression function $\phi(x)$ is shown in Fig. 3. Estimators $\hat{R}_n(x, \hat{a}_n^{(4)})$, $\hat{R}_n(x, \tilde{a}_n^{(4)})$ and $\hat{f}_n(x)$ are calculated with $Q = Q' = 0,4$ and $Q = Q' = 0,5$.

For the estimator $\hat{a}_n^{(4)}$, the Mean Squared Error (MSE) cannot drop down below some threshold value around 8. For $\tilde{a}_n^{(4)}$ the quality of estimation is much more better. The presented figures prove that the estimator $\tilde{a}_n^{(4)}$ leads to the significantly better results than the estimator $\hat{a}_n$ does.

The estimator $\hat{\phi}_n(x, \tilde{a}_n)$ differs more from the regression function $\phi(x)$ if the coefficient α takes higher values. The MSE values as a function of n, for α equal to $0,2$, $0,5$ and $0,7$, is presented in Fig. 4. The parameters $Q = Q'$ are set to $0,5$.

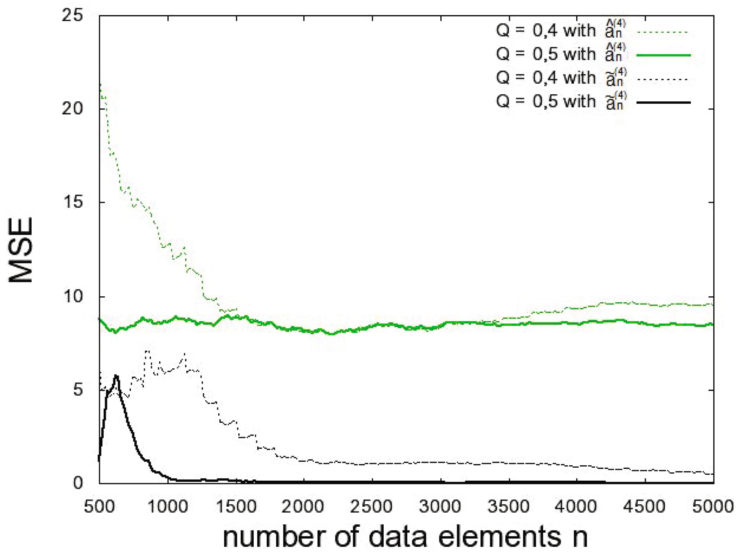


Fig. 3. The Mean Squared Error as a function of number of data elements n, for estimators $\hat{a}_n^{(4)}$ and $\tilde{a}_n^{(4)}$, $Q = Q' = 0,4$ and $Q = Q' = 0,5$, $\alpha = 0,2$

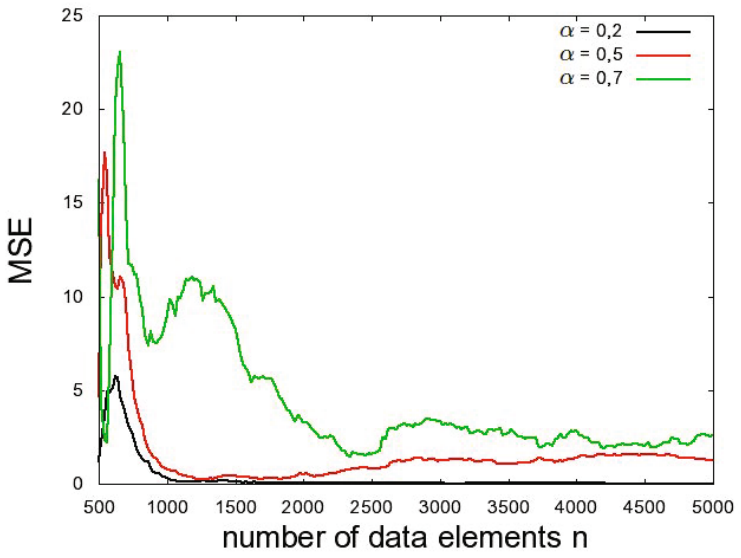


Fig. 4. The Mean Squared Error as a function of number of data elements n, for estimator $\tilde{a}_n^{(4)}$, $Q = Q' = 0,5$ and three different values of α: $0,2$, $0,4$ and $0,5$

For $\alpha = 0,5$ and $\alpha = 0,7$ inequalities (28) do not hold. Therefore, the MSE cannot converge to 0, as it is in the case of $\alpha = 0,2$.

To present the final result of the function $\phi(x)$ estimation, the estimators $\hat{\phi}_n(x, \hat{a}_n)$ and $\hat{\phi}_n(x, \tilde{a}_n)$ are compared with function (29). The results obtained for $Q = Q' = 0,5$, $\alpha = 0,2$ and $n = 5000$ are depicted in Fig. 5.

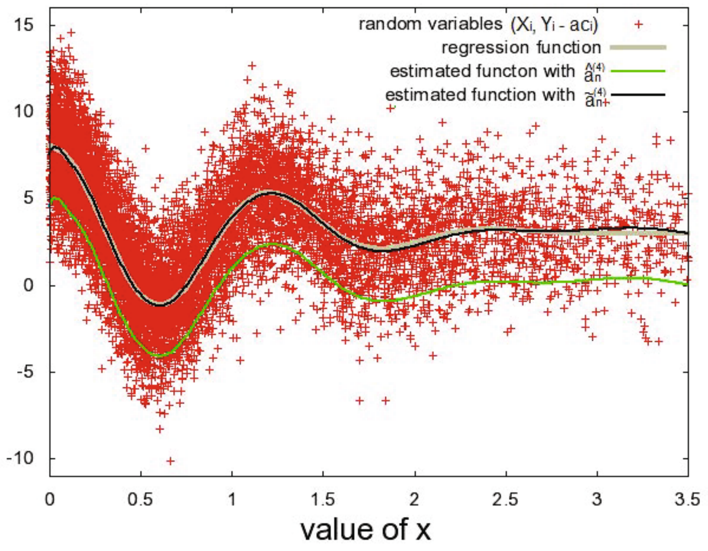


Fig. 5. The comparison of estimators $\hat{\phi}_n(x, \hat{a}_n)$ and $\hat{\phi}_n(x, \tilde{a}_n)$ with the regression function $\phi(x)$, for $\alpha = 0,2$, $Q = Q' = 0,5$, $n = 5000$. Points denote the input-output random variables in the form $(X_i, Y_i - ac_i)$.

6 Final Remarks

In the paper we applied recursive least squares method, in combining with general regression neural network, for learning in a non-stationary environment. The orthogonal series-type kernel was applied to design the general regression neural networks. Sufficient conditions for convergence in probability were given and simulation results were presented. Further work can be concentrated on handling of time-varying noise by making use of supervised and unsupervised neural networks [2], [3], [9] and neurofuzzy structures developed in [8], [28]-[31], [31], [33] and [34].

Acknowledgments. The paper was prepared under project operated within the Foundation for Polish Science Team Programme co-financed by the EU European Regional Development Fund, Operational Program Innovative Economy 2007-2013, and also supported by National Science Centre NCN.

References

1. Albert, A.E., Gardner, L.A.: Stochastic Approximation and Nonlinear Regression, vol. (42). MIT Press, Cambridge (1967)
2. Bilski, J., Rutkowski, L.: A fast training algorithm for neural networks. IEEE Transactions on Circuits and Systems II 45, 749–753 (1998)
3. Cierniak, R., Rutkowski, L.: On image compression by competitive neural networks and optimal linear predictors. Signal Processing: Image Communication - a Eurasip Journal 15(6), 559–565 (2000)
4. Gałkowski, T., Rutkowski, L.: Nonparametric recovery of multivariate functions with applications to system identification. Proceedings of the IEEE 73, 942–943 (1985)
5. Gałkowski, T., Rutkowski, L.: Nonparametric fitting of multivariable functions. IEEE Transactions on Automatic Control AC-31, 785–787 (1986)
6. Greblicki, W., Rutkowska, D., Rutkowski, L.: An orthogonal series estimate of time-varying regression. Annals of the Institute of Statistical Mathematics 35, Part A, 147–160 (1983)
7. Greblicki, W., Rutkowski, L.: Density-free Bayes risk consistency of nonparametric pattern recognition procedures. Proceedings of the IEEE 69(4), 482–483 (1981)
8. Nowicki, R.: Rough Sets in the Neuro-Fuzzy Architectures Based on Monotonic Fuzzy Implications. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 510–517. Springer, Heidelberg (2004)
9. Patan, K., Patan, M.: Optimal Training Strategies for Locally Recurrent Neural Networks. Journal of Artificial Intelligence and Soft Computing Research 1(2), 103–114 (2011)
10. Rutkowski, L.: Sequential estimates of probability densities by orthogonal series and their application in pattern classification. IEEE Transactions on Systems, Man, and Cybernetics SMC-10(12), 918–920 (1980)
11. Rutkowski, L.: Sequential estimates of a regression function by orthogonal series with applications in discrimination, New York, Heidelberg, Berlin. Lectures Notes in Statistics, vol. 8, pp. 236–244 (1981)
12. Rutkowski, L.: On system identification by nonparametric function fitting. IEEE Transactions on Automatic Control AC-27, 225–227 (1982)
13. Rutkowski, L.: Orthogonal series estimates of a regression function with applications in system identification. In: Probability and Statistical Inference, pp. 343–347. D. Reidel Publishing Company, Dordrecht (1982)
14. Rutkowski, L.: On Bayes risk consistent pattern recognition procedures in a quasi-stationary environment. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4(1), 84–87 (1982)
15. Rutkowski, L.: On-line identification of time-varying systems by nonparametric techniques. IEEE Transactions on Automatic Control AC-27, 228–230 (1982)
16. Rutkowski, L.: On nonparametric identification with prediction of time-varying systems. IEEE Transactions on Automatic Control AC-29, 58–60 (1984)
17. Rutkowski, L.: Nonparametric identification of quasi-stationary systems. Systems and Control Letters 6, 33–35 (1985)
18. Rutkowski, L.: The real-time identification of time-varying systems by nonparametric algorithms based on the Parzen kernels. International Journal of Systems Science 16, 1123–1130 (1985)

19. Rutkowski, L.: A general approach for nonparametric fitting of functions and their derivatives with applications to linear circuits identification. IEEE Transactions Circuits Systems CAS-33, 812–818 (1986)
20. Rutkowski, L.: Sequential pattern recognition procedures derived from multiple Fourier series. Pattern Recognition Letters 8, 213–216 (1988)
21. Rutkowski, L.: Nonparametric procedures for identification and control of linear dynamic systems. In: Proceedings of 1988 American Control Conference, June 15-17, pp. 1325–1326 (1988)
22. Rutkowski, L.: An application of multiple Fourier series to identification of multivariable nonstationary systems. International Journal of Systems Science 20(10), 1993–2002 (1989)
23. Rutkowski, L.: Nonparametric learning algorithms in the time-varying environments. Signal Processing 18, 129–137 (1989)
24. Rutkowski, L., Rafajłowicz, E.: On global rate of convergence of some nonparametric identification procedures. IEEE Transaction on Automatic Control AC-34(10), 1089–1091 (1989)
25. Rutkowski, L.: Identification of MISO nonlinear regressions in the presence of a wide class of disturbances. IEEE Transactions on Information Theory IT-37, 214–216 (1991)
26. Rutkowski, L.: Multiple Fourier series procedures for extraction of nonlinear regressions from noisy data. IEEE Transactions on Signal Processing 41(10), 3062–3065 (1993)
27. Rutkowski, L., Gałkowski, T.: On pattern classification and system identification by probabilistic neural networks. Applied Mathematics and Computer Science 4(3), 413–422 (1994)
28. Rutkowski, L.: A New Method for System Modelling and Pattern Classification. Bulletin of the Polish Academy of Sciences 52(1), 11–24 (2004)
29. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, Melbourne, December 2-5, vol. 3, pp. 1428–1431 (2001)
30. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)
31. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)
32. Specht, D.F.: A general regression neural network. IEEE Transactions on Neural Networks 2, 568–576 (1991)
33. Starczewski, J., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
34. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)
35. Szegö, G.: Orthogonal Polynomials, vol. 23. American Mathematical Society Coll. Publ.(1959)
36. Wilks, S.S.: Mathematical Statistics. John Wiley, New York (1962)

On the Application of the Parzen-Type Kernel Probabilistic Neural Network and Recursive Least Squares Method for Learning in a Time-Varying Environment

Maciej Jaworski[1] and Yoichi Hayashi[2]

[1] Department of Computer Engineering, Czestochowa University of Technology, Armii Krajowej 36, 42-200 Czestochowa, Poland
maciej.jaworski@kik.pcz.pl

[2] Department of Computer Science, Meiji University, Tama-ku, Kawasaki 214-8571, Japan
hayashiy@cs.meiji.ac.jp

Abstract. This paper presents the Parzen kernel-type regression neural network in combination with recursive least squares method to solve problem of learning in a time-varying environment. Sufficient conditions for convergence in probability are given. Simulation experiments are presented and discussed.

1 Introduction

A concept of probabilistic neural networks and general regression neural networks was first proposed by Specht [36], [37]. These structures are useful nonparametric tools for providing nonlinear regression in a stationary (see e.g. [4], [6], [7], [9], [12], [14]-[17], [23]-[25] and [28]-[31]) and nonstationary environment (see [8], [18]-[22], [26] and [27]). Nonlinear regresion can be applied, for example, in the analysis of multiple input-single output (MISO) systems. Such systems are associated with a wide range of biomedical, economic and engineering problems. Let us consider a MISO system, described by the following equation

$$Y_i = \phi(X_i) + ac_i + Z_i. \tag{1}$$

Pairs (X_i, Y_i), $X_i \in A \subset \mathbb{R}^p$, $Y_i \in \mathbb{R}$, $i \in \{1, \ldots, n\}$, where X_i are independent and identically distributed random variables. The probability density function $f(x)$ of variables X_i is not known a priori. The zero-mean random variables Z_i represent the noise and the variance of Z_i is bounded

$$Var(Z_i) = E[Z_i^2] = \sigma_z^2 < \infty,\ i \in \{1, \ldots, n\}. \tag{2}$$

Elements of deterministic, monotonically increasing sequence c_i ($\lim_{i\to\infty} |c_i| = \infty$) are known. In this paper we present the algorithm, which allows to estimate the parameter a and the regression function $\phi(x)$.

2 Algorithm for Learning of Parameter *a*

To estimate the value of constant a, the recursive least squares method will be applied [2]. The estimator $\hat{a}_n$ is given in the form

$$\hat{a}_n = \hat{a}_{n-1} + \frac{c_n}{\sum_{j=0}^{n} c_j^2} \left(Y_n - \hat{a}_{n-1} c_n \right). \tag{3}$$

To calculate the estimator $\hat{\phi}_n(x, \hat{a}_n)$ of regression function $\phi(x)$, the parameter a is assumed to be equal to $\hat{a}_n$. However, if the expected value of $\phi(X_i)$ is non-zero (i.e $\int_A \phi(x) f(x) dx \neq 0$), then the estimator $\hat{a}_n$ is biased

$$\hat{a}_n = a + E[R(X_i)] \frac{\sum_{j=1}^{n} c_j}{\sum_{j=1}^{n} c_j^2}. \tag{4}$$

The bias, for particular number of data elements n, can lead to the significant difference between the estimator $\hat{\phi}_n(x, \hat{a}_n)$ and the function $\phi(x)$. To deal with this problem, a linear regression method can be applied in some particular cases. For example, if $c_n = n^t$, $n \in \mathbb{N}$, then estimator $\hat{a}_n$ is given by

$$\hat{a}_n = a + Bn^{-t}, \tag{5}$$

where B is some unknown constant. Given a sequence of estimator values $\hat{a}_i$, $i \in \{n_0, \ldots, n\}$, one can propose the new estimator $\tilde{a}_n$ of a in the following form

$$\tilde{a}_n = \frac{\sum_{i=n_0}^{n} \frac{a_i}{c_i} \sum_{i=n_0}^{n} \frac{1}{c_i} - \sum_{i=n_0}^{n} a_i \sum_{i=n_0}^{n} \frac{1}{c_i^2}}{\sum_{i=n_0}^{n} \frac{1}{c_i} \sum_{i=n_0}^{n} \frac{1}{c_i} - (n - n_0 + 1) \sum_{i=n_0}^{n} \frac{1}{c_i^2}}. \tag{6}$$

Estimators $\hat{a}_i$ for low values of i demonstrate relatively high variances. Therefore, in formula (6) it is recommended to set the value of n_0 sufficiently high. However, an important disadvantage of such an approach is that the estimators $\tilde{a}_n$ can be calculated only for $n > n_0$.

3 Probabilistic Neural Network for Estimation of Regression Function

In this section, the Probabilistic Neural Network scheme will be proposed for estimation of regression function $\phi(x)$. Since the actual value of parameter a is not known, the considered estimator is a function of estimator $\overline{a}_n$, where $\overline{a}_n = \hat{a}_n$ given by (3) or $\overline{a}_n = \tilde{a}_n$ given by (6). To perform the nonlinear regression, the output variables Y_i of MISO system (1) need to be transformed

$$V_i(\overline{a}_n) = Y_i - \overline{a}_n c_i. \tag{7}$$

Defining $R(x) = \phi(x)f(x)$, regression function $\phi(x)$, in each point x at which $f(x) \neq 0$, can be written as follows

$$\phi(x) = \frac{\phi(x)f(x)}{f(x)} = \frac{R(x)}{f(x)}. \tag{8}$$

Estimators $\tilde{R}_n(x, \overline{a}_n)$ and $\tilde{f}_n(x)$ of functions $R(x)$ and $f(x)$ can be established using nonparametric method based on kernel functions $K_n : \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}$ and $K'_n : \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}$

$$\tilde{R}_n(x, \overline{a}_n) = \frac{1}{n}\sum_{i=1}^{n} V_i(\overline{a}_n)K_n(x, X_i), \tag{9}$$

$$\tilde{f}_n(x) = \frac{1}{n}\sum_{i=1}^{n} K'_n(x, X_i). \tag{10}$$

In this paper kernel functions based on Parzen kernels $K : \mathbb{R} \rightarrow \mathbb{R}$ are considered. Any function can be used as a Parzen kernel if satisfies the following conditions

$$\sup_{w\in\mathbb{R}} |K(w)| < \infty, \tag{11}$$

$$\int_{\mathbb{R}} |K(w)|dw < \infty, \tag{12}$$

$$\lim_{\|w\|\rightarrow\infty} \|w\|^p |K(w)| = 0, \tag{13}$$

$$\int_{\mathbb{R}} K(w)dw = 1. \tag{14}$$

Then, the kernel functions $K_n(x, u)$ and $K'_n(x, u)$ are given in the forms

$$K_n(x, u) = \frac{1}{h_n^p}\prod_{k=1}^{p} K\left(\frac{x_k - u_k}{h_n}\right), \tag{15}$$

$$K'_n(x, u) = \frac{1}{h_n'^p}\prod_{k=1}^{p} K\left(\frac{x_k - u_k}{h'_n}\right), \tag{16}$$

where x_k and u_k denote the k-th coordinate of vectors x and u, h_n and h'_n are known sequences, satisfying $\lim_{n\rightarrow\infty} h_n = 0$ and $\lim_{n\rightarrow\infty} h'_n = 0$. Particularly, sequences h_n and h'_n can be equal.

In estimators (9) and (10) for each variable X_i the same kernel function is used ($K_n(x, u)$ and $K'_n(x, u)$ respectively). Alternatively, estimators for functions $R(x)$ and $f(x)$ can be proposed with the use of different kernel functions for each X_i

$$\hat{R}_n(x, \overline{a}_n) = \frac{1}{n}\sum_{i=1}^{n} V_i(\overline{a}_n)K_i(x, X_i), \tag{17}$$

$$\hat{f}_n(x) = \frac{1}{n}\sum_{i=1}^{n} K_i'(x, X_i). \tag{18}$$

The main advantage of this approach is that the estimators $\hat{R}_n(x, \overline{a}_n)$ and $\hat{f}_n(x)$ can be expressed in a recursive way

$$\hat{R}_i(x, \overline{a}_n) = \frac{i-1}{i}\hat{R}_{i-1}(x, \overline{a}_n) + \frac{1}{i}V_i(\overline{a}_n)K_i(x, X_i),\ \hat{R}_0(x, \overline{a}_n) = 0, \tag{19}$$

$$\hat{f}_i(x) = \frac{i-1}{i}\hat{f}_{i-1}(x) + \frac{1}{i}K_i(x, X_i),\ \hat{f}_0(x) = 0, \tag{20}$$

Finally, according to formula (8), the estimator $\hat{\phi}_i(x, \overline{a}_n)$ is given by

$$\hat{\phi}_i(x, \overline{a}_n) = \frac{\hat{R}_i(x, \overline{a}_n)}{\hat{f}_i(x)},\ i \in \{1, \ldots, n\}, \tag{21}$$

The presented algorithm of finding the estimator $\hat{\phi}_n(x, \overline{a}_n)$ is the general regression neural network [36]. The network is shown in Fig. 1.

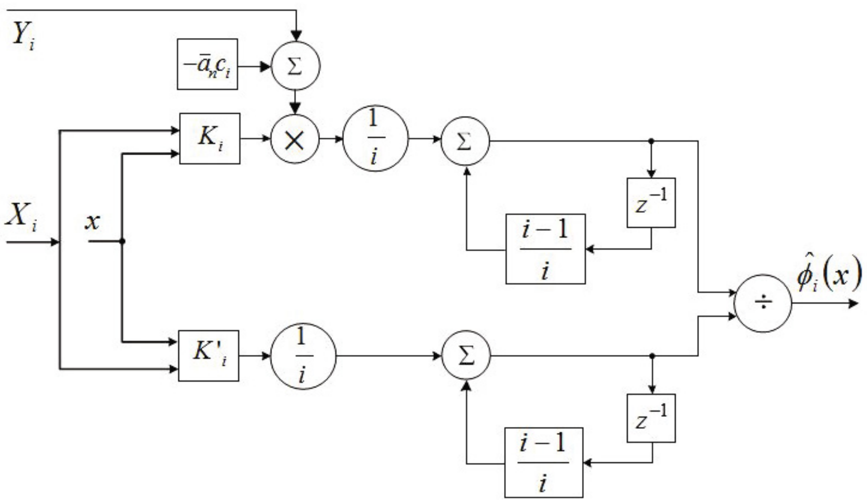


Fig. 1. The block diagram of the general regression neural network for estimating the regression function $\phi(x)$

4 Main Result

The following theorems ensures the convergence of the algorithm presented in sections 2 and 3

Theorem 1. *If (2) holds and the following assumptions are satisfied*

$$\int_A \phi^2(x) f(x) dx < \infty, \tag{22}$$

$$\lim_{n\to\infty} \left(\frac{\sum_{i=1}^n c_i}{\sum_{i=1}^n c_i^2} \right) = 0, \tag{23}$$

then

$$\hat{a}_n \overset{n\to\infty}{\longrightarrow} a \text{ in probability.} \tag{24}$$

Proof. The convergence in probability can be proven analyzing the convergence of expression $E[\hat{a}_n - a]^2$.

Theorem 2. *If (11), (12), (13) and (14) holds and additionally the following conditions are satisfied*

$$\overline{a}_n \overset{n\to\infty}{\longrightarrow} a \text{ in probability,} \tag{25}$$

$$\lim_{n\to\infty} h'_n = 0,\ \lim_{n\to\infty} \left(n^{-2} \sum_{i=1}^n h_i'^{-p} \right) = 0, \tag{26}$$

$$\lim_{n\to\infty} h_n = 0,\ \lim_{n\to\infty} \left(n^{-2} \sum_{i=1}^n h_i^{-p} \right) = 0, \tag{27}$$

then

$$\hat{\phi}_n(x, \overline{a}_n) \overset{n\to\infty}{\longrightarrow} \phi(x) \text{ in probability.} \tag{28}$$

Proof. The theorem can be proven using the convergence (25), convergence of the estimator $\hat{\phi}_n(x, \overline{a}_n)$ [1] and Theorem 4.3.8 in [40].

Example
Let us assume that conditions (2) and (22) are satisfied. If the sequence c_i is taken in the form

$$c_i = i^t, t > 0, \tag{29}$$

then

$$\frac{\sum_{i=1}^n c_i}{\sum_{i=1}^n c_i^2} = O(i^{-t}). \tag{30}$$

In the light of assumption (23) the algorithm described in section 2 is convergent for any value of $t > 0$. To investigate convergence of the algorithm from section 3, one can propose the following form of sequences h_n and h'_n

$$h_n = Dn^{-H},\ h'_n = D'n^{-H'},\quad D, D' > 0,\quad H, H' > 0. \tag{31}$$

The assumptions (26) and (27) of the Theorem 2 are held if the parameters H and H' satisfy the following inequalities

$$0 < H < 1,\ \ 0 < H' < 1. \tag{32}$$

5 Experimental Results

In the following experiments the quality of the regression function estimators $\hat{\phi}_n(x, \hat{a}_n)$ and $\hat{\phi}_n(x, \tilde{a}_n)$ is examined. The value of the parameter n_0 in estimator $\hat{\phi}_n(x, \tilde{a}_n)$ is set to 400. The real value of parameter a is equal to $1,5$. The one-dimensional regression function $\phi(x)$ is considered, which is given by

$$\phi(x) = 10 * \exp\left(3 * (\sin x - 1)\right) + 1. \tag{33}$$

The random variables X_i come from the normal distribution $N(4, 4)$ and the noise variables Z_i are generated form the standard normal distribution $N(0, 1)$. Sequence c_i is given in the form (29) with $t = 0,4$. As the Parzen kernel $K(w)$, in kernel functions (15) and (16), the Epanechnikov kernel is proposed

$$K(w) = \begin{cases} \frac{3}{4}\left(1 - w^2\right), & w \in [-1; 1], \\ 0, & w \in (-\infty; -1) \cup (1; \infty). \end{cases} \tag{34}$$

It is easily seen that the Epanechnikov kernel satisfies conditions (11)-(14). The sequences h'_n and h_n are taken in the form (31), with parameters $D = D' = 2,0$ and $Q = Q' = 0,3$, which satisfy inequalities (32).

In Figure 2 the values of estimators $\hat{a}_n$ and $\tilde{a}_n$ for different numbers of data elements n are presented. The estimators are also compared with the real value of a.

Both estimators converge to the value $a = 1,5$, however the estimator $\tilde{a}_n$ is significantly better for each considered number of data elements n. The value of estimator $\overline{a}_n$, where $\overline{a}_n$ can be taken as $\hat{a}_n$ or as $\tilde{a}_n$, affects the quality of the estimation of the function $\phi(x)$. If the actual value of a were known, the unbiased estimator $\hat{\phi}_n(x, a)$ of the regression function could be calculated. The difference between the estimator $\overline{a}_n$ and the value of a introduces some bias to the estimator $\hat{\phi}_n(x, \overline{a}_n)$. The bias is proportional to the mentioned difference $(\overline{a}_n - a)$. This effect is presented in Fig. 3.

The Mean Squared Error (MSE) for the estimator $\hat{\phi}_n(x, \tilde{a}_n)$ is very close to zero value, while for the estimator $\hat{\phi}_n(x, \hat{a}_n)$ it holds at the level of $MSE \approx 10$. This results proves the usability of the estimator $\tilde{a}_n$.

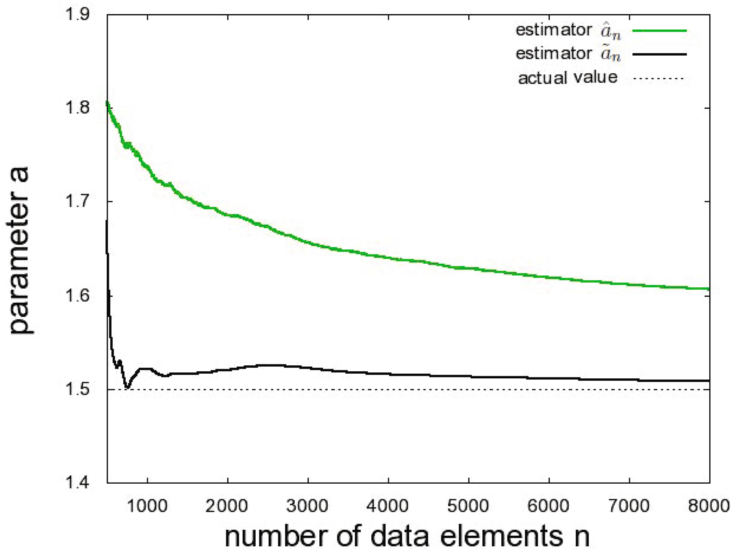


Fig. 2. Comparison of the estimators $\hat{a}_n$ and $\tilde{a}_n$ with the actual value of parameter a

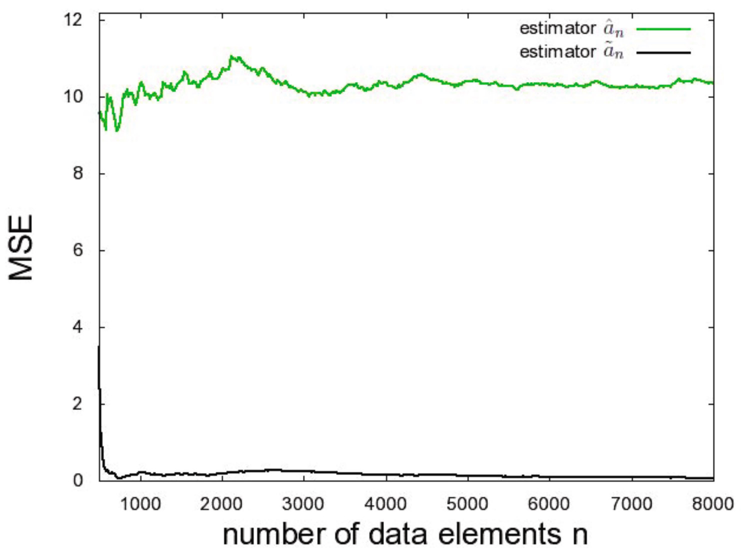


Fig. 3. The Mean Squared Error of estimator $\hat{\phi}_n(x, \overline{a}_n)$ as a function of number of data elements n, for $\overline{a}_n = \hat{a}_n$ and $\overline{a}_n = \tilde{a}_n$

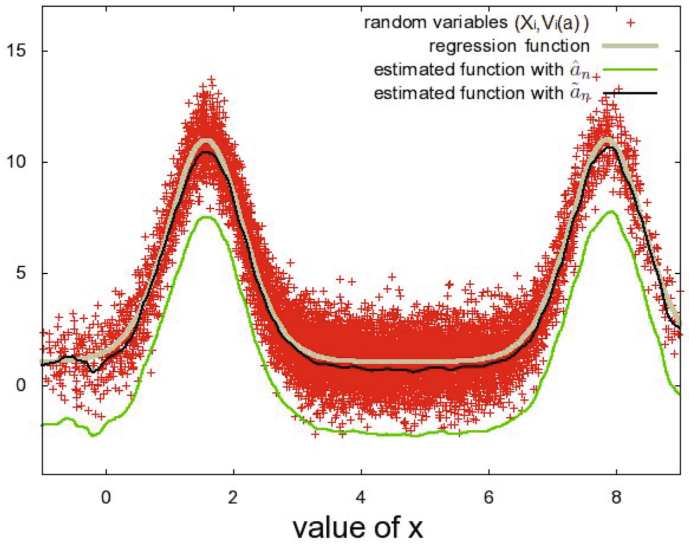


Fig. 4. Estimators $\hat{\phi}_n(x, \hat{a}_n)$ and $\hat{\phi}_n(x, \tilde{a}_n)$ in comparison with the function $\phi(x)$. Points represent the pairs of random variables $(X_i, V_i(a))$.

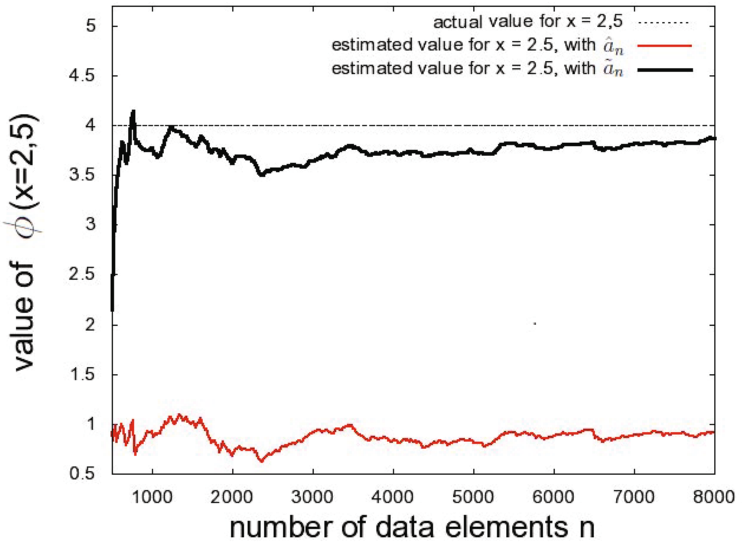


Fig. 5. The values of $\hat{\phi}_n(2,5\,, \hat{a}_n)$ and $\hat{\phi}_n(2,5\,, \tilde{a}_n)$ as functions of n in comparison with the actual value $\phi(2,5) \approx 3,998$

In Figure 4 estimators $\hat{\phi}_n(x, \hat{a}_n)$ and $\hat{\phi}_n(x, \tilde{a}_n)$, obtained for $n = 5000$, are presented in comparison with the regression function (33).

The estimators $\hat{\phi}_n(x, \overline{a}_n)$ ($\overline{a}_n = \hat{a}_n$ or $\overline{a}_n = \tilde{a}_n$) actually try to fit the function to the pairs $(X_i, V_i(\overline{a}_n))$ instead of $(X_i, V_i(a))$. This explains why the bias of the estimator $\hat{\phi}_n(x, \hat{a}_n)$ is such large.

Figure 5 shows the values of considered estimator in one particular point $x = 2,5$ for different numbers of data elements n. The value of the real regression function in this point is $\phi(2, 5) \approx 3,998$.

6 Conclusions

In the paper the Parzen kernel-type regression neural network in combination with recursive least squared method were presented to solve problem of learning in a time-varying environment. Sufficient conditions for convergence in probability were given. Simulation experiments were presented and discussed. Our on-going work is focused on adaptation of supervised and unsupervised neural networks (see e.g. [3], [5] and [13]) and neuro-fuzzy structures (see e.g. [10], [11], [32]-[35] and [38]-[39]) for learning in time-varying environments.

Acknowledgments. The paper was prepared under project operated within the Foundation for Polish Science Team Programme co-financed by the EU European Regional Development Fund, Operational Program Innovative Economy 2007-2013, and also supported by National Science Centre NCN.

References

1. Ahmad, I.A., Lin, P.E.: Nonparametric sequential estimation of multiple regression function. Bulletin of Mathematical Statistics 17, 63–75 (1976)
2. Albert, A.E., Gardner, L.A.: Stochastic Approximation and Nonlinear Regression, vol. (42). MIT Press, Cambridge (1967)
3. Bilski, J., Rutkowski, L.: A fast training algorithm for neural networks. IEEE Transactions on Circuits and Systems II 45, 749–753 (1998)
4. Cacoullos, P.: Estimation of a multivariate density. Annals of the Institute of Statistical Mathematics 18, 179–190 (1965)
5. Cierniak, R., Rutkowski, L.: On image compression by competitive neural networks and optimal linear predictors. Signal Processing: Image Communication - a Eurasip Journal 15(6), 559–565 (2000)
6. Gałkowski, T., Rutkowski, L.: Nonparametric recovery of multivariate functions with applications to system identification. Proceedings of the IEEE 73, 942–943 (1985)
7. Gałkowski, T., Rutkowski, L.: Nonparametric fitting of multivariable functions. IEEE Transactions on Automatic Control AC-31, 785–787 (1986)
8. Greblicki, W., Rutkowska, D., Rutkowski, L.: An orthogonal series estimate of time-varying regression. Annals of the Institute of Statistical Mathematics 35, Part A, 147–160 (1983)

9. Greblicki, W., Rutkowski, L.: Density-free Bayes risk consistency of nonparametric pattern recognition procedures. Proceedings of the IEEE 69(4), 482–483 (1981)
10. Nowicki, R.: Rough Sets in the Neuro-Fuzzy Architectures Based on Non-monotonic Fuzzy Implications. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 518–525. Springer, Heidelberg (2004)
11. Nowicki, R., Pokropińska, A.: Information Criterions Applied to Neuro-Fuzzy Architectures Design. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 332–337. Springer, Heidelberg (2004)
12. Parzen, E.: On estimation of a probability density function and mode. Analysis of Mathematical Statistics 33(3), 1065–1076 (1962)
13. Patan, K., Patan, M.: Optimal Training Strategies for Locally Recurrent Neural Networks. Journal of Artificial Intelligence and Soft Computing Research 1(2), 103–114 (2011)
14. Rutkowski, L.: Sequential estimates of probability densities by orthogonal series and their application in pattern classification. IEEE Transactions on Systems, Man, and Cybernetics SMC-10(12), 918–920 (1980)
15. Rutkowski, L.: Sequential estimates of a regression function by orthogonal series with applications in discrimination, New York, Heidelberg, Berlin. Lectures Notes in Statistics, vol. 8, pp. 236–244 (1981)
16. Rutkowski, L.: On system identification by nonparametric function fitting. IEEE Transactions on Automatic Control AC-27, 225–227 (1982)
17. Rutkowski, L.: Orthogonal series estimates of a regression function with applications in system identification. In: Probability and Statistical Inference, pp. 343–347. D. Reidel Publishing Company, Dordrecht (1982)
18. Rutkowski, L.: On Bayes risk consistent pattern recognition procedures in a quasi-stationary environment. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4(1), 84–87 (1982)
19. Rutkowski, L.: On-line identification of time-varying systems by nonparametric techniques. IEEE Transactions on Automatic Control AC-27, 228–230 (1982)
20. Rutkowski, L.: On nonparametric identification with prediction of time-varying systems. IEEE Transactions on Automatic Control AC-29, 58–60 (1984)
21. Rutkowski, L.: Nonparametric identification of quasi-stationary systems. Systems and Control Letters 6, 33–35 (1985)
22. Rutkowski, L.: The real-time identification of time-varying systems by nonparametric algorithms based on the Parzen kernels. International Journal of Systems Science 16, 1123–1130 (1985)
23. Rutkowski, L.: A general approach for nonparametric fitting of functions and their derivatives with applications to linear circuits identification. IEEE Transactions Circuits Systems CAS-33, 812–818 (1986)
24. Rutkowski, L.: Sequential pattern recognition procedures derived from multiple Fourier series. Pattern Recognition Letters 8, 213–216 (1988)
25. Rutkowski, L.: Nonparametric procedures for identification and control of linear dynamic systems. In: Proceedings of 1988 American Control Conference, June 15-17, pp. 1325–1326 (1988)
26. Rutkowski, L.: An application of multiple Fourier series to identification of multi-variable nonstationary systems. International Journal of Systems Science 20(10), 1993–2002 (1989)
27. Rutkowski, L.: Nonparametric learning algorithms in the time-varying environments. Signal Processing 18, 129–137 (1989)

28. Rutkowski, L., Rafajłowicz, E.: On global rate of convergence of some nonparametric identification procedures. IEEE Transaction on Automatic Control AC-34(10), 1089–1091 (1989)
29. Rutkowski, L.: Identification of MISO nonlinear regressions in the presence of a wide class of disturbances. IEEE Transactions on Information Theory IT-37, 214–216 (1991)
30. Rutkowski, L.: Multiple Fourier series procedures for extraction of nonlinear regressions from noisy data. IEEE Transactions on Signal Processing 41(10), 3062–3065 (1993)
31. Rutkowski, L., Gałkowski, T.: On pattern classification and system identification by probabilistic neural networks. Applied Mathematics and Computer Science 4(3), 413–422 (1994)
32. Rutkowski, L.: A New Method for System Modelling and Pattern Classification. Bulletin of the Polish Academy of Sciences 52(1), 11–24 (2004)
33. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, Melbourne, December 2-5, vol. 3, pp. 1428–1431 (2001)
34. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)
35. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)
36. Specht, D.F.: A general regression neural network. IEEE Transactions on Neural Networks 2, 568–576 (1991)
37. Specht, D.F.: Probabilistic neural networks. Neural Networks 3, 109–118 (1990)
38. Starczewski, J., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
39. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)
40. Wilks, S.S.: Mathematical Statistics. John Wiley, New York (1962)

Learning in Rough-Neuro-Fuzzy System for Data with Missing Values

Bartosz A. Nowak and Robert K. Nowicki

Department of Computer Engineering, Czestochowa University of Technology,
Al. Armii Krajowej 36, 42-200 Czestochowa, Poland
{bartosz.nowak,robert.nowicki}@kik.pcz.pl

Abstract. Rough-neuro-fuzzy systems offer suitable way for classifying data with missing values. The paper presents a new implementation of gradient learning in the case of missing input data which has been adapted for rough-neuro-fuzzy classifiers. We consider the system with singleton fuzzification, Mamdani-type reasoning and center average defuzzification. Several experiments based on common benchmarks illustrating the performance of trained systems are shown. The learning and testing of the systems has been performed with various number of missing values.

Keywords: missing values, rough fuzzy, classification, back-propagation.

1 Introduction

Data classification is one of often challenge for computational intelligence. Usually data bases are collected using questionnaires, medical examinations or outputs from different types of measuring devices. Every sample is described by few parameters, nearly always more than one. It is very often that part of the data is missing. Reason that led to a lack of some information is very important in choosing relevant method of classification. In literature [1], [2] have been proposed three main types of databases with missing data:

– MCAR (missing completely at random) – probability of missing parameter v_i is not related with value of other parameters $v_j, j \neq i$ neither true or hypothetical value of v_i (if exist). As an example could be outputs of questionnaires send by unstable link, with error detection, which deletes corrupted parts of transmission. As result, parts of data is lost, and probability of lacks in database is true random. This conditions is very strict and is rarely fulfilled in practice.
– MAR (missing at random) – probability of missing parameter v_i depends on other parameters $v_j, j \neq i$ however it not depends on true or hypothetical value of v_i (if exist). For example there is form with questions: “How old are you?”, “What is your bank account number?”. It is obvious that younger people will more often restraint themselves from answering the second question.

– MNAR (missing not at random) – probability of missing parameter v_i is connected with its true or hypothetical value. The same connection as in MCAR, but with assumption that v_i is more often unavailable when its original value is 0 due to problem in transmission, could be presented as an example.

The main approaches for classification with missing values are:

1. Imputation – method base on filling lacks in the database with valid values using various methods, e.g: imputing average value of attribute, imputing random value in range of occurrence, determining lost value based on most similar samples using EM-algorithm [3], k-nn [4] or others [5] and replacing incomplete sample $\mathbf{v}^{(i)}$ by a set of complete samples, which base on $\mathbf{v}^{(i)}$ [6].
2. Marginalisation – part of database that has unavailable data is simply deleted or marginalised thus reducing dimensionality of database, e.g: list-wise deletion – samples that have lacks of some values are erased (probably the simplest of all marginalisation methods, but often used in real-life, suitable only to MCAR databases with very low rate of missing), pair-wise deletion – according to analysis requirements only samples that has missing needed attributes are ignored and attribute deletion – attributes that have lacks are deleted.
3. Rough-neuro-fuzzy system as generalisation of neuro-fuzzy system are capable to handle data with missing values [7].

The paper proposes implementation of back-propagation as learning algorithm for rough-neuro-fuzzy decision system [7] with Mamdani-type implication and CA defuzzification.

2 Rough-Neuro-Fuzzy Systems

The rough neuro-fuzzy classifier which is under consideration has been described in [7]. It is based on classic neuro-fuzzy system which implements singleton fuzzification, Mamdani-type reasoning and Centre Average defuzzification. The answer of such MIMO neuro-fuzzy system adapted for classification tasks has a form

$$\mu_{\omega_j}(x) = \overline{z}_j = \frac{\sum\limits_{r=1}^{N} \overline{z}_j^r \cdot \mu_{A^r}(\mathbf{v})}{\sum\limits_{r=1}^{N} \mu_{A^r}(\mathbf{v})}. \tag{1}$$

As was stated, the output value is interpreted as the membership degree of object x to class ω_j. Classified object is described by vector of features $\mathbf{v} = [v_1, v_2, \ldots, v_n]$. The fuzz sets $A^r = A_1^r \times A_2^r \times \ldots \times A_n^r$ and parameters $\overline{z}_j^r$ corresponding to rules in form

$$\begin{array}{c} R^r : \textbf{IF } v_1 \text{ is } A_1^r \text{ AND } v_2 \text{ is } A_2^r \text{ AND } \ldots \text{AND } v_n \text{ is } A_n^r \\ \textbf{THEN } x \in \omega_1(\overline{z}_1^r), x \in \omega_2(\overline{z}_2^r) \ldots x \in \omega_m(\overline{z}_m^r) \end{array}. \tag{2}$$

2.1 Rough-Neuro-Fuzzy Classifier

The classifier presented in previous section assume that values of all n conditional attributes v_i are known. Now we assume that values of some attributes are missing. So, we divide the set Q of all n attributes into two subset P and G as follows

$$\begin{array}{ll} v_i \in P & \text{if value for } v_i \text{ is known,} \\ v_i \in G & \text{if value for } v_i \text{ is missing.} \end{array} \tag{3}$$

Formally, we have the vector of known values $\overline{\mathbf{v}}_P \in \mathbb{V}_P$ and vector of unknown values, but we known that $\overline{\mathbf{v}}_G \in \mathbb{V}_G$. In consequent, the membership function of fuzzy set A^r is unknown. However, we can approximate it by rough fuzzy set which is the pair $\left\{\overline{\widetilde{P}}A^r, \underline{\widetilde{P}}A^r\right\}$. $\overline{\widetilde{P}}A^r$ is the $\widetilde{P}$–upper approximation of fuzzy set A^r and $\underline{\widetilde{P}}A^r$ is its $\widetilde{P}$–lower approximation. Their membership function, as in [8], is defined by

$$\begin{aligned} \mu_{\overline{\widetilde{P}}A}(\overline{\mathbf{v}}) &= \underset{\overline{\mathbf{v}}_G \in \mathbb{V}_G}{sup}\ \mu_A(\overline{\mathbf{v}}_P, \overline{\mathbf{v}}_G)\ , \\ \mu_{\underline{\widetilde{P}}A}(\overline{\mathbf{v}}) &= \underset{\overline{\mathbf{v}}_G \in \mathbb{V}_G}{inf}\ \mu_A(\overline{\mathbf{v}}_P, \overline{\mathbf{v}}_G)\ . \end{aligned} \tag{4}$$

Using any t–norm in definition of Cartesian product, w have

$$\begin{aligned} \mu_{\overline{\widetilde{P}}A}(\overline{\mathbf{v}}) &= T\left(\underset{i:v_i \in P}{T}(\mu_{A_i}(\overline{v}_i)), \underset{i:v_i \in G}{T} \sup_{\overline{v}_i \in \mathbb{V}_i} \mu_{A_i}(\overline{v}_i)\right)\ , \\ \mu_{\underline{\widetilde{P}}A}(\overline{\mathbf{v}}) &= T\left(\underset{i:v_i \in P}{T}(\mu_{A_i}(\overline{v}_i)), \underset{i:v_i \in G}{T} \inf_{\overline{v}_i \in \mathbb{V}_i} \mu_{A_i}(\overline{v}_i)\right)\ . \end{aligned} \tag{5}$$

It is obvious that $\mu_{\underline{\widetilde{P}}A}(\overline{\mathbf{v}}) \le \mu_{\overline{\widetilde{P}}A}(\overline{\mathbf{v}})$. When $\overline{\mathbf{v}}$ has no lacks then $\overline{\widetilde{P}}A = \underline{\widetilde{P}}A$ and $\mu_A(\overline{\mathbf{v}}) = \mu_{\overline{\widetilde{P}}A}(\overline{\mathbf{v}}) = \mu_{\underline{\widetilde{P}}A}(\overline{\mathbf{v}})$.

Following the approximation expressed sets A^r we obtain the approximate answer of classifier $\left\{\mu_{\overline{\widetilde{P}}\omega_j}(x), \mu_{\underline{\widetilde{P}}\omega_j}(x)\right\} = \left\{\overline{\overline{z}_j}, \underline{\overline{z}_j}\right\}$. As has been proven in [9]

$$\begin{aligned} \overline{\overline{z}_j} &= \frac{\sum_{r=1}^{N} \overline{z}_j^r \mu_{\overline{\widetilde{P}}A^r}(\mathbf{v})}{\sum_{r=1}^{N}\left(\overline{z}_j^r \mu_{\overline{\widetilde{P}}A^r}(\mathbf{v}) + \neg\overline{z}_j^r \mu_{\underline{\widetilde{P}}A^r}(\mathbf{v})\right)}, \\ \underline{\overline{z}_j} &= \frac{\sum_{r=1}^{N} \overline{z}_j^r \mu_{\underline{\widetilde{P}}A^r}(\mathbf{v})}{\sum_{r=1}^{N}\left(\overline{z}_j^r \mu_{\underline{\widetilde{P}}A^r}(\mathbf{v}) + \neg\overline{z}_j^r \mu_{\overline{\widetilde{P}}A^r}(\mathbf{v})\right)}, \end{aligned} \tag{6}$$

where negation operator $\neg\overline{z}_j^r = 1 - \overline{z}_j^r$.

To date learning algorithm has not been proposed. Whole decision system can be presented in network form.

3 Learning Algorithm

One of main advantages of neuro-fuzzy system over fuzzy system, mentioned in literature [10] is possibility of use gradient learning method as in neural networks.

The method, working in neuro-fuzzy systems, can be adapted to rough-neuro-fuzzy classifier. In implemented system only c_i^r, σ_i^r are the parameters that are subject to learn. On account of assumption $z_j^r \in \{0,1\}$ parameters z_j^r are no suitable to gradient learning. Error is computed providing sample $\overline{\mathbf{v}}_p$ on inputs and comparing obtained result with requested output pattern $\mathbf{d} = [d_1, d_2, \ldots, d_m]$.

Proposed method additionally ensure that values c_i^r are always in range of occurrence v_i, i.e. $[v_{i\min}, v_{i\max}]$. If the boundary aren't known, the minimum and maximum values funded in learning set is obtained. Correction of system's parameters is performed to minimise sum of squares for every error on outputs. In the paper we propose probably the simplest error measure Q. We we would like to equate lower $\underline{\overline{z}}_j$ and upper $\overline{\overline{z}}_j$ outputs with binary affiliation of given sample to every class. It leads to following definition of Q:

$$Q(c_i^r, \sigma_i^r) = \sum_{j=1}^{m} \left(\left(d_j - \overline{\overline{z}_j}\right)^2 + \left(d_j - \underline{\overline{z}_j}\right)^2 \right). \tag{7}$$

Decreasing of $Q(c_i^r, \sigma_i^r)$ is done by gradient method, which in consecutive iterations changes the values of parameters c_l^k and σ_l^k as follow:

$$c_i^r(t+1) = c_i^r(t) + \varDelta c_i^r(t); \quad \sigma_i^r(t+1) = \sigma_i^r(t) + \varDelta \sigma_i^r(t)\ , \tag{8}$$

where t is counter of iteration and $\varDelta c_l^k, \varDelta \sigma_l^k$ are the corrections defined by

$$\varDelta c_l^k = \eta \left(-\frac{\partial Q}{\partial c_i^r} \right); \quad \varDelta \sigma_l^k = \eta \left(-\frac{\partial Q}{\partial \sigma_l^k} \right). \tag{9}$$

Symbol $\eta \in (0,1)$ means the coefficient of learning.

For rough neuro-fuzzy classifier defined by eq. (6), the derivatives from eq. (9) are expressed by

$$-\frac{\partial Q}{\partial c_l^k}(\mathbf{v}) = 2 \sum_{j=1}^{m} \left(\begin{array}{l} \frac{\overline{\epsilon}_j}{(u\_d_j)^2} \left(u\_d_j \overline{z}_l^k \frac{\partial \mu_{\overline{\widetilde{P}}A^k}}{\partial c_l^k} - u\_n_j \left(\overline{z}_l^k \frac{\partial \mu_{\overline{\widetilde{P}}A^k}}{\partial c_l^k} + \neg \overline{z}_l^k \frac{\partial \mu_{\underline{\widetilde{P}}A^k}}{\partial c_l^k} \right) \right) + \\ + \frac{\underline{\epsilon}_j}{(d\_d_j)^2} \left(d\_d_j \overline{z}_l^k \frac{\partial \mu_{\underline{\widetilde{P}}A^k}}{\partial c_l^k} - d\_n_j \left(\overline{z}_l^k \frac{\partial \mu_{\underline{\widetilde{P}}A^k}}{\partial c_l^k} + \neg \overline{z}_l^k \frac{\partial \mu_{\overline{\widetilde{P}}A^k}}{\partial c_l^k} \right) \right) \end{array} \right). \tag{10}$$

and

$$-\frac{\partial Q}{2\partial \sigma_l^k}(\mathbf{v}) = 2 \sum_{j=1}^{m} \left(\begin{array}{l} \frac{\overline{\epsilon}_j}{(u\_d_j)^2} \left(u\_d_j \overline{z}_l^k \frac{\partial \mu_{\overline{\widetilde{P}}A^k}}{\partial \sigma_l^k} - u\_n_j \left(\overline{z}_l^k \frac{\partial \mu_{\overline{\widetilde{P}}A^k}}{\partial \sigma_l^k} + \neg \overline{z}_l^k \frac{\partial \mu_{\underline{\widetilde{P}}A^k}}{\partial \sigma_l^k} \right) \right) + \\ + \frac{\underline{\epsilon}_j}{(d\_d_j)^2} \left(d\_d_j \overline{z}_l^k \frac{\partial \mu_{\underline{\widetilde{P}}A^k}}{\partial \sigma_l^k} - d\_n_j \left(\overline{z}_l^k \frac{\partial \mu_{\underline{\widetilde{P}}A^k}}{\partial \sigma_l^k} + \neg \overline{z}_l^k \frac{\partial \mu_{\overline{\widetilde{P}}A^k}}{\partial \sigma_l^k} \right) \right) \end{array} \right). \tag{11}$$

where the difference between desired values of lower and upper values can be different:

$$\overline{\epsilon}_j = d_j - \overline{\overline{z}_j}; \ \underline{\epsilon}_j = d_j - \underline{\overline{z}_j} \tag{12}$$

and the symbols u_n_j, u_d_j, d_n_j and d_d_j are defined as follow

$$\begin{array}{l} u\_n_j = \sum_{r=1}^{N} \overline{z}_j^r \mu_{\overline{\widetilde{P}}A^r}(\mathbf{v});\ u\_d_j = \sum_{r=1}^{N} \left(\overline{z}_j^r \mu_{\overline{\widetilde{P}}A^r}(\mathbf{v}) + \neg \overline{z}_j^r \mu_{\underline{\widetilde{P}}A^r}(\mathbf{v}) \right); \\ d\_n_j = \sum_{r=1}^{N} \overline{z}_j^r \mu_{\underline{\widetilde{P}}A^r}(\mathbf{v});\ d\_d_j = \sum_{r=1}^{N} \left(\overline{z}_j^r \mu_{\underline{\widetilde{P}}A^r}(\mathbf{v}) + \neg \overline{z}_j^r \mu_{\overline{\widetilde{P}}A^r}(\mathbf{v}) \right). \end{array} \tag{13}$$

Modifications of fuzzy sets are performed many times, consecutively after computing gradient of error for every sample.

The $\widetilde{P}$–lower and $\widetilde{P}$–upper approximation of antexedent fuzzy sets A^q is defined by eq. (5). It is obvious, the form derivatives $\frac{\partial\mu_{\overline{\widetilde{P}}A^q}}{\partial\sigma_i^q}(\mathbf{v})$, $\frac{\partial\mu_{\overline{\widetilde{P}}A^q}}{\partial c_i^q}(\mathbf{v})$, $\frac{\partial\mu_{\underline{\widetilde{P}}A^q}}{\partial\sigma_i^q}(\mathbf{v})$, $\frac{\partial\mu_{\underline{\widetilde{P}}A^q}}{\partial c_i^q}(\mathbf{v})$ are various for case of known value of attribute v_i and for case of missing one. Further we will propose corresponding forms. At first, it is assumed that values c_i^q are always in the range $[v_{i\min}, v_{i\max}]$. It will be provide by a corrections introduced in the learning procedure (16). Thus, the expression $\sup_{\overline{v}_l\in\mathbb{V}_l}\mu_A(\overline{v}_l)$ in (5) become a constant. When sets A_l^k is normal (i.e. the height $h\left(A_l^k\right)=1$), we have

$$\sup_{\overline{v}_l\in\mathbb{V}_l}\mu_{A_l^k}(\overline{v}_l)=\mu_{A_l^k}(\overline{c}_l^k)=1. \tag{14}$$

In the same way, the expression $\inf_{\overline{v}_l\in\mathbb{V}_l}\mu_A(\overline{v}_l)$ in (5) takes the form

$$\inf_{\overline{v}_l\in\mathbb{V}_l}\mu_{A_l^k}(\overline{v}_l)=\min\left\{\mu_{A_l^k}(v_{l\min}),\mu_{A_l^k}(v_{l\max})\right\}. \tag{15}$$

Following (14) and (15) we can replace unknown input value v_l with one of: $\{c_l^r, v_{l\max}, v_{l\min}\}$. During computation of $\frac{\partial\mu_{\overline{\widetilde{P}}A^k}}{\partial\sigma_l^k}(\mathbf{v})$, $\frac{\partial\mu_{\overline{\widetilde{P}}A^k}}{\partial c_l^k}(\mathbf{v})$ when v_l has no value it is substituted by the centre of fuzzy set, and during computation of $\frac{\partial\mu_{\underline{\widetilde{P}}A^k}}{\partial\sigma_l^k}(\mathbf{v})$, $\frac{\partial\mu_{\underline{\widetilde{P}}A^k}}{\partial c_l^k}(\mathbf{v})$ in the same condition, then value of v_l is replaced by $v_{l\min}$ or $v_{l\max}$ depending which is more distant from centre of fuzzy set. Because $\mu_{A_l^k}(\overline{c}_l^k)$ are constant then $\frac{\partial\mu_{\overline{\widetilde{P}}A_l^k}}{\partial c_l^k}=\frac{\partial\mu_{\overline{\widetilde{P}}A_l^k}}{\partial\sigma_l^k}=0$.

$$\begin{array}{c}\text{if } v_i \text{ is not missing:}\\
\begin{array}{cc}\frac{\partial\mu_{\overline{\widetilde{P}}A^q}}{\partial c_i^q}(\mathbf{v})=2\mu_{\overline{\widetilde{P}}A^q}\frac{v_i-c_i^q}{(\sigma_i^q)^2}\ ; & \frac{\partial\mu_{\overline{\widetilde{P}}A^q}}{\partial\sigma_i^q}(\mathbf{v})=2\mu_{\overline{\widetilde{P}}A^q}\frac{(v_i-c_i^q)^2}{(\sigma_i^q)^3}\ ;\\
\frac{\partial\mu_{\underline{\widetilde{P}}A^q}}{\partial c_i^q}(\mathbf{v})=2\mu_{\underline{\widetilde{P}}A^q}\frac{v_i-c_i^q}{(\sigma_i^q)^2}\ ; & \frac{\partial\mu_{\underline{\widetilde{P}}A^q}}{\partial\sigma_i^q}(\mathbf{v})=2\mu_{\underline{\widetilde{P}}A^q}\frac{(v_i-c_i^q)^2}{(\sigma_i^q)^3}\end{array}\\
\text{if } v_i \text{ is missing:}\\
\begin{array}{cc}\frac{\partial\mu_{\overline{\widetilde{P}}A^q}}{\partial c_i^q}(\mathbf{v})=0\ ; & \frac{\partial\mu_{\overline{\widetilde{P}}A^q}}{\partial\sigma_i^q}(\mathbf{v})=0\ ;\\
\frac{\partial\mu_{\underline{\widetilde{P}}A^q}}{\partial c_i^q}(\mathbf{v})=2\mu_{\underline{\widetilde{P}}A^q}\frac{v_{i,\mathrm{far}}^q-c_i^q}{(\sigma_i^q)^2}\ ; & \frac{\partial\mu_{\underline{\widetilde{P}}A^q}}{\partial\sigma_i^q}(\mathbf{v})=2\mu_{\underline{\widetilde{P}}A^q}\frac{(v_{i,\mathrm{far}}^q-c_i^q)^2}{(\sigma_i^q)^3}\end{array}\end{array},$$

where

$$v_{i,\mathrm{far}}^q=\begin{cases}v_{i\min} & \text{if } |v_{i\min}-c_i^q|>|v_{i\max}-c_i^q|\\ v_{i\max} & \text{if } |v_{i\min}-c_i^q|<|v_{i\max}-c_i^q|\\ c_i^q & \text{if } |v_{i\min}-c_i^q|=|v_{i\max}-c_i^q|\end{cases},$$

and $v_{i\min}$, $v_{i\max}$ are relevant minimum or maximum value of attribute c_i among all learning samples.

Corrections Δc_l^k and $\Delta\sigma_l^k$ are not introduced directly to avoid situation that c_i^r exceed range of v_i occurrence, and situation that $\sigma_l^k\le 0$. So, there are replaced by $(\Delta c_l^k)^*$ and $(\Delta\sigma_l^k)^*$ defined as follows:

$$(\Delta c_l^k)^* = \begin{cases} v_{i\min} - c_l^k & \text{if } v_{l\min} \geq c_i^r + \Delta c_i^r \\ \Delta c_l^k & \text{if } v_{l\min} < c_i^r + \Delta c_i^r < v_{l\max} \\ v_{i\max} - c_l^k & \text{if } c_i^r + \Delta c_i^r \geq v_{l\max}\ , \end{cases}$$
$$(\Delta \sigma_l^k)^* = \begin{cases} 0 & \text{if } 0 \geq \sigma_l^k + \Delta \sigma_l^k \\ \Delta \sigma_l^k & \text{if } 0 < \sigma_l^k + \Delta \sigma_l^k\ . \end{cases} \tag{16}$$

3.1 Back-Propagation Algorithm

Application of back-propagation method can significantly simplify notation of derivatives. Corrections are calculated starting from outputs, as shown in (Fig. 1). Corrections are computed using several steps, where $j = 1 \ldots m$, $r = 1 \ldots N$, $i = 1 \ldots n$:

$$\overline{\epsilon\_d_j} = -\frac{\overline{\epsilon}_j u\_n_j}{(u\_d)^2}; \quad \underline{\epsilon\_d_j} = -\frac{\underline{\epsilon}_j d\_n_j}{(d\_d)^2}$$
$$\overline{\epsilon\_n_j} = \frac{\overline{\epsilon}_j}{u\_d} + \overline{\epsilon\_d_j}; \quad \underline{\epsilon\_n_j} = \frac{\underline{\epsilon}_j}{d\_d} + \underline{\epsilon\_d_j}$$
$$\overline{\epsilon\_act_r} = \sum_{j=1}^{m} \left(z_j^r \overline{\epsilon\_n_j} + \neg z_j^r \underline{\epsilon\_d_j} \right); \quad \underline{\epsilon\_act_r} = \sum_{j=1}^{m} \left(z_j^r \underline{\epsilon\_n_j} + \neg z_j^r \overline{\epsilon\_d_j} \right)$$
$$\overline{\epsilon\_c\_\sigma_r} = \overline{\epsilon\_act_r} \mu_{\overline{\widetilde{P}}A}; \quad \underline{\epsilon\_c\_\sigma_r} = \underline{\epsilon\_act_r} \mu_{\underline{\widetilde{P}}A}$$

if v_i is not missing:

$$\Delta c_i^r = \eta \frac{v_i - c_i^r}{(\sigma_i^r)^2} \left(\overline{\epsilon\_c\_\sigma_r} + \underline{\epsilon\_c\_\sigma_r} \right); \quad \Delta \sigma_i^r = \eta \frac{(v_i - c_i^r)^2}{(\sigma_i^r)^3} \left(\overline{\epsilon\_c\_\sigma_r} + \underline{\epsilon\_c\_\sigma_r} \right),$$

if v_i is missing:

$$\Delta c_i^r = \eta \underline{\epsilon\_c\_\sigma_r} \frac{v_{i,far}^r - c_i^r}{(\sigma_i^r)^2}; \qquad \Delta \omega_i^r = \eta \underline{\epsilon\_c\_\sigma_r} \frac{(v_{i,far}^r - c_i^r)^2}{(\sigma_i^r)^3}\ .$$

After each iteration same corrections as in (16) are applied.

3.2 Starting Values and Learning Parameters

Some numbers, which characterized network and learning parameters was chosen in empirical way. Value η is the most important factor, that has direct influence result of learning. Starting values describing fuzzy sets in antecedents are random with assumption:

$$c_i^r \in [v_{i\min}, v_{i\max}] \qquad , i = 1, 2 \ldots n$$
$$\sigma_i^r \in [0.5 \tfrac{3(v_{i\max} - v_{i\min})}{N}, 1.5 \tfrac{3(v_{i\max} - v_{i\min})}{N}] \quad r = 1, 2 \ldots N \quad .$$

Values of $\overline{\mathbf{z}}$ are constant and don't change during learning:

$$\overline{z}_j^r = \begin{cases} 1 & \text{if } j - 1 = (r - 1 \bmod m) \\ 0 & \text{if } j - 1 \neq (r - 1 \bmod m)\ . \end{cases}$$

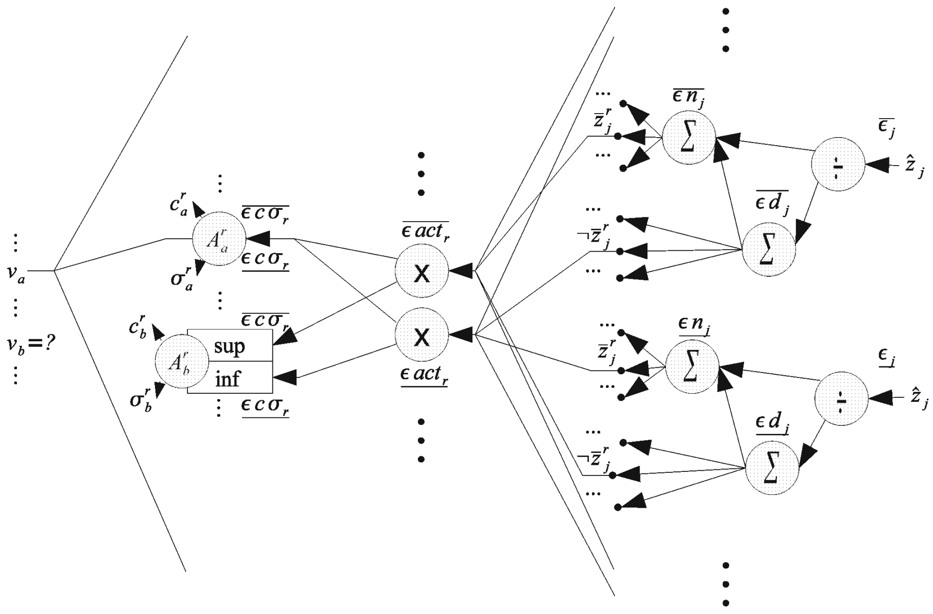

v_a
$v_b = ?$
c_a^r
σ_a^r
A_a^r
c_b^r
σ_b^r
A_b^r
sup
inf
$\overline{\epsilon c\sigma_r}$
$\underline{\epsilon c\sigma_r}$
$\overline{\epsilon act_r}$
$\underline{\epsilon act_r}$
X
$\overline{z}_j^r$
$\neg\overline{z}_j^r$
$\underline{z}_j^r$
$\neg\underline{z}_j^r$
$\overline{\epsilon n_j}$
$\overline{\epsilon d_j}$
$\underline{\epsilon n_j}$
$\underline{\epsilon d_j}$
$\overline{\epsilon_j}$
$\underline{\epsilon_j}$
$\hat{z}_j$

Fig. 1. Back-propagation algorithm in rough-neuro-fuzzy classifier with Mamdani implication and CA defuzzification

To avoid bias, when quantities of samples in each class differs, proper samples in database are duplicated to enforce same number of sample in each class. Order of samples is random. Quantity of iteration is 500 multiplied by number of samples. Selected back-propagation parameter $\eta = 0.1$. Using these values system usually achieves, at least local, optimum after about 200 multiplied by number of samples iterations.

4 Testing Procedure and Results

Implemented system was tested using three common databases from UCI Machine Learning Repository [11], i.e. Glass Identification, Iris data set and Breast Cancer Wisconsin. In the first benchmark the number of class i reduced to 2 (window glass and other), the other ones was applied without any changes.

4.1 Method of Generation Lacks in Database

To test proficiency of proposed learning algorithm mentioned databases artificially generated lacks was inducted. To simulate MCAR mechanism pseudo-random algorithm was used. This method tried to uniform missing rate in samples of each class, and avoid situation where all attributes in sample are missing and all values in attribute are missing.

4.2 Algorithm of Performance Calculation

Performance of the network is counted for database of test samples $\mathbf{V_t}$. To properly evaluate system that has not equal quantity of samples in differ classes performance is average probability of properly classified samples rate in every class:

$$perf(\mathbf{V_t}) = \frac{1}{m} \sum_{j=1}^{m} \frac{1}{\parallel \omega_j \parallel} \sum_{s:\mathbf{v}^s \in \omega_j} g\_c_t \ ,$$

where $\parallel \omega_j \parallel$ is quantity of samples in class ω_j and g_c_t indicates if t-th sample, $x(t)$, is properly classified, what is calculated as follows

$$g\_c_t = \begin{cases} 1 & \text{if } \forall j = 1, 2 \ldots m : \begin{pmatrix} \left(\overline{\overline{z_j}}(t) > 0.5 \wedge \underline{z_j}(t) > 0.5 \right) | x(t) \in \omega_j \\ \wedge \\ \left(\overline{\overline{z_j}}(t) < 0.5 \wedge \underline{z_j}(t) < 0.5 \right) | x(t) \notin \omega_j \end{pmatrix} \\ 0 & \text{else} \end{cases}$$

As an example on not proper system can be tomatoes classifier. Possible responses are: "good" or "bad". Probability that vegetable belong to first class is 0.9 and system always give that response. Applied performance's rate method gives result $\frac{1}{2}(1 + 0) = 0.5$, whilst standard, which is the rate of proper classified samples, would give 0.9 as outcome.

System was tested by 10-fold cross validation. Database was divided into 10 parts, which have nearly same, if possible same, number of samples and nearly same occurrence of each class. Test was conducted 10 times always using different part as testing set and the rest parts as learning set. For every case performance is average for each of 10 tests. Starting parameters of network and missing placement was set differently every time.

5 Final Remarks

In the paper the learning methods suitable for rough neuro-fuzzy classifiers and for learning database with missing values is proposed. It is the one of a few possible solutions - probably the simplest one. It based on the structures of rough neuro-fuzzy classifiers, which are composed, in principle, of two neuro-fuzzy systems coupled in specific way. The error measure defined in proposed method is just a sum of error measured defined typical for separate neuro-systems. Moreover, the problem of missing features in learning pattern has been solved. The experiments performed on benchmark database confirmed the effectiveness of this method in wide range of missing rate both in learning and testing patterns. The authors expect to get even better results using the method proposed in the article to the ensembles of classifiers [12], [13]. Research has shown that it is possible to learn these systems using backpropagation [14]. Especially interesting are the properties of ensembles of rough fuzzy classifiers. In future research it would be interesting to implement our methodology for learning other neuro–fuzzy systems, e.g. relational [13] [15], flexible [16], [17] and type 2 [18], [19].

Table 1. Performance of window glass identification

miss. rate in test set [%]	missing rate in learning set [%]							
	0	5	10	25	0	5	10	25
	rules per class = 1				rules per class = 2			
0	0.9240	0.9268	0.9031	0.9147	0.8591	0.9299	0.9432	0.9031
5	0.8110	0.8333	0.7982	0.8348	0.7317	0.8600	0.8606	0.8295
10	0.7248	0.7453	0.7830	0.7904	0.6431	0.7366	0.7151	0.7656
25	0.5075	0.6282	0.5570	0.5915	0.3648	0.4510	0.5335	0.6545
	rules per class = 3				rules per class = 4			
0	0.9239	0.9493	0.8999	0.9168	0.9216	0.9131	0.8793	0.8899
5	0.7570	0.8596	0.8603	0.8537	0.7879	0.8388	0.8057	0.8462
10	0.6672	0.7138	0.7481	0.8340	0.6303	0.7677	0.7061	0.7831
25	0.3058	0.4320	0.5651	0.6358	0.3503	0.5365	0.4799	0.6288

Table 2. Performance of iris classification

miss. rate in test set [%]	missing rate in learning set [%]							
	0	5	10	25	0	5	10	25
	rules per class = 1				rules per class = 2			
0	0.9200	0.8867	0.8333	0.8667	0.9533	0.9533	0.8800	0.8800
5	0.8467	0.8467	0.7333	0.7800	0.8333	0.8667	0.8067	0.8267
10	0.7600	0.8200	0.7200	0.7400	0.7333	0.8333	0.7600	0.7267
25	0.5133	0.5600	0.5267	0.4600	0.5133	0.5667	0.5267	0.5600
	rules per class = 3				rules per class = 4			
0	0.9733	0.9000	0.9067	0.9467	0.9533	0.9200	0.9533	0.9533
5	0.8600	0.8067	0.8467	0.9067	0.8133	0.8400	0.8867	0.8667
10	0.7200	0.7400	0.7667	0.7867	0.7200	0.7467	0.8067	0.7867
25	0.4400	0.5400	0.5400	0.5600	0.4733	0.5533	0.5400	0.6400

Table 3. Performance of Wisconsin breast cancer recognition

miss. rate in test set [%]	missing rate in learning set [%]							
	0.2543	5	10	25	0.2543	5	10	25
	rules per class = 1				rules per class = 2			
0.2543	0.9573	0.9576	0.9577	0.9414	0.9520	0.9648	0.9628	0.9536
5	0.8785	0.9232	0.9224	0.9170	0.8652	0.9264	0.9326	0.9210
10	0.8051	0.8580	0.8819	0.8660	0.7998	0.8726	0.8793	0.8799
25	0.6029	0.6195	0.6470	0.6538	0.5172	0.6351	0.6709	0.6746
	rules per class = 3				rules per class = 4			
0.2543	0.9575	0.9658	0.9565	0.9553	0.9564	0.9578	0.9594	0.9302
5	0.8310	0.9181	0.9237	0.9173	0.8445	0.9268	0.9273	0.8991
10	0.7203	0.8526	0.8694	0.8685	0.7319	0.8378	0.8688	0.8534
25	0.4087	0.5830	0.6573	0.7034	0.3908	0.5565	0.6388	0.6600

References

1. Rubin, D.B.: Interference and missing data. Biometrika 63, 581–592 (1976)
2. Little, R.J.A., Rubin, D.B.: Statistical analysis with missing data, 2nd edn. Wiley–Interscience (2002)
3. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the em algorithm. Journal of the Royal Statistical Society. Series B 39, 1–38 (1977)
4. Song, Q., Shepperd, M., Chen, X., Liu, J.: Can k-nn imputation improve the performance of c4.5 with small software project data sets? a comparative evaluation. The Journal of Systems & Software 81(12), 2361–2370 (2008)
5. Walczak, B., Massart, D.L.: Dealing with missing data: Part i. Chemometrics and Intelligent Laboratory Systems 58(1), 15–27 (2001)
6. Sartori, N., Salvan, A., Thomaseth, K.: Multiple imputation of missing values in a cancer mortality analysis with estimated exposure dose. Computational Statistics and Data Analysis 49(3), 937–953 (2005)
7. Nowicki, R.: Rough–neuro–fuzzy structures for classification with missing data. IEEE Trans. on Systems, Man, and Cybernetics—Part B: Cybernetics 39 (2009)
8. Dubois, D., Prade, H.: Putting rough sets and fuzzy sets together. In: Sowiski, R. (ed.) Intelligent Decision Support: Handbook of Applications and Advances of the Rough Sets Theory, pp. 203–232. Kluwer, Dordrecht (1992)
9. Nowicki, R.K.: On combining neuro–fuzzy architectures with the rough set theory to solve classification problems with incomplete data. IEEE Trans. on Knowledge and Data Engineering 20(9), 1239–1253 (2008)
10. Rutkowski, L.: New Soft Computing Techniques for System Modeling, Pattern Classification and Image Processing. Springer, Heidelberg (2004)
11. Mertz, C.J., Murphy, P.M.: UCI respository of machine learning databases, `http://www.ics.uci.edu/pub/machine-learning-databases`
12. Korytkowski, M., Rutkowski, L., Scherer, R.: From Ensemble of Fuzzy Classifiers to Single Fuzzy Rule Base Classifier. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2008. LNCS (LNAI), vol. 5097, pp. 265–272. Springer, Heidelberg (2008)
13. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)
14. Korytkowski, M., Scherer, R., Rutkowski, L.: On combining backpropagation with boosting. In: 2006 International Joint Conference on Neural Networks, Vancouver, BC, Canada, pp. 1274–1277 (2006)
15. Scherer, R.: Neuro-fuzzy Systems with Relation Matrix. In: Rutkowski, L., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2010. LNCS, vol. 6113, pp. 210–215. Springer, Heidelberg (2010)
16. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, December 2-5, vol. 3, pp. 1428–1431 (2001)
17. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)
18. Starczewski, J., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
19. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)

Diameter of the Spike-Flow Graphs of Geometrical Neural Networks

Jaroslaw Piersa

Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Torun, Poland
piersaj@mat.umk.pl

Abstract. Average path length is recognised as one of the vital characteristics of random graphs and complex networks. Despite a rather sparse structure, some cases were reported to have a relatively short lengths between every pair of nodes, making the whole network available in just several hops. This small-worldliness was reported in metabolic, social or linguistic networks and recently in the Internet. In this paper we present results concerning path length distribution and the diameter of the spike-flow graph obtained from dynamics of geometrically embedded neural networks. Numerical results confirm both short diameter and average path length of resulting activity graph. In addition to numerical results, we also discuss means of running simulations in a concurrent environment.

Keywords: geometrical neural networks, path length distribution, graph diameter, small-worldliness.

1 Introduction

Neural networks, while already turned out a worthy tool in machine learning, still constitute a potent source of knowledge about the nature of biological brain. Recently, a growing number of mathematical models were developed and studied in order to shed some light [9,10].

One of the most striking facts about brain networks is its sparsity. Recall, that number of neurons in human brain is put at 10^{11} while the number of synapses 10^{15}, clearly this places brain somewhere in-between regular lattice and a fully connected graph.

In [11] a flexible and mathematically feasible model of neural activity was put forward. It was mathematically proven and numerically confirmed, that its degree distribution obeys a power law, moreover the exponent value is close to results obtained from fMRI scans of human brain [7]. Continuing the research, we have decided to look closer on other commonly discussed features of the model. The aim of this work is to present strictly numerical findings (although, supported by random graph theory) about the average path length of spike-flow graphs, which tend emerge as a result of an self-organization process accompanying the dynamics of the system. Despite its sparsity, the graph turns out to have relatively short diameter, which again bears similarity to features reported in real brain networks [3]. However complex the structure of the network might be, it still admits passing of information between any areas within just a few edge traverses. This striking feature is believed to be one of the foundations of brain's resilience to noise and random failures [2,3]. In addition to presenting obtained results, we also provide a brief discussion about numerical details of the simulation, focusing on paralleisation possibilities.

The article is organised as follows: we present the network model and the simulation dynamics in Section 2. Than we present obtained results of the path length distribution in Section 3. Numerical, technical and implementation details are discussed in Section 4 and the paper is concluded with Section 5.

2 Simulation Model

We start with description of the underlying structural network $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of neurons and $\mathcal{E}$ stands for synaptic connections. Given a radius $R > 1$ of the two-dimensional sphere $S_2 \subset \mathbb{R}^3$ and the expected density of neurons $\rho \gg 1$ we pick number of neurons N in the system randomly from Poissonian distribution $\mathcal{P}(\rho \cdot 4\pi R^2)$, namely with parameter λ scaling as density times surface of the sphere. Each of N neurons v is than independently picked from the uniform distribution on the sphere surface. Additionally, it receives its initial *charge* $\sigma_v \geq 0$. The charge will be a subject to a dynamics, though we admit only non-negative integer values for σ_v.

A set of synapses is constructed in as follows: for each pair of neurons u, v a symmetric *synaptic connection* $\{u, v\}$ is added to $\mathcal{E}$ independently with probability

$$\mathbb{P}(\{u,v\} \in \mathcal{E}) = \begin{cases} d(u,v)^{\alpha} & d(u,v) \geq 1 \\ 1 & \text{otherwise}, \end{cases} \tag{1}$$

where $d()$ stands for euclidean distance between neurons u and v. The exponent α is fixed at the value -2.5. If successfully included, the synapse $e = \{u, v\}$ receives its *weight* w_{uv} independently generated from the gaussian distribution $w_{uv} \sim N(0, s^2)$. The weight indicates an excitatory or inhibitory (when negative) type of the synapse.

A network *energy* function depending on the charge stored in each of neurons is defined as

$$E(\bar{\sigma}) = \sum_{(u,v) \in \mathcal{E}} w_{u,v} |\sigma_v - \sigma_u|. \tag{2}$$

Having presented the structural network, we describe the dynamics. We adopt a variation of the celebrated Ising spin glass model, though extended for multiple values of σ.

– At each step we pick a random pair of neurons $u, v \in \mathcal{V}$, such that they are connected with a synapse $\{u, v\} \in \mathcal{E}$ and the charge stored in u is non-zero $\sigma_u \geq 1$,

– We try to transfer a single unit of charge from u to v through the synapse, in other words $\sigma_u := \sigma_u - 1$ and $\sigma_v := \sigma_v + 1$,
– if this transfer reduces energy of the system, it is accepted, and the dynamics proceeds to next iteration,
– if this transfer increases the energy by ΔE, then it is accepted with probability $\mathbb{P}(u \to v) = \exp(-\beta \Delta E)$ and rejected otherwise.

The parameter $\beta \gg 1$ stands for an inverse temperature and is assumed to be large. Though the dynamics can run arbitrarily long, it is terminated after hitting given number of iterations or reaching the state, where no further transfers are accepted.

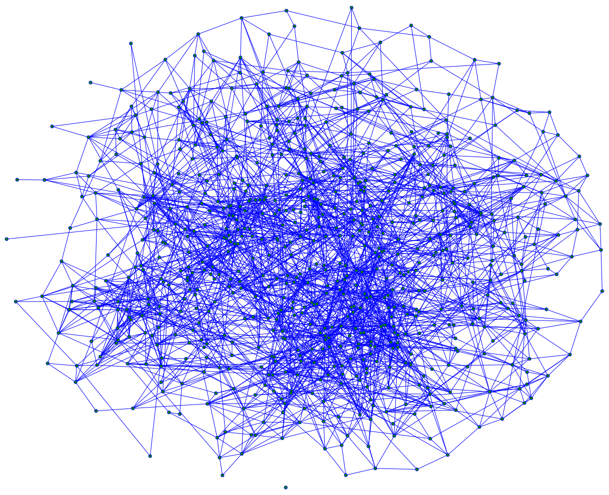


Fig. 1. A subgraph of resulting network, plot includes 719 neurons and 2500 synapses. Spatial coordinates were remapped from a sphere to a surface for better visibility. Despite a small sample, hubs (node with large degree) are clearly noticeable.

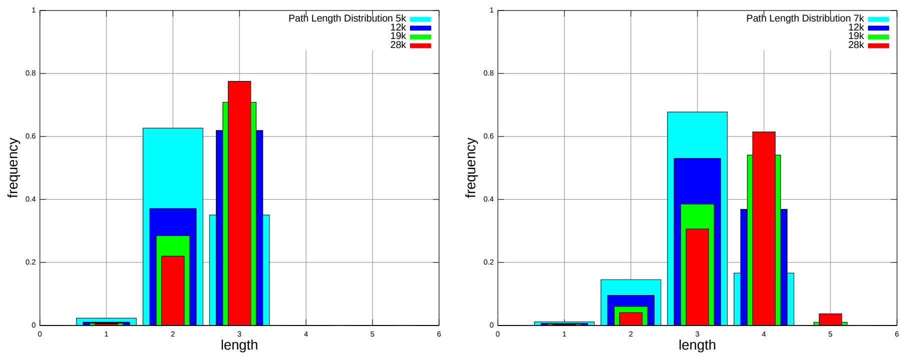
3 Path Length Distribution Results

As pointed out in [11] the the dynamics leads to concentration of the charge in small number of nodes. For each synapse $e = \{u, v\}$ a total amount of charge which flew either from u to v or from v to u is recorded and denoted as d_e. Reaching a predefined threshold value θ qualifies an edge as a vital in the evolution process. A graph built from the vital edges will be referred as a *spike flow graph*, namely $\mathcal{G}_1 = (\mathcal{V}, \mathcal{E}_1 = \{e \in \mathcal{E} : d_e \geq \theta\})$. A value $\theta = 1$ is assumed throughout rest of the work, which results in spike flow graph consisting of all edges participating in the dynamics. A small subgraph of the resulting spike flow network is depicted on Fig. 1.

As a *minimal path length* between neurons u and v we understand classical definition, i.e. $l(u, v)$ is a minimum number of edges $(e_1 = \{u_1, v_1\}, .., e_n = \{u_n, v_n\})$, such that

– they start at u and terminate at v, $u_1 = u$, $v_n = v$,
– they are incident, $\forall_{i=1..n-1} v_i = u_{i+1}$,
– they are included in resulting spike flow graph $\forall_i e_i \in \mathcal{E}_1$.

For the sake of simplicity we assume $l(u, u) = 0$.



(a) $\alpha = -2.5$ (b) $\alpha = -3.5$

Fig. 2. A plot of path length distribution of the spike-flow graph obtained from a simulation of networks counting 5 to 28k neurons. The distribution slowly increases its mode as the number of nodes grows.

Obtained results of the path lengths are presented on Fig. 2. The results were collected from simulations of networks counting between 300 and up to $5.5 \cdot 10^4$ neurons for exponent of the connectivity function (Eq. 1) $\alpha = -2.5$ and $\alpha = -3.5$. As shown in [12] a resulting spike-flow graph obeys a power law degree distribution, this type of graphs also frequently exhibits a small world phenomenon [2].

Fig. 3 presents the dependency between the size of the simulation sample and corresponding average path value. It presents a slow growth, but in all of our cases is still bounded by 4. A slow increase of the average path length is observable, which again is in agreement with theoretical results of classical random graphs theory. Also note, that for our small samples the path length distribution is focused on 3 values. The the rest (number of paths of length up to the graph diameter — see below) is non zero, though negligible.

The *diameter of the graph* is defined as a maximum value of shortest path length $l(u, v)$ between every pair of nodes (u, v) in the graph. Its plot is provided on Fig 3. Recall that for Erdős-Réyni random graph model the diameter grows logarithmically as the network size increases, see [5] Ch. 5 and 7 for rigorous proof. Clearly the diameter can be arbitrarily larger than an average path length,

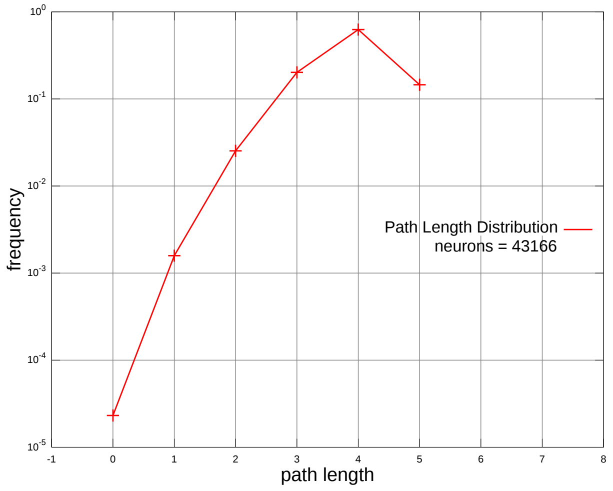


Fig. 3. A semi-log plot of the path length distribution of the spike-flow graph. The number of paths of length 0 (selfloops) and 1, though negligible, is non zero. Number of neurons $N \simeq 40$k, $\alpha = -3.5$.

however in our cases it is bounded by 6 for our samples, which means that starting in any neuron, the whole network is available in no more than 6 hops. While our model is not a ER model, this feature make it strikingly similar to WWW [1] or social graphs [6]. A few reports suggest small-worldliness to be present also in brain activity networks [3].

Small discrepancy between l for different values of α seems to originate from difference in underlying structural graphs. Lower α yields less synapses in the structural network $\mathcal{G}$, while the spike-flow graph consists exclusively of synapses present in $\mathcal{G}$. The formula of g was suggested in [7].

While the clustering coefficient is an aim of ongoing research we can provide a brief preliminary results concerning this value. After classical theory [5] we define a *clustering coefficient of the node* v as a ratio of existing edges in the neighbourhood of v to all possible edges.

$$C(u) = \frac{|\{e = (w,v) : e \in \mathcal{E} \wedge (w,u) \in \mathcal{E} \wedge (v,u) \in \mathcal{E}\}|}{|\{(w,v) : (w,u) \in \mathcal{E} \wedge (v,u) \in \mathcal{E}\}|} \tag{3}$$

The *clustering coefficient of the graph* C is defined as an average of clustering coefficients of its nodes.

$$C = \frac{1}{|\mathcal{V}|} \sum_{u \in \mathcal{V}} C(u) \tag{4}$$

Recall, that for ER graph model the clustering coefficient is relatively small (it is equal to the average connectivity of the graph). As it can be see in Table 1 for spike-flow graph in our model the actual clustering coefficient is 2-3 orders of magnitude higher then its average degree, which again is in agreement with fMRI data [3].

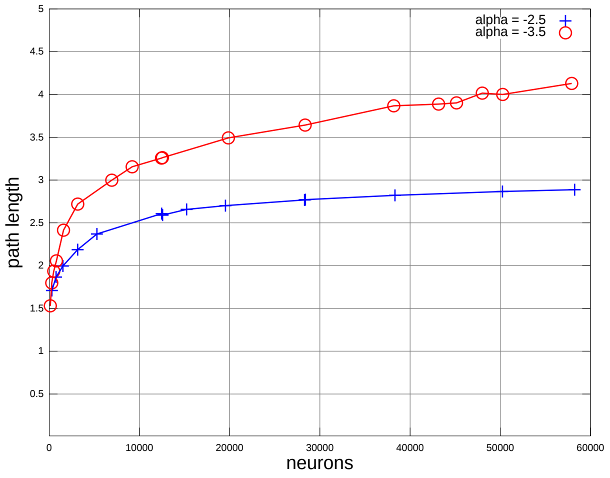


Fig. 4. A plot of the average path length of the spike-flow graph vs sample size

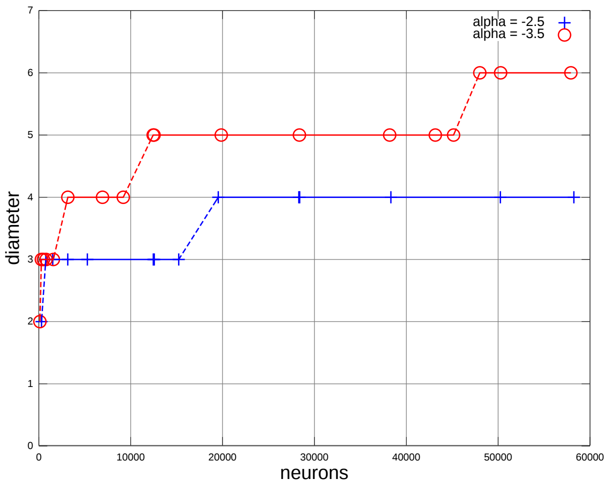


Fig. 5. A plot of the diameter (maximum path length connecting a pair of nodes) of the spike-flow graph vs sample size. Note, that slope segments indicate values, which are not known (diameter must be an integer number).

Table 1. Calculated clustering coefficient and average connectivity for obtained spike-flow graphs, $\alpha = -2.5$. Note that the *average degree* is also a theoretical clustering coefficient for equivalent ER random graph, while by *clustering coefficient* we mean an actual, numerically obtained value for our model.

Neurons	Synapses	Average degree	Clustering coefficient
3155	0.18M	0.038	0.243
12568	0.8M	0.010	0.207
19531	1.3M	0.006	0.198
24630	1.7M	0.005	0.194
28501	2.0M	0.004	0.191
38322	2.6M	0.0035	0.187
50241	3.5M	0.0027	0.183
58244	4.1M	0.0023	0.182

4 Numerical Details

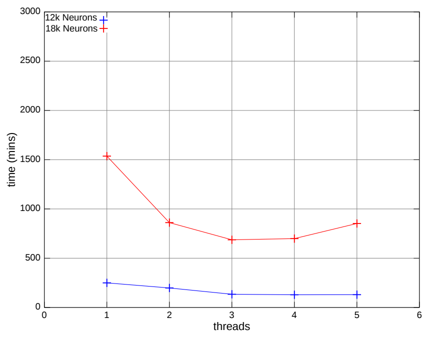
The simulations were carried on spheres with radii varying up to $R = 30$ with constant density $\rho = 10$, see Sec. 2. This yields samples counting up to 55k neurons.Additionally, we expect an average 500 units of charge amount per neuron to be present in the network. In most cases a strict limit of iterations turned out a sufficient terminating condition.

The simulation itself was paralleled to take advantage of multiprocessing environment. However, due to highly unpredictable dynamics and the usage of control-flow instructions one finds particularly demanding to put graphical computing units into a good use [12]. We decided to implement a task division, i.e. splitting the total number of iterations among parallel threads. Sparse though the resulting network might be, it still requires visiting all of the neighbour neurons before deciding whether (or not) to accept the transfer. Along with growing number of threads it increases frequency of waiting for the lock to be released and thus affects the speedup. The overall efficiency was about 66% for three threads.

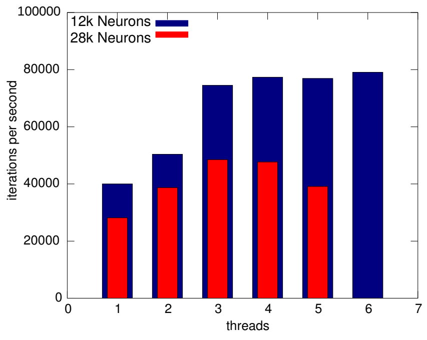
For calculation of the path length distribution a classical Dijkstra algorithm has been adapted with a domain-division paralleisation. The results of calculation times are presented on Fig. 4. The computation time seems reasonable, when take into account sequential collecting data from threads. Obtained efficiency is about .78 for 4 threads, which is the number of computing cores. The results were obtained on Core Quad 2GHz CPU + 4GB RAM + Fedora 11–14 (32bit, PAE) system.

Some of the simulations were also run of the infrastructure of the PG-Grid, who kindly provided their computing resources. The timing and speedup results from those simulations, while still being collected and analysed, are beyond the scope of this work.

Additionally a Monte Carlo was also implemented, however we found the time, required to yield a results with a satisfactory precision, comparative to the time required by the former algorithm. Although, for larger samples MC may turn out indispensable.

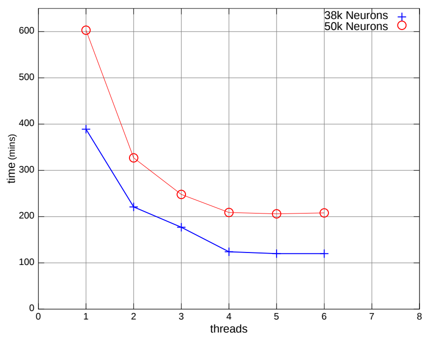


(a) Computation time

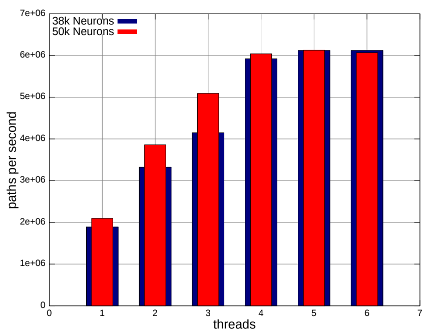


(b) Iterations per second

Fig. 6. Computation times obtained for the dynamics. Reprinted from [12].



(a) Computation time.



(b) Number of paths calculated per second.

Fig. 7. Computation times obtained and number of path lengths calculated per second for sample 38k neurons. The results were obtained on 4 cores CPU.

5 Conclusion and Future Work

In this work we have presented our findings concerning path length distribution in spike-flow graphs of neural network. The distribution turns out concentrated on few values and its mean value is relatively low, making the whole network available to reach within just a few hops. Along with high clicqueliness determined by the clustering coefficient, this suggests a complicated structure of the resulting network, even despite not so complicated dynamics. Clearly, the network bears resemblances to social graphs or WWW networks, which also tend to exhibit a small-world phenomenon.

It might turn out interesting to see how the network evolves to its final shape throughout the simulation. This unfortunately requires either more computational power or reduction of the size of the network, in order to keep the simulation time reasonable. As an additional aim of future work, we point at considering a directed version of the spike flow graph, which is natural feature of WWW networks [1]. Since the dynamics in our network is directed, the distinction between a input and output synapses seems to be a forgone conclusion. One more vital aspect frequently discussed when working with complex networks is clustering coefficient [2,3]. While it was bearly mentioned in in Sec. 3, a coherent comparison between the results obtained from fMRI and our model is a focus of ongoing research.

Acknowledgements. The author would like to mention the unparalleled support from prof. Tomasz Schreiber, Associate Professor at Faculty of Mathematics and Computer Science, NCU, a bright young scientist, who carried researches in mathematics, computer science and physics, prematurely died in December 2010.

The author acknowledges a cooperation and a dash of red ink from dr Filip Piękniewski.

Additionally, the author is grateful to PL-Grid Project[1] for providing a computing infrastructure for simulations.

The work has been partially supported by Polish Ministry of Science and grant project 2011/01/N/ST6/01931.

The author would also like to acknowledge the support of the European Social Fund and Government of Poland as a part of Integrated Operational Program for Regional Development, ZPORR, Activity 2.6 ”Regional Innovation Strategies and Knowledge Transfer” of Kuyavian-Pomeranian Voivodeship in Poland.

References

1. Albert, R., Jeong, H., Barabasi, A.L.: Diameter of the World-Wide Web. Nature 401 (September 9, 1999)
2. Albert, R., Barabasi, A.L.: Statistical mechanics of complex networks. Reviews of Modern Physics 74 (January 2002)
3. Bassett, D.S., Bullmore, E.: Small-World Brain Networks. The Neuroscientist 12(6) (2006)
4. Bullmore, E., Sporns, O.: Complex brain networks: graph theoretical analysis of structural and functional systems. Nature Reviews, Neuroscience 10 (March 2009)
5. Chung, F., Lu, L.: Complex graphs and networks. In: Conference Board of the Mathematical Sciences. American Mathematical Society (2006)
6. Csermely, P.: Weak links: the universal key to the stability of networks and complex systems. Springer, Heidelberg (2009)

[1] `http://www.plgrid.pl`

7. Eguiluz, V., Chialvo, D., Cecchi, G., Baliki, M., Apkarian, V.: Scale-free brain functional networks. Physical Review Letters, PRL 94, 018102 (2005)
8. Piekniewski, F.: Spontaneous scale-free structures in spike flow graphs for recurrent neural networks. Ph.D. dissertation, Warsaw University, Warsaw, Poland (2008)
9. Piekniewski, F., Schreiber, T.: Spontaneous scale-free structure of spike flow graphs in recurrent neural networks. Neural Networks 21(10), 1530–1536 (2008)
10. Piekniewski, F.: Spectra of the Spike Flow Graphs of Recurrent Neural Networks. In: Alippi, C., Polycarpou, M., Panayiotou, C., Ellinas, G. (eds.) ICANN 2009, Part II. LNCS, vol. 5769, pp. 603–612. Springer, Heidelberg (2009)
11. Piersa, J., Piekniewski, F., Schreiber, T.: Theoretical model for mesoscopic-level scale-free self-organization of functional brain networks. IEEE Transactions on Neural Networks 21(11) (November 2010)
12. Piersa, J., Schreiber, T.: Scale-free degree distribution in information-flow graphs of geometrical neural networks. Simulations in concurren environment (in Polish). Accepted for Mathematical Methods in Modeling and Analysis of Concurrent Systems — Postproceedings, Poland (July 2010)
13. Schreiber, T.: Spectra of winner-take-all stochastic neural networks, arXiv 3193(0810), pp. 1–21 (October 2008), http://arxiv.org/PS_cache/arxiv/pdf/0810/0810.3193v2.pdf
14. Watts, D., Strogatz, S.: Collective dynamics of 'small-world' networks. Nature 393, 440–442 (1998)

Weak Convergence of the Recursive Parzen-Type Probabilistic Neural Network in a Non-stationary Environment

Lena Pietruczuk[1] and Jacek M. Zurada[2]

[1] Department of Computer Engineering, Czestochowa University of Technology, ul. Armii Krajowej 36, 42-200 Czestochowa, Poland
lena.pietruczuk@kik.pcz.pl

[2] Department of Electrical and Computer Engineering, University of Louisville, 405 Lutz Hall, Louisville, KY 40292, USA
jacek.zurada@louisville.edu

Abstract. A recursive version of general regression neural networks is presented. Weak convergence is proved in a case of nonstationary noise. Experimental results are given.

1 Introduction

Probabilistic neural networks proposed by Specht [35] are net structures corresponding to nonparametric density and regression estimates developed to solve stationary (see e.g. [2], [4], [6], [7], [9], [13]-[16], [22]-[24], [27]-[30], [38] and [39]) and nonstationary problems (see e.g. [8], [17]-[21], [25] and [26]). In a letter case it was assumed in literature that noise was stationary.

Let us consider the following system

$$Y_i = \phi(X_i) + Z_i,\ i = 1, \ldots, n \tag{1}$$

where $X_1, \ldots, X_n$ is a sequence of independent and identically distributed variables in R^p with probability density function f, ϕ is an unknown function and $Z_1, \ldots, Z_n$ are independent random variables with unknown distributions such that

$$E[Z_i] = 0, \qquad Var[Z_i] = d_i, \qquad \text{for } i = 1 \ldots, n. \tag{2}$$

It should be emphasized that the variance of Z_i is not equal for all i. The problem is to estimate function ϕ, in the case of time varying noise Z_i. In this paper we will apply the Parzen kernel-type regression neural network and prove its convergence even if variance of noise diverges to infinity.

2 Probabilistic Neural Network

To estimate the regression function $\phi(x)$ we use the recursive version of Parzen kernel procedures

$$K_n(x, u) = h_n^{-p} K\left(\frac{x - u}{h_n}\right), \tag{3}$$

$$K'_n(x,u) = h_n'^{-p} K\left(\frac{x-u}{h'_n}\right), \tag{4}$$

where K is an appropriately selected function in the form such that

$$||K||_\infty < \infty, \tag{5}$$

$$\int |K(y)|dy < \infty, \tag{6}$$

$$\lim_{y\longrightarrow R} |yK(y)| = 0, \tag{7}$$

$$\int_R K(y)dy = 1 \tag{8}$$

and h_n, h'_n are certain sequences of numbers. Let

$$\hat{\phi}_n(x) = \frac{\hat{R}_n(x)}{\hat{f}_n(x)} \tag{9}$$

be the estimator of regression function

$$\phi_(x) = \frac{R_(x)}{f_(x)} \tag{10}$$

where $R(x) = f(x)\phi(x)$. The estimators $\hat{f}_n$ and $\hat{R}_n$ are in the form

$$\hat{f}_i(x) = \frac{1}{n}\sum_{i=1}^{n} K_i(x, X_i). \tag{11}$$

and

$$\hat{R}_i(x) = \frac{1}{n}\sum_{i=1}^{n} Y_i K_i(x, X_i). \tag{12}$$

Then estimator $\hat{\phi}_n(x)$ takes the form

$$\hat{\phi}_n(x) = \frac{\sum_{i=1}^{n} Y_i h_i^{-p} K(\frac{x-X_i}{h_i})}{\sum_{i=1}^{n} h_i^{-p} K(\frac{x-X_i}{h'_i})} \tag{13}$$

which is known in the literature under the name probabilistic neural network [35]. Observe that procedures (11) and (12) can be presented in the following form

$$\hat{f}_i(x) = \frac{1}{n}\sum_{i=1}^{n} h_i^{-p} K\left(\frac{x-X_i}{h_i}\right) \tag{14}$$

and

$$\hat{R}_i(x) = \frac{1}{n}\sum_{i=1}^{n} Y_i h_i^{-p} K\left(\frac{x-X_i}{h_i}\right). \tag{15}$$

The block diagram of recursive generalized regression neural network is presented in Fig. 1.

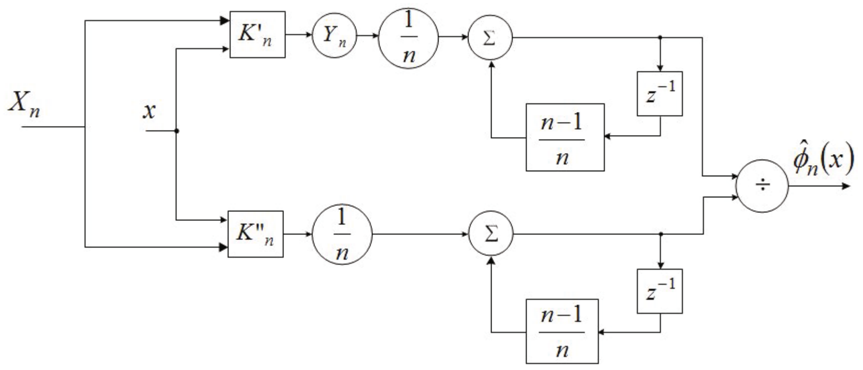


Fig. 1. The block diagram of recursive GRNN

Theorem 1. *If*

$$s_i = \sup_x\{(\sigma_i^2 + \phi^2(x))f(x)\} < \infty,\ i = 1, \dots, n \tag{16}$$

$$||x||^p K(x) \longrightarrow 0, \qquad ||x|| \longrightarrow \infty, \tag{17}$$

$$n^{-2}\sum_{i=1}^{n} h_i'^{-p} \to 0, \tag{18}$$

$$n^{-2}\sum_{i=1}^{n} h_i^{-p} s_i \to 0, \tag{19}$$

$$h_i' \to 0, \qquad h_i \to 0, \tag{20}$$

then $\phi_n(x) \xrightarrow{n} \phi(x)$ *in probability.*

Proof. It is sufficient to show that

$$\hat{f}_n(x) \longrightarrow f(x) \quad \text{with probability one} \tag{21}$$

and

$$\hat{R}_n(x) \longrightarrow R(x) \quad \text{with probability one.} \tag{22}$$

Convergence (21) under condition (18) was proved in [2]. Therefore it is enough to show that (22) holds.

Obviously

$$\left|\hat{R}_n(x) - R(x)\right| \le \left|\hat{R}_n(x) - E\left[\hat{R}_n(x)\right]\right| + \left|E\left[\hat{R}_n(x)\right] - R(x)\right|. \tag{23}$$

Observe that

$$Var(\hat{R}_n(x)) = Var(n^{-1}\sum_{i=1}^{n} Y_i K_i(x, X_i)) = n^{-2}\sum_{i=1}^{n}\left(Var\left[h_i^{-p} Y_i K\left(\frac{x - X_i}{h_i}\right)\right]\right) \le \tag{24}$$

$$\leq n^{-2}\sum_{i=1}^{n} h_i^{-p}\int_{R^p} E\left[Y_i^2|X_i=u\right] f(u)K^2\left(\frac{x-X_i}{h_i}\right)du \leq \tag{25}$$

$$\leq n^{-2}\sum_{i=1}^{n} h_i^{-p}\int_{R^p} (\sigma_i^2+\phi^2(x))f(u)||K||_\infty\left(\frac{x-X_i}{h_i}\right)du \leq \tag{26}$$

$$\leq n^{-2}\sum_{i=1}^{n} h_i^{-p}2s_i||K||_\infty. \tag{27}$$

Therefore $Var(\hat{R}_n(x)) \xrightarrow{n} 0$ if condition (19) holds and, consequently, $|\hat{R}_n(x) - ER_n(x)| \xrightarrow{n} 0$ in probability. Since $|E\hat{R}_n(x) - R(x)| \xrightarrow{n} 0$ under condition (17) (see [2]), then the result is established.

2.1 Example for Specific Sequence $\mathbf{d_n}$

We consider the case when $d_n = O(n^\alpha)$ for $\alpha > 0$. We choose the sequences h'_n and h_n to be in the form

$$h'_n = D'n^{-H'}, \qquad h_n = Dn^{-H}, \tag{28}$$

where D, D', H and H' are constants. Then conditions (18)-(20) are satisfied if

$$H' < 1 \text{ and } H + \alpha < 1. \tag{29}$$

3 Simulation Results

In the following simulations we estimate the regression of the function

$$\phi(x) = x\sin(x). \tag{30}$$

Let us assume that input data come from the normal distribution with the mean equal to 0 and the standard deviation equal to 3.9. First we select the kernel function. For the purpose of this paper we choose triangular kernel and assume that $h'_n = h_n$.

By examining the relationship between the value of the mean square error and the value of the parameter $H = H'$ in (28) we obtained the results illustrated in Fig. 2 for $D = D' = 6$ and $\alpha = 0.35$. The experiment was performed on the set of 6500 data. As we can see with increasing value of H the error of the algorithm decreases. For $H = 0$ the mean square error (MSE) is equal to $4,6158$ and for $H = 0.3$ the value of this error is equal to $0,0501$.

In Figure (3) we show the dependence between the number of elements and the value of the MSE. We assume that $H = H' = 0.4$, $D = D' = 1$ and $\alpha = 0.3$. Even with increasing variance of Z_n, corresponding to the increasing number of elements, the accuracy of the algorithm improves. For $n = 6000$ the MSE is equal to 0.0705 and for $n = 50000$ it decreases to 0.0331.

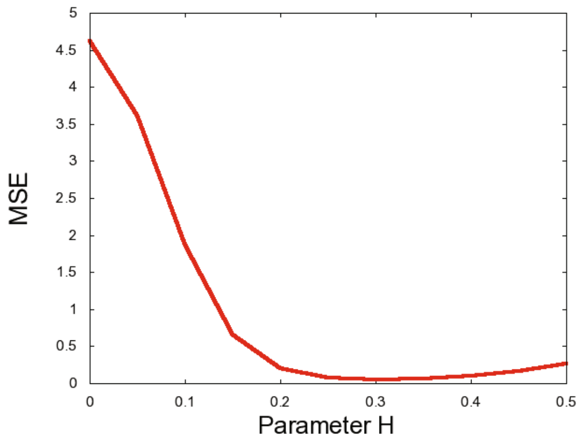


Fig. 2. The dependence between the value of parameter H and the value of the MSE

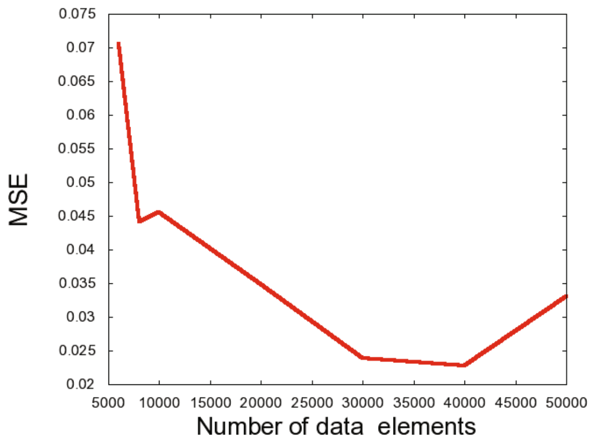


Fig. 3. The dependence between the number of data elements N and the value of the MSE

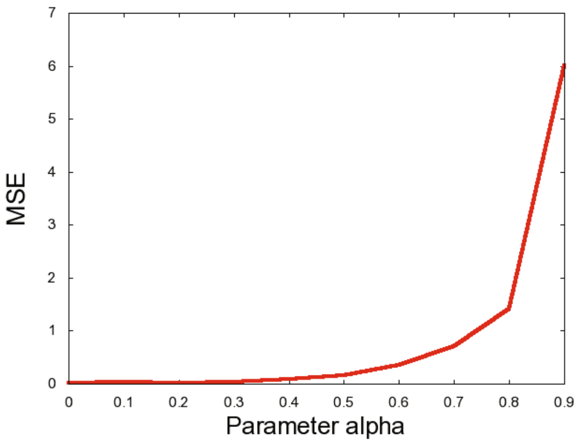


Fig. 4. The dependence between the value of parameter α and the value of the MSE

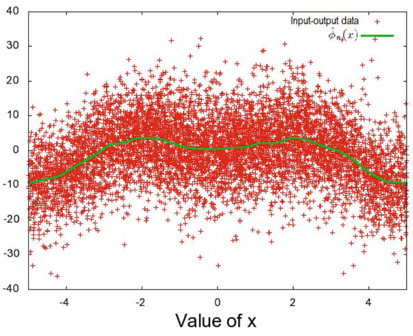


Fig. 5. The obtained value of estimator for $x = 1$ and different number of data elements n

In Figure (4) we show the dependence between the value of parameter α and the MSE. In this case $n = 5500$, $D = 5.9$ and $H = H' = 0.29$. From conditions (29) we can see, that for this value of H and H', value of α should not be bigger than 0.71. For small value of α the error is small and for α outside the permissible range the error increases very fast.

In Figure (5) we can see how the value of $\phi_n(x)$ is changing for $x = 1$ with increasing value of n. The horizontal line show the value of $\phi(x)$ for this x.

In Figure (6) we can see input-output data with $\alpha = 0.5$ and obtained estimator values. Input data are from the normal distribution $\mathcal{N}(0, (4.9)^2)$. In this experiment $H = H' = 0.3$, $D = D' = 6.2$ and $n = 10000$.

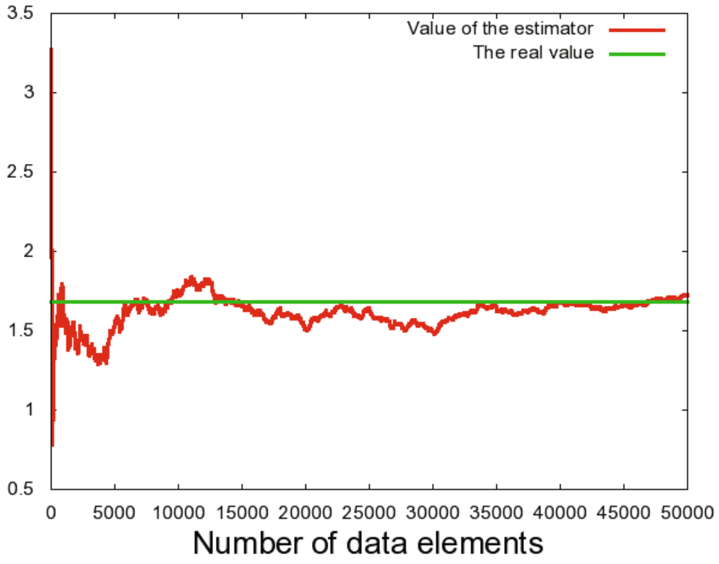


Fig. 6. The input-output data and the obtained estimator values

4 Conclusion and Future Work

In this paper we presented a recursive version of general regression neural networks and we proved the weak convergence in a case of nonstationary noise. In the future work the application of neuro-fuzzy structures [10], [31]-[34], [36], [37] or supervised and unsupervised neural networks [3], [5], [12] can be study in the case of non-stationary environment. Finaly, it should be noted that procedure (9) can be applied to data streams [1].

Acknowledgments. This paper was prepared under project operated within the Foundation for Polish Science Team Programme co-financed by the EU European Regional Development Fund, Operational Program Innovative Economy 2007-2013, and also supported by the National Science Center NCN.

References

1. Aggarwal, C.: Data Streams. Models and Algorithms. Springer, New York (2007)
2. Ahmad, I.A., Lin, P.E.: Nonparametric sequential estimation of multiple regression function. Bulletin of Mathematical Statistics 17, 63–75 (1976)
3. Bilski, J., Rutkowski, L.: A fast training algorithm for neural networks. IEEE Transactions on Circuits and Systems II 45, 749–753 (1998)
4. Chu, C.K., Marron, J.S.: Choosing a kernel regression estimator. Statistical Science 6, 404–436 (1991)

5. Cierniak, R., Rutkowski, L.: On image compression by competitive neural networks and optimal linear predictors. Signal Processing: Image Communication - a Eurasip Journal 15(6), 559–565 (2000)
6. Gałkowski, T., Rutkowski, L.: Nonparametric recovery of multivariate functions with applications to system identification. Proceedings of the IEEE 73, 942–943 (1985)
7. Gałkowski, T., Rutkowski, L.: Nonparametric fitting of multivariable functions. IEEE Transactions on Automatic Control AC-31, 785–787 (1986)
8. Greblicki, W., Rutkowska, D., Rutkowski, L.: An orthogonal series estimate of time-varying regression. Annals of the Institute of Statistical Mathematics 35, Part A, 147–160 (1983)
9. Greblicki, W., Rutkowski, L.: Density-free Bayes risk consistency of nonparametric pattern recognition procedures. Proceedings of the IEEE 69(4), 482–483 (1981)
10. Nowicki, R.: Rough Sets in the Neuro-Fuzzy Architectures Based on Non-monotonic Fuzzy Implications. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 518–525. Springer, Heidelberg (2004)
11. Parzen, E.: On estimation of a probability density function and mode. Analysis of Mathematical Statistics 33(3), 1065–1076 (1962)
12. Patan, K., Patan, M.: Optimal training strategies for locally recurrent neural networks. Journal of Artificial Intelligence and Soft Computing Research 1(2), 103–114 (2011)
13. Rutkowski, L.: Sequential estimates of probability densities by orthogonal series and their application in pattern classification. IEEE Transactions on Systems, Man, and Cybernetics SMC-10(12), 918–920 (1980)
14. Rutkowski, L.: Sequential estimates of a regression function by orthogonal series with applications in discrimination, New York, Heidelberg, Berlin. Lectures Notes in Statistics, vol. 8, pp. 236–244 (1981)
15. Rutkowski, L.: On system identification by nonparametric function fitting. IEEE Transactions on Automatic Control AC-27, 225–227 (1982)
16. Rutkowski, L.: Orthogonal series estimates of a regression function with applications in system identification. In: Probability and Statistical Inference, pp. 343–347. D. Reidel Publishing Company, Dordrecht (1982)
17. Rutkowski, L.: On Bayes risk consistent pattern recognition procedures in a quasi-stationary environment. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4(1), 84–87 (1982)
18. Rutkowski, L.: On-line identification of time-varying systems by nonparametric techniques. IEEE Transactions on Automatic Control AC-27, 228–230 (1982)
19. Rutkowski, L.: On nonparametric identification with prediction of time-varying systems. IEEE Transactions on Automatic Control AC-29, 58–60 (1984)
20. Rutkowski, L.: Nonparametric identification of quasi-stationary systems. Systems and Control Letters 6, 33–35 (1985)
21. Rutkowski, L.: The real-time identification of time-varying systems by nonparametric algorithms based on the Parzen kernels. International Journal of Systems Science 16, 1123–1130 (1985)
22. Rutkowski, L.: A general approach for nonparametric fitting of functions and their derivatives with applications to linear circuits identification. IEEE Transactions Circuits Systems CAS-33, 812–818 (1986)
23. Rutkowski, L.: Sequential pattern recognition procedures derived from multiple Fourier series. Pattern Recognition Letters 8, 213–216 (1988)

24. Rutkowski, L.: Nonparametric procedures for identification and control of linear dynamic systems. In: Proceedings of 1988 American Control Conference, June 15-17, pp. 1325–1326 (1988)
25. Rutkowski, L.: An application of multiple Fourier series to identification of multivariable nonstationary systems. International Journal of Systems Science 20(10), 1993–2002 (1989)
26. Rutkowski, L.: Nonparametric learning algorithms in the time-varying environments. Signal Processing 18, 129–137 (1989)
27. Rutkowski, L., Rafajłowicz, E.: On global rate of convergence of some nonparametric identification procedures. IEEE Transaction on Automatic Control AC-34(10), 1089–1091 (1989)
28. Rutkowski, L.: Identification of MISO nonlinear regressions in the presence of a wide class of disturbances. IEEE Transactions on Information Theory IT-37, 214–216 (1991)
29. Rutkowski, L.: Multiple Fourier series procedures for extraction of nonlinear regressions from noisy data. IEEE Transactions on Signal Processing 41(10), 3062–3065 (1993)
30. Rutkowski, L., Gałkowski, T.: On pattern classification and system identification by probabilistic neural networks. Applied Mathematics and Computer Science 4(3), 413–422 (1994)
31. Rutkowski, L.: A New Method for System Modelling and Pattern Classification. Bulletin of the Polish Academy of Sciences 52(1), 11–24 (2004)
32. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, Melbourne, December 2-5, vol. 3, pp. 1428–1431 (2001)
33. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)
34. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)
35. Specht, D.F.: A general regression neural network. IEEE Transactions Neural Networks 2, 568–576 (1991)
36. Starczewski, L., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
37. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)
38. Wegman, E.J., Davies, H.I.: Remarks on some recursive estimators of a probability density. Annals of Statistics 7, 316–327 (1979)
39. Yamato, H.: Sequential estimation of a continuous probability density function and the mode. Bulletin of Mathematical Statistics 14, 1–12 (1971)

Strong Convergence of the Parzen-Type Probabilistic Neural Network in a Time-Varying Environment

Lena Pietruczuk[1] and Meng Joo Er[2]

[1] Department of Computer Engineering, Czestochowa University of Technology, ul. Armii Krajowej 36, 42-200 Czestochowa, Poland
lena.pietruczuk@kik.pcz.pl

[2] Nanyang Technological University, School of Electrical and Electonic Engineering, 50 Nanyang Avenue, Singapore 639798
emjer@ntu.edu.sg

Abstract. In this paper general regression neural networks are applied to handle nonstationary noise. Strong convergence is established. Experiments conducted on synthetic data show good performance in the case of finite length of data samples.

1 Introduction

In 90's a concept of probabilistic neural networks and general regression neural networks was introduced by Specht [34],[35]. Both structures are implementations of nonparametric estimates working in a stationary (see e.g. [3], [4], [7], [12]-[15], [21]-[23] and [26]-[29]) and nonstationary environment (see [6], [16]-[20], [24] and [25]). In this paper we will consider the problem of estimation of the regression function with non-stationary noise Z_n. The system is of the form

$$Y_i = \phi(X_i) + Z_i, \tag{1}$$

where $X_1, \ldots, X_n$ is a sequence of some independent and equally distributed variables in R^p with probability density function f, ϕ is an unknown function and $Z_1, \ldots, Z_n$ are independent random variables with unknown distributions such that:

$$E[Z_i] = 0, \qquad Var[Z_i] = d_i, \qquad \text{for } i \geq 1. \tag{2}$$

The problem is to estimate the regression function ϕ assuming different values of variances of random variables Z_i. In this paper the regression estimate is based on the Parzen kernel. We will prove its strong convergence even if variance of noise Z_i diverges to infinity.

2 Algorithm and Main Result

For estimation of the regression function $\phi(x)$ we use the Parzen kernel estimate. Le us define

$$K_n(x,u) = h_n^{-p} K\left(\frac{x-u}{h_n}\right), \tag{3}$$

$$K_n'(x,u) = h_n^{'-p} K\left(\frac{x-u}{h_n'}\right), \tag{4}$$

where the kernel K satisfies the conditions

$$||K||_\infty < \infty, \tag{5}$$

$$\int |K(y)|dy < \infty, \tag{6}$$

$$\lim_{y \longrightarrow \infty} |yK(y)| = 0, \tag{7}$$

$$\int_{R^p} K(y)dy = 1 \tag{8}$$

and h_n, h_n' are certain sequences of numbers. Let

$$\hat{\phi}_n(x) = \frac{\hat{R}_n(x)}{\hat{f}_n(x)} \tag{9}$$

be the estimator of the regression function

$$\phi(x) = \frac{R(x)}{f(x)}. \tag{10}$$

The functions $\hat{f}_n$ and $\hat{R}_n$ are in the form

$$\hat{f}_n(x) = \frac{1}{n} \sum_{i=1}^{n} K_n(x, X_i) \tag{11}$$

and

$$\hat{R}_n(x) = \frac{1}{n} \sum_{i=1}^{n} Y_i K_n(x, X_i). \tag{12}$$

Then estimator $\hat{\phi}_n(x)$ takes the form

$$\hat{\phi}_n(x) = \frac{\sum_{i=1}^{n} Y_i K(\frac{x-X_i}{h_n})}{\sum_{i=1}^{n} K(\frac{x-X_i}{h_n'})} \tag{13}$$

which is known in the literature under the name probabilistic neural network [34]. The block diagram of generalized regression neural network is shown in Fig. 1.

Theorem 1. *If*

$$s_i = \sup_x \{(\sigma_i^2 + \phi^2(x))f(x)\} < \infty, \; i = 1, \ldots, n \tag{14}$$

$$||x||^p K(x) \longrightarrow 0 \qquad ||x|| \longrightarrow \infty, \tag{15}$$

$$\sum_{n=1}^{\infty} n^{-2} h_n'^{-p} < \infty, \tag{16}$$

$$\sum_{n=1}^{\infty} n^{-2} h_n'^{-p} s_n < \infty, \tag{17}$$

$$h_n' \to 0, \qquad h_n \to 0, \tag{18}$$

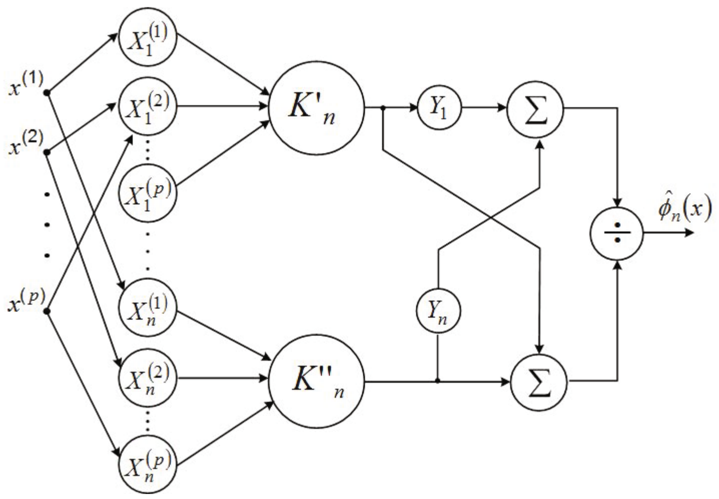


Fig. 1. Generalized regression neural network

then $\phi_n(x) \xrightarrow{n} \phi(x)$ *with probability one for each* x *where* $\phi(x)$ *is continuous.*

Proof. It is sufficient to show that

$$\hat{f}_n(x) \longrightarrow f(x) \quad \text{with probability one for each x where } f(x) \text{ is continuous} \tag{19}$$

and

$$\hat{R}_n(x) \longrightarrow R(x) \quad \text{with probability one for each x where } R(x) \text{ is continuous.} \tag{20}$$

Convergence of (19) under condition (16) was proved in [5]. Therefore it is enough to show that (20) it true. Obviously

$$\left|\hat{R}_n(x) - R(x)\right| \leq \left|\hat{R}_n(x) - E\left[\hat{R}_n(x)\right]\right| + \left|E\left[\hat{R}_n(x)\right] - R(x)\right|. \tag{21}$$

Observe that

$$\left|\hat{R}_n(x) - E\left[\hat{R}_n(x)\right]\right| = \left|\frac{1}{n}\sum_{i=1}^{n} Y_i K_n(x, X_i) - E\left[\frac{1}{n}\sum_{i=1}^{n} Y_i K_n(x, X_i)\right]\right| = \tag{22}$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left[Y_i K_n(x, X_i) - E\left[Y_i K_n(x, X_i)\right]\right] \tag{23}$$

and

$$\sum_{i=1}^{\infty}\frac{1}{i^2} Var\left(Y_i K_n(x, X_i)\right) = \sum_{i=1}^{\infty}\frac{1}{i^2} Var\left(Y_i h_n^{-p} K\left(\frac{x - X_i}{h_n}\right)\right) \leq \tag{24}$$

$$\leq \sum_{i=1}^{\infty} \frac{1}{i^2} E\left[Y_i^2 h_n^{-2p} K^2\left(\frac{x - X_i}{h_n}\right)\right] =$$

$$\sum_{i=1}^{\infty} \frac{1}{i^2} \int_{R^p} E\left[Y_i^2 | X_i = u\right] f(u) ||K||_\infty h_n^{-2p} K\left(\frac{x - X_i}{h_n}\right) du \leq \tag{25}$$

$$\leq \sum_{i=1}^{\infty} \frac{1}{i^2} \int_{R^p} 2(\sigma_n^2 + \phi^2(x)) f(u) ||K||_\infty h_n^{-2p} K\left(\frac{x - X_i}{h_n}\right) du \leq \tag{26}$$

$$\leq \sum_{i=1}^{\infty} \frac{1}{i^2} 2s_i ||K||_\infty h_n^{-2p} \int_{R^p} K\left(\frac{x - X_i}{h_n}\right) du = 2||K||_\infty \sum_{i=1}^{\infty} \frac{1}{i^2} s_i h_n^{-p} \tag{27}$$

Therefore $\sum_{i=1}^{\infty} i^{-2} Var(Y_i K_n(x, X_i)) \leq \infty$ if condition (17) holds and, by using the strong law of big numbers [9], $|\hat{R}_n(x) - E\hat{R}_n(x)| \xrightarrow{n} 0$ with probability one. Since $|E\hat{R}_n(x) - R(x)| \xrightarrow{n} 0$ under condition (15) (see [5] and [8]), then the result is established.

Example
Let us consider the case when $d_n = O(n^\alpha)$ for $\alpha > 0$ and $p = 1$. For this problem we choose sequences h'_n and h_n to be in the form

$$h'_n = D' n^{-H'}, \qquad h_n = D n^{-H} \tag{28}$$

where D, D', H and H' are constants. Then the conditions (16)-(18) are satisfied for

$$0 < H' < 1 \qquad \text{and} \qquad 0 < H + \alpha < 1. \tag{29}$$

3 Simulation Results

In the following simulations we will test algorithm (13) assuming that input data are from normal distribution with the mean equal to 0 and the standard deviation equal to 4.5. The function ϕ is of the form

$$\phi(x) = (x^4 - 4x^3 + 5x^2 + 3x - 10) \sin(2x). \tag{30}$$

First we select the kernel function. For the purpose of this paper we choose Epanecznikow kernel [8] and we assume that $h'_n = h_n$.

The first experiment examines the relationship between the value of the mean square error and the value of the parameter $H = H'$ in (28). The obtained results are illustrated in Fig. 1 for $\alpha = 0.4$. The experiment was performed on the set of 6000 input data with values of parameter $D = D'$ equal to 0.4, 0.6 and 0.8. As we can see with increasing value of H the error of the algorithm decreases.

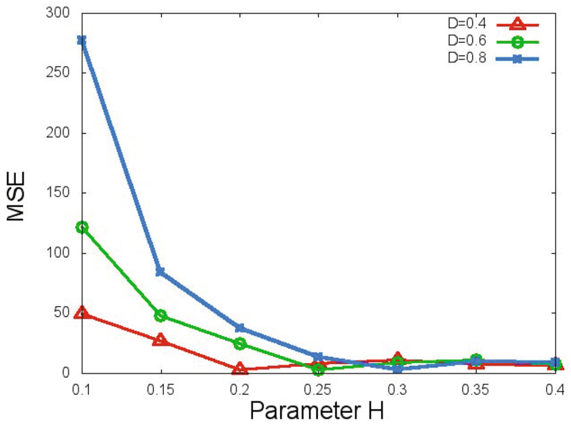


Fig. 2. The dependence between the value of parameter H and the value of the MSE

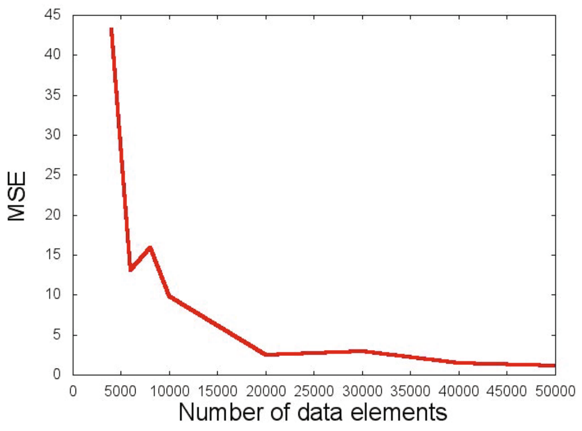


Fig. 3. The dependence between the number of data elements N and the value of the MSE

The best results were obtained for bigger values of parameter H. The smallest value of the MSE was equal to 2.65 for $H = 0.3$ and $D = 0.8$. The biggest value of the MSE was equal to $277, 24$ for $H = 0.1$ and $D = 0.8$.

In Figure 2 we show the dependence between the number of elements and the value of the mean square error. We assume that $H = H' = 0.3$, $D = D' = 0.8$ and $\alpha = 0.35$. Even with increasing variance of Z_n, corresponding to the increasing number of elements, the accuracy of the algorithm improves. For $n = 4000$ the MSE has the biggest value equal to 43.16 and the best results were obtained for $n = 50000$. In this case the MSE was equal to $1, 08$.

In Figure 3 we show the dependence between the value of parameter α and the error of the algorithm. In this case $n = 7000$, $D = 0.8$ and $H = H' = 0.3$. From conditions (29) we can see, that for this value of H and H', value of α

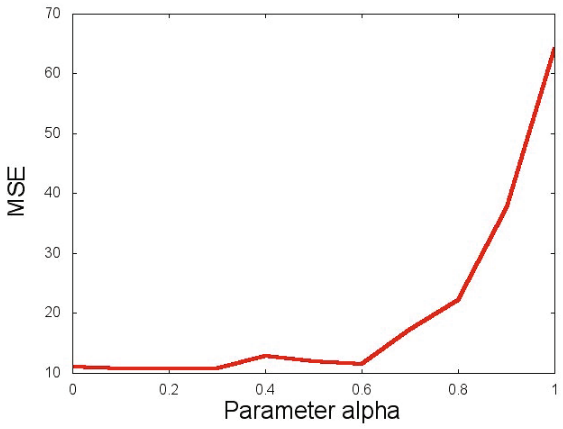


Fig. 4. The dependence between the value of parameter α and the value of the MSE

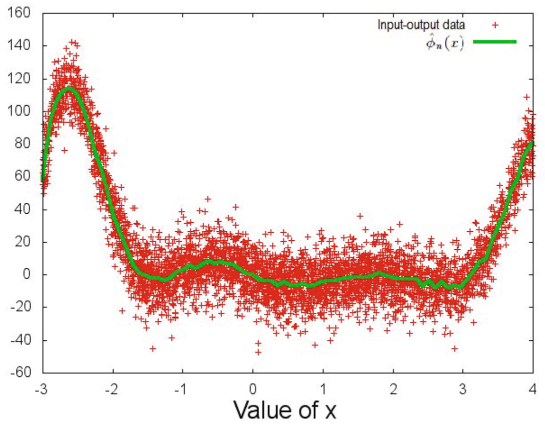


Fig. 5. The input-output data and the obtained estimator values

should be not bigger than 0.7. The best results were obtained for $\alpha = 0.3$. The MSE was equal to 10.70. For $\alpha = 1$ the MSE was equal to $64, 20$.

In Figure 4 we can see the input-output data with $\alpha = 0.6$ and obtained estimator values. Input data were coming from the normal distribution $\mathcal{N}(0, (4.5)^2)$. In this experiment $H = H' = 0.3$, $D = D' = 0.9$ and $n = 7000$.

4 Conclusion and Future Work

In this paper we applied the general regression neural networks to handle non-stationary noise and we established the strong convergence. The next stage of research can be an application of supervised and unsupervised neural networks [1], [2], [11] or neuro-fuzzy structures [10], [30]-[33], [36], [37] to the study of noisy data.

Acknowledgments. This paper was prepared under project operated within the Foundation for Polish Science Team Programme co-financed by the EU European Regional Development Fund, Operational Program Innovative Economy 2007-2013, and also supported by the National Science Center NCN.

References

1. Bilski, J., Rutkowski, L.: A fast training algorithm for neural networks. IEEE Transactions on Circuits and Systems II 45, 749–753 (1998)
2. Cierniak, R., Rutkowski, L.: On image compression by competitive neural networks and optimal linear predictors. Signal Processing: Image Communication - a Eurasip Journal 15(6), 559–565 (2000)
3. Gałkowski, T., Rutkowski, L.: Nonparametric recovery of multivariate functions with applications to system identification. Proceedings of the IEEE 73, 942–943 (1985)
4. Gałkowski, T., Rutkowski, L.: Nonparametric fitting of multivariable functions. IEEE Transactions on Automatic Control AC-31, 785–787 (1986)
5. Greblicki, W., Pawlak, M.: Nonparametric system identification. Cambridge University Press (2008)
6. Greblicki, W., Rutkowska, D., Rutkowski, L.: An orthogonal series estimate of time-varying regression. Annals of the Institute of Statistical Mathematics 35, Part A, 147–160 (1983)
7. Greblicki, W., Rutkowski, L.: Density-free Bayes risk consistency of nonparametric pattern recognition procedures. Proceedings of the IEEE 69(4), 482–483 (1981)
8. Härdle, W.: Applied Nonparametric Regression. Cambridge University Press, Cambridge (1990)
9. Loeve, M.: Probability Theory. Springer, Heidelberg (1977)
10. Nowicki, R.: Rough Sets in the Neuro-Fuzzy Architectures Based on Non-monotonic Fuzzy Implications. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 518–525. Springer, Heidelberg (2004)
11. Patan, K., Patan, M.: Optimal training strategies for locally recurrent neural networks. Journal of Artificial Intelligence and Soft Computing Research 1(2), 103–114 (2011)
12. Rutkowski, L.: Sequential estimates of probability densities by orthogonal series and their application in pattern classification. IEEE Transactions on Systems, Man, and Cybernetics SMC-10(12), 918–920 (1980)
13. Rutkowski, L.: Sequential estimates of a regression function by orthogonal series with applications in discrimination, New York, Heidelberg, Berlin. Lectures Notes in Statistics, vol. 8, pp. 236–244 (1981)
14. Rutkowski, L.: On system identification by nonparametric function fitting. IEEE Transactions on Automatic Control AC-27, 225–227 (1982)
15. Rutkowski, L.: Orthogonal series estimates of a regression function with applications in system identification. In: Probability and Statistical Inference, pp. 343–347. D. Reidel Publishing Company, Dordrecht (1982)

16. Rutkowski, L.: On Bayes risk consistent pattern recognition procedures in a quasi-stationary environment. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4(1), 84–87 (1982)
17. Rutkowski, L.: On-line identification of time-varying systems by nonparametric techniques. IEEE Transactions on Automatic Control AC-27, 228–230 (1982)
18. Rutkowski, L.: On nonparametric identification with prediction of time-varying systems. IEEE Transactions on Automatic Control AC-29, 58–60 (1984)
19. Rutkowski, L.: Nonparametric identification of quasi-stationary systems. Systems and Control Letters 6, 33–35 (1985)
20. Rutkowski, L.: The real-time identification of time-varying systems by nonparametric algorithms based on the Parzen kernels. International Journal of Systems Science 16, 1123–1130 (1985)
21. Rutkowski, L.: A general approach for nonparametric fitting of functions and their derivatives with applications to linear circuits identification. IEEE Transactions Circuits Systems CAS-33, 812–818 (1986)
22. Rutkowski, L.: Sequential pattern recognition procedures derived from multiple Fourier series. Pattern Recognition Letters 8, 213–216 (1988)
23. Rutkowski, L.: Nonparametric procedures for identification and control of linear dynamic systems. In: Proceedings of 1988 American Control Conference, June 15-17, pp. 1325–1326 (1988)
24. Rutkowski, L.: An application of multiple Fourier series to identification of multivariable nonstationary systems. International Journal of Systems Science 20(10), 1993–2002 (1989)
25. Rutkowski, L.: Nonparametric learning algorithms in the time-varying environments. Signal Processing 18, 129–137 (1989)
26. Rutkowski, L., Rafajłowicz, E.: On global rate of convergence of some nonparametric identification procedures. IEEE Transaction on Automatic Control AC-34(10), 1089–1091 (1989)
27. Rutkowski, L.: Identification of MISO nonlinear regressions in the presence of a wide class of disturbances. IEEE Transactions on Information Theory IT-37, 214–216 (1991)
28. Rutkowski, L.: Multiple Fourier series procedures for extraction of nonlinear regressions from noisy data. IEEE Transactions on Signal Processing 41(10), 3062–3065 (1993)
29. Rutkowski, L., Gałkowski, T.: On pattern classification and system identification by probabilistic neural networks. Applied Mathematics and Computer Science 4(3), 413–422 (1994)
30. Rutkowski, L.: A New Method for System Modelling and Pattern Classification. Bulletin of the Polish Academy of Sciences 52(1), 11–24 (2004)
31. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, Melbourne, December 2-5, vol. 3, pp. 1428–1431 (2001)
32. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)
33. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)

34. Specht, D.F.: A general regression neural network. IEEE Transactions Neural Networks 2, 568–576 (1991)
35. Specht, D.F.: Probabilistic neural networks. Neural Networks 3, 109–118 (1990)
36. Starczewski, L., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
37. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)

Learning in a Time-Varying Environment by Making Use of the Stochastic Approximation and Orthogonal Series-Type Kernel Probabilistic Neural Network

Jacek M. Zurada[1] and Maciej Jaworski[2]

[1] Department of Electrical and Computer Engineering, University of Louisville, 405 Lutz Hall, Louisville, KY 40292, USA
jacek.zurada@louisville.edu

[2] Department of Computer Engineering, Czestochowa University of Technology, Armii Krajowej 36, 42-200 Czestochowa, Poland
maciej.jaworski@kik.pcz.pl

Abstract. In the paper stochastic approximation, in combining with general regression neural network, is applied for learning in a time-varying environment. The orthogonal-type kernel is applied to design the general regression neural networks. Sufficient conditions for weak convergence are given and simulation results are presented.

1 Introduction

In literature, a nonlinear regression issue with the use of nonparametric methods has been widely studied. One of them are probabilistic neural networks, developed by Specht [35]. These are net structures, corresponding to nonparametric density and regression estimates, designed to solve stationary (see e.g. [5], [6], [8], [13]-[16], [22]-[24] and [27]-[30]) and nonstationary problems (see e.g. [7], [17]-[21], [25] and [26]). It should be noted that the second case was considered in literature only with a stationary noise.

Given n pairs of random variables (X_i, Y_i), $i \in \{1, \ldots, n\}$, where $X_i \in \mathbf{A} \subset \mathbb{R}^p$, $Y_i \in \mathbb{R}$, the aim of nonlinear regression is to find a function $\phi(x)$, which reflects the dependency of X_i and Y_i as best as possible. Variables X_i are independent and identically distributed, however their probability density function $f(x)$ is unknown. In general, the relation between X_i nd Y_i can be expressed in the form

$$Y_i = \phi(X_i) + Z_i,\ i \in \{1, \ldots, n\}, \tag{1}$$

where random variables Z_i represent the noise. The only assumptions about the noise variables are as follows

$$E[Z_i] = 0,\ \ E[Z_i^2] = \sigma_i^2 \leq \sigma_Z^2,\ \ i \in \{1, \ldots, n\}. \tag{2}$$

In this paper, the generalized nonlinear regression problem is considered. In this case, an additional deterministic perturbation is introduced to the system described by (1)

$$Y_i = \phi(X_i) + ac_i + Z_i,\ i \in \{1, \dots, n\}, \tag{3}$$

where $c_i,\ i \in \{1, \dots, n\}$ are elements of some known sequence $(\lim_{i\to\infty} |c_i| = \infty)$ and a is an unknown constant. The task of the presented generalized nonlinear regression is to estimate the value of parameter a and to find the model function $\phi(x)$ simultaneously.

2 Algorithm

The algorithm of solving the generalized nonlinear regression problem will be considered as a two-step process:

- first, an estimator $\hat{a}_n$ of the parameter a is calculated, with the use of values of $Y_1, \dots, Y_n$,
- second, assuming that a is equal to $\hat{a}_n$, the estimator $\hat{\phi}_n(x, \hat{a}_n)$ of function $\phi(x)$ is established with the use of $(X_1, Y_1), \dots, (X_n, Y_n)$.

The first part of the algorithm can be performed with the application of the stochastic approximation. It is easily seen that formula (3) leads to the following equation

$$a = \frac{E[Y_i]}{c_i} - \frac{\int_{\mathbf{A}} \phi(x) f(x) d\mathbf{x}}{c_i}. \tag{4}$$

Let $\hat{A}_i$ denotes the estimator of $E[Y_i]$. Knowing that $\lim_{i\to\infty} |c_i| = \infty$, the estimator $\hat{a}_i$ of prameter a can be proposed in the form

$$\hat{a}_i = \frac{\hat{A}_i}{c_i}. \tag{5}$$

Estimator $\hat{A}_i$ can be treated as a solution of a trivial equation

$$M_i(\overline{A}_i) = E[Y_i] - \overline{A}_i = 0. \tag{6}$$

Therefore, the values of $E[Y_i],\ i \in \{1, \dots, n\}$ can be estimated with the Robbins-Monro stochastic approximation procedure [12]

$$\hat{A}_i = (1 - \gamma_i)\hat{A}_{i-1} + \gamma_i Y_i, \tag{7}$$

where $\hat{A}_0 = 0$ and γ_i is some known sequence. Combining (5) with (7) one obtains the formula for estimator $\hat{a}_i$

$$\hat{a}_i = \frac{(1 - \gamma_i)\hat{A}_{i-1} + \gamma_i Y_i}{c_i}, \quad \hat{A}_0 = 0. \tag{8}$$

After the value of $\hat{a}_n$ is computed, one can perform the second step of the nonlinear regression algorithm. The following transformation of random variables Y_i is introduced

$$V_{n,i} = Y_i - \hat{a}_n c_i. \tag{9}$$

In view of (3) one can write

$$V_{n,i} = \phi(X_i) + Z_i + (a - \hat{a}_n)c_i. \tag{10}$$

Then, the commonly known regression procedures can be applied to find the function $\phi(x)$ for pairs $(X_i, V_{n,i}),\ i \in \{1, \ldots, n\}$. In this paper the nonparametric approach with orthogonal series is proposed.

The function $\phi(x)$, at each point x such that $f(x) \neq 0$, can be expressed as follows

$$\phi(x) = \frac{\phi(x)f(x)}{f(x)} \overset{def.}{=} \frac{R(x)}{f(x)}. \tag{11}$$

Estimators $\hat{R}_n(x, \hat{a}_n)$ and $\hat{f}_n(x)$ of functions $R(x)$ and $f(x)$ are calculated separately. Then, the estimator $\hat{\phi}_n(x, \hat{a}_n)$ of function $\phi(x)$ is defined as a quotient of $\hat{R}_n(x, \hat{a}_n)$ and $\hat{f}_n(x)$.

It is assumed that probability density function can be decomposed into some orthogonal series $g_j(x),\ j \in \mathbb{N}$

$$f(x) \sim \sum_{j=0}^{\infty} a_j g_j(x), \tag{12}$$

where $|g_j(x)| \leq G_j,\ \forall_{x \in A},\ \forall_{j \in \mathbb{N}}$. The coefficients a_j can be estimated by the appropriate arithmetic means

$$a_{n,j} = \frac{1}{n} \sum_{i=1}^{n} g_j(X_i). \tag{13}$$

Then, as an estimator of function $f(x)$ one can take

$$\hat{f}_n(x) = \sum_{j=0}^{N(n)} a_{n,j} g_j(x) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=0}^{N(n)} g_j(X_i) g_j(x), \tag{14}$$

where $N(n)$ is some function of n, satisfying $\lim_{n \to \infty} N(n) = \infty$.

Function $R(x)$ can be decomposed into orthogonal series in similar way

$$R(x) \sim \sum_{j=0}^{\infty} b_j g_j(x). \tag{15}$$

As an estimator of b_j one can propose

$$b_{n,j} = \frac{1}{n}\sum_{i=1}^{n} V_{n,i} g_j(X_i). \tag{16}$$

Analogously to (14), the estimator of function $R(x)$ is expressed as follows

$$\hat{R}_n(x, \hat{a}_n) = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=0}^{M(n)} V_{n,i} g_j(X_i) g_j(x), \tag{17}$$

where $M(n)$ satisfies $\lim_{n\to\infty} M(n) = \infty$. Finally, combining (14) and (17), the estimator of function $\phi(x)$ is obtained

$$\hat{\phi}_n(x, \hat{a}_n) = \frac{\hat{R}_n(x, \hat{a}_n)}{\hat{f}_n(x)} = \frac{\sum_{i=1}^{n}\sum_{j=0}^{M(n)} V_{n,i} g_j(X_i) g_j(x)}{\sum_{i=1}^{n}\sum_{j=0}^{N(n)} g_j(X_i) g_j(x)}. \tag{18}$$

3 Main Result

The following theorem allows to determine the forms of sequences γ_i and c_i, which provide the convergence of estimator (8)

Theorem 1. *If (2) holds and additionally the following conditions are satisfied*

$$\int_A \phi^2(x) f(x) dx < \infty, \tag{19}$$

$$\gamma_i = \delta i^{-r},\ \delta > 0,\ 0 < r < 1, \tag{20}$$

$$|c_{i+1} - c_i| = O(i^{-q}),\ q > r, \tag{21}$$

$$c_i = O(i^t),\ 0 < 2t < r, \tag{22}$$

then

$$\hat{a}_i \overset{i\to\infty}{\longrightarrow} a \quad \text{in probability}. \tag{23}$$

Proof. The theorem can be proven by making use of the Dvoretzky Theorem [4].

In estimator (18) the sequences $M(n)$ and $N(n)$ have to be chosen. Theorem 2 allows to choose them properly, ensuring the convergence of estimator $\hat{\phi}_n(x, \hat{a}_n)$.

Theorem 2. *If (2), (19) and (23) holds and the following conditions are true*

$$\lim_{n\to\infty} N(n) = \infty,\quad \lim_{n\to\infty}\left[\frac{1}{n}\left(\sum_{j=1}^{N(n)} G_j^2\right)^2\right] = 0, \tag{24}$$

$$\lim_{n\to\infty} M(n) = \infty, \quad \lim_{n\to\infty} \left[\frac{1}{n} \left(\sum_{j=1}^{M(n)} G_j^2 \right)^2 \right] = 0, \tag{25}$$

then

$$\hat{\phi}_n(x, \hat{a}_n) \overset{n\to\infty}{\longrightarrow} \phi(x) \quad \text{in probability} \tag{26}$$

at each point x at which series (12) and (15) are convergent to $f(x)$ and $R(x)$, respectively.

Proof. The theorem can be proven combining convergence (23), weak convergence of estimator (18) [14] and Theorem 4.3.8 in [38]. Conditions for convergence of series (12) and (14) are given in [1].

Example
Let us assume that the sequence c_i is of the form $c_i = i^t$. Then $c_i = O(i^t)$ and $|c_{i+1} - c_i| = O(i^{t-1})$. According to conditions (21) and (22), the sequence γ_i from (20) should be given in the following form

$$\gamma_i = \delta i^{-r}, \ \delta > 0, \ 0 < 2t < r < 1 - t. \tag{27}$$

This mean, that the exponent t should belong to the interval $\left(0, \frac{1}{3}\right)$. In estimators (14) and 17), one of the possible choices of an orthogonal series g_j is the Fourier orthogonal series. The functions, defined on the interval $[a_F; b_F]$, are given by

$$g_j(x) = \begin{cases} \dfrac{1}{\sqrt{b_F - a_F}}, & j = 0, \\ \sqrt{\dfrac{2}{b_F - a_F}} \sin\left(2\pi j \dfrac{x - a_F}{b_F - a_F}\right), & j \bmod 2 = 1, \\ \sqrt{\dfrac{2}{b_F - a_F}} \cos\left(2\pi j \dfrac{x - a_F}{b_F - a_F}\right), & j \bmod 2 = 0, \ j \neq 0. \end{cases} \tag{28}$$

Each function $g_j(x)$ is bounded by a constant $G_j = C$. Assumption (24) of Theorem 2 can be expressed as

$$\lim_{n\to\infty} \left[\frac{1}{n} \left(\sum_{j=1}^{N(n)} G_j^2 \right)^2 \right] = \lim_{n\to\infty} \left[\frac{N^2(n) C^2}{n} \right] = 0. \tag{29}$$

Then, the sequence $N(n)$ can be proposed in the following form

$$N(n) = \lceil D' n^{Q'} \rceil, \ D' > 0, \ Q' < \frac{1}{2}. \tag{30}$$

The analogous form can be obtained for the function $M(n) = \lceil Dn^Q \rceil$, in the light of condition (25)

$$M(n) = \lceil Dn^Q \rceil,\ D > 0,\ Q < \frac{1}{2}. \tag{31}$$

4 Experimental Results

In the following simulations, the system described by equation (3) is examined, with the real value of parameter $a = 2$ and the function $\phi(x)$ given by

$$\phi(x) = \frac{10x}{1 + x^2},\ x \in [-5 : 5]. \tag{32}$$

Random variables X_i are generated from the uniform distribution, where $X_i \in [-5 : 5]$. Noise variables Z_i are taken from the standard normal distribution $N(0, 1)$. Sequence c_i is taken in the form

$$c_i = i^t,\ t = 0,3. \tag{33}$$

Then, in view of condition (27), the exponent r in the sequence γ_i should belong to the interval $(0,6; 0,7)$. In the presented simulations r is set to $0,65$, i. e.

$$\gamma_i = i^{-r},\ r = 0,65. \tag{34}$$

To estimate the functions $R(x)$ and $f(x)$, the orthogonal Fourier series (28) is applied. Functions $N(n)$ and $M(n)$ are given in the forms (30) and (31), respectively, with $D = D' = 2$. Simulation are performed for several allowed, distinct values of parameters $Q = Q'$

In Figure 1 the convergence of estimator $\hat{a}_n$ is presented.

The estimator converges to the actual value of a with the increasing number of data elements n, although it demonstrates a relatively high variance. These variations affect the quality of estimation of function $\phi(x)$, what is shown in Fig. 2. Although the trend of the Mean Squared Error (MSE), as a function of n, is decreasing, it follows the variations of estimator $\hat{a}_n$.

In Figure 3 an example result of estimator $\hat{\phi}_n(x, \hat{a}_n)$ is presented, for $Q = 0,4$ and the number of data elements $n = 5000$. The result is compared with the function (32).

5 Final Remarks

In the paper we applied stochastic approximation in connection with general regression neural network for learning in a time-varying environment. The orthogonal series-type kernel was applied to design the general regression neural networks. Sufficient conditions for weak convergens were given and simulation results were presented. In future work it would be interesting to apply neuro-fuzzy structures [9], [10], [31]-[34], [36], [37] and supervised and unsupervised neural networks [2], [3], [11] for learning in a time-varying environment.

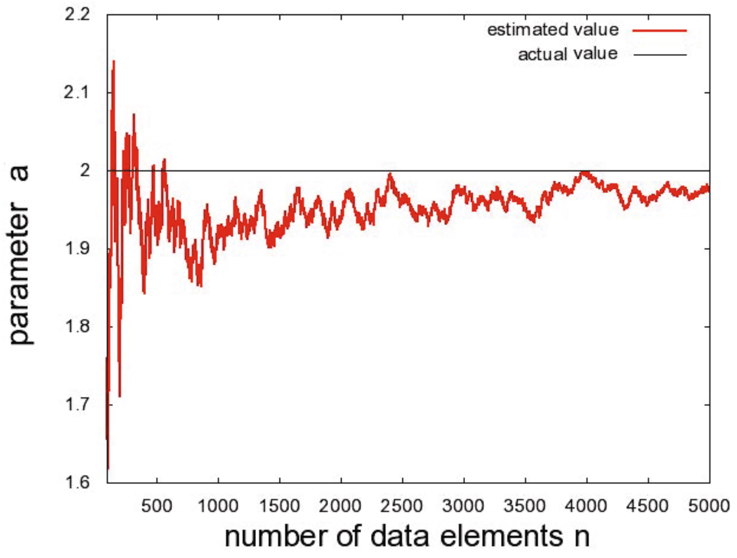


Fig. 1. Convergence of the estimator $\hat{a}_n$ to the actual value of parameter a

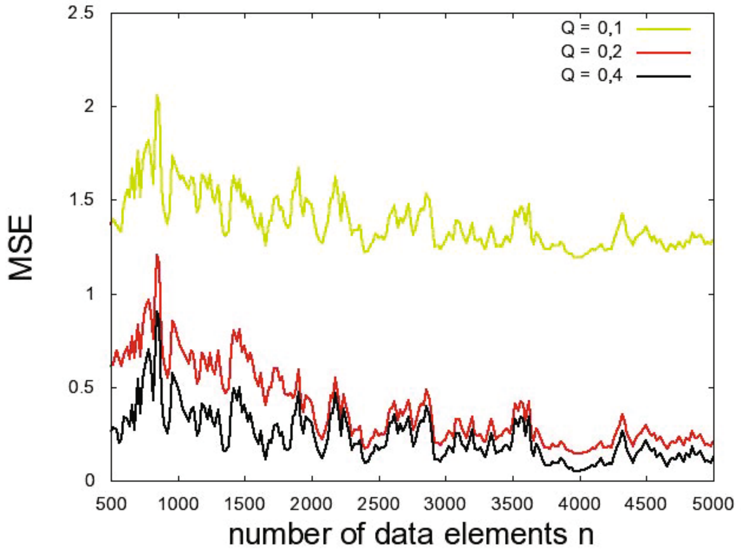


Fig. 2. Dependence of the Mean Squared Error on the number of data elements n, for three different values of parameter: $Q = Q' = 0,1$, $Q = Q' = 0,2$ and $Q = Q' = 0,4$

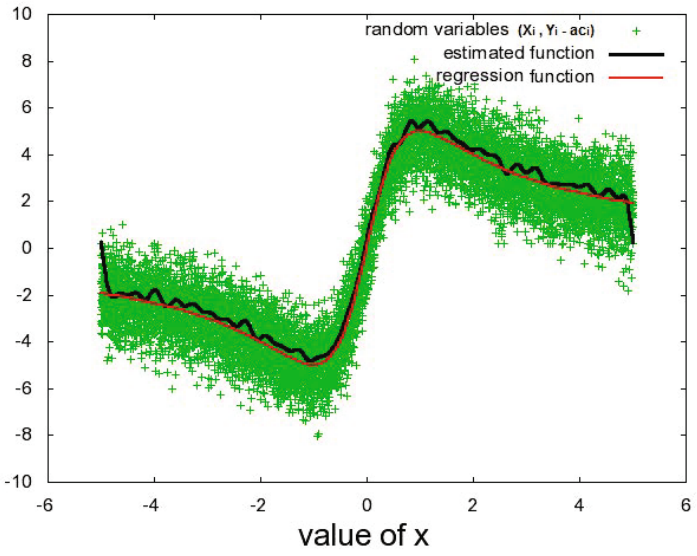


Fig. 3. Comparison of the real function $\phi(x)$ with the estimator $\hat{\phi}_n(x, \hat{a}_n)$, obtained for $n = 5000$ and $Q = 0,4$. Points represent the random variables $(X_i, Y_i - ac_i)$

Acknowledgments. The paper was prepared under project operated within the Foundation for Polish Science Team Programme co-financed by the EU European Regional Development Fund, Operational Program Innovative Economy 2007-2013, and also supported by National Science Centre NCN.

References

1. Alexits, G.: Convergence Problems of Orthogonal Series, Budapest, Academia and Kiado, pp. 261–264 (1961)
2. Bilski, J., Rutkowski, L.: A fast training algorithm for neural networks. IEEE Transactions on Circuits and Systems II 45, 749–753 (1998)
3. Cierniak, R., Rutkowski, L.: On image compression by competitive neural networks and optimal linear predictors. Signal Processing: Image Communication - a Eurasip Journal 15(6), 559–565 (2000)
4. Dvoretzky, A.: On stochastic approximation. In: Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, vol. 1, pp. 39–56. University of California Press (1956)
5. Gałkowski, T., Rutkowski, L.: Nonparametric recovery of multivariate functions with applications to system identification. Proceedings of the IEEE 73, 942–943 (1985)
6. Gałkowski, T., Rutkowski, L.: Nonparametric fitting of multivariable functions. IEEE Transactions on Automatic Control AC-31, 785–787 (1986)
7. Greblicki, W., Rutkowska, D., Rutkowski, L.: An orthogonal series estimate of time-varying regression. Annals of the Institute of Statistical Mathematics 35, Part A, 147–160 (1983)

8. Greblicki, W., Rutkowski, L.: Density-free Bayes risk consistency of nonparametric pattern recognition procedures. Proceedings of the IEEE 69(4), 482–483 (1981)
9. Nowicki, R.: Rough Sets in the Neuro-Fuzzy Architectures Based on Monotonic Fuzzy Implications. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 510–517. Springer, Heidelberg (2004)
10. Nowicki, R.: Nonlinear modelling and classification based on the MICOG defuzzifications. Journal of Nonlinear Analysis, Series A: Theory, Methods and Applications 7(12), 1033–1047 (2009)
11. Patan, K., Patan, M.: Optimal Training Strategies for Locally Recurrent Neural Networks. Journal of Artificial Intelligence and Soft Computing Research 1(2), 103–114 (2011)
12. Robbins, H., Monro, S.: A stochastics approximation method. Annals Mathematics of Statistics 22, 400–407 (1951)
13. Rutkowski, L.: Sequential estimates of probability densities by orthogonal series and their application in pattern classification. IEEE Transactions on Systems, Man, and Cybernetics SMC-10(12), 918–920 (1980)
14. Rutkowski, L.: Sequential estimates of a regression function by orthogonal series with applications in discrimination, New York, Heidelberg, Berlin. Lectures Notes in Statistics, vol. 8, pp. 236–244 (1981)
15. Rutkowski, L.: On system identification by nonparametric function fitting. IEEE Transactions on Automatic Control AC-27, 225–227 (1982)
16. Rutkowski, L.: Orthogonal series estimates of a regression function with applications in system identification. In: Probability and Statistical Inference, pp. 343–347. D. Reidel Publishing Company, Dordrecht (1982)
17. Rutkowski, L.: On Bayes risk consistent pattern recognition procedures in a quasi-stationary environment. IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-4(1), 84–87 (1982)
18. Rutkowski, L.: On-line identification of time-varying systems by nonparametric techniques. IEEE Transactions on Automatic Control AC-27, 228–230 (1982)
19. Rutkowski, L.: On nonparametric identification with prediction of time-varying systems. IEEE Transactions on Automatic Control AC-29, 58–60 (1984)
20. Rutkowski, L.: Nonparametric identification of quasi-stationary systems. Systems and Control Letters 6, 33–35 (1985)
21. Rutkowski, L.: The real-time identification of time-varying systems by nonparametric algorithms based on the Parzen kernels. International Journal of Systems Science 16, 1123–1130 (1985)
22. Rutkowski, L.: A general approach for nonparametric fitting of functions and their derivatives with applications to linear circuits identification. IEEE Transactions Circuits Systems CAS-33, 812–818 (1986)
23. Rutkowski, L.: Sequential pattern recognition procedures derived from multiple Fourier series. Pattern Recognition Letters 8, 213–216 (1988)
24. Rutkowski, L.: Nonparametric procedures for identification and control of linear dynamic systems. In: Proceedings of 1988 American Control Conference, June 15-17, pp. 1325–1326 (1988)
25. Rutkowski, L.: An application of multiple Fourier series to identification of multivariable nonstationary systems. International Journal of Systems Science 20(10), 1993–2002 (1989)
26. Rutkowski, L.: Nonparametric learning algorithms in the time-varying environments. Signal Processing 18, 129–137 (1989)

27. Rutkowski, L., Rafajłowicz, E.: On global rate of convergence of some nonparametric identification procedures. IEEE Transaction on Automatic Control AC-34(10), 1089–1091 (1989)
28. Rutkowski, L.: Identification of MISO nonlinear regressions in the presence of a wide class of disturbances. IEEE Transactions on Information Theory IT-37, 214–216 (1991)
29. Rutkowski, L.: Multiple Fourier series procedures for extraction of nonlinear regressions from noisy data. IEEE Transactions on Signal Processing 41(10), 3062–3065 (1993)
30. Rutkowski, L., Gałkowski, T.: On pattern classification and system identification by probabilistic neural networks. Applied Mathematics and Computer Science 4(3), 413–422 (1994)
31. Rutkowski, L.: A New Method for System Modelling and Pattern Classification. Bulletin of the Polish Academy of Sciences 52(1), 11–24 (2004)
32. Rutkowski, L., Cpałka, K.: A general approach to neuro - fuzzy systems. In: Proceedings of the 10th IEEE International Conference on Fuzzy Systems, Melbourne, December 2-5, vol. 3, pp. 1428–1431 (2001)
33. Rutkowski, L., Cpałka, K.: A neuro-fuzzy controller with a compromise fuzzy reasoning. Control and Cybernetics 31(2), 297–308 (2002)
34. Scherer, R.: Boosting Ensemble of Relational Neuro-fuzzy Systems. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 306–313. Springer, Heidelberg (2006)
35. Specht, D.F.: Probabilistic neural networks. Neural Networks 3, 109–118 (1990)
36. Starczewski, J., Rutkowski, L.: Interval type 2 neuro-fuzzy systems based on interval consequents. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 570–577. Physica-Verlag, Springer-Verlag Company, Heidelberg, New York (2003)
37. Starczewski, J., Rutkowski, L.: Connectionist Structures of Type 2 Fuzzy Inference Systems. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Waśniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 634–642. Springer, Heidelberg (2002)
38. Wilks, S.S.: Mathematical Statistics. John Wiley, New York (1962)

Accelerating BST Methods for Model Reduction with Graphics Processors

Peter Benner[1], Pablo Ezzatti[2],
Enrique S. Quintana-Ortí[3], and Alfredo Remón[3]

[1] Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg, Germany
benner@mpi-magdeburg.mpg.de
[2] Centro de Cálculo-Instituto de Computación, Universidad de la República, Montevideo, Uruguay
pezzatti@fing.edu.uy
[3] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain
{quintana,remon}@icc.uji.es

Abstract. Model order reduction of dynamical linear time-invariant system appears in many scientific and engineering applications. Numerically reliable SVD-based methods for this task require $\mathcal{O}(n^3)$ floating-point arithmetic operations, with n being in the range $10^3 - 10^5$ for many practical applications. In this paper we investigate the use of graphics processors (GPUs) to accelerate model reduction of large-scale linear systems via Balanced Stochastic Truncation, by off-loading the computationally intensive tasks to this device. Experiments on a hybrid platform consisting of state-of-the-art general-purpose multi-core processors and a GPU illustrate the potential of this approach.

Keywords: Model reduction, linear dynamical systems, Lyapunov equations, SVD-based methods, GPUs.

1 Introduction

Model order reduction is an important numerical tool to reduce the time and cost required for the design of optimal controllers in many industrial processes where dynamics can be modeled by a linear time-invariant (LTI) system of the form:

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t), \quad t > 0, \quad x(0) = x^0, \\ y(t) &= Cx(t) + Du(t), \quad t \geq 0. \end{aligned} \tag{1}$$

Here, $x(t)$ contains the states of the system, with $x^0 \in \mathbb{R}^n$ the initial state, $u(t) \in \mathbb{R}^m$ and $y(t) \in \mathbb{R}^p$ contain the inputs and outputs, respectively, and $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{p\times n}$, $D \in \mathbb{R}^{p\times m}$. The system in (1) can also be described by the associated transfer function matrix (TFM) $G(s) = C(sI_n - A)^{-1}B + D$. A particularly important property is that the number of states (also known as the state-space dimension or the order) of the system, n, is in general much larger than m and p.

The goal of model reduction is to find a reduced-order LTI system,

$$\begin{aligned} \dot{\hat{x}}(t) &= \hat{A}\hat{x}(t) + \hat{B}u(t), \quad t > 0, \quad \hat{x}(0) = \hat{x}^0, \\ \hat{y}(t) &= \hat{C}\hat{x}(t) + \hat{D}u(t), \quad t \geq 0, \end{aligned} \tag{2}$$

of order r, with $r \ll n$, and associated TFM $\hat{G}(s) = \hat{C}(sI_n - \hat{A})^{-1}\hat{B} + \hat{D}$ which approximates the dynamics of the original system defined by $G(s)$. The reduced-order realization (2) can then replace the original model of order n in subsequent simulations or processes, thus simplifying these tasks considerably. Model order reduction of large-scale systems appears, e.g., in thermal, thermo-mechanical, electro-mechanical and acoustic finite element models [1]. We consider a system to be large-scale if $n \sim \mathcal{O}(1,000) - \mathcal{O}(100,000)$, while, often, $m, p \sim \mathcal{O}(10) - \mathcal{O}(100)$.

The numerical method for model order reduction considered in this paper is based on the so-called state-space truncation approach and requires, at an initial stage, the solution of a Lyapunov and a Riccati equation. The reduced-order system is then obtained using a variant of the balanced stochastic truncation (BST) method [2], which only requires dense linear algebra computations. Although there exist several other approaches for model order reduction (see, e.g., [1, 3] and the references therein), those are specific for a certain subset of problems and often do not possess relevant properties such as error bounds, preservation of stability and passivity, or phase information. A comparison of the numerical properties of SVD-based methods (as Balanced Stochastic Truncation, BST) and Krylov subspace methods can be found in [1].

The Lyapunov and Riccati equations are solved in our algorithms via the matrix sign function, which yields a computational cost for the global model order reduction procedure of $\mathcal{O}(n^3)$ flops (floating-point arithmetic operations). This calls for the application of high performance computing in the reduction of models with n in the order of thousands or larger.

Recent work on the implementation of the BLAS specification and some relevant linear algebra operations included in LAPACK [4–7] has demonstrated the potential of graphics processors (GPUs) to yield high performance for the execution of dense linear algebra operations, specially if they can be cast in terms of matrix-matrix products. In [8] we built upon these works to deal with the solution of the standard Lyapunov equation on a GPU. Here, we extend this work by tackling the different stages in BST methods for model reduction of linear systems, namely, the solution of the Lyapunov and Riccati equations, the computation of the SVD, and other auxiliary computations. The target architecture is a hybrid platform consisting of a general-purpose multicore processor and a GPU. We exploit these two resources by designing a hybrid numerical algorithm for model order reduction that performs fine-grain computations on the CPU while off-loading computationally intensive operations to the GPU.

The rest of the paper is structured as follows. In Section 2 we briefly review the BST method for model order reduction, including the Lyapunov solver, the sign function-based Riccati solver and the remaining stages of the method. In Section 3 high performance implementations for a hybrid CPU-GPU platform are described. In Section 4 we present experimental results that expose the parallelism attained by the numerical algorithms on a platform consisting of two Intel QuadCore processors connected to an NVIDIA Tesla C2050 GPU. Finally, in Section 5 we provide a few concluding remarks and future lines of work.

2 Model Reduction Methods Based on SVD

Relative error methods attempt to minimize the relative error $\|\Delta_r\|_\infty$, defined implicitly by $G - \hat{G} = G\Delta_r$. Among these, BST and its variants are particularly popular [9–11]. Due to their high computational cost, BST methods have been used only for problems of moderate dimension, i.e., models of state-space dimension in the order of hundreds. The implementation included in the *Subroutine Library in Control Theory* – SLICOT[1] [12], available for MATLAB® and Fortran 77, made feasible to target systems with a few thousands of state-space variables on nowadays standard desktop computers, but larger problems remain un-affordable, unless a cluster of computers and a message-passing library as PLiCMR [13] is employed.

BST is a technique where the reduced order model is obtained truncating a balanced stochastic realization. Such a realization is obtained as follows. Define $\Phi(s) = G(s)G^T(-s)$, and let W be a *square minimum phase right spectral factor* of Φ, i.e., $\Phi(s) = W^T(-s)W(s)$. As D has full row rank, $E = DD^T$ is positive definite and a minimal state-space realization (A_W, B_W, C_W, D_W) of W is given by (see [14, 15])

$$\begin{array}{ll} A_W = A, & B_W = BD^T + W_cC^T, \\ C_W = E^{-\frac{1}{2}}(C - B_W^T X_W), & D_W = E^{\frac{1}{2}}. \end{array} \tag{3}$$

Here, W_c is the controllability Gramian of $G(s)$ given by the solution of the Lyapunov equation

$$AW_c + W_cA^T + BB^T \ = \ 0 \tag{4}$$

while W_o is the observability Gramian of $W(s)$ obtained as the *stabilizing* solution of the algebraic Riccati equation (ARE)

$$\begin{aligned} 0 = {} & (A - B_W E^{-1} C)^T W_o + W_o (A - B_W E^{-1} C) + W_o B_W E^{-1} B_W^T W_o + \\ & C^T E^{-1} C. \end{aligned} \tag{5}$$

In the following subsections we revisit the sign function methods for the solution of Lyapunov and Riccati equations introduced in [16] and [17] respectively. For the solution of the Lyapunov equation, the algorithm introduced in [8] has demonstrated to be highly efficient on hybrid architectures equipped with a GPU. This Lyapunov solver provides a low-rank approximation to the full-rank factor of the solution matrix.

[1] Available from `http://www.slicot.org`

2.1 Solution of the Lyapunov Equation

The matrix sign function was introduced in [17] as an efficient tool to solve stable (standard) Lyapunov equations. The variant of the Newton iteration method for the matrix sign function in Algorithm `CECLNC` [16] can be employed for the solution of a Lyapunov equation (like that in (4)).

Algorithm `CECLNC`:

```
$A_0 \leftarrow A$, $\tilde{S}_0 \leftarrow B^T$
$k \leftarrow 0$
repeat
    Compute the rank-revealing QR (RRQR) decomposition
        $\frac{1}{\sqrt{2c_k}} \left[ \tilde{S}_k,\ \ c_k \tilde{S}_k A_k^{-T} \right] = Q_s \begin{bmatrix} U_s \\ 0 \end{bmatrix} \mathit{\Pi}_s$
    $\tilde{S}_{k+1} \leftarrow U_s \mathit{\Pi}_s$
    $A_{k+1} \leftarrow \frac{1}{\sqrt{2}} \left( A_k / c_k + c_k A_k^{-1} \right)$
    $k \leftarrow k + 1$
until $\|A_k - I\|_1 < \tau_l \|A_k\|_1$
```

The number of columns of factor $\tilde{S}$ is doubled at each iteration and, in consequence, the computational and storage costs associated with its update grow at each iteration. To moderate this increase, and the number of columns in the factors, an RRQR factorization is computed at each step. This approach yields important gains when the number of iterations that are required for convergence is large. Note that Q_s is not accumulated as it is not needed in further computations. This reduces the cost of the RRQR significantly. For simplicity, we do not detail this compression procedure; see [18].

On convergence, after j iterations, $\tilde{S} = \frac{1}{\sqrt{2}} \tilde{S}_j$, of dimension $\tilde{k}_c \times n$, is the full (row-)rank approximation of S, so that $W_c = S^T S \approx \tilde{S}^T \tilde{S}$.

The Newton iteration for the sign function method usually presents a fast convergence rate, which is ultimately quadratic. Initial convergence can be accelerated using several techniques. In our case, we employ a scaling factor defined by the parameter

$$c_k = \sqrt{\|A_k\| / \|A_k^{-1}\|}.$$

In the convergence test, τ_l is a tolerance threshold for the iteration that is usually set as a function of the problem dimension and the machine precision ε. In particular, to avoid stagnation in the iteration, we set $\tau_l = n \cdot \sqrt{\varepsilon}$ and perform one or two additional iteration steps after the stopping criterion is satisfied. Due to the quadratic convergence of the Newton iteration, this is usually enough to reach the attainable accuracy. The RRQR decomposition can be obtained by means of the traditional QR factorization with column pivoting [18] plus a reliable rank estimator.

2.2 Solution of the Riccati Equation

The solution of an Algebraic Riccati Equation (ARE) of the form

$$F^T X + XF - XGX + Q \; = \; 0, \tag{6}$$

can be obtained from the stable invariant subspace of the Hamiltonian matrix defined in [19]

$$H = \begin{bmatrix} F & G \\ -Q & -F^T \end{bmatrix}. \tag{7}$$

This solution can be obtained computing the matrix sign function of H [17]:

$$\text{sign}(H) = Y = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}, \tag{8}$$

and then, solving the overdetermined system

$$\begin{bmatrix} Y_{11} \\ Y_{12} + I_n \end{bmatrix} X = \begin{bmatrix} I_n - Y_{21} \\ -Y_{11} \end{bmatrix} \tag{9}$$

(e.g., applying the least squares method).

Algorithm GECRSG summarizes the steps to solve an ARE with this method.

```
Algorithm GECRSG:

$H_0 \leftarrow \begin{bmatrix} F & G \\ -Q & -F^T \end{bmatrix}$
$k \leftarrow 0$
repeat
    $H_{k+1} \leftarrow \frac{1}{2}\left(H_k/d_k + d_k H_k^{-1}\right)$
until $\|H_{k+1} - H_k\|_1 < \tau_r \|H_k\|_1$
Solve
    $\begin{bmatrix} Y_{11} \\ Y_{12} + I_n \end{bmatrix} X = \begin{bmatrix} I_n - Y_{21} \\ -Y_{11} \end{bmatrix}$
```

The scaling factor d_k and the tolerance threshold τ_r can be defined here following the ideas presented for c_k and τ_l, respectively, in the previous subsection.

3 High Performance Implementation on a Hybrid Architecture

In this section, we describe an efficient implementation of the BST model order reduction method, specially designed for hybrid platforms composed of a multicore CPU connected to a single GPU. The objective of the hybrid implementation is to reduce the computational time of the BST method, executing each operation on the most convenient architecture and reducing the amount of data transfers between the two components. Specifically, the most expensive computations are executed on the GPU while the fine-grain operations are executed on the CPU.

3.1 Hybrid Implementation of the Lyapunov Solver

The most time consuming operation of Algorithm CECLNC is the update of A_{k+1}. This is due to the large dimension of A and the significant computational cost of the matrix inversion.

The hybrid algorithm to accelerate the solution of this equation proceeds as follows. At the beginning of each iteration, the CPU transfers matrix A_k to the GPU. Then, the CPU and the GPU cooperate in the inversion of matrix A_k, which is returned to the CPU upon completion. The rest of the operations are performed on the CPU since they require a minor computational effort and can be efficiently executed on a multicore processor, e.g. invoking parallel implementations of BLAS and LAPACK to compute the linear algebra operations or using a simple OpenMP-based parallelization in other cases.

The inversion algorithm is based on the Gauss-Jordan elimination method [20], since this is a highly parallel algorithm. The implementation, presented in [21], includes some performance-enhancing techniques, as the processing by blocks to exploit the hierarchical organization of the memory, hybrid and concurrent computing (CPU + GPU) to increment the resource utilization, look-ahead techniques [22] to avoid bottlenecks, padding to accelerate the memory accesses on the GPU, and multilevel blocks strategies to improve throughput of both devices.

3.2 Hybrid Implementation of the Riccati Solver

This stage can be divided into three steps:

– First, it is necessary to build matrix H performing some matrix-matrix multiplications (see equations (3)-(5)). As the dimensions of the matrices involved in these operations are moderate, the related computational cost is moderate as well. For this reason, and with the aim to reduce data transfers overheads, those operations are performed on the CPU.
– Second, the sign function for the extended matrix (7) is computed. The proposal is based on the efficient matrix inversion kernel described in subsection 3.1 and the utilization of OpenMP to parallelize a loop, executed on the CPU, that simultaneously computes the matrix addition, the matrix scaling, and the matrix norm required for the evaluation of the loop guard; see Algorithm GECRSG. Note that the solution of the Riccati equation via this solver involves matrices which are twice as big as those that appear in the sign function solver for the Lyapunov equation.
– Finally, the overdetermined system is solved. To do so, a multi-thread version of routine GEQP3 (included in LAPACK) is employed. Other minor operations are also executed on the CPU and parallelized using OpenMP.

3.3 Remaining Stages of the BST Method

Once the low rank factor from the controllability gramian (S) and the observability gramian (W_o) have been computed from the solution of the Lyapunov and the Riccati equations respectively, only some minor operations with moderate computational effort are required to obtain the reduced order model.

The main computations in this step include some matrix-matrix products involving matrices of relatively small dimension. All these operations require a moderate number of arithmetic operations and, therefore, can be efficiently computed on the CPU using `BLAS`. Computing them on the CPU avoids data transfers and the associated overhead.

4 Numerical Experiments

In this section we evaluate the parallel performance of the BST model order reduction method. The target platform consists of two Intel Xeon QuadCore E5520 processors (2.27GHz) with 24GB of RAM connected to an Nvidia Tesla C2050 via a PCI-e bus. Multi-threaded implementation of BLAS, from the Intel MKL library (version 11.1) for the general-purpose processor and from Nvidia CUBLAS (version 3.2) for the GPU are used.

We compare three different implementations: a sequential one (BST_SCPU) that is executed on a single CPU core (used as the reference implementation), a parallel multi-thread routine (BST_MTCPU) that exploits all the cores from the CPU, and a hybrid CPU-GPU implementation (BST_HYB) that executes operations concurrently on the GPU and the CPU cores.

We employ double precision arithmetic for the solution of two instances of the `STEEL_I` model reduction problem [23], extracted from the Oberwolfach benchmark collection (University of Freiburg)[2]. This model arises in the manufacturing process of steel profiles. The goal is to design a control that yields moderate temperature gradients when the rail is cooled down. The mathematical model corresponds to the boundary control for a 2-D heat equation. A finite element discretization, followed by adaptive refinement of the mesh, results in the example in this benchmark. The problem dimensions depend of the discretization mesh; the two versions employed are $\texttt{STEEL\_I}_{1357}$ with $n = 1,357$, $m = 7$, $p = 6$; and $\texttt{STEEL\_I}_{5177}$ with $n = 5,177$, $m = 7$, $p = 6$.

Table 1 summarizes the results (in seconds) obtained with all the implementations evaluated. The execution time dedicated to solve the Lyapunov equation is shown in column 2; columns 3, 4 and 5 report the time required to initialize matrix H, compute sign(H) and solve the overdetermined system, respectively; column 6 displays the accumulated time. All the times given in Table 1 include the costs to perform all the necessary CPU-GPU data transfers.

Note that most of the time is dedicated to compute the solution of the Riccati equation, in particular the computation of sign(H) (column 4). The rest of the time is basically spent in the Lyapunov equation solver. A careful study of these

[2] `http://www.imtek.de/simulation/benchmark/`

Table 1. Execution time (in secs.) of the different steps of the BST method for the STEEL_I problem

Implementation	Lyapunov solver	H init.	sign(H)	System solver	Total time(s)
STEEL_I$_{1357}$					
BST_SCPU	7.74	0.10	118.15	3.23	129.22
BST_MTCPU	1.68	0.05	22.34	0.57	24.64
BST_HYB	9.46	0.05	10.93	0.57	21.01
STEEL_I$_{5177}$					
BST_SCPU	334.16	1.52	6404.65	325.34	7065.67
BST_MTCPU	63.75	0.86	1127.87	25.05	1217.53
BST_HYB	26.82	0.78	292.93	24.92	345.48

two operations demonstrates that the computational effort is concentrated in the calculation of matrix inverses. This operation is accelerated in the BST_MTCPU implementation using multi-thread codes. The BST_HYB variant improves the parallelization of the matrix inversion procedure using the Gauss-Jordan elimination method, which is more suitable for its execution on parallel architectures, and off-loading part of the computations to the GPU.

The times reported for the STEEL_I$_{1357}$ instance show a notable benefit from the usage of the multicore (BST_MTCPU) and the hybrid (BST_HYB) implementation, which are respectively 5 and 6 times faster than the sequential implementation. From the results obtained for STEEL_I$_{5177}$ we can conclude that these differences are even higher for larger problems. In this case, BST_MTCPU is nearly 6 times faster than BST_SCPU, while BST_HYB is more than 20 times faster. The reason is that larger problems present a higher inherent parallelism which can be leveraged by the massively parallel architecture of the GPU.

5 Concluding Remarks

We have presented two high performance parallel implementations for the BST method for model reduction. Variant BST_MTCPU is optimized for its execution on a multicore CPU, while BST_HYB targets hybrid platforms composed of a CPU and a GPU. BST_HYB exploits the capabilities of both architectures, the multicore CPU and the many-core GPU, yielding a high performance implementation of the BST model reduction technique. Two levels of parallelism are exploited in this implementation: at the inner level, multithread lineal algebra kernels included in the BLAS library (MKL and CUBLAS) are employed to compute the most time-consuming linear algebra operations; at the outer level, operations proceed concurrently in both architectures, overlapping computations on the CPU and the GPU.

Experimental results on a platform consisting of a state-of-the-art general-purpose multi-core processor and a pre-Fermi GPU show that model order reduction of large-scale linear systems can be significantly accelerated using this kind of platforms.

The promising results obtained encourage us to further improve the developed implementations. On-going and future work include:

– Exploit the use of multiple GPUs to further reduce the computational time and increase the dimension of the affordable problems.
– Evaluate the use of mixed precision techniques that allow to perform most of the computations in single precision arithmetic.

References

1. Antoulas, A.: Approximation of Large-Scale Dynamical Systems. SIAM Publications, Philadelphia (2005)
2. Benner, P., Quintana-Ortí, E.S., Quintana-Ortí, G.: Efficient numerical algorithms for balanced stochastic truncation. Internat. J. in Applied Mathematics and Computer Science 1(1), 15–21 (2005)
3. Freund, R.: Reduced-order modeling techniques based on Krylov subspaces and their use in circuit simulation. In: Datta, B. (ed.) Applied and Computational Control, Signals, and Circuits, vol. 1, ch. 9, pp. 435–498. Birkhäuser, Boston (1999)
4. Volkov, V., Demmel, J.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49 (May 2008), http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html
5. Bientinesi, P., Igual, F.D., Kressner, D., Quintana-Ortí, E.S.: Reduction to Condensed Forms for Symmetric Eigenvalue Problems on Multi-core Architectures. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 387–395. Springer, Heidelberg (2010)
6. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S., Quintana-Ortí, G.: Exploiting the capabilities of modern GPUs for dense matrix computations. Concurrency and Computation: Practice and Experience 21, 2457–2477 (2009)
7. Ltaif, H., Tomov, S., Nath, R., Du, P., Dongarra, J.: A scalable high performance cholesky factorization for multicore with gpu accelerators. University of Tennessee, LAPACK Working Note 223 (2009)
8. Benner, P., Ezzatti, P., Quintana-Ortí, E.S., Remón, A.: Using Hybrid CPU-GPU Platforms to Accelerate the Computation of the Matrix Sign Function. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009. LNCS, vol. 6043, pp. 132–139. Springer, Heidelberg (2010)
9. Desai, U., Pal, D.: A transformation approach to stochastic model reduction. IEEE Trans. Automat. Control AC–29, 1097–1100 (1984)
10. Green, M.: Balanced stochastic realization. Linear Algebra Appl. 98, 211–247 (1988)
11. Varga, A., Fasol, K.H.: A new square–root balancing–free stochastic truncation model reduction algorithm. In: Prepr. 12th IFAC World Congress, Sydney, Australia, vol. 7, pp. 153–156 (1993)

12. Varga, A.: Task II.B.1 – selection of software for controller reduction. The Working Group on Software (WGS), SLICOT Working Note 1999–18 (December 1999), http://www.win.tue.nl/niconet/NIC2/reports.html
13. Benner, P., Quintana-Ortí, E.S., Quintana-Ortí, G.: State-space truncation methods for parallel model reduction of large-scale systems. Parallel Comput. 29, 1701–1722 (2003)
14. Anderson, B.: An algebraic solution to the spectral factorization problem. IEEE Trans. Automat. Control AC-12, 410–414 (1967)
15. Anderson, B.: A system theory criterion for positive real matrices. SIAM J. Cont. 5, 171–182 (1967)
16. Benner, P., Quintana-Ortí, E.S., Quintana-Ortí, G.: Solving linear-quadratic optimal control problems on parallel computers. Optimization Methods Software 23(6), 879–909 (2008)
17. Roberts, J.: Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. Internat. J. Control 32, 677–687 (1980); Reprint of Technical Report No. TR-13, CUED/B-Control, Cambridge University, Engineering Department (1971)
18. Golub, G., Van Loan, C.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
19. Benner, P., Byers, R., Quintana-Ortí, E.S., Quintana-Ortí, G.: Solving algebraic Riccati equations on parallel computers using Newton's method with exact line search. Parallel Comput. 26(10), 1345–1368 (2000)
20. Gerbessiotis, A.V.: Algorithmic and Practical Considerations for Dense Matrix Computations on the BSP Model, Oxford University Computing Laboratory, PRG-TR 32 (1997), http://web.njit.edu/~alexg/pubs/papers/PRG3297.ps
21. Ezzatti, P., Quintana-Ortí, E.S., Remón, A.: Using graphics processors to accelerate the computation of the matrix inverse. The Journal of Supercomputing (2011), http://dx.doi.org/10.1007/s11227-011-0606-4, doi:10.1007/s11227-011-0606-4
22. Strazdins, P.: A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, Tech. Rep. TR-CS-98-07 (1998)
23. Benner, P., Saak, J.: A semi-discretized heat transfer model for optimal cooling of steel profiles. In: Benner, P., Mehrmann, V., Sorensen, D. (eds.) Dimension Reduction of Large-Scale Systems. Lecture Notes in Computational Science and Engineering. Springer, Heidelberg (2005)

Reducing Thread Divergence in GPU-Based B&B Applied to the Flow-Shop Problem

Imen Chakroun[1], Ahcène Bendjoudi[2], and Nouredine Melab[1]

[1] Université Lille 1 CNRS/LIFL, INRIA Lille Nord Europe
Cité scientifique - 59655, Villeneuve d'Ascq cedex, France
{imen.chakroun,nouredine.melab}@lifl.fr

[2] CEntre de Recherche sur l'Information Scientifique et Technique (CERIST)
Division Théorie et Ingénierie des Systèmes Informatiques DTISI,
3 rue des frères Aissou, 16030 Ben-Aknoun, Algiers, Algeria
abendjoudi@cerist.dz

Abstract. In this paper,we propose a pioneering work on designing and programming B&B algorithms on GPU. To the best of our knowledge, no contribution has been proposed to raise such challenge. We focus on the parallel evaluation of the bounds for the Flow-shop scheduling problem. To deal with thread divergence caused by the bounding operation, we investigate two software based approaches called thread data reordering and branch refactoring. Experiments reported that parallel evaluation of bounds speeds up execution up to 54.5 times compared to a CPU version.

Keywords: Branch and Bound, Data Parallelism, GPU Computing, Thread Divergence, Flow-shop Scheduling.

1 Introduction

Solving to optimality large size combinatorial optimization problems using a Branch and Bound algorithm (B&B) is CPU time intensive. Although B&B allows to reduce considerably the exploration time using a bounding mechanism, the computation time remains significant and the use of parallelism to speed up the execution has become an attractive way out. Because of their tremendous computing power and remarkable cost efficiency, GPUs (Graphic Processing Units) have been recently revealed as a powerful way to achieve high performance on long-running scientific applications [9]. However, while several parallel B&B strategies based on large computer clusters and grids have been proposed in the litterature [7], to the best of our knowledge no contribution has been proposed for designing B&B algorithms on GPUs. Indeed, the efficient parallel B&B approaches proposed in the literature [2] do not immediately fit GPU architecture and have to be revisited.

B&B algorithms are characterized by four basic operations: branching, bounding, selection and elimination. For most combinatorial problems, bounding is a very time consuming operation. Indeed, a bounding function is used to compute the estimated optimal solution called *lower bound* of the problem being tackled.

For this reason, and in order to reach higher computing performance, we focus on a GPU based B&B algorithm using a parallel evaluation of the bounds. This parallel strategy is a node-based approach. It does not aim to modify the search trajectory, neither the dimension of the B&B tree nor its exploration. The main objective is to speed up the evaluation of the lower bounds associated to the sub-problems using a GPU CUDA-based computing without changing the semantics of the execution.

The design and programming paradigm proposed in CUDA is based on the Simple Program Multiple Data (SPMD) model. However, its execution model is Single Instruction Multiple Data (SIMD) which is well suited for regular functions (kernels) but represent a challenging issue for irregular computations. In this paper, we address such issue on the Flow-shop scheduling problem for which the bounding function is irregular leading to thread divergence. Indeed, if sub-problems evaluated in parallel by a warp of threads (32 threads in the G80 GPU model) are located at different levels of the search tree, the threads may diverge. This means that at a given time they execute different instruction flows. This behavior is due to the bounding function for the Flow-shop problem which is composed of several conditional instructions and loops that depend on the data associated to the sub-problem on which it is applied. We investigate two approaches called thread data reordering and branch refactoring to deal with thread divergence for the Flow-shop scheduling on GPU.

The remainder of the paper is organized as follows: in Section 2, we present the different B&B parallel existing models focusing on the parallel evaluation of bounds. We also highlight the issues and challenges to deal with the irregular nature of the bounding operation of the Flow-shop problem. In Section 3, we analyse the thread divergence scenarios for this problem, our case study. While in Section 4, we show how to reduce thread divergence using a judicious thread data remapping, we detail in Section 5 some software optimizations useful to get around control flow instructions. Finally, some perspectives of our work are proposed in Section 6.

2 GPU-Based Parallel B&B: Issues and Challenges

Solving exactly a combinatorial optimization problem consists in finding the solution having the optimal cost. For this purpose, the B&B algorithm is based on an implicit enumeration of all the solutions of the problem being solved. The space of potential solutions (search space) is explored by dynamically building a tree which root node represents the initial problem. The leaf nodes are the possible solutions and the internal nodes are subspaces of the total search space. The construction of such a tree and its exploration are performed using four operators: branching, bounding, selection and pruning. The bounding operation is used to compute the estimated optimal solution called "lower bound" of the problem being tackled. The pruning operation uses this bound to decide whether to prune the node or to continue its exploration. A selection or exploration strategy selects one node among all pending nodes according to defined priorities.

The priority of a node could be based on its depth in the B&B tree which leads to a depth-first exploration strategy. A priority based on the breadth of the node is called a breadth-first exploration. A best first selection strategy could also be used. It is based on the presumed capacity of the node to yield good solutions.

Thanks to the pruning operator, B&B allows to reduce considerably the computation time needed to explore the whole solution space. However, the exploration time remains significant and parallel processing is thus required. In [7], three parallel models are identified for B&B algorithms: (1) the parallel multi-parametric model (2), the parallel tree exploration, and (3) the parallel evaluation of the bounds. The model (1) consists in launching simultaneously several B&B processes. These processes differ by one or more operator(s), or have the same operators differently parameterized. The trees explored in this model are not necessarily the same. Model (2) consists in launching several B&B processes to explore simultaneously different paths of the same tree.

Unlike the two previous models, model (3) suppose the launching of only one B&B process. It does not assume to parallelize the whole B&B algorithm but only the bounding operator. Each calculation unit evaluates the bounds of a distinct pool of nodes. This approach perfectly suits GPU computing. In fact, bounding is often a very time consuming operation. In this paper, we focus on the design and implementation of a B&B algorithm on GPU based on the parallel evaluation of the lower bounds.

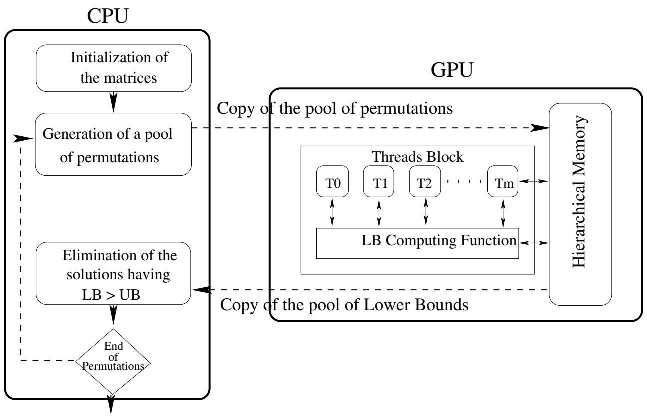

CPU
Initialization of the matrices
Generation of a pool of permutations
Elimination of the solutions having LB > UB
End of Permutations
GPU
Copy of the pool of permutations
Threads Block
T0
T1
T2
Tm
LB Computing Function
Hierarchical Memory
Copy of the pool of Lower Bounds

Fig. 1. GPU-based evaluation of bounds

As illustrated in Figure 1, the idea is to generate a pool of subnodes on CPU using the branching operator and to send it to GPU where each thread handles one node. The subnodes are then evaluated in parallel, the resulting lower bounds are moved back to CPU where the remaining selection and elimination operators are applied.

Using Graphics Processing Units have become increasingly popular in High-Performance Computing. A large number of optimizations have been proposed to improve the performance of GPU programs. The majority of these optimizations target the GPU memory hierarchy by adjusting the pattern of accesses to the device memory [9]. In contrast, there has been less work on optimizations that tackle another fundamental aspect of GPU performance, namely its SIMD execution model. This is the main challenge we are facing in our work. When a GPU application runs, each GPU multiprocessor is given one or more thread block(s) to execute. Those threads are partitioned into warps[1] that get scheduled for execution. At any clock cycle, each processor of the multiprocessor selects a half warp that is ready to execute the same instruction on different data. The GPU SIMD model assumes that a warp executes one common instruction at a time. Consequently, full efficiency is realized when all 32 threads of a warp agree on their execution path. However, if threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken. Threads that are not on that path are disabled, and when all paths complete, the threads converge back to the same execution path. This phenomenon is called *thread divergence* and often causes serious performance degradations.

The parallel evaluation of bounds is a node-based parallelism. This feature implies an irregular computation depending on the data of each node. Irregularities calculation are reflected in several flow control instructions that would conduct to different behaviors. As we explained before, such data-dependent conditional branches are the main cause of thread divergence. In the following section, we discuss such conditional instruction we encounter in the lower bound of the Flow-shop permutation problem.

3 Thread Divergence in the Flow-Shop Lower Bound

The permutation Flow-shop problem is a very known NP-hard combinatorial optimization problem. It can be formulated as a set of N jobs J_1, J_2..J_N to be scheduled in the same order on M machines. The machines are critical resources as each machine can not be simultaneously assigned to two jobs. Each job J_i is composed of M consecutive tasks ti_1..ti_M, where t_{ij} designates the j^{th} task of the job J_i requiring the machine M_j. To each task t_{ij} is associated a processing time p_{ij}. The goal is to find a permutation schedule that minimizes the total processing time called makespan.

The effectiveness of B&B algorithms resides in the use of a good estimation (lower bound for the maximization problem) of the optimal solution. In that purpose we use the most known lower bound for the permutation Flow-shop problem with M machines; the one proposed by Lageweg et al. [6] with $O(M^2 N log N)$ complexity. This bound is based mainly on Johnson's theorem [5], which provides a procedure for finding an optimal solution with 2 machines. Johnson algorithm assumes to assign jobs at the beginning or at the end of the schedule depending of the processing time of that job.

[1] We assume using the G80 model in which a warp is a pool of 32 threads.

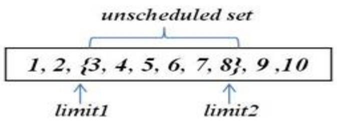


Fig. 2. Representation of the thread input

Starting from this principle of Johnson's algorithm, we designed a thread input as a set of unscheduled jobs, an index representing the start of the range of unscheduled jobs namely LIMIT1 and an index addressing the end of the range of the unscheduled jobs namely LIMIT2 (see Figure 2). Each thread would pick one of the unscheduled jobs, schedule it and calculate the corresponding makespan.

The Flow-shop permutation lower bound we adopted clearly provides a good estimation of the cost of a solution. However, its implementation on GPU perfectly echoes the thread divergence. Actually, it counts almost a dozen of control instructions namely "if" and "for". The example above shows a piece of our code that exhibits thread divergence.

```
int thread_idx = blockIdx.x * blockDim.x + threadIdx.x;

1. if( pool[thread_idx].limit1 != 0 )
      time = TimeMachines[1] ;
   else
      time = TimeArrival[1] ;

2. if( TimeMachinesEnd[pool[thread_idx].permutation[0]] > minima )
   {
      nbTimes++ ;
      minima = TimeMachinesEnd[pool[thread_idx].permutation[0]];
   }

3. for(int k = 0 ; k < pool[thread_idx].limit1; k++)
      jobTime = jobEnd[k] ;
```

Consider the first "if" scenario. Let us suppose the values of LIMIT of the first 31 threads of a warp are not null except one. When that warp encounters the conditional instruction "if", only one thread passes through the condition checking and performs the assignment instruction. All the other 31 threads will be idle waiting for the thread 32 to be completed. The big deal in this case is that no other warps are allowed to run on that multiprocessor meanwhile because the warp is not completely idle. The same problem is encountered with the "for" loop. Suppose LIMIT1 of the first 31 threads of a warp is null while its value for the thread 32 is quite important. In that case, the 31 threads have to stay idle and wait until the other thread finishes its loop. The gap could be quite important since the value of LIMIT1 could be high depending on the size of the permutation being evaluated.

Some present techniques for handling branch divergence either demand hardware support [1] or require host-GPU interaction [11], which incurs overhead. Some other works such as [3] intervene at the code level. They expose a branch distribution method that aims to reduce the divergent portion of a branch by factoring out structurally similar code from the branch paths. In our work, we have also opted for software-based optimizations like [3]. In fact, we figure out how to literally rewrite the branching instructions into basic ones in order to make thread execution paths uniform. We also demonstrate that we could ameliorate performances only by judiciously reordering data being assigned to each thread.

4 Thread-Data Reordering

As explained in Section 2, any flow control instruction (if, switch, for, while) can significantly affect the instruction throughput by causing threads of the same warp to diverge. If this happens, the different paths are executed in a serial way, increasing the total amount of instructions executed for this warp. It is important here to note that the threads execution path are data-dependent that is the data input set of a thread determines its behavior in a given kernel. Starting from this observation, we propose to reorder the data sets that the GPU threads work on.

The purpose of thread-data reordering is essentially to find an appropriate mapping between threads and input sets. In our work, we propose a reordering based on the data of the thread rather than its identifier like it is usually done [11]. Indeed, since the data of a given sub problem depend on its level in the search tree, the idea is to generate the pool of nodes to be evaluated in parallel from the same level unless from close levels on the tree. To do so we used the breadth-first exploration strategy (BFS) to generate the pool. Breadth first exploration expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. Initially the pool contains just the root. At each iteration, the node at the head of the pool is expanded. The generated nodes are then added to the tail of the pool. Using breadth-first branching guarantees that nodes belonging to the same level in the tree have much in common than other nodes generated by other decomposition paradigms namely depth first (DFS) or best first (BEFS) branching. Particularly in our case, nodes generated from the same father node have the same LIMIT1 and LIMIT2 but have different jobs to schedule.

To evaluate the performance of the proposed approach we run experiments over Flow-shop instances proposed by Taillard in [10]. Taillard's benchmarks are the best known problem instances for basic model with makespan objective. Each Taillard's instance NxM defines the number N of jobs to be scheduled and the number of machines M on which the jobs are scheduled. For each experiment, different pool sizes and problem instances are considered. The approach has been implemented using C-CUDA 4.0. The experiments have been carried out using a an Intel Xeon E5520 bi-processor coupled with a GPU device. The bi-processor is 64-bit, quad-core and has a clock speed of 2.27GHz. The GPU device is an Nvidia Tesla C2050 with 448 CUDA cores (14 multiprocessors with 32 cores each) and a 2.8GB global memory.

Table 1. Time measurements on GPU without data reordering (using DFS) and with data reordering (using BFS)

Pool Size \ Instances		20x20	50x20	100x20	200x20	500x20
4096	DFS	191.1	214.41	242.28	283.4	383.84
16x256	BFS	186.75	204.58	238.1	275.43	366.47
8192	DFS	198.73	218.53	248.39	293.69	408.2
32x256	BFS	194.22	209.26	241.46	287.18	404.72
16384	DFS	224.82	253.36	300.21	351.14	546.62
64x256	BFS	216.86	235.78	291.10	325.93	521.80
32768	DFS	231.02	276.91	340.15	426.98	761.38
128x256	BFS	219.5	259.16	316.5	410.52	711.86
65536	DFS	269.8	318.52	405.8	568.76	1133.97
256x256	BFS	253.96	301.81	380.95	543.84	1090.22

The obtained results are reported in Table 1. Each pool size in the first column is expressed as bxt where b and t designates respectively the number of blocks and the number of threads within a block. Each node in the pool is evaluated by exactly one thread. We notice that although the speed-up is not impressive, reordering data makes the kernel run faster than with a pool generated via a DFS exploration strategy. For a same pool size, the acceleration grows accordingly with the permutation size associated to the instance. For example, in the instance corresponding to the scheduling of 500 jobs over 20 machines, the permutation size is equal to 500. For this instance, a pool of 4096 sub-problems would contain nodes from the same level of the exploration tree whereas the same pool generated in instance 20 jobs over 20 machines would have sub-problems from different levels. This explains why the acceleration follows the permutation size behavior. Generating the pool to evaluate using a breadth first strategy guarantees for large instances to have nodes from almost the same level. This allows to reduce the impact of conditional instructions that depends of the values of LIMIT1 and LIMIT2.

5 Branch Refactoring

To reduce the divergence caused by conditional instructions, we use the branch refactoring approach. This latter consists in rewriting the conditional instructions so that threads of the same warp execute an uniform code avoiding their divergence. To do that, we study in the following different "if" scenarios and propose some optimizations accordingly. Consider a generalization of the if-else statement of the scenario (1) exposed in the Section 3. In this case, the conditional expression compares the content of a variable to 0. The idea is to replace this conditional expression by two functions namely f and g as explained in the Equation [1].

$$\begin{array}{lcl} if(x \neq 0) & & if(x \neq 0) \\ \quad a = b[1]; & & \quad a = b[1] + 0 \times c[1]; \\ else & \Rightarrow & else \\ \quad a = c[1]; & & \quad a = 0 \times b[1] + c[1]; \end{array}$$

$$\Rightarrow a = f(x) \times b[1] + g(x) \times c[1];$$

$$where:\ \text{f(x)}=\begin{cases} f(x) = 0\ if \quad x = 0 \\ 1 \qquad\qquad else \end{cases} \quad and \quad \text{g(x)}=\begin{cases} g(x) = 1\ if \quad x = 0 \\ 0 \qquad\qquad else \end{cases} \tag{1}$$

The behavior of f and g fits the trigonometric function cosinus. This function returns values between 0 and 1. Particularly, we defined an integer variable to which we assign the result of the cosinus function as quoted in the above code. The value taken would only be 0 or 1 since it would be rounded to 0 if it is not equal to 1. In order to increase performance we used CUDA runtime math operations: __sinf(x), __expf(x) and so forth. Those functions are mapped directly to the hardware level [8]. They are faster but provide lower accuracy which does not matter in our case because we do round results to int. The throughput of __sinf(x), __cosf(x), __expf(x) is 1 operation per clock cycle [8].

Result of branch rewriting for the scenario (1)

```
int coeff = __cosf(pool[tid].limit1);
time = (1 - coeff) * TimeMachines[1] + coeff * TimeArrival[1];
```

Let us now consider a scenario with an "if" statement which compares two values between themselves like shown in Equation 2.

$$\begin{array}{ll} & if(x \gneqq y) \quad a = b[1]; \quad \Rightarrow \quad if(x - y \geq 1) \quad a = b[1]; \\ \Rightarrow & if(x - y - 1 \geq 0) \qquad a = b[1]; \qquad (x, y) \in N \\ \Rightarrow & a = f(x, y) \times b[1] + g(x, y) \times a; \\ & where: \qquad \text{f(x,y)}=\begin{cases} 1\ if\ x - y - 1 \geq 0 \\ 0\ if\ x - y - 1 < 0 \end{cases} \\ & and \qquad \text{g(x,y)}=\begin{cases} 0\ if\ x - y - 1 \geq 0 \\ 1\ if\ x - y - 1 < 0 \end{cases} \end{array} \tag{2}$$

We do the same transformations than before using the exponential function. The exponential is a positive function which is equal to 1 when applied to 0. Thus, if x is less than y __expf(x-y-1) returns a value between 0 and 1. If we round this result to an integer value we obtain 0. Now, if x is greater than y __expf(x-y-1) return a value greater than 1 and since we get the minimum between 1 and the exponential, the returned result would be 1. This behavior exactly satisfies our prerequisites. The "if" instruction is now equivalent to:

```
int coeff = min(1, __expf(x - y - 1));
a = coeff * b[1] + ( 1 - coeff ) * a ;
```

The effectiveness of both transformations was tested on the same configuration used in Section 4. Table 2 compares the parallel efficency of the GPU-based evaluation of bounds with and without using code optimizations. The reported speed ups are calculated relatively to the sequentiel version considering the ratio between the measured execution times.

Table 2. Parallel speedup obtained with/without code optimization

Pool Size \ Instances		20x20	50x20	100x20	200x20	500x20
4096	refactored	1.17	2.14	3.24	6.93	10.24
16x256	basic	1.05	1.89	3.06	5.17	9.66
8192	refactored	2.25	4.20	6.35	10.77	18.44
32x256	basic	2.01	3.72	5.99	9.88	17.39
16384	refactored	4.00	7.96	10.52	18.92	28.55
64x256	basic	3.58	7.04	9.92	17.36	26.93
32768	refactored	7.99	11.90	19.52	30.02	41.76
128x256	basic	7.13	10.53	18.42	27.54	39.40
65536	refactored	10.43	15.15	32.38	45.76	54.53
256x256	basic	9.31	13.41	30.55	41.98	51.44

The reported accelerations no doubtly prove that bound evaluation parallelization on top of GPU provides an efficient way for speed up B&B algorithms. In fact, the GPU-based evaluation runs up to 51.44 times faster than the CPU one. The other important result, is that using thread divergence reduction improves the classic GPU acceleration. This acceleration improvement grows acccordingly to the size of the problem instance and the size of the pool of sub-problems considered in the experiment. Indeed, with a pool of 65536 nodes and an instance of 500 jobs and 20 machines a speed up of x54,5 is achieved while it reaches only x10,2 with a pool of 4096 nodes.

6 Conclusion and Future Work

In this work, we have investigated using GPUs to improve the performance of B&B algorithms. To the best of our knowledge, no contribution has been proposed to raise such challenge. We focused on the parallel evaluation of the bounds for the Flow-shop permutation problem. In order to face out irregularities caused by data dependent branching and leading to thread divergence, we have proposed some software based optimizations. Experiments reported that parallel evaluation of bounds speed up executions up to 54.5 times compared to a CPU version. This promising results could be easily improved when the approach is combined with an optimized data access to GPU memory spaces.

As future contribution, we aim to focus on memory management issues related to data inputs for combinatorial optimization problems. Indeed, when working with such structures usually large in size many memory transactions are performed leading to a global loss of performance. Our direction for future work is also to generate the pool of subproblems directly on GPU. This modification would reduce the transfer time of data structures from CPU to GPU. The challenging issue of this approach is to find an efficient mapping between a thread id and the nodes to generate.

References

1. Fung, W., Sham, I., Yuan, G., Aamodt, T.: Dynamic warp formation and scheduling for efficient gpu control flow. In: MICRO 2007: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, pp. 407–420 (2007)
2. Gendron, B., Crainic, T.G.: Parallel Branch and Bound Algorithms: Survey and Synthesis. Operations Research 42, 1042–1066 (1994)
3. Han, T., Abdelrahman, T.S.: Reducing branch divergence in GPU programs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4), Article 3, 8 pages. ACM, New York (2011)
4. Jang, B., et al.: Exploiting memory access patterns to improve memory performance in data-parallel architectures. IEEE Trans. on Parallel and Distributed Systems 22(1), 105–118 (2011)
5. Johnson, S.M.: Optimal two and three-stage production schedules with setup times included. Naval Research Logistis Quarterly 1, 61–68 (1954)
6. Lenstra, J.K., Lageweg, B.J., Rinnooy Kan, A.H.G.: A General bounding scheme for the permutation Flow-shop problem. Operations Research 26(1), 53–67 (1978)
7. Melab, N.: Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul. HDR thesis, LIFL, USTL (Novembre 2005)
8. NVIDIA CUDA C Programming Best Practices Guide, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf
9. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.-Z., Baghsorkhi, S.S., Hwu, W.W.: Program optimization carving for gpu computing. J. Parallel Distributed Computing 68(10), 1389–1401 (2008)
10. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of European Research 23, 661–673 (1993)
11. Zhang, E.Z., Jiang, Y., Guo, Z., Shen, X.: Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In: Proceedings of the 24th ACM International Conference on Supercomputing (ICS 2010), pp. 115–126. ACM, New York (2010)

A GPU-Based Approximate SVD Algorithm

Blake Foster, Sridhar Mahadevan, and Rui Wang

Department of Computer Science
Univ. of Massachusetts, Amherst, MA 01003, USA
{blfoster,mahadeva,ruiwang}@cs.umass.edu

Abstract. Approximation of matrices using the Singular Value Decomposition (SVD) plays a central role in many science and engineering applications. However, the computation cost of an exact SVD is prohibitively high for very large matrices. In this paper, we describe a GPU-based approximate SVD algorithm for large matrices. Our method is based on the QUIC-SVD introduced by [6], which exploits a tree-based structure to efficiently discover a subset of rows that spans the matrix space. We describe how to map QUIC-SVD onto the GPU, and improve its speed and stability using a blocked Gram-Schmidt orthogonalization method. Using a simple matrix partitioning scheme, we have extended our algorithm to out-of-core computation, suitable for very large matrices that exceed the main memory size. Results show that our GPU algorithm achieves 6∼7 times speedup over an optimized CPU version of QUIC-SVD, which itself is orders of magnitude faster than exact SVD methods.

Keywords: SVD, GPU, cosine trees, out-of-core computation.

1 Introduction

The Singular Value Decomposition (SVD) is a fundamental operation in linear algebra. Matrix approximation using SVD has numerous applications in data analysis, signal processing, and scientific computing. Despite its popularity, the SVD is often restricted by its high computation cost, making it impractical for very large datasets. In many practical situations, however, computing the full-matrix SVD is not necessary; instead, we often need only the k largest singular values, or an approximate SVD with controllable error. In such cases, an algorithm that computes a low-rank SVD approximation is sufficient, and can significantly improve the computation speed for large matrices.

A series of recent papers have studied using matrix sampling to solve the low-rank matrix approximation (LRMA) problem. These algorithms construct a basis made up of rows or linear combinations of rows sampled from the matrix, such that the projection of the matrix onto the basis has bounded error (more specifically, the error is statistically bounded with high probability). Common sampling-based methods include length-squared sampling [4,2] and random projection sampling [3,12]. In this paper, we focus on a method called QUIC-SVD, recently introduced by [6]. QUIC-SVD exploits a tree-based structure to perform fast sampled-based SVD approximation with automatic error control. The main benefit compared to previous work is that it iteratively selects samples that are both adaptive and representative.

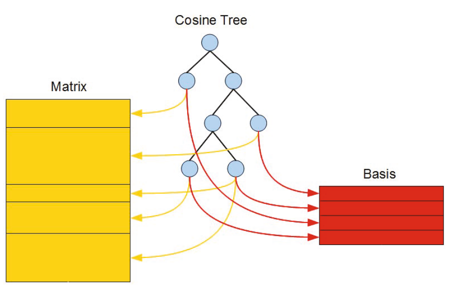
Our goal is to map the QUIC-SVD algorithm onto the graphics processing unit (GPU) to further improve its efficiency. Modern GPUs have emerged as low-cost massively parallel computation platforms that provide very high floating point performance and memory bandwidth. In addition, the availability of high-level programming languages such as CUDA has significantly lowered the programming barrier for the GPU. These features make the GPU a suitable and viable solution for solving many computationally intensive tasks in scientific computing. We describe how we implemented the QUIC-SVD algorithm on the GPU, and demonstrate its speedup (about 6$\sim$7 times) over an optimized CPU version, which itself is orders of magnitude faster than exact SVD methods. We also describe a matrix partitioning scheme that easily adapts the algorithm to out-of-core computation, suitable for very large matrices. We have tested our algorithm on dense matrices up to 22,000$\times$ 22,000, as reported in Section 3.

Related Work. Acceleration of matrix decomposition algorithms on modern GPUs has received significant attention in recent years. Galoppo et al. [5] reduced matrix decomposition and row operations to a series of rasterization problems on the GPU, and Bondhugula et al. [1] provided a GPU-based implementation of SVD using fragment shaders and frame buffer objects. Since then, the availability of general programming language such as CUDA has made it possible to program the GPU without relying on the graphics pipeline. In [13], a number of matrix factorization methods are implemented using CUDA, including LU, QR and Cholesky, and considerable speedup is achieved over optimized CPU algorithms. GPU-based QR decomposition was also studied by [9] using blocked Householder reflections. Recently, Lahabar et al. [10] presented a GPU-based SVD algorithm built upon the Golub-Reinsch method. They achieve up to 8$\times$ speedup over an Intel MKL implementation running on dual core CPU. Like most existing work (including commercial GPU-based linear algebra toolkit such as CULA [7]), their focus is on solving the exact SVD. In contrast, our goal is to solve approximate SVD on the GPU, which can provide additional performance gain for many large-scale problems in practical applications.

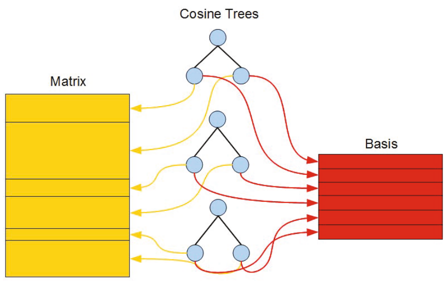
2 Algorithm

2.1 Overview

Given an $m \times n$ matrix A (where n is the smaller dimension), the SVD factors A into the product of three matrices: $A = U\Sigma V^{\mathrm{T}}$ where U and V are both orthogonal matrices ($U^{\mathrm{T}}U = I$ and $V^{\mathrm{T}}V = I$) and Σ is a diagonal matrix storing the singular values. An exact SVD takes $O(mn^2)$ time to compute and thus is expensive for large matrices. To approximate the SVD, we can construct a subspace basis that captures the intrinsic dimensionality of A by sampling rows or taking linear combinations of rows. The final SVD can be extracted by performing an exact SVD on the subspace matrix, which is a much smaller



(a) Single cosine tree



(b) Partitioned cosine trees

Fig. 1. (a) shows a single cosine tree; (b) shows a set of cosine trees constructed using our partitioning scheme, such that all trees collectively build a common basis set. Yellow arrow indicates the matrix rows that a tree node owns; red arrow indicates a vector inserted into the basis set, which is the mean vector of the rows owned by a node.

than the original matrix. For example, if the intrinsic dimensionality of A is approximately k, where $k \ll n$, the computation cost is now reduced to $O(mnk)$.

The QUIC-SVD [6] is a sample-based approximate SVD algorithm. It iteratively builds a row subspace that approximates A with controlled L_2 error. The basis construction is achieved using a binary tree called *cosine tree*, as shown in Figure 1(a), where each node represents a collection (subset) of the matrix rows. To begin, a root node is built that represents all rows (i.e. the entire matrix A), and the mean (average) vector of the rows is inserted into the initial basis set. At each iteration, a leaf node $\mathbf{n}_s$ in the current tree is selected for splitting. The selection is based on each node's estimated error, which predicts whether splitting a node is likely to lead to a maximal reduction in the matrix approximation error. To perform the splitting, a pivot row $\mathbf{r}_p$ is sampled from the selected node $\mathbf{n}_s$ according to the length-squared distribution. Then, $\mathbf{n}_s$ is partitioned into two child nodes. Each child node owns a subset of rows from $\mathbf{n}_s$, selected by their dot products with the pivot row. Specifically, rows closer to the minimum dot product value are inserted to the left child, and the remaining are inserted to the right child. Finally, the mean vector of each subset is added to the basis set, replacing the mean vector contributed by the parent node.

Figure 1(a) shows the cosine tree constructed at an intermediate step of the algorithm. Each leaf node represents a subset of rows, and contributes a mean vector to the current basis set. As the tree is split further, the basis set expands. The process terminates when the whole matrix approximation error is estimated to fall below a relative threshold:

$$\|A - \widehat{A}\|_F^2 = \|A - A\widehat{V}\widehat{V}^{\mathrm{T}}\|_F^2 \leq \epsilon \|A\|_F^2$$

where $\widehat{V}$ is the approximate row basis set constructed using the algorithm, $\widehat{A} = A\widehat{V}\widehat{V}^{\mathrm{T}}$ is the reconstructed matrix with the approximate SVD, and $\|\cdot\|_F$ denotes the Frobenius norm. This error is calculated using a Monte Carlo estimation routine, which estimates the projection error of A onto the row basis set $\widehat{V}$. The error estimation routine returns an error upper bound with confidence $1 - \delta$, where δ is an adjustable parameter. Therefore, when δ is small, the error is bounded by the returned value with high probability.

The Monte Carlo estimation works as follows. First, randomly select s rows from the matrix using length-squared distribution, where s is logarithmic to the number of rows. Second, project each selected row $\mathbf{r}_i$ to the current basis set $\widehat{V}$, calculate the squared length of the projection $\|\mathbf{r}_i\widehat{V}\|_F^2$, and divide it by the probability of selecting that row (recall the probability is proportional to the squared length of that row). This results in a weighted squared magnitude $wSqMag_i$ for each selected row. Intuitively, if a row is well represented by the basis $\widehat{V}$, the projection will preserve its squared magnitude. Finally, the mean and variance of $wSqMag_i$ is used to model the statistics of all rows in A projected onto $\widehat{V}$, from which the error bound can be calculated. Details can be found in [6].

This Monte Carlo estimation routine is also used to estimate the error contributed by a node, in order to prioritize the selection of nodes for splitting. Intuitively, nodes with large error are not well-approximated by the current basis, and thus splitting them is likely to yield the largest benefit.

Whenever a vector is inserted into the basis set, it is orthogonalized against the existing basis vectors using Gram-Schmidt orthogonalization. This is necessary for the Monte Carlo error estimation and SVD extraction. Once the tree building terminates, the current basis set accurately captures the row subspace of A, and the final SVD can be extracted from the basis set by solving a much smaller SVD problem.

In summary, the main computation loop involves the following steps: 1) select a leaf node with the maximum estimated error; 2) split the node and create two child nodes; 3) the mean vector of each child is inserted into the basis set and orthonormalized (while the one contributed by the parent is removed); 4) estimate the error of each child node; 5) estimate the error of the whole matrix approximation, and terminate when it's sufficiently small. For more details, we refer the reader to [6].

2.2 GPU Implementation

We implemented QUIC-SVD using the CUDA programming language, in conjunction with the CULA [7] library to extract the final SVD. We found that most of the computation is spent on the following two parts: 1) computing vector inner products and row means for node splitting; 2) Gram-Schmidt orthogonalization. Therefore in the following we focus on discussing these two parts. The computation for each tree node is spread across the entire GPU device. Note that as we assume the input matrix is low-rank, the number of nodes we need to create is small relative to the matrix dimensions.

When we split a node, we need to compute the inner product of every row with the pivot row (which is selected by sampling length-squared distribution of the rows). Since a node does not necessarily span contiguous rows, we could not use a simple matrix-vector multiplication call to accomplish this step. Rearranging the rows of each node into contiguous chunks after each split was not an option, as this would incur excessive memory traffic. Instead, we maintain an index array at each node to point to the rows that the node owns, and then use a custom CUDA kernel to compute all inner products in parallel. To maintain memory coherence, we assign each CUDA block to a row. Thus all threads in a block cooperatively work on a single row, which is stored contiguously in memory. Next, the rows are split into two subsets based on their inner products with the pivot row. Specifically, we first use a parallel reduction to compute the minimum and maximum inner product values, then assign a subset label to each row based on whether its inner product value is closer to the minimum or maximum. For each subset we again use a custom CUDA kernel to compute the mean vector, which will be inserted into the basis set.

When we add a new mean vector to the basis, it must be orthonormalized with respect to the existing basis vectors with the Gram-Schmidt process. Given a set of orthonormal basis vectors $\mathbf{v}_1, ..., \mathbf{v}_k$ and a new basis vector $\mathbf{r}$, the classical Gram-Schmidt process would compute

$$\mathbf{r}' = \mathbf{r} - p_{\mathbf{v}_1}(\mathbf{r}) - ... - p_{\mathbf{v}_k}(\mathbf{r}), \quad \text{and} \quad \mathbf{v}_{k+1} = \mathbf{r}'/\|\mathbf{r}'\|$$

where $p_{\mathbf{v}}(\mathbf{r}) = (\mathbf{r} \cdot \mathbf{v})\,\mathbf{v}$ denotes the projection of $\mathbf{r}$ onto $\mathbf{v}$. Both the projection and subtraction can be done in parallel, but the numerical stability is poor. The modified Gram-Schmidt process subtracts the projection vector sequentially:

$$\mathbf{r}_1 = \mathbf{r} - p_{\mathbf{v}_1}(\mathbf{r}); \quad \mathbf{r}_2 = \mathbf{r}_1 - p_{\mathbf{v}2}(\mathbf{r}_1); \quad ... \quad \mathbf{r}' = \mathbf{r}_k = \mathbf{r}_{k-1} - p_{\mathbf{v}_k}(\mathbf{r}_{k-1})$$

This is mathematically the same, but the numerical stability is improved greatly. Unfortunately, this formulation serializes the computation and cannot be easily parallelized.

To exploit the benefits of both, we propose to use a *blocked* Gram-Schmidt process [8], which involves partitioning the basis vectors into κ blocks (subsets). Within each block, we use the classical Gram-Schmidt to gain parallelism; and across blocks we use the modified Gram-Schmidt to gain numerical stability. Specifically, assume the current set of basis vectors is partitioned into the following κ blocks: $V_1, ..., V_\kappa (\kappa \ll k)$. We will then compute

$$\mathbf{u}_1 = \mathbf{r} - \mathrm{GS}(\mathbf{r}, V_1),\ \mathbf{u}_2 = \mathbf{u}_1 - \mathrm{GS}(\mathbf{u}_1, V_2),\ ... \,\mathbf{u}_\kappa = \mathbf{u}_{\kappa-1} - \mathrm{GS}(\mathbf{u}_{\kappa-1}, V_\kappa) = \mathbf{r}'$$

where $\mathrm{GS}(\mathbf{u}, V)$ denotes the standard Gram-Schmidt orthogonalization of $\mathbf{u}$ with respect to basis subset V. Note that when $\kappa = 1$ or $\kappa = k$, the algorithm degenerates to the classical or the modified Gram-Schmidt respectively. We set κ such that each block contains approximately 20 basis vectors, and we have found that this provides a good tradeoff between speed and numerical stability.

Among the other steps, the Monte Carlo error estimation is straightforward to implement on the GPU; selecting a splitting node is achieved with a priority queue [6] maintained on the CPU; and the extraction of the final SVD is performed with the CULA toolkit. The cost of SVD extraction is insignificant as it only involves computing the SVD of a $k \times k$ matrix. The priority queue is implemented on the CPU because it's inefficient to implement such a data structure on the GPU. Moreover, as the priority queue requires only a small amount of data to be transferred between the CPU and GPU, it incurs very little overhead.

2.3 Partitioned Version

To accommodate large datasets, we introduce a partitioned version of the algorithm that can process matrices larger than GPU or even main memory size. While the original QUIC-SVD algorithm [6] did not consider out-of-core computation, we found that the structure of the cosine tree lends itself naturally to partitioning. To begin, we split the matrix A into s submatrices $A_1, ..., A_s$, each containing $\lceil m/s \rceil$ consecutive rows from A. Next, we run QUIC-SVD on each submatrix A_i sequentially. A naive algorithm would then simply merge the basis set constructed for each submatrix A_i to form a basis for the whole matrix. While this would give correct results, it would introduce a lot of redundancy (as each basis set is computed independently), and consequently reduce efficiency.

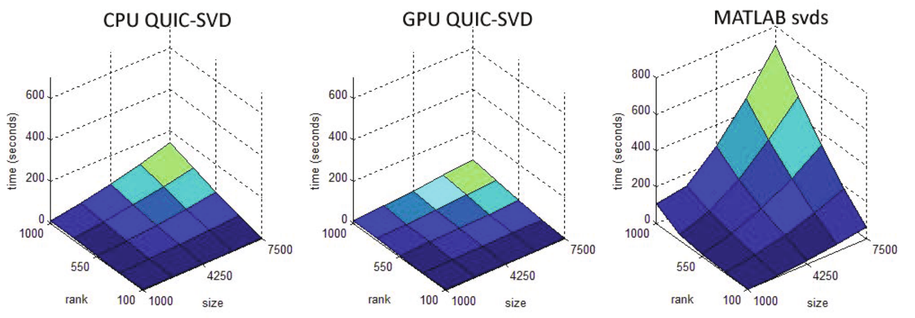
We make a small modification to the algorithm to eliminate redundancy. We build an individual cosine tree for each submatrix A_i, but all submatrices share a common basis set. The algorithm processes the submatrices sequentially in order. When processing submatrix A_i, the corresponding matrix rows are loaded into GPU memory, and a new cosine tree is constructed. The basis set from previous submatrices is used as the initial basis. If the error estimated from this basis is already below the given threshold, the algorithm will stop immediately and proceed to the next submatrix. Intuitively this means submatrix A_i is already well represented by the current basis set, hence no update is necessary. Otherwise, the algorithm processes A_i in the same way as the non-partitioned version, and the basis set is expanded accordingly. Once we are done with the current submatrix, the GPU memory storing the matrix rows is overwritten with the next submatrix.

After a complete pass through every submatrix, we observe that the whole matrix approximation error is equal to the sum of the each subset's approximation error, which is bounded by the given relative error threshold. In other words:

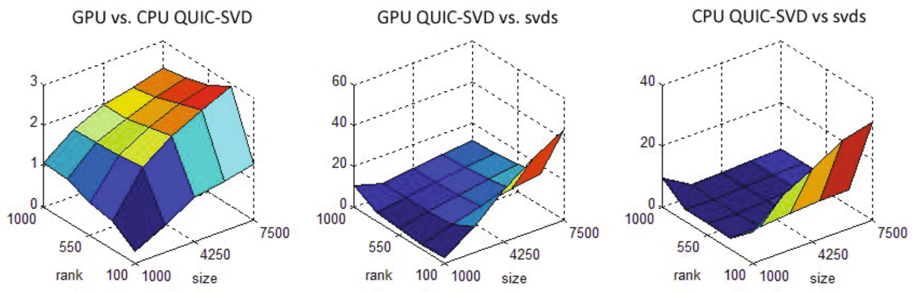
$$\|A - \widehat{A}\|_F^2 = \textstyle\sum_{i=1}^{s} \|A_i - \widehat{A}_i\|_F^2 \leq \sum_{i=1}^{s} \epsilon \|A_i\|_F^2 = \epsilon \|A\|_F^2$$

where $\widehat{A}_i = A_i \widehat{V} \widehat{V}^{\mathrm{T}}$ is the submatrix A_i reconstructed using the row basis V. The equalities in the above equation hold due to the definition of the squared Frobenius norm, which sums over the squares of individual elements. Thus by controlling the relative error of each submatrix, we can bound the error of the whole matrix in the end. Figure 1(b) shows an example of three cosine trees sharing a common basis.

Note that by using partitioning, only a fraction of the matrix data are loaded to GPU memory at a time, allowing for out-of-core computation. However, a downside with this method is that it serializes the processing of submatrices, thus is not suitable for parallel computation on multiple GPU devices.



(a) Running time reported for each of the three algorithms listed.



(b) Plots of speedup factors comparing each pair of algorithms.

Fig. 2. Performance and speedup comparisons for the following three algorithms: CPU QUIC-SVD, GPU QUIC-SVD, and MATLAB's `svds`. The input matrices are randomly generated with size ranging from 1000^2 to 7500^2 and rank ranging from 100 to 1000.

One approach to address this issue would be to process one submatrix independently on each GPU, and then merge the results on a single GPU. The merging step is essentially performing another QUIC-SVD. The related error analysis of this approach remains for future work.

SVD Extraction. Given a matrix $A \in \mathbb{R}^{m\times n}$ and a basis $\widehat{V} \in \mathbb{R}^{n\times k}$, QUIC's SVD-extraction procedure first projects A onto the basis, resulting in an $m \times k$ matrix $P = A\widehat{V}$. It then computes an exact SVD on the $k \times k$ matrix $P^{\mathrm{T}}P$, resulting in $U'\Sigma'V'^{\mathrm{T}} = P^{\mathrm{T}}P$. Note that this step can be replaced by an eigen-decomposition of P. Finally, the approximate SVD of A is extracted as $V = \widehat{V}V'$, $\Sigma = \sqrt{\Sigma'}$, and $U = PV'\Sigma^{-1}$.

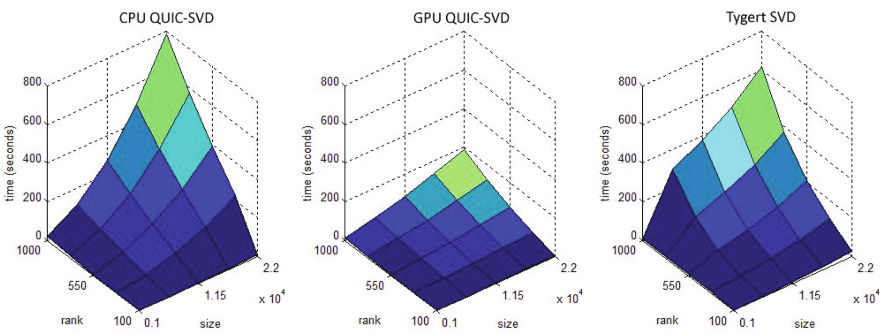
We assume that P can fit in memory, since $k \ll n$. The matrix A cannot fit in memory, so we once again load A into memory one block at a time. Given a block A_i, the corresponding block of $P_i \subset P$ is $A_i\widehat{V}$. After we have completed a pass over all of A, the entire P is in memory. We then proceed with the rest of the computation as described above.

3 Results

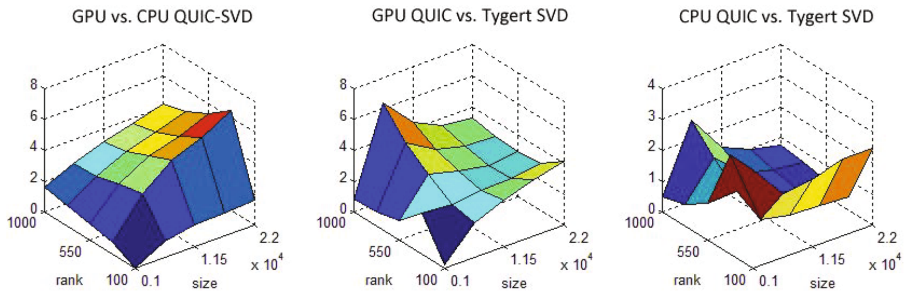
For testing and evaluation, we compared results of our GPU-based algorithm to the following three implementations: 1) a multi-threaded CPU version of QUIC-SVD; 2) MATLAB `svds` routine; and 3) the Tygert SVD [11], which is a fast CPU-based approximate SVD algorithm built upon random projection. To make fair comparisons, we have optimized the CPU version as much as we can. We used Intel Math Kernel Library for linear algebra operations and OpenMP for all applicable loops. In each test case, we plot the running time as well as the speedup over a range of matrix sizes and ranks. We use random matrices for testing. Given a rank k and size n, we first generate an $n \times k$ matrix and a $k \times n$ matrix filled with uniform random numbers between $[-1, 1]$; we then multiply them to obtain an $n \times n$ matrix of rank k. Our experimental results were collected on a PC with an Intel Core i7 2.66 GHz CPU (which has 8 hyperthreads), 6 GB of RAM, and an NVIDIA GTX 480 GPU. Both the CPU and GPU algorithms use double-precision arithmetic. For QUIC-SVD, we set the relative error threshold $\epsilon = 10^{-12}$, and $\delta = 10^{-12}$ (in Monte Carlo error estimation) for all experiments. All timings for the GPU implementation includes both the data transfer time (to and from the GPU) and actual computation time (on the GPU).

Figure 2(a) shows a performance comparison of our GPU implementation vs. the CPU implementation of QUIC-SVD as well as MATLAB `svds`. The matrix size ranges from $1,000^2$ to $7,500^2$ (the largest that `svds` could handle on our system), and the matrix rank ranges from 100 to 1000. All three algorithms were run with the same input and similar accuracy, measured by the L_2 error of SVD approximation. Figure 2(b) plots the speedup factor for the same tests. In addition, we show the speedup factor of the CPU version of QUIC-SVD over `svds`. We observe that the CPU QUIC-SVD is up to 30 times faster than `svds`, and our GPU implementation is up to 40 times faster. In both cases, the maximum speedup is achieved under a large and low-rank matrix. This makes sense because matrices with lower ranks favor the QUIC-SVD algorithm. From this plot we can see that the speedup primarily comes from the QUIC-SVD algorithm itself. If we compare the GPU and the CPU versions of QUIC-SVD alone, the maximum speedup of the GPU version is about 3 times (note that the two have their peak performances at different points). Although this is a moderate speedup, it will become more significant for larger matrices (shown below), as the GPU's parallelism will be better utilized.

Figure 3(a) shows a performance comparison of our GPU and CPU implementations to Tygert SVD [11], which is a very fast approximate SVD algorithm that exploits random projection. Here we set the size of the test matrices to range from $1,000^2$ to $22,000^2$, and the rank to range from 100 to 1000. As a $22,000^2$ (double precision) matrix is too large to fit in GPU memory, we used our partitioned version with 4 partitions. Again all three algorithms were run with the same input and comparable accuracy. Figure 3(b) plots the speedup factor for each pair of the tests. Note that the CPU version and Tygert algorithm have comparable performance, while the GPU version is up to 7 times faster than



(a) Running time reported for each of the three algorithms listed.



(b) Plots of speedup factors comparing each pair of algorithms.

Fig. 3. Performance and speedup comparison for the following three algorithms: CPU QUIC-SVD, GPU QUIC-SVD, and Tygert SVD. The input matrices are randomly generated with size ranging from 1000^2 to 22000^2 and rank ranging from 100 to 1000.

either. Although the GPU version does not perform as well on small matrices due to the data setup and transfer overhead, its benefits are evident for large-scale matrices.

4 Conclusions and Future Work

In summary, we have presented a GPU-based approximate SVD algorithm. Our method builds upon the QUIC-SVD algorithm introduced by [6], which exploits a tree-based structure to efficiently discover the intrinsic subspace of the input matrix. Results show that our GPU algorithm achieves 6$\sim$7 times speedup over an optimized CPU implementation. Using a matrix partitioning scheme, we have extended our algorithm to out-of-core computation, suitable for very large matrices.

In ongoing work, we are modifying our GPU algorithm to work with sparse matrices. This is important as large-scale matrices tend to be sparse. We will also test our algorithm in practical applications. One application we are particularly interested in is extracting singular vectors from large graph Laplacians. This is instrumental for certain machine learning problems such as manifold alignment and transfer learning. Finally, we have found that the Monte Carlo error estimation is taking a considerable amount of overhead. We would like to investigate more efficient error-estimation schemes.

Acknowledgments. We would like to thank Alexander Gray for providing details of the QUIC-SVD algorithm. This work is supported by NSF grant FODAVA-1025120.

References

1. Bondhugula, V., Govindaraju, N., Manocha, D.: Fast SVD on graphics processors. Tech. rep., UNC Chapel Hill (2006)
2. Deshpande, A., Vempala, S.: Adaptive Sampling and Fast Low-Rank Matrix Approximation. In: Díaz, J., Jansen, K., Rolim, J.D.P., Zwick, U. (eds.) APPROX 2006 and RANDOM 2006. LNCS, vol. 4110, pp. 292–303. Springer, Heidelberg (2006)
3. Friedland, S., Niknejad, A., Kaveh, M., Zare, H.: Fast Monte-Carlo low rank approximations for matrices. In: Proc. of IEEE/SMC International Conference on System of Systems Engineering (2006)
4. Frieze, A., Kannan, R., Vempala, S.: Fast Monte-Carlo algorithms for finding low-rank approximations. J. ACM 51, 1025–1041 (2004)
5. Galoppo, N., Govindaraju, N.K., Henson, M., Manocha, D.: LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In: Proc. of the 2005 ACM/IEEE Conference on Supercomputing, p. 3 (2005)
6. Holmes, M., Gray, A., Isbell, C.L.: QUIC-SVD: Fast SVD using Cosine trees. In: Proc. of NIPS, pp. 673–680 (2008)
7. Humphrey, J.R., Price, D.K., Spagnoli, K.E., Paolini, A.L., Kelmelis, E.J.: CULA: hybrid GPU accelerated linear algebra routines. In: Proc. SPIE, p. 7705 (2010)
8. Jalby, W., Philippe, B.: Stability analysis and improvement of the block Gram-Schmidt algorithm. SIAM J. Sci. Stat. Comput. 12, 1058–1073 (1991)
9. Kerr, A., Campbell, D., Richards, M.: QR decomposition on GPUs. In: Proc. of the 2nd Workshop on GPGPU, pp. 71–78 (2009)
10. Lahabar, S., Narayanan, P.J.: Singular value decomposition on GPU using CUDA. In: Proc. of IEEE International Symposium on Parallel & Distributed Processing, pp. 1–10 (2009)
11. Rokhlin, V., Szlam, A., Tygert, M.: A randomized algorithm for principal component analysis. SIAM J. Matrix Anal. Appl. 31, 1100–1124 (2009)
12. Sarlos, T.: Improved approximation algorithms for large matrices via random projections. In: Proc. of the 47th Annual IEEE Symposium on Foundations of Computer Science, pp. 143–152 (2006)
13. Volkov, V., Demmel, J.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. Tech. rep., UC Berkeley (2008)

Automatic CUDA Code Synthesis Framework for Multicore CPU and GPU Architectures

Hanwoong Jung[1], Youngmin Yi[2], and Soonhoi Ha[1]

[1] School of EECS, Seoul National University,
Seoul, Korea
{jhw7884,sha}@iris.snu.ac.kr
[2] School of ECE, University of Seoul,
Seoul, Korea
ymyi@uos.ac.kr

Abstract. Recently, general purpose GPU (GPGPU) programming has spread rapidly after CUDA was first introduced to write parallel programs in high-level languages for NVIDIA GPUs. While a GPU exploits data parallelism very effectively, task-level parallelism is exploited as a multi-threaded program on a multicore CPU. For such a heterogeneous platform that consists of a multicore CPU and GPU, we propose an automatic code synthesis framework that takes a process network model specification as input and generates a multithreaded CUDA code. With the model based specification, one can explicitly specify both function-level and loop-level parallelism in an application and explore the wide design space in mapping of function blocks and selecting the communication methods between CPU and GPU. The proposed technique is complementary to other high-level methods of CUDA programming.

Keywords: GPGPU, CUDA, model-based design, automatic code synthesis.

1 Introduction

Relentless demand for high computing power is leading us to a many core era where tens or hundreds of processors are integrated in a single chip. Graphics Processor Units (GPUs) are the most prevailing many-core architecture today. Compute Unified Device Architecture (CUDA) is a programming framework recently introduced by NVIDIA, which enables general purpose computing on Graphics Processing Units (GPGPU). With massive parallelism of GPGPU, CUDA has been very successful for acceleration of a wide range of applications in various domains. CUDA is essentially a C/C++ programming language with several extensions for GPU thread execution and synchronization as well as GPU-specific memory access and control. Its popularity has grown rapidly, as it allows programmers to write parallel program in high-level languages and achieve huge performance gain by utilizing the massively parallel processors in a GPU effectively.

Although GPU computing can increase the performance of the task by exploiting data parallelism, it is common that not all tasks can be executed in GPUs as they expose insufficient data parallelism or they require more memory space than the GPU can provide, and so on. While a GPU exploits data parallelism very effectively, task-level parallelism is more easily exploited as a multi-threaded program on a multi-core CPU. Thus, to maximize the overall application performance, one should exploit the underlying heterogeneous parallel platforms that consist of both multi-core CPU and GPU.

It is very challenging to write parallel programs on heterogeneous parallel platforms. It is a well-known fact that high performance is not easily achieved by parallel programming. It is reported that about 100-fold performance improvement is achieved by optimizing the parallel program in a heterogeneous system that consists of a host and a GPU [1]. But optimizing a parallel program is very laborious and time-consuming. To help CUDA programming, several programming models and tools have been proposed, which will be reviewed in the next section. In this paper, we propose a novel model-based parallel programming methodology.

We specify a program with a task graph where a node represents a code segment and an arc represents the dependency and interaction between two adjacent nodes. The task graph has the same execution semantics as dataflow graphs: a node becomes executable when it receives data from the predecessor nodes [2] [3] and an arc represents a FIFO queue that stores the data samples transferred from the source node to the destination node. Such dataflow models express the task-level parallelism of an application naturally. A node can be mapped to a CPU processor core or to a GPU. We identify a data-parallel node that may be mapped to the GPU, to explicitly express the data parallelism. A data-parallel node is associated with a *kernel* to be executed in the GPU.

From this specification, one can explore a wide design space that is constructed according to how design choices are combined: which node should be mapped to the CPU and implemented in C, or mapped to the GPU and implemented in CUDA kernel? And we can select the types of communication APIs between CPU and GPU and kernel invocation methods. Therefore, in this paper, we propose an automatic code synthesis framework that takes as input a dataflow graph (similar to KPN graph) and generates CUDA code for heterogeneous GPGPUs and multi-core CPU platforms.

2 Related Work

Since CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer, and of manually optimizing the utilization of the GPU memory, there is keen interest in developing a high-level method of CUDA programming without worrying about the complexity of the underlying GPU architecture.

*hi*CUDA [4] is an example that has been proposed as a directive-based programming language for CUDA programming. It uses compiler directives to specify parallel execution region of the application similarly to OpenMP. A *hi*CUDA compiler translates it to an equivalent CUDA program without any modification to the original sequential C code.

In our framework, it is assumed that the kernel function is given. Therefore, if we assume the kernel function code can be obtained using *hi*CUDA, our framework is complementary to the works.

GPUSs [5] is a directive-based programming model for exploiting task parallelism as well as data parallelism. So it focuses on heterogeneous systems with general purpose processors and multiple GPUs. Also it supports other targets such as Cell BE, SMP with almost the same user experience. Although its runtime system keeps track of the input and output of each kernel in order to reduce the communication, they do not support asynchronous communication between the CPU and the GPU for overlapping the data transfer and the kernel execution yet to our best knowledge.

StreamIt [6] is a programming model especially targeting for streaming applications. It is based on a dataflow model with a block-level abstraction: a basic unit for a block is called a *filter*. The *filter*s can be connected in a commonly occurred predefined fashion into composite blocks such as pipelines, split-joins, and feedback loops. Like our framework, *StreamIt* supports other heterogeneous platforms such as Cell BE. It is not known to us, however, how to explore the design space of CUDA programming.

Jacket [7] is a runtime platform for executing *MATLAB* code on GPGPUs. It supports simple casting functions to create GPU data structure allowing programmers to describe the application with the native *MATLAB* language. Through simply casting the input data, native *MATLAB* functions wrap those input into GPU functions.

3 Motivation: CUDA Programming

Typical GPUs consist of hundreds of processing cores that are organized in a hierarchical fashion. A computational unit of GPU that is executed by a number of CUDA threads is called the kernel. A kernel is defined as a function with the special annotation, and the kernel is launched in the GPU when the kernel function is called in the host code. The number of threads to be launched can be configured when the kernel is called.

CUDA assumes heterogeneous architectures that consist of different type of processors (i.e., CPUs and GPUs) and separate memory spaces. GPU has its own global memory separated from the host memory; therefore, to execute a kernel in GPU, input data needs to be copied to the GPU memory from the host memory. Likewise, the result needs to be copied from the GPU memory to the host memory after the kernel is completed.

A CUDA program usually uses a fork-join model of parallel execution: when it meets a parallelizable section of the code, it defines the section as a kernel function that will be executed in the GPU by launching a massive number of threads (fork). And the host CPU waits until it receives the result back from the GPU (join) and continues. The maximal performance gain can be formulated as follows:

$$G(x) = \frac{S+P}{S+C+\dfrac{P}{N}} = \frac{1}{x+\dfrac{1-x}{N}+\dfrac{C}{S+P}} \tag{1}$$

where S and P mean the execution time of the sequential section and the parallelizable section to be accelerated by GPU and C means the communication and synchronization overhead between CPU and GPU. N is the effective number of processors in the GPU, which is usually very large, and x presents the ratio of the sequential section in the application. It is observed that C is not negligible and often becomes the limiting factor of the performance gain. In equation (1), if P increases and C remains small, synchronous communication is commonly used between the host CPU and the GPU.

In this paper, however, we also consider the case where P is not dominant so that it is important to use task-parallelism that is included in S, and reduce the communication overhead C in equation (1). So we utilize the multi-core CPU maximally by making a multi-threaded program, in which a thread is mapped to the GPU if the thread has a massive data parallelism inside. And to overlap kernel execution and data communication between the CPU and the GPU, asynchronous CUDA APIs are also considered.

Therefore the design space of CUDA programming is defined by the following design axes: mapping of tasks to the processing elements, number of GPUs, communication protocols, and the communication optimization methods that will be discussed later in section 6. In this paper, we explore the design space of CUDA programming for a given task graph specification graph.

4 The Proposed CUDA Code Synthesis Framework

Fig. 1 illustrates the overall flow of the proposed CUDA code synthesis framework. An application is specified in a task graph based on the CIC model that has been proposed as a parallel programming model [8]. A CIC task is a concurrent process that communicates with other tasks through FIFO channels. We assume that the body of the task is defined in a separate file suffixed by .cic. If a task has data-parallelism inside, we assume that two definitions of the task are given, one for a CPU thread implementation and the other for a GPU kernel implementation. Architecture information is separately specified that defines the number of cores in the CPU and the number of GPUs available.

In the mapping stage, the programmer decides which task node to execute on which component in the target architecture. While multiple tasks can be mapped onto a single GPU, a single task cannot be mapped onto multiple GPUs. After mapping is determined, programmers should decide further design configurations such as communication methods between CPU and GPU, and the number of CUDA threads, and the grid configuration for each GPU task (i.e., CUDA kernel).

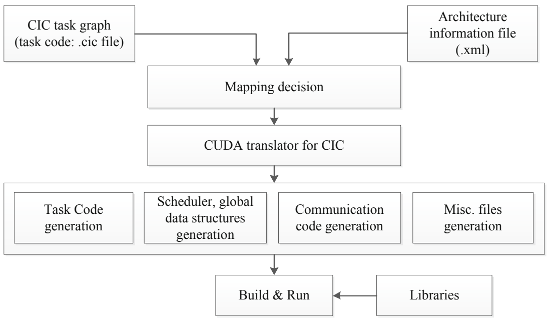


Fig. 1. Overall flow of the proposed CUDA code synthesis framework

Once these decisions have been made, we generate intermediate files for subsequent code synthesis steps. The code synthesizer generates target executable code based on the mapping and configuration information. In task code generation, GPU task is synthesized into a CUDA kernel code. Since the kernel code itself is already given, the synthesizer only adds variable declarations for parameters and includes the header files that declare the generic API prototypes. The code synthesizer also generates a main scheduler function that creates the threads and manages global data structures for tasks, channels, and ports. In the current implementation, we support both POSIX threads and Windows threads. In communication code generation, the synthesizer generates communication code between CPU host and GPU device by redefining the macros such as $MQ\_SEND(), MQ\_RECEIVE()$, used as generic communication APIs in the CIC model. As will be explained in the next section, we support various types of communication methods. This information is kept in an intermediate file (*gpusetup.xml*). There are additional files necessary for building process: for example $Makefile$ help the developer build programs easier.

```
__global__ void Vector_Add(int* C, int* A, int* B){
    const int ix = blockDim.x * blockIdx.x + threadIdx.x;
    C[ix] = A[ix] + B[ix];
}
TASK_GO {
    MQ_RECEIVE(port_input1, DEVICE_BUFFER(input1), sizeof(int) * SIZE);
    MQ_RECEIVE(port_input2, DEVICE_BUFFER(input2), sizeof(int) * SIZE);
    KERNEL_CALL(Vector_Add, DEVICE_BUFFER(output), DEVICE_BUFFER
                        (input1), DEVICE_BUFFER(input2));
    MQ_SEND(port_output, DEVICE_BUFFER(output), sizeof(int) * SIZE);
}
```

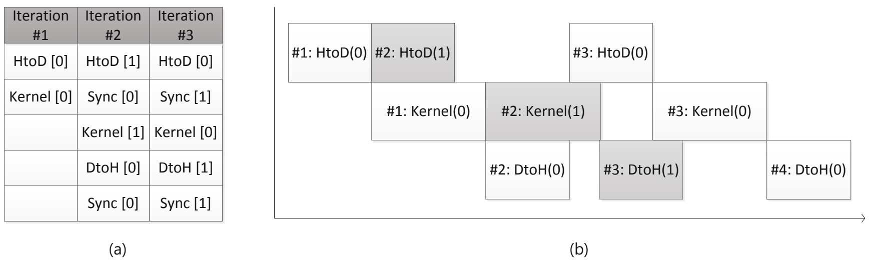
Fig. 2. A CUDA CIC task file (VecAdd.cic)

Fig. 2 shows a simple CIC task code (VecAdd.cic) that defines a CUDA kernel function. The main body of the task is described in the $TASK\_GO$ section that uses various macros whose prototypes are given as templates in the user interface of the proposed framework. The $DEVICE\_BUFFER(port\_name)$ API indicates GPU memory space for the channel and GPU memory allocation code is automatically generated by the code synthesizer relieving the programmers burden. GPU memory de-allocation code is also automatically generated at the wrap-up stage of the task execution. The kernel launch is define by the $KERNEL\_CALL(kernelfunctionname, parameter1, parameter2, ...)$ macro.

5 Code Generation for CPU and GPU Communication

5.1 Synchronous/Asynchronous Communication

CUDA programming platform supports both of synchronous and asynchronous communications. With the synchronous communication method, the CPU task that calls the methods can only proceed after the communication completes. Asynchronous communication is usually better for higher throughput but synchronous communication methods require less memory space than the asynchronous ones where additional memory allocations for *stream* buffers are required. Kernel launches are by default asynchronous executions so that the function call returns immediately and the CPU thread can proceed during the launched kernel is executed. Communications (i.e., memory copy) performed by CUDA APIs that are suffixed with *async* also behave in this manner: instead of blocking the CPU until the memory copy is finished, it returns immediately. Moreover, using *stream*s the communication and kernel execution can be overlapped hiding the communication time as shown in Fig. 3. Kernel executions and memory copy with different *stream*s do not have any dependency, and therefore can be executed asynchronously overlapping their operations. On the contrary, operations with the same *stream* should be serialized. The same *stream* is used



Iteration #1	Iteration #2	Iteration #3
HtoD [0]	HtoD [1]	HtoD [0]
Kernel [0]	Sync [0]	Sync [1]
	Kernel [1]	Kernel [0]
	DtoH [0]	DtoH [1]
	Sync [0]	Sync [1]

Fig. 3. (a) Asynchronous calls with the *stream* ID and (b) the execution timeline of each *stream*

to specify the dependency between CUDA operations and a synchronization function (i.e., *streamSynchronize* ()) denoted as "Sync" in Fig. 3 should be called later to guarantee the completion of the actual copying.

5.2 Bypass/Clustering Communication

To reduce the communication overhead, we define two optimization methods in the proposed code synthesizer, *bypass* and *clustering* as depicted in Fig. 4. By default, the data in a channel is copied into the local buffer in a task. If we want to accelerate the task utilizing GPUs, we should copy the data in the local buffer into the GPU device memory. Hence, it copies the data twice. To reduce such a copy overhead, we implement *bypass* communication. In *bypass* method, the data in the channel is copied to the device memory directly, not through the local buffer.

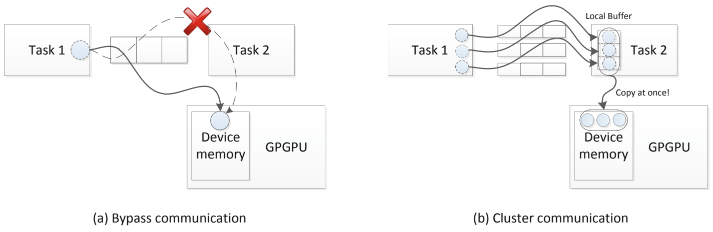


Fig. 4. *Bypass/Clustering* communication

In case there are more than one input/output channels and the size of data is not large, the setup overhead of Direct Memory Access (DMA) becomes significant, sometimes even larger than the actual memory copy overhead. To reduce this overhead, we support *clustering* communication: After the data in all of the input channels are copied into the local buffer inside the task, we send all the data into the device memory at once. This local cluster buffer is declared and allocated by the code synthesizer automatically freeing the programmers from detailed implementation.

$$\sum_{i=1}^{N}(Di * DMAcost + DMAsettingtime) \tag{2}$$

$$\sum_{i=1}^{N}(Di * Memcpycost) + \sum_{i=1}^{N}(Di) * DMAcost + DMAsettingtime \tag{3}$$

(Di: Sample data size of channel ith)

The total execution time of the *bypass* method and the *clustering* method is formalized in equation (2) and (3) respectively. Comparing these two values, we choose the better technique for communication optimization.

6 Experiments

For experiments, we used Intel Core i7 CPU (2.8GHz) with Debian 2.6.32-29 Linux distribution and two Tesla M2050 GPUs. For CUDA programming, we used NVIDIA GPU Computing SDK 3.1 and CUDA toolkit v3.2 RC2.

6.1 Matrix Multiplication

We performed experiments with a simple matrix multiplication example to compare communication overhead between the *bypass* method and the *clustering* method. Two tasks send matrices to a task which multiply two matrices. So there are two input channels in the task.

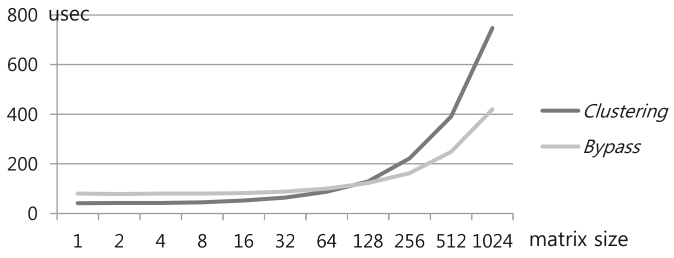


Fig. 5. Communication cost for two communication methods

Fig. 5 shows the communication time (in usec units) of two methods varying the data size (in KBs). When the data size is smaller than 128 KB, it takes less time with the *clustering* method. Otherwise, the *bypass* method is better.

6.2 Lane Detection Algorithm

With a real-life example of lane-detection algorithm, we performed the design space exploration of CUDA programming in the proposed framework. As shown in the Fig. 6, the algorithm consists of two filter chains; one is for detecting the lane in the frame and the other is for providing clearer image display to the driver. Tasks with gray color indicate that they can be mapped to GPU. We used the *Highway.yuv* video clip which consists of 300 frames of HD size (1280x720).

Table 1 shows the execution time of each task on a CPU core and a GPU, obtained by profiling. As can be seen in the table, filter tasks have enough data parallelism to be run on a GPU. Since our target platform contains two GPUs, we can use two GPUs in the mapping stage. As of now we perform manual mapping based on the profiling information of Table 1.

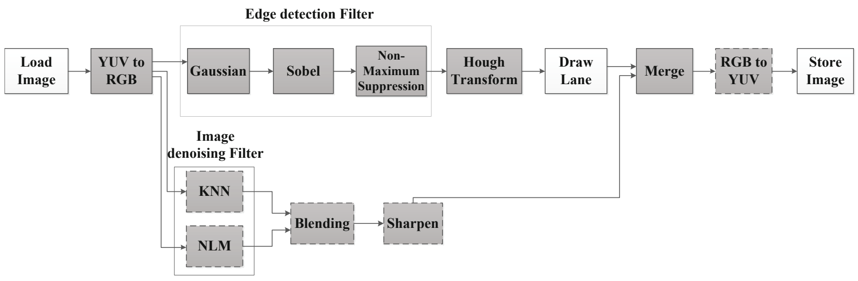


Fig. 6. Task graph of lane detection application

Table 1. Profiling information of tasks (unit: usec)

Task	CPU	GPU	Task	CPU	GPU	Task	CPU	GPU
LoadImage	479	-	KNN	4963268	1615	YUVtoRGB	70226	265
NLM	6911048	13740	Gaussian	389758	1110	Blending	62069	294
Sobel	36616	181	Sharpen	336404	714	Non-max	473716	1752
Merge	45500	245	Hough	369178	2820	RGBtoYUV	76848	300
Draw Lane	3740	-	StoreImage	1068	-	-	-	-

In this experiment, we compared the following three cases: 1) All tasks are mapped on the CPU 2) All GPU-mappable tasks are mapped on a single GPU 3) All GPU-mappable tasks of each filter chain are mapped on each GPU (tasks with solid line in Fig. 6: GPU_0, tasks with dashed line in Fig. 6: GPU_1). Since all tasks have only one or two input ports, we used the *bypass* method. The result is shown in Table 2. Note that we also varied the number of *streams* for asynchronous communication to change the depth of pipelining.

By using one GPU, we could get about 140X speed-up compared using only one CPU core. When we used two GPUs, we could further increase the performance by more than 20% from the gain with one GPU. With asynchronous communications when using one GPU, we could increase the performance gain by 20%. The gain was reduced to 13% when we used two GPUs because the data transfer itself between the device memories in GPU took little time compared to the transfer between the CPU memory and the GPU memory.

Table 2. Results of design space exploration (unit: sec)

CPU	2109.500			
	Sync	Async 2	Async 3	Async 4
1 GPU	12.485	10.654	10.645	10.653
2 GPUs	9.845	9.254	9.168	8.992

* "Async N" denotes asynchronous communication with N *streams*.

7 Conclusions

In this paper, we propose a novel CUDA programming framework that is based on a dataflow model for application specification. The proposed code synthesizer supports various communication methods, so that a user can select suitable communication methods by simply changing the configuration parameters through the GUI. Also we can change the mapping of tasks easily, which increases the design productivity drastically. The proposed methodology could be applied for other platforms such as Cell BE and multi-processor systems. We verified the viability of the proposed technique with the real-life example.

Acknowledgements. This work was supported by Seoul R&BD Program (JP090955), the 2010 Research Fund of the University of Seoul, the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center), National Research Foundation of Korea Grant funded by the Korean Government (NRF-2011-0013479) and research Grant of Education and Research Foundation College of Engineering, Seoul National University.

References

1. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands-on Approach, pp. 78–79. Morgan Kaufmann Publisher (2010)
2. Kahn, G.: The semantics of a simple language for parallel programming. In: Proceedings of IFIP Congress, vol. 74, pp. 471–475 (1974)
3. Lee, E.A., Messerschmitt, D.G.: Synchronous Data Flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
4. Han, T.D., Abdelrahman, T.S.: hiCUDA: A High-level Language for GPU programming. IEEE Transactions on Parallel and Distributed Systems 22(1), 78–90 (2011)
5. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 851–862. Springer, Heidelberg (2009)
6. Udupa, A., Govindarajan, R., Thazhuthaveetil, M.J.: Software Pipelined Execution of Stream Programs on GPUs. In: Symposium on Code Generation and Optimization, pp. 200–209 (2009)
7. Accelereyes, http://wiki.accelereyes.com/wiki/index.php/Jacket_Documentation
8. Kwon, S., et al.: A Retargetable Parallel-Programming Framework for MPSoC. In: TODAES, vol. 13, pp. 1–18 (July 2008)

Accelerating the Red/Black SOR Method Using GPUs with CUDA

Elias Konstantinidis and Yiannis Cotronis

Department of Informatics and Telecommunications,
University of Athens, 157 84 Ilisia
Athens, Greece
{ekondis,cotronis}@di.uoa.gr

Abstract. This work presents our strategy, applied optimizations and results in our effort to exploit the computational capabilities of GPUs under the CUDA environment in solving the Laplacian PDE. The parallelizable red/black SOR method was used. Additionally, a program for the CPU, featuring OpenMP, was developed as a performance reference. Significant performance improvements were achieved by using optimization methods which proved to have substantial speedup in performance. Eventually, a direct comparison of performance of both versions was realised. A 51x speedup was measured for the CUDA version over the CPU version, exceeding 134GB/sec bandwidth. Memory access patterns prove to be a critical factor in efficient program execution on GPUs and it is, therefore, appropriate to follow data reorganization in order to achieve the highest feasible memory throughput.

Keywords: GPU computing, CUDA, SOR, PDEs, red/black.

1 Introduction

Traditionaly, conventional processors have been used to solve computational problems. Modern graphics processors (*GPUs*) have become coprocessors with significantly more computational power than general purpose processors. Their large computational potential have turned them to a special challenge for solving general-purpose problems with large computational burden. Thus, application programming environments have been developed like the proprietary CUDA (Compute Unified Development Architecture) by NVidia [1,8] and the OpenCL (Open Computing Language) [4] which is supported by many hardware vendors.

CUDA environment is rapidly evolving and has become adequately mature [1]. It provides an elegant way for writing GPU parallel programs, by using a kind of extended C language, without involving other graphics APIs.

Partial differential equations (PDEs) constitute an important sector of the computational science field. To solve a PDE (i.e. Laplace equation) numerically an iterative method can be employed [7]. One of the prefered methods is the SOR (Successive OverRelaxation) method which provides a good rate of convergence.

The next section presents the related work. In section 3 the red/black method is analyzed and in section 4 the various implementation options are discussed, where a different kernel was developed for each significant improvement. Performance results are presented in section 5, where the most critical factors affecting performance are emphasized and the conclusions follow in section 6.

2 Related Work

The simpler Jacobi method has been applied in CUDA for solving problems like PDEs [12] but it is not suggested due to its slow convergence. Datta has investigated stencil kernels in a variety of multicore and manycore architectures [16]. OpenGL and CUDA implementations have been compared [9] in performance and CUDA environment proves to be more efficient and flexible. Hybrid implementations have also been published [15] utilizing both CPUs and GPUs.

The Gauss-Seidel method, providing faster convergence, has been applied in CUDA implementations to accelerate fluid dynamic simulations [13,14]. GPUs have also been used to accelerate Gauss-Seidel by making use of shading languages, like Cg, before native GPU computing languages, like CUDA, were devised [17].

The SOR method can lead to even faster convergence [7]. It has been applied to medical analysis [10] as well as to computational fluid dynamics [11], as these kind of problems require a large number of calculations to be performed.

3 Red/Black SOR Method

The red/black SOR method belongs to the iterative methods family like the Jacobi. The red black coloring allows easy parallelization. The calculation of all elements in the same color can be handled independently of the others since there is no data dependence between them(figure 1). Therefore, this problem is an ideal one for parallelization.

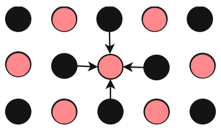


Fig. 1. Red point values are affected only by the neighbouring black points

Moving on to the iterative process, values are calculated in two phases. At first, red elements get updated and then black elements follow. Every point is updated according to the neighbour point values(figure 1), as the equations (1) and (2) indicate. The boundary values are assumed constant and predetermined.

$$u_{i,j}^{k+1} = (1-\omega)u_{i,j}^{k} + \omega(u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k})/4 \tag{1}$$

$$u_{i,j}^{k+1} = (1-\omega)u_{i,j}^{k} + \omega(u_{i-1,j}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j-1}^{k+1} + u_{i,j+1}^{k+1})/4 \tag{2}$$

The ω is the relaxation factor which allows faster convergence. The speed of convergence is not a subject of this paper, thus the selected value is irrelevant.

This calculation is executed iteratively until the matrix values converge. Checking for convergence is performed by calculating the sum of squared differences between the current values and the previous iteration values.

4 Implementations

It's important to realize that this application is bounded to memory bandwidth and not to calculation throughput. Assuming that cache is being used efficiently, N^2 access reads and $N^2/2$ access writes per red or black calculation stage are required. This equals to $3xN^2$ total accesses for a full iteration. For each element about 6 floating point operations are required. So, for the whole matrix $6xN^2$ flops and $3xN^2$ accesses are required, thus the mean flops per element access ratio is 2, or $\frac{1}{2}$ flops per byte access ratio, as single floating point precision has been used. A satisfactory ratio is considered to be at least 10, which is much higher [6]. The GTX280 GPU provides almost 1 TeraFlop peak computational power and about 142GB/sec bandwidth, making a ratio of about 7 flops per byte. Consequently, the performance is dependent on the memory bandwidth capability of the GPU device.

4.1 GPU Implementations

Although the red/black method provides inherent parallelism, optimizing for CUDA can be challenging. Memory access patterns, different memory types present factors that have to be carefully considered in efficient implementations.

All kernels were developed in plain CUDA C code, thus no other high level GPU libraries have been used (i.e. CUBLAS [2]).

Global Memory Usage Only. A simple CUDA implementation makes use of global memory only. For efficient implementations the access pattern should follow the recommendations for CUDA devices about using memory coalescing [1]. Generally, for memory coalesced accesses all accesses of a warp should concern addresses within the same memory segment. The exact requirements are described in the CUDA programming guide [1].

Memory coalescing can not be satisfied for reading all three elements in a row($u_{x-1,y}$, $u_{x,y}$ and $u_{x+1,y}$) on CC(*compute capability*) devices below 1.2. Fortunately, there are alternative ways for fetching these elements as it will be shown, by using other memory types (i.e. shared memory) via thread cooperation.

In order to satisfy memory coalescing the compute efficiency had to be decreased. Half of the threads remain idle which causes serious under-utilization.

Texture Memory Usage. Memory access reads are performed through texture memory. Texture memory features an intermediate memory cache which is optimized for 2D locality.

However, as texture memory is not writable, all writes are performed to global memory like previously. Thus, the low efficiency problem remains as the same access pattern is employed for memory writes. Memory coalescing on writing, enforces half threads to remain idle. A memory structure reorganization is required to confront idleness.

Element Reordering with Global Memory Usage. Each iteration works by reading all red and black elements and writing back black elements or by reading all black and red elements and writing back red elements. In order to avoid sparse element accesses and to enforce coalescing, it would be better to reorder matrix elements so to have the elements grouped according to its color. Therefore, the matrix is split into 2 independent matrices, one having only the red elements and the other only the black elements (figure 2). Each element position (i, j) is transformed to a new one on the new matrix $(i/2, j)$ which is the red one if $(i + j) \mod 2 = 0$ or otherwise the black one.

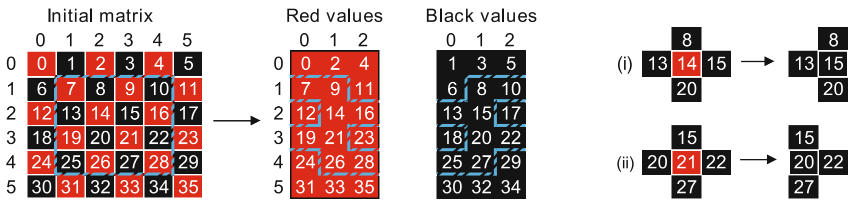


Fig. 2. Matrix element reordering. Mapping of odd numbered row elements (i) and even numbered row elements (ii) to the reordered matrix elements.

Working with 2 distinct matrices for each color allows to use warp threads at full efficiency while preseving coalescing. In each iteration either all red elements are calculated or black elements are calculated. The total accessed bytes that get discarded are reduced and average efficiency is increased.

Assuming that threads of a thread block can share data element values, all read accesses are performed in a coalesced manner except for the overhanging halo elements (figure 3). These halo elements are read individualy without coalescing, as they belong to different memory segments, which is inefficient (32 byte transactions per element on below CC 1.2 devices [1]). Data sharing between threads is accomplished through global memory cache on CC 2.x devices or, as it will be shown, by using shared memory or texture memory.

The data reordering procedure requires some time which can be significant on large matrices. Fortunately, due to the large number of iterations, the benefit compensates its cost. Although, it could be efficiently implemented on the GPU, it was left out of scope of this work and a CPU implementation was used.

The addressing of elements is now more complex. The neighbours of an element are 3 vertical elements plus one on the left or right, depending on the row number (figure 2: i, ii).

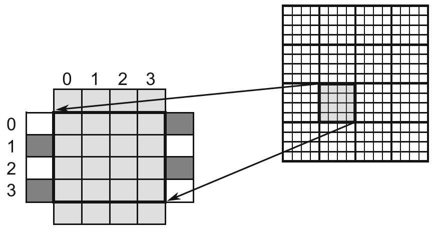


Fig. 3. Read accesses by a typical 4x4 thread block. Elements in light grey color are read in a coalesced manner. Elements in dark grey color are read individually. The elements in the 4x4 box maps to threads of a block.

Element Reordering with Texture Memory Usage. Another optimization was applied by combining the use of texture memory with the previous optimization. This resolves the non-coalesced element accessing issue via texture caching that devices without global memory cache suffer. The inherent 2D locality nature of the problem helps improve performance by using texture memory.

Element Reordering with Shared Memory Usage. Alternatively, shared memory is used to cache block elements into fast gpu memory and reduce bandwidth requirements. Shared memory is very fast but it's use is more complicated since it is not used implicitly like texture memory cache but as a *software managed cache* instead.

First, a group of elements that fits to shared memory is read cooperatively by a block of threads as already shown (figure 3). Using a synchronization point (*__syncthreads()*) after loading data, the access through shared memory is ensured to be in sync. Thereafter, all calculation data are fetched from shared memory whose latency is two orders of magnitude lower than that of global memory (when not cached) [3]. The destination element is accessed through global memory and its access is coalesced.

The disadvantage of shared memory is the requirement of inserting synchronization points and the additional instruction overhead in code. Synchronization points present a point for stalling a multiprocessor as when threads of a block wait on a synchronization point total threads ready for execution are reduced. Thus, the chances for latency hiding through multithreading get decreased. Additionally, shared memory copies from/to global memory require extra instructions contrary to cache memory.

For a thread block of size MxN, a $(M + 2) \times (N + 2)$ space of elements is used in shared memory. Shared memory elements are accessed sequentially by warp threads, thus this method provides bank conflict free accesses to shared memory [1]. To have conflict free accesses M should be a mupliple of 16 for 1.x compute capability devices or a mupliple of 32 for 2.x compute capability devices.

4.2 Common Optimization Strategies

Additionally, to optimize kernels a few common optimization guidelines were applied. In order to optimize the compute efficiency of the kernel the amount of thread computation was increased. In the initial code each thread computed a single element value. Having more values to be computed by each thread the part of useful computation is increased. Additionally, memory accesses are decreased since the values needed for a thread to calculate two neighboring elements are overlapped. However, register usage is increased which in turn can reduce the device occupancy. We call the number of elements computed by each thread as *thread granularity*.

The group of elements that is calculated by a single thread was selected to be in vertical order to accomodate the requirements for conflict free shared memory accessing. In such way warp threads access consecutive elements of a row in shared memory and bank conflicts are avoided. In contrast, choosing elements horizontally would induce bank conflicts.

5 Performance Results

An execution of each kernel can give a picture of their performance. The kernels were executed on a GTX280 GPU for a 2050×2050 matrix (table 1) with convergence detection. Convergence was detected after running for 3201 iterations.

For each kernel execution two time metrics have been measured. The time for the pure kernel execution is mentioned as *calculation time* and the full execution time is mentioned as *execution time*. Execution time includes both PCI-Express data transfer time overhead and, possibly, the element reordering time overhead. Times were measured by the standard clock() function and always in unit of seconds.

As expected, the versions with reordered elements performed better than the first two. Furthermore, the calculation time is only slightly less than the full execution time so the element reordering process does not consume significant proportion of the total execution time. The data transfer time is negligible for this size of problem and as the precision of clock() function is not enough to measure it precisely, both times appear equal in the case of kernel 2.

At first glance the texture version seems to be the best kernel. Surprisingly, the shared memory kernel performs worse than the global memory kernel. The instruction overhead, register pressure and synchronization cost for reading data to shared memory, synchronizing, processing, synchronizing again and finally writing back data to global memory diminishes the benefits of the low latency shared memory. However, this may be missleading because at this point each thread calculates just one element in every kernel (granularity=1).

Next, a number of executions for different configurations were performed for global memory, texture memory and shared memory versions (all using reordering) on the 1026x1026 problem size for 10000 total iterations. Each configuration is differentiated by block size and thread granularity, both of which affect the pattern of elements in shared memory.

Table 1. Execution times in seconds of all kernels for matrix 2050x2050 on GTX280 GPU with convergence detection

Kernel	Calculation time	Execution time
Kernel 1: Global memory (no reordering)	5.60	5.63
Kernel 2: Texture memory (no reordering)	4.34	4.34
Kernel 3: Global memory (reordered)	2.92	2.96
Kernel 4: Texture memory (reordered)	2.29	2.34
Kernel 5: Global+shared memory (reordered)	3.99	4.04

Table 2. Block size and thread granularity tuning on GTX280 GPU

Configuration		Execution time			
Block size	Thread granularity	Global mem (#3)	Texture mem (#4)	Shared mem (#5)	Notes
16x16	1	2.683	2.309	3.432	
16x8	1	2.605	2.199	3.447	
32x8	1	2.698	2.324	3.478	
32x4	1	2.605	2.215	3.510	
64x4	1	2.698	2.386	3.525	
64x2	1	2.621	2.293	3.650	
16x16	2	2.730	2.137	2.839	
16x8	2	2.574	2.044	2.808	
32x2	2	2.527	2.028	3.010	
64x2	2	2.558	2.106	2.932	
16x8	4	2.792	2.511	2.558	
32x6	4	2.636	2.340	2.589	
32x2	4	2.511	1.981	2.621	Best score (global memory)
64x2	4	2.699	2.324	2.590	
64x4	4	2.745	2.386	2.683	
64x4	8	2.745	2.308	2.699	
64x2	8	2.745	2.277	2.215	Best score (shared memory)
32x2	8	2.542	1.950	2.230	Best score (texture memory)
32x4	8	2.667	2.340	2.246	
32x2	16	2.667	2.074	2.917	
64x2	16	2.808	2.184	3.853	

The GT200 based GTX280 (compute capability 1.3) and the GF100 (Fermi) based GTX480 (compute capability 2.0) were used in the experiments. The GT200 and Fermi architectures are much different in various characteristics. The most important addition regarding this computation is the addition of level 1 and level 2 global memory cache in Fermi based GPUs. The results of the tuning process for GTX280 GPU are depicted on table 2.

The GTX280 seems to perform best with texture memory (1.95 secs) using block size of 32x2 and thread granularity of 8, so each thread block actually calculates a grid of 32x16 elements. Using shared memory the calculation finishes after 2.215 secs by using block size of 64x2 and thread granularity of 8 elements, so each block works on a grid of 64x16 elements. Global memory, as expected, is the worst as its best configuration takes 2.511 secs to finish. So, after changing the thread granularity the usage of shared memory proves better than global memory only, which was expected. In contrast, the GTX480 performs best with its global memory kernel, in a 64x4 block size with thread granularity of 4 elements per thread, which finished the calculation in just 1.004 secs and the shared memory kernel, in a 64x2 block size with thread granularity of 8, follows with 1.111 secs. The texture memory kernel is the worst in this case as it finishes in 1.25 secs by using 16x12 block configuration and a granularity of just 1.

In summary the CC 1.x device performs slightly better with texture memory usage and worse when using only global memory, where in contrast, the CC 2.0 device performs best with just global memory and worse when using texture memory. This reflects the great advantage that the cache memory offers when spacial and temporal locality exists in the nature of a problem.

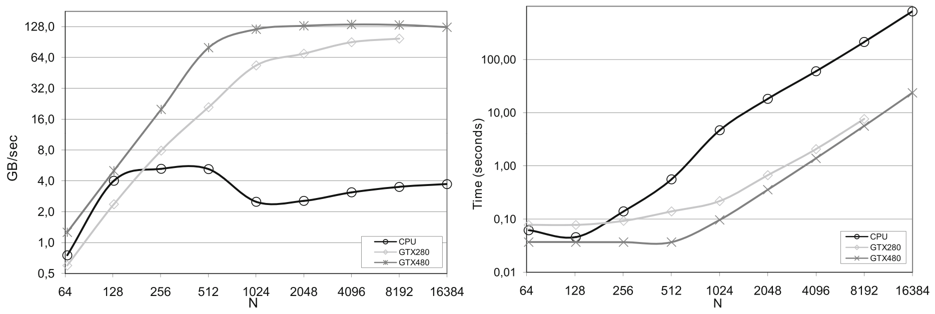


Fig. 4. Calculation throughput and calculation times achieved with CPU, GTX280(k#4) & GTX480(k#3) GPUs

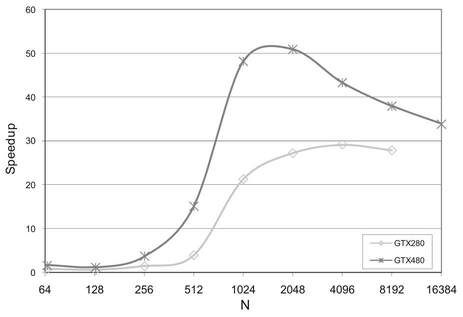


Fig. 5. Calculation speed-up achieved on GTX280 (k#4) & GTX480 (k#3) over CPU

Finally, the best versions of both GPUs were compared with the CPU version. The CPU version is a typical OpenMP [5] application which does not make use of any special optimizations (i.e. manual vectorization). The CPU used in the experiment was an Intel Core 2 Quad Q9550 (2.83GHz) on a different host system than of these used for the GPU executions, as the latter featured quite inferior CPUs. The Microsoft Visual Studio C++ 2008 compiler was used for the CPU version and the optimization flags were enabled ("/arch:SSE2 /fp:fast /openmp /O2"). However, no automatic vectorization was noticed in the generated machine code. The experiments were run for problem sizes from 66x66 to 16386x16386 elements for 1000 iterations. The results of the execution are depicted on table 3. The throughput was calculated by using the calculation time and the assumption that 3xN^2 accesses are required for each iteration. The discrepancies between execution times and calculation times are due to the element reordering procedure and the PCI-Express limited throughput, so the execution time is partly CPU dependent. The proportion of this extra time is relatively small, with an exception on the last case (16386x16386), where the large discrepancy is due to the limited main memory of the host which was equiped with only 1GB RAM and it forced the excessive use of virtual memory. Moreover, the last experiment was not possible to run on the GTX280 as it required more than 1GB of device memory which was not available on this device.

Table 3. Throughput achieved on GTX480, GTX280 and CPU (1000 iterations)

	CPU Intel Core 2 Quad Q9550 @ 2.83GHz (OpenMP)		GPU NVidia GTX280 (texture mem. k#4)					GPU NVidia GTX480 (global mem. k#3)				
Matrix size	Execution-Calculation time	Mean throughput GB/sec	Execution time	Calculation time	GPU Throughput GB/sec	Calculation Speed-up	Overall speed-up	Execution time	Calculation time	GPU Throughput GB/sec	Calculation Speed-up	Overall speed-up
66x66	0.06	0.754	0.09	0.08	0.599	0.79	0.66	0.04	0.04	1.263	1.68	1.63
130x130	0.05	4.022	0.09	0.08	2.372	0.59	0.49	0.04	0.04	5.025	1.24	1.21
258x258	0.14	5.259	0.09	0.09	7.917	1.51	1.51	0.04	0.04	19.948	3.78	3.50
514x514	0.56	5.227	0.16	0.14	20.981	4.01	3.60	0.05	0.04	80.019	15.19	11.96
1026x1026	4.68	2.507	0.25	0.22	53.580	21.37	18.80	0.13	0.10	120.799	48.25	36.56
2050x2050	18.32	2.561	0.75	0.67	69.904	27.30	24.49	0.47	0.36	130.625	51.02	39.30
4098x4098	60.54	3.098	2.37	2.08	90.391	29.18	25.54	1.79	1.40	134.429	43.40	33.75
8194x8194	214.14	3.503	8.58	7.66	97.927	27.96	24.96	7.22	5.63	133.222	38.03	29.66
16386x16386	804.91	3.727	-	-	-	-	-	687.57	23.74	126.378	33.91	1.17

The throughput and calculation times achieved by all three devices are depicted in figure 4 in logarithmic scale. The speed-up observed for each GPU compared to the CPU is depicted in figure 5.

The total throughput achieved by the GTX480 exceeded 134GB/sec. On the GTX280 the throughput reached nearly to 98GB/sec. Theoretical numbers for both are about 177GB/sec and 142GB/sec, which means that throughput percentage utilization is about 75% and 69%, respectively. Thus, these percentage numbers seem adequate for exploiting the bandwidth capabilities of GPUs.

6 Conclusion

The advent of compute capability devices with cache memory proves that in some cases the use of special memory types is not as critical as it was in previous generation GPUs. Simpler implementations can be quite efficient and programming for performance on GPUs does not necessarily prove the use of special memory types as a mandatory requirement.

Beyond vendor suggested optimizations, data reordering proves to be a critical one as it can provide significant performance improvements. Data elements should be reordered in memory in such way that frequent accesses are coalesced and no redundant memory transfers are performed in critical parts of code. Increased data locality, in combination with data coalescing helps maximizing memory throughput which is the most critical factor affecting performance in memory bounded applications.

Evidently, reordering poses a trade-off. Data reordering can be time consuming for very large matrices. Fortunately, this procedure can also be moved to the GPU in order to exploit its high throughput but this was left for future work.

Acknowledgment. This research was partially funded by the University of Athens Special Account of Research Grants no 10812.

References

1. NVidia CUDA Reference Manual v. 3.1 NVidia (2010)
2. NVidia CUDA Toolkit 3.1 CUBLAS Library NVidia (2010)
3. NVidia CUDA C Best Practices Guide Version 3.1 NVidia (2010)
4. The OpenCL Specification Khronos group (2009)
5. OpenMP Application Program Interface version 3.0 OpenMP Architecture Review Board (2008)
6. Kirk, D.B., Hwu, W.W.: Programming Massively Parallel Processors. Morgan Kaufmann (2009)
7. Burden, R.L., Faires, D.: Numerical Analysis, 7th edn. Brooks Cole (2000)
8. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. ACM Queue 6(2), 40–53 (2008)
9. Amorim, R., Haase, G., Liebmann, M., Weber, R.: Comparing CUDA and OpenGL implementations for a Jacobi iteration SpezialForschungsBereich F 32, 025 (December 2008)
10. Ha, L., Krger, J., Joshi, S., Silva, C.T.: Multiscale Unbiased Diffeomorphic Atlas Construction on Multi-GPUs. GPU Computing Gems. Morgan Kaufmann (2011)
11. Komatsu, K., Soga, T., Egawa, R., Takizawa, H., Kobayashi, H., Takahashi, S., Sasaki, D., Nakahashi, K.: Parallel processing of the Building-Cube Method on a GPU platform Computers & Fluids (2011) (in press, corrected proof)
12. Cecilia, J.M., Garcìa, J.M., Ujaldòn, M.: CUDA 2D Stencil Computations for the Jacobi Method Para 2010 - State of the Art in Scientific and Parallel Computing (2010)
13. Amador, G., Gomes, A.: A CUDA-based Implementation of Stable Fluids in 3D with Internal and Moving Boundaries. In: 2010 International Conference on Computational Science and Its Applications (2010)
14. Amador, G., Gomes, A.: CUDA-based Linear Solvers for Stable Fluids. In: International Conference on Information Science and Applications, ICISA (2010)
15. Venkatasubramanian, S., Vuduc, R.W.: Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In: 23rd International Conference on Supercomputing (2009)
16. Datta, K., Murphy, M., Volkovand, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: ACM/IEEE Conference on Supercomputing (2008)
17. Krüger, J., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. In: ACM SIGGRAPH 2003 (2003)

Dense Affinity Propagation on Clusters of GPUs

Marcin Kurdziel and Krzysztof Boryczko

AGH University of Science and Technology,
Faculty of Electrical Engineering, Automatics, Computer Science and Electronics,
Department of Computer Science,
al. A. Mickiewicza 30,
30-059 Krakow, Poland
{kurdziel,boryczko}@agh.edu.pl

Abstract. This article focuses on implementation of Affinity Propagation, a state of the art method for finding exemplars in sets of patterns, on clusters of Graphical Processing Units. When finding exemplars in dense, non-metric data Affinity Propagation has $O(n^2)$ memory complexity. This limits the size of problems that can fit in the Graphical Processing Unit memory. We show, however, that dense Affinity Propagation can be distributed on multiple Graphical Processing Units with low communication-to-computation ratio. By exploiting this favorable communication pattern we propose an implementation which can find exemplars in large, dense data sets efficiently, even when run over slow interconnect.

Keywords: Affinity Propagation, multi-GPU implementation, clustering.

1 Introduction

Clustering, a fundamental unsupervised learning method, seeks groups of related patterns, i.e. clusters, in data. Classical approaches to this problem rely on selecting exemplar patterns which serve as the centers of clusters. In order to construct meaningful clusters, exemplars are selected in a way that maximize similarity between them and patterns in their clusters. Thus one usually attempts to minimize a cost function of the form:

$$J\left(C_{1,\ldots,k},\theta_{1,\ldots,k}\right)=\sum_{i=1}^{k}\sum_{\mathbf{x}\in C_i} d\left(\mathbf{x},\theta_i\right), \tag{1}$$

where $C_{1,\ldots,k}$ are the clusters, $\theta_{1,\ldots,k}$ are their corresponding exemplar patterns and $d\left(\cdot,\cdot\right)$ is a dissimilarity measure. It turns out, however, that finding global minimum of Eq. 1, even in metric cases, is an NP-hard problem. Only approximate solutions are available in polynomial time (see e.g. [4]). Heuristic techniques are therefore employed for selecting exemplar patterns, with k-means algorithm [9] being the most widely used approach. An important drawback of the k-means algorithm is its propensity to stuck in poor local minima of the cost function. Typically one needs to run k-means a number of times, each run starting from a different initial choice of exemplars, and then select the best of solution found.

Recently a new method for finding exemplars in data sets was proposed [5]. This algorithm, called Affinity Propagation (AP), finds exemplars by passing carefully designed, real-valued messages between patterns. It turns out that a single run of AP usually finds clusters with the cost value (Eq. 1) well below the one obtained by several thousand runs of k-means [5]. Another strength of affinity propagation is its ability to find clusters in non-metric data. One example reported in [5] demonstrates successful application of AP to data given by similarities[1] which are not even symmetric.

We show that AP is very well suited for execution in massively parallel environments. Iteration of AP over dense similarity matrix can be split between a number of nodes with relatively modicum amount of inter-node communication. We exploit this for efficient implementation of dense AP on clusters of Graphical Processing Units (GPUs), thereby allowing for clustering data sets which would not fit in the memory available on a single GPU.

2 Related Work

Graphical Processing Units were recently employed in high-performance implementations of several clustering techniques, including hierarchical clustering algorithms [15], fuzzy clustering [1], clustering by mixture decomposition with Expectation Maximization [8] and document clustering with flocking model [16]. Several related works focused on implementing classical k-means algorithm on GPU [6] [2] [12] [14] [8], with one approach [13] focusing on implementing classical k-means on a cluster of conventional nodes while offloading certain operations to GPU units. The most related work, however, was presented in [7]. While previous works focused on classical k-means, ref. [7] presents a GPU implementation of approximate AP.

A major roadblock in implementing AP on GPUs is its memory requirement. When clustering n patterns, dense AP requires three $n \times n$ matrices, one for the pairwise pattern similarities and two other for the exchanged messages. In an attempt to alleviate this problem the approximation scheme proposed in [7] employs Z-curve, a simple locality preserving space filling curve, to construct a sparse band matrix approximation of the $n \times n$ input similarities. With this approach messages exchanged during AP also fit in band matrices. Overall, the approximation significantly reduces AP's memory requirement and allows for very efficient GPU implementation.

The main drawback of the approximation scheme proposed in [7] is its limited applicability in terms of similarity measures as compared to the exact AP. Band

[1] Exemplar selection may also be expressed in terms of maximizing a quality function $S\left(C_{1,\ldots,k}, \theta_{1,\ldots,k}\right) = \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} s\left(\mathbf{x}, \theta_i\right)$, where $s\left(\cdot, \cdot\right)$ is a similarity measure. This is the formulation used in AP.

matrix approximation with space filling curve requires that patterns be points in a Cartesian space with input being a Gram matrix whose kernel function is inversely proportional to the distances between the points [7]. Exact AP imposes no such requirements and works well in dense non-metric cases. This is especially important given the increasing amount of non-vectorial data gathered in various experimental techniques. In molecular biology, for example, patterns may represent sequences of nucleic acids or structures of proteins and will often require complex, non-metric similarity measures (see e.g. [11, chapters 3, 6, 11]) with dense similarity matrices. Method given in [7] cannot be applied to dense non-metric instances of this kind. This article approaches AP k-clustering on GPU from other direction – instead of seeking a sparse approximation to the similarity matrix we seek to fit the AP matrices in the available device memory and decrease the clustering time by distributing the execution of AP over multiple GPUs. That way we can employ GPUs for efficient AP clustering of large, dense non-metric data.

3 Affinity Propagation on Clusters of GPUs

Affinity Propagation finds exemplars by iteratively exchanging two kinds of messages, i.e. *responsibilities* and *availabilities*, between every pair of patterns [5]. Iterations begin with the availabilities set to zero and are carried out until message values converge. In the proposed parallelization scheme matrices storing the messages, denoted by $\mathbf{R}$ and $\mathbf{A}$ respectively, as well as the similarity matrix $\mathbf{S}$ are split row-wise between the GPUs. That is, the k-th GPU stores rows $\lfloor \frac{n}{M} \rfloor \cdot (k-1)$ ... $\lfloor \frac{n}{M} \rfloor \cdot k$, where n is the number of patterns and M is the number of GPUs. All three matrices are stored in the row-major order, with rows padded to facilitate coalesced memory accesses.

The responsibility message $\mathbf{R}_{ij}$ quantifies the fitness of pattern $\mathbf{x}_j$ as an exemplar for pattern $\mathbf{x}_i$ [5] and is calculated as:

$$\mathbf{R}_{ij}^{(t)} = \mathbf{S}_{ij} - \max_{k \neq j} \left(\mathbf{A}_{ik}^{(t-1)} + \mathbf{S}_{ik} \right) , \tag{2}$$

where indices $(t-1)$ and (t) denote values from the previous and the current iteration, respectively[2]. This update to the matrix $\mathbf{R}$ can be carried out in $O(n^2)$ operations by finding the maximal and the second maximal element in each row of $\mathbf{A}^{(t-1)} - \mathbf{S}$ [5, Matlab implementation in Supporting Online Material]. In each row, the responsibilities are then updated using the corresponding maximal element, except at the position of the maximal element itself, where the second maximal element is used instead.

We implemented the GPU kernel responsible for finding the maximal elements following results of the performance evaluation given in [3]. Each row is

[2] In the actual implementation, new responsibility values must be set to weighted means of previous responsibility values and the values given by Eq. 2 [5]. This weighting is necessary for the algorithm convergence. It increases the number of accesses to the GPUs device memory but does not disrupt memory access patterns.

thus processed by one work group[3]. The k-th work item in the i-th work group processes elements (i,k), $(i,k+w)$, $(i,k+2w)$, ..., where w is the size of the work group, and stores the results in the local device memory. After scanning the whole row, work group performs tree-reduction over the partial results in the local memory. Limiting the number of work groups to one per row has the benefit of reducing the number of local memory barriers [3]. The available level of paralelizm is not affected negatively due to multiple rows being processed concurrently. The kernel performing updates to the responsibilities is straightforward – each row is processed by one work group, which fetches the maximal and second maximal element for that row and performs updates according to Eq. 2. Overall, these kernels involve $O(n)$ non-coalesced global memory accesses to the arrays storing the maximal elements. This is tolerable given the $O(n^2)$ coalesced memory accesses while scanning matrices $\mathbf{A}$ and $\mathbf{S}$ and updating $\mathbf{R}$. This part of the algorithm requires no communication between the GPUs.

After updating the responsibilities, AP proceeds with the availability messages. The availability message $\mathbf{A}_{ij}$ quantifies the fitness of the pattern $\mathbf{x}_i$ to be a member of the cluster formed by $\mathbf{x}_j$ [5] and is calculated as[4]:

$$\mathbf{A}_{ij}^{(t)} = \begin{cases} \min\left\{0,\ \mathbf{R}_{jj}^{(t)} + \sum\limits_{k \neq i,j} \max\left\{0, \mathbf{R}_{kj}^{(t)}\right\}\right\}, & \text{if } i \neq j \\ \sum\limits_{k \neq i} \max\left\{0, \mathbf{R}_{ki}^{(t)}\right\}, & \text{otherwise} \end{cases} \tag{3}$$

To implement this update efficiently we need to calculate the sum

$$u_j^{(t)} = \mathbf{R}_{jj}^{(t)} + \sum_{k \neq j} \max\left\{0, \mathbf{R}_{kj}^{(t)}\right\} \tag{4}$$

for each column j of the matrix $\mathbf{R}^{(t)}$ [5, Matlab implementation in Supporting Online Material]. Availabilities can then be calculated with local matrix updates: $\mathbf{A}_{ij}^{(t+1)} = \min\left\{0, u_j^{(t)} - \max\left\{0, \mathbf{R}_{ij}^{(t)}\right\}\right\}$ for $i \neq j$ and $\mathbf{A}_{jj}^{(t+1)} = u_j^{(t)} - \mathbf{R}_{jj}^{(t)}$. Note, however, that $\mathbf{R}^{(t)}$ is split row-wise between the GPUs. Each GPU therefore calculates partial sums over the rows assigned to it. These partial sums are transferred to the host memory, summed up in a global all-to-all reduction operation and then written back to the device global memory. The reduction between the nodes is carried out with an MPI All-Reduce operation [10].

In the kernel calculating the partial sums each work item processes a single column, keeping the running sum in the local memory. Because $\mathbf{R}^{(t)}$ is stored in the row-major order with row padding and each work group processes a block

[3] The implementation of the multi-GPU AP was done in OpenCL – see *The OpenCL Specification v1.0*, Chapter 2, for definition of work item, work group and related concepts.

[4] As with responsibilities, in the actual implementation updates to availabilities also involve weighting of old and new message values [5], which increases device memory usage but does not disrupt memory access patterns.

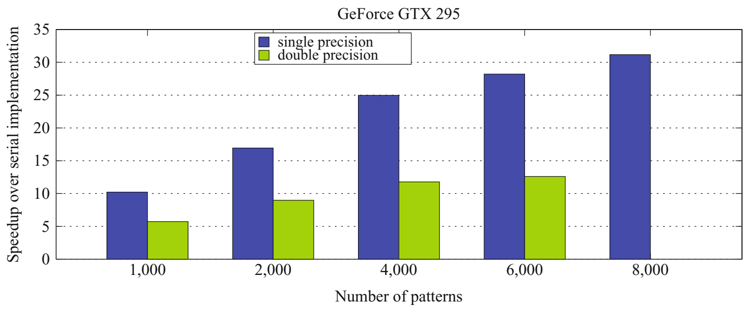
of columns, accesses to the device global memory are coalesced. To remove one conditional from the summation loop, work item treats all summed elements uniformly and then performs a fix for the diagonal element with a single non-coalesced global memory read. In the kernel updating the availabilities, each row of $\mathbf{A}_{ij}^{(t+1)}$ is processed by one work group. Here all memory accesses are coalesced. Overall, the calculation of availabilities involves $O(n^2)$ accesses to the device global memory, including n non-coalesced accesses, two $O(n)$-byte transfers between host and device memory and one n-element all-to-all summation between the GPU nodes.

Iterative updates to availabilities and responsibilities constitute most of AP's computational cost. Identification of exemplars and assignment of patterns to clusters is, in comparison, a relatively inexpensive operation. One way to pinpoint exemplars in AP is to scan main diagonals of converged matrices $\mathbf{R}$ and $\mathbf{A}$ and find patterns θ_i for which $\mathbf{R}_{ii} + \mathbf{A}_{ii} > 0$ [5, Matlab implementation in Supporting Online Material]. Each non-exemplar pattern is then assigned to the cluster formed by exemplar that is most similar it.

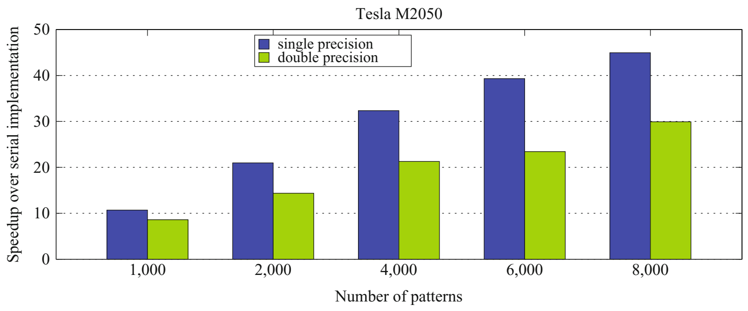
Implemented on a cluster of GPUs, identification of exemplars involves $O(n)$ non-coalesced memory accesses when scanning the diagonals of the dense matrices $\mathbf{R}$ and $\mathbf{A}$. Indices of the exemplars are then transferred to the host, exchanged between the GPU nodes in a single all-to-all gather operation and written back to the devices' global memory. Distribution of exemplar indices between the nodes is carried out with an MPI Gather-to-all operation [10]. Kernel that assigns non-exemplar patterns to exemplars resembles the kernel that find maxima in $\mathbf{A}^{(t-1)} - \mathbf{S}$ during updates to responsibilities. Each row of the similarity matrix, $\mathbf{S}_{i(\cdot)}$, that corresponds to a non-exemplar pattern $\mathbf{x}_i$ is scanned by one work group, with the k-th work item processing elements $\mathbf{S}_{ik},\ \mathbf{S}_{i(k+w)},\ \ldots$, where w is the size of the work group. Each work item finds maximum among those of the processed elements whose second indices correspond to exemplar patterns. A tree-reduction is then performed in the device local memory to find the row-wise maximum.

4 Performance Evaluation

The performance of the multi-GPU AP was evaluated on two clusters. First cluster consists of four dual GPU, NVIDIA GeForce GTX 295 nodes with Intel Xeon E5540 processors and 4xQDR Infiniband interconnect. Second cluster consists of 24 nodes, each one with two NVIDIA Tesla M2050 units and two Intel Xeon X5670 processors. We used 1Gbit Ethernet interconnect in this cluster. The GeForce GPU can use up to 900MB of global device memory. Tesla units have 3GB of global memory. Our multi-GPU AP implementation employs CUDA SDK version 3.2.16, MVAPICH library for the Infiniband interconnect and Intel MPI library for the Ethernet interconnect. In addition to the multi-GPU AP we also developed serial AP C code used as a reference implementation in the reported performance evaluation. Performance was evaluated on single and double precision random similarity matrices, ranging in size from $1,000 \times 1,000$ to $56,000 \times 56,000$ for the single precision cases and to $40,000 \times 40,000$ for the double precision cases. Each test case was run five times and median execution time of $1,000$ AP iterations was used in the performance evaluation.



(A)



(B)

Fig. 1. Speedup on (A) one GeForce GTX 295 GPU and (B) one Tesla M2050 unit over the serial CPU implementation

Figure 1 reports speedup on 1 GPU over the serial CPU implementation. In single precision arithmetic we achieve roughly 10-fold speedup on small test cases. For $8,000 \times 8,000$ single precision similarity matrix speedup exceeds 30 on GeForce GPU and 40 on Tesla unit. With double precision arithmetic performance decreases by roughly 50% on the GeForce GPU[5] and slightly less so on the Tesla units. Overall, the GPU implementation provide a sizable decrease in execution time.

To investigate how this performance level translates to multi-GPU environments, we have performed benchmark runs with fixed-size and variable-size test cases. The fixed-size, $8,000 \times 8,000$ single precision and $6,000 \times 6,000$ double precision test cases were run on the GeForce GTX 295 cluster with Infiniband interconnect. Performance numbers from these tests are reported in Fig. 2. For

[5] The $8,000 \times 8,000$ double precision test case does not fit in the global memory of one GeForce GTX 295 GPU, and is therefore not reported in Fig. 1.

problems of these size, the parallelization efficiency on eight GPUs exceeds 75%. It is worth noting, that one iteration of AP takes less than 6ms in these settings, and the whole clustering procedure can be accomplished in a matter of seconds.

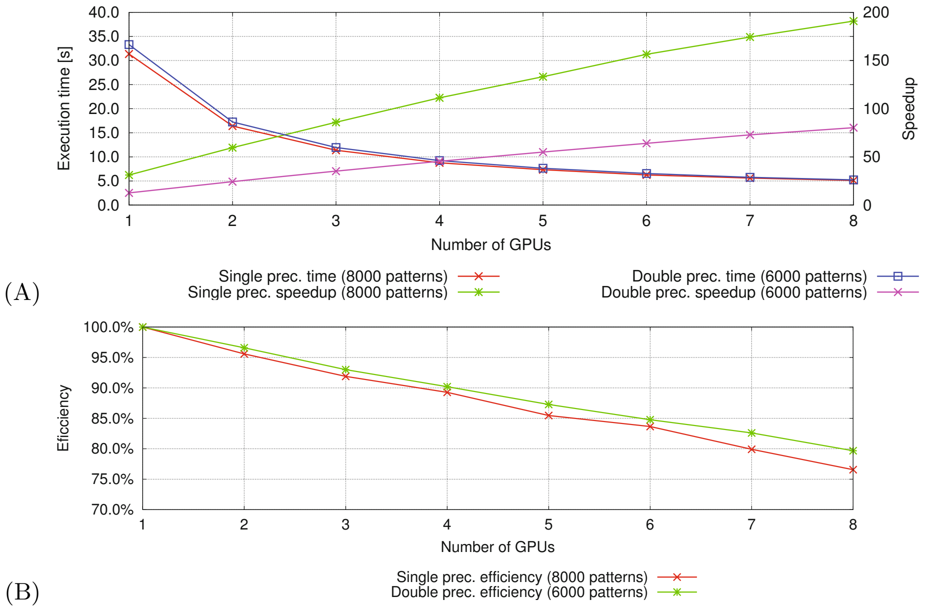


Fig. 2. Execution time, speedup over the serial implementation (A) and parallelization efficiency (B) for fixed-size similarity matrices with multi-GPU AP on GeForce GTX 295 and Infiniband interconnect

The variable-size test cases were designed to investigate the efficiency of multi-GPU AP when clustering large datasets. In these test cases the size of similarity matrices was increased proportionally to the number of employed GPUs. The global device memory on the GPUs was therefore kept filled. Tests were carried out on both GeForce GPUs connected by Infiniband network and Tesla M2050 units connected by 1Gbit Ethernet. The results are reported in Fig. 3. Note that even on the Ethernet interconnect there is only a marginal increase in execution time when problem size increases proportionally to the number of employed GPUs. On Infiniband interconnect execution time remains roughly constant. For the largest variable-size test cases we also measured time spent in the MPI All-Reduce operations[6]. On the GeForce GTX 295 cluster the total time spent in the MPI All-Reduce operations of 1000 AP iterations was approx. 2.4s in the largest single

[6] There is also an MPI Gather-to-all operation carried out after exemplar patterns are identified. Note however, that it is carried out only once and over $O(n)$ elements. Its cost is therefore insignificant compared to the total cost of the MPI All-Reduce operation, which also involve $O(n)$ elements but is invoked in every AP iteration.

precision case and approx. 0.8s in the largest double precision case. On the Tesla M2050 cluster these operations took in total approx. 5.5s in the largest single precision case and approx. 6.9s in the largest double precision case. Overall, the time spent in MPI operations in these test cases did not exceed several seconds.

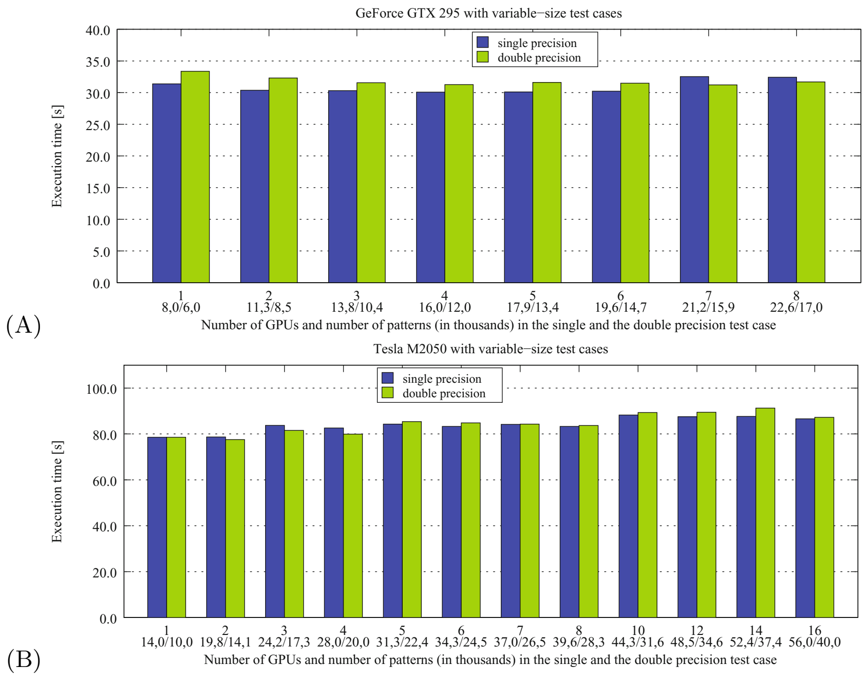


Fig. 3. Execution time with multi-GPU AP when the size of the similarity matrix increases proportionally to the number of employed GPUs. Results are reported for (A) GeForce GTX 295 GPUs with Infiniband interconnect and (B) Tesla M2050 units with 1Gbit Ethernet interconnect.

5 Conclusions

Results reported in Sect. 4 confirm that dense AP is very well suited for execution in parallel environments. One GPU alone provides several tens-fold decrease in execution time when compared to the serial CPU implementation. This performance level translates well to multi-GPU environments. In the multi-GPU implementation iteration of dense AP requires only one all-to-all reduction over $O(n)$-element array, even though GPUs process three floating point matrices, each one $O(n^2)$ in size. By running AP on multiple GPUs one can not only decrease the time needed to find the exemplars but, even more importantly, find exemplars in data which would not fit in global device memory of a single GPU. The largest test case reported in Fig. 3 involves a 12GB similarity matrix. With 16 Tesla units interconnected by 1Gbit Ethernet this test case takes a similar amount of time per AP iteration as an 800MB similarity matrix on one unit.

Our final remark concerns the arithmetic precision in multi-GPU AP implementation. At first glance it may seem not obvious whether double precision arithmetics is needed in multi-GPU AP. Double precision implementation suffers roughly 50% performance penalty when compared to the single precision implementation. Nevertheless, our experience suggests that in poorly convergent cases double precision implementation is less likely to fall into oscillations. This seems important mainly when working with large similarity matrices.

Acknowledgements. This work was founded by the Polish Ministry of Science and Higher Education grant no. N N519 443039 and AGH University of Science and Technology grant no. 11.11.120.865. For the Tesla cluster we used the infrastructure at the Academic Computer Centre CYFRONET AGH, provided by the POWIEW project. The project is co-funded by the European Regional Development Fund (ERDF) as a part of the Innovative Economy program.

References

1. Anderson, D., Luke, R., Keller, J.: Incorporation of non-euclidean distance metrics into fuzzy clustering on graphics processing units. In: Melin, P., Castillo, O., Ramrez, E., Kacprzyk, J., Pedrycz, W. (eds.) Analysis and Design of Intelligent Systems using Soft Computing Techniques. AISC, vol. 41, pp. 128–139. Springer, Heidelberg (2007)
2. Cao, F., Tung, A.K.H., Zhou, A.: Scalable Clustering Using Graphics Processors. In: Yu, J.X., Kitsuregawa, M., Leong, H.-V. (eds.) WAIM 2006. LNCS, vol. 4016, pp. 372–384. Springer, Heidelberg (2006)
3. Catanzaro, B.: OpenCL optimization case study: Simple reductions. White paper, AMD Developer Central (2010), http://developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study-Simple-Reductions.aspx
4. Charikar, M., Guha, S., Tardos, É., Shmoys, D.: A constant–factor approximation algorithm for the k–median problem. Journal of Computer and System Sciences 65(1), 129–149 (2002)
5. Frey, B., Dueck, D.: Clustering by passing messages between data points. Science 315(5814), 972–976 (2007)
6. Hall, J., Hart, J.: GPU acceleration of iterative clustering. In: The ACM Workshop on General Purpose Computing on Graphics Processors. Manuscript Accompanying Poster at GP2 (2004)
7. Hussein, M., Abd-Almageed, W.: Efficient band approximation of gram matrices for large scale kernel methods on GPUs. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009. ACM, New York (2009)
8. Ma, W., Agrawal, G.: A translation system for enabling data mining applications on GPUs. In: Proceedings of the 23rd International Conference on Supercomputing, pp. 400–409. ACM (2009)
9. MacQueen, J.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, pp. 281–297. University of California Press, Berkeley (1967)

10. Message Passing Interface Forum: MPI: A message passing interface standard (1995)
11. Pevsner, J.: Bioinformatics and functional genomics. Wiley-Blackwell (2009)
12. Shalom, S.A.A., Dash, M., Tue, M.: Efficient K-Means Clustering Using Accelerated Graphics Processors. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2008. LNCS, vol. 5182, pp. 166–175. Springer, Heidelberg (2008)
13. Takizawa, H., Kobayashi, H.: Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing. The Journal of Supercomputing 36, 219–234 (2006)
14. Wu, R., Zhang, B., Hsu, M.: Clustering billions of data points using GPUs. In: Proceedings of the Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop, pp. 1–6. ACM (2009)
15. Zhang, Q., Zhang, Y.: Hierarchical clustering of gene expression profiles with graphics hardware acceleration. Pattern Recognition Letters 27(6), 676–681 (2006)
16. Zhang, Y., Mueller, F., Cui, X., Potok, T.: Large-scale multi-dimensional document clustering on GPU clusters. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–10. IEEE (2010)

High-Performance Pseudo-Random Number Generation on Graphics Processing Units

Nimalan Nandapalan[1], Richard P. Brent[1,2], Lawrence M. Murray[3], and Alistair P. Rendell[1]

[1] Research School of Computer Science
[2] Mathematical Sciences Institute,
The Australian National University
[3] CSIRO Mathematics, Informatics and Statistics

Abstract. This work considers the deployment of pseudo-random number generators (PRNGs) on graphics processing units (GPUs), developing an approach based on the xorgens generator to rapidly produce pseudo-random numbers of high statistical quality. The chosen algorithm has configurable state size and period, making it ideal for tuning to the GPU architecture. We present a comparison of both speed and statistical quality with other common GPU-based PRNGs, demonstrating favourable performance of the xorgens-based approach.

Keywords: Pseudo-random number generation, graphics processing units, Monte Carlo.

1 Introduction

Motivated by compute-intense Monte Carlo methods, this work considers the tailoring of pseudo-random number generation (PRNG) algorithms to graphics processing units (GPUs). Monte Carlo methods of interest include Markov chain Monte Carlo (MCMC) [5], sequential Monte Carlo [4] and most recently, particle MCMC [1], with numerous applications across the physical, biological and environmental sciences. These methods demand large numbers of random variates of high statistical quality. We have observed in our own work that, after acceleration of other components of a Monte Carlo program on GPU [14,15], the PRNG component, still executing on the CPU, can bottleneck the whole procedure, failing to produce numbers as fast as the GPU can consume them. The aim, then, is to also accelerate the PRNG component on the GPU, without compromising the statistical quality of the random number sequence, as demanded by the target Monte Carlo applications.

Performance of a PRNG involves both speed and quality. A metric for the former is the number of random numbers produced per second (RN/s). Measurement of the latter is more difficult. Intuitively, for a given sequence of numbers, an inability to discriminate their source from a truly random source is indicative of high quality. Assessment may be made by a battery of tests which attempt to identify flaws in the sequence that are not expected in a truly random sequence.

These might include, for example, tests of autocorrelation and linear dependence. Commonly used packages for performing such tests are the DIEHARD [11] and TestU01 [9] suites.

The trade-off between speed and quality can take many forms. Critical parameters are the *period* of the generator (the length of the sequence before repeating) and its *state size* (the amount of working memory required). Typically, a generator with a larger state size will have a larger period. In a GPU computing context, where the available memory per processor is small, the state size becomes a critical design component. As a conventional PRNG produces a single sequence of numbers, an added challenge in the GPU context is to concurrently produce many uncorrelated streams of numbers.

Existing work in this area includes the recent release of NVIDIA's CURAND [17] library, algorithms in the Thrust C++ library [6] and elsewhere [20,8], and early work for graphics applications [7]. Much of this work uses simple generators with small state sizes and commensurately short periods, in order not to exceed the limited resources that a GPU provides to individual threads. While such generators are potentially very fast, the statistical quality of numbers produced is not necessarily adequate for modern Monte Carlo applications, and in some cases can undermine the procedure enough to cause convergence to the wrong result. Other recent work follows a trend in generators inspired by hashing algorithms, like those found in cryptographic analysis [21,22,18]. These generators are conceptually different to sequential generators, and while performance is comparable, a proper analysis of each exceeds the scope of this work.

The Mersenne Twister [13] is the *de facto* standard for statistical applications and is used by default in packages such as MATLAB. It features a large state size and long period, and has recently been ported to GPUs [19]. However, it has a fixed and perhaps over-large state size, and is difficult to tune for optimal performance on GPUs. In this work we adapt the xorgens algorithm [3,2]. The attraction of this approach is the flexible choice of period and state size, facilitating the optimisation of speed and statistical quality within the resource constraints of a particular GPU architecture.

We begin with a brief overview of CUDA, then discuss qualitative testing of PRNGs, and algorithms including the Mersenne Twister for Graphic Processors (MTGP), CURAND and xorgens. We then describe our adaptation of the xorgens algorithm for GPUs. Finally, the results of testing these generators are presented and some conclusions drawn.

1.1 The NVIDIA Compute Unified Device Architecture (CUDA) and the Graphics Processing Unit (GPU)

The Compute Unified Device Architecture (CUDA) was introduced by the NVIDIA Corporation in November 2006 [16]. This architecture provides a complete solution for general purpose GPU programming (GPGPU), including new hardware, instruction sets, and programming models. The CUDA API allows communication between the CPU and GPU, allowing the user to control the execution of code on the GPU to the same degree as on the CPU.

A GPU resides on a *device*, which usually consists of many *multiprocessors* (MPs), each containing some *processors*. Each CUDA compatible GPU device has a globally accessible memory address space that is physically separate from the MPs. The MPs have a local shared memory space for each of the processors associated with the MP. Finally, each processor has its own set of registers and processing units for performing computations.

There are three abstractions central to the CUDA software programming model, provided by the API as simple language extensions:

– A hierarchy of *thread* groupings – a thread being the smallest unit of processing that can be scheduled by the device.
– Shared memory – fast sections of memory common to the threads of a group.
– Barrier synchronisation – a means of synchronising thread operations by halting threads within a group until all threads have met the barrier.

Threads are organised into small groups of 32 called *warps* for execution on the processors, which are Single-Instruction Multiple-Data (SIMD) and implicitly synchronous. These are organised for scheduling across the MPs in *blocks*. Thus, each block of threads has access to the same shared memory space. Finally, each block is part of a grid of blocks that represents all the threads launched to solve a problem. These are specified at the invocation of *kernels* – functions executed on the GPU device – which are managed by ordinary CPU programs, known as *host* code.

As a consequence of the number of in-flight threads supported by a device, and the memory requirements of each thread, not all of a given GPU device's computational capacity can be used at once. The fraction of a device's capacity that can be used by a given kernel is known as its *occupancy*.

1.2 Statistical Testing: TestU01

Theoretically, the performance of some PRNGs on certain statistical tests can be predicted, but usually this only applies if the test is performed over a complete period of the PRNG. In practice, statistical testing of PRNGs over realistic subsets of their periods requires empirical methods [9,11].

For a given statistical test and PRNG to be tested, a test statistic is computed using a finite number of outputs from the PRNG. It is required that the distribution of the test statistic for a sequence of uniform, independently distributed random numbers is known, or at least that a sufficiently good approximation is computable [10]. Typically, a *p-value* is computed, which gives the probability that the test statistic exceeds the observed value.

The p-value can be thought of as the probability that the test statistic or a larger value would be observed for perfectly uniform and independent input. Thus the p-value itself should be distributed uniformly on $(0, 1)$. If the p-value is extremely small, for example of the order 10^{-10}, then the PRNG definitely *fails* the test. Similarly if $1 - p$ is extremely small. If the p-value is not close to 0 or 1, then the PRNG is said to *pass* the test, although this only says that the test failed to detect any problem with the PRNG.

Typically, a whole battery of tests is applied, so that there are many p-values, not just one. We need to be cautious in interpreting the results of many such tests; if performing N tests, it is not exceptional to observe that a p-value is smaller than $1/N$ or larger than $1 - 1/N$. The TestU01 library presented by L'Ecuyer[9] provides a thorough suite of tests to evaluate the statistical quality of the sequence produced by a PRNG. It includes and improves on all of the tests in the earlier DIEHARD package of Marsaglia [11].

1.3 The Mersenne Twister for Graphic Processors

The MTGP generator is a recently-released variant of the well known Mersenne Twister [13,19]. As its name suggests, it was designed for GPGPU applications. In particular, it was designed with parallel Monte Carlo simulations in mind. It is released with a parameter generator for the Mersenne Twister algorithm to supply users with distinct generators on request (MTGPs with different sequences). The MTGP is implemented in NVIDIA CUDA [16] in both 32-bit and 64-bit versions. Following the popularity of the original Mersenne Twister PRNG, this generator is a suitable standard against which to compare GPU-based PRNGs.

The approach taken by the MTGP to make the Mersenne Twister parallel can be explained as follows. The next element of the sequence, x_i, is expressed as some function, h, of a number of previous elements in the sequence, say

$$x_i = h(x_{i-N}, x_{i-N+1}, x_{i-N+M}).$$

The parallelism that can be exploited in this algorithm becomes apparent when we consider the pattern of dependency between further elements of the sequence:

$$\begin{aligned} x_i &= h(x_{i-N}, x_{i-N+1}, x_{i-N+M}) \\ x_{i+1} &= h(x_{i-N+1}, x_{i-N+2}, x_{i-N+M+1}) \\ &\vdots \\ x_{i+N-M-1} &= h(x_{i-M-1}, x_{i-M}, x_{i-1}) \\ x_{i+N-M} &= h(x_{i-M}, x_{i-M+1}, x_i). \end{aligned}$$

The last element in the sequence, which produces x_{i+N-M}, requires the value of x_i, which has not yet been calculated. Thus, only $N - M$ elements of the sequence produced by a Mersenne Twister can be computed in parallel.

As N is fixed by the Mersenne prime chosen for the algorithm, all that can be done to maximise the parallel efficiency of the MTGP is careful selection of the constant M. This constant, specific to each generator, determines the selection of one of the previous elements in the sequence in the recurrence that defines the MTGP. Thus, it has a direct impact on the quality of the random numbers generated, and the distribution of the sequence.

1.4 CURAND

The CUDA CURAND Library is NVIDIA's parallel PRNG framework and library. It is documented in [16]. The default generator for this library is based on the XORWOW algorithm introduced by Marsaglia[12]. The XORWOW algorithm is an example of the *xorshift* class of generators.

Generators of this class have a number of advantages. The algorithm behind them is particularly simple when compared to other generators such as the Mersenne Twister. This results in simple generators which are very fast but still perform well in statistical tests of randomness.

The idea of the xorshift class generators is to combine two terms in the pseudo-random sequence (integers represented in binary) using left/right shifts and "exclusive or" (xor) operations to produce the next term in the sequence. Shifts and xor operations can be performed quickly on computing architectures, typically faster than operations such as multiplication and division. Also, generators designed on this principle generally do not require a large number of values in the sequence to be retained (i.e. a large state space) in order to produce a sequence of satisfactory statistical quality.

1.5 Xorgens

Marsaglia's original paper [12] only gave xorshift generators with periods up to $2^{192}-1$. Brent[3] recently proposed the *xorgens* family of PRNGs that generalise the idea and have period $2^n - 1$, where n can be chosen to be any convenient power of two up to 4096. The xorgens generator has been released as a free software package, in a C language implementation (most recently xorgens version 3.05 [2]).

Compared to previous xorshift generators, the xorgens family has several advantages:

- A family of generators with different periods and corresponding memory requirements, instead of just one.
- Parameters are chosen optimally, subject to certain criteria designed to give the best quality output.
- The defect of linearity over GF(2) is overcome efficiently by combining the output with that of a Weyl generator.
- Attention has been paid to the initialisation code (see comments in [3,2] on proper initialisation), so that the generators are suitable for use in a parallel environment.

For details of the design and implementation of the xorgens family, we refer to [3,2]. Here we just comment on the combination with a Weyl generator. This step is performed to avoid the problem of linearity over GF(2) that is common to all generators of the Linear-Feedback Shift Register class (such as the Mersenne Twister and CURAND). A Weyl generator has the following simple form:

$$w_k = w_{k-1} + \omega \mod 2^w,$$

where ω is some odd constant (a recommended choice is an odd integer close to $2^{w-1}(\sqrt{5}-1)$). The final output of an xorgens generator is given by:

$$w_k(I + R^\gamma) + x_k \mod 2^w, \tag{1}$$

where x_k is the output before addition of the Weyl generator, γ is some integer constant close to $w/2$, and R is the right-shift operator. The inclusion of the term R^γ ensures that the least-significant bits have high linear complexity (if we omitted this term, the Weyl generator would do little to improve the quality of the least-significant bit, since (w_k mod 2) is periodic with period 2).

As addition mod 2^w is a non-linear operation over GF(2), the result is a mixture of operations from two different algebraic structures, allowing the sequence produced by this generator to pass all of the empirical tests in BigCrush, including those failed by the Mersenne Twister. A bonus is that the period is increased by a factor 2^w (though this is not free, since the state size is increased by w bits).

2 XorgensGP

Extending the xorgens PRNG to the GPGPU domain is a nontrivial endeavour, with a number of design considerations required. We are essentially seeking to exploit some level of parallelism inherent in the flow of data. To realise this, we examine the recursion relation describing the xorgens algorithm:

$$x_i = x_{i-r}(I + L^a)(I + R^b) + x_{i-s}(I + L^c)(I + R^d).$$

In this equation, the parameter r represents the degree of recurrence, and consequently the size of the state space (in words, and not counting a small constant for the Weyl generator and a circular array index). L and R represent left-shift and right-shift operators, respectively. If we conceptualise this state space array as a circular buffer of r elements we can reveal some structure in the flow of data. In a circular buffer, x, of r elements, where $\texttt{x[i]}$ denotes the i^{th} element, x_i, the indices i and $i + r$ would access the same position within the circular buffer. This means that as each new element x_i in the sequence is calculated from $\texttt{x[i} - \texttt{r]}$ and $\texttt{x[i} - \texttt{s]}$, the result replaces the r^{th} oldest element in the state space, which is no longer necessary for calculating future elements.

Now we can begin to consider the parallel computation of a sub-sequence of xorgens. Let us examine the dependencies of the data flow within the buffer $\texttt{x}$ as a sequence is being produced:

$$\begin{aligned}
x_i &= x_{i-r}A + x_{i-s}B \\
x_{i+1} &= x_{i-r+1}A + x_{i-s+1}B \\
&\vdots \\
x_{i+(r-s)} &= x_{i-r+(r-s)}A + x_{i-s+(r-s)}B \\
&= x_{i-s}A + x_{i+r-2s}B \\
&\vdots \\
x_{i+s} &= x_{i-r+s}A + x_{i-s+s}B \\
&= x_{i-r+s}A + x_iB.
\end{aligned}$$

If we consider the concurrent computation of the sequence, we observe that the maximum number of terms that can be computed in parallel is

$$\min(s, r - s).$$

Here r is fixed by the period required, but we have some freedom in the choice of s. It is best to choose $s \approx r/2$ to maximise the inherent parallism. However, the constraint $\mathrm{GCD}(r, s) = 1$ implies that the best we can do is $s = r/2 \pm 1$, except in the case $r = 2$, $s = 1$. This provides one additional constraint, in the context of xorgensGP versus (serial) xorgens, on the parameter set $\{r, s, a, b, c, d\}$ defining a generator. Thus, we find the thread-level parallelism inherent to the xorgens class of generators.

In the CUDA implementation of this generator we considered the approach of producing independent subsequences. With this approach the problem of creating one sequence of random numbers of arbitrary length, L, is made parallel by p processes by independently producing p subsequences of length L/p, and gathering the results. With the block of threads architecture of the CUDA interface and this technique, it is a logical and natural decision to allocate each subsequence to a block within the grid of blocks. This can be achieved by providing each block with its own local copy of a state space via the shared memory of an MP, and then using the thread-level parallelism for the threads within this block. Thus, the local state space will represent the same generator, but at different points within its period (which is sufficiently long that overlapping sequences are extremely improbable).

Note that, in contrast to MTGP, each generator is identical in that only one parameter set $\{r, s, a, b, c, d\}$ is used. The main advantage of this is that parameters can be known at compile time, allowing the compiler to make optimisations that would not be available if the parameters were dynamically allocated at runtime. This results in fewer registers being required by each thread, and so improved occupancy of the device. For the generator whose test results are given in §3, we used the parameters $(r, s, a, b, c, d) = (128, 65, 15, 14, 12, 17)$.

3 Results

We now present an empirical comparison of existing GPU PRNGs against our implementation of xorgensGP. All experiments were performed on an NVIDIA GeForce GTX 480 and a single GPU on the NVIDIA GeForce GTX 295 (which is a dual GPU device), using the CUDA 3.2 toolkit and drivers. Performance results are presented in Table 1, and qualitative results in Table 2.

We first compared the memory footprint of each generator. This depends on the algorithm defining the generator. The CURAND generator was determined to have the smallest memory requirements of the three generators compared, and the MTGP was found to have the greatest. The MTGP has the longest period ($2^{11213} - 1$), and the CURAND generator has the shortest period ($2^{192} - 2^{32}$).

Next, we compared the random number throughput (RN/s) of each generator on the two different devices. This was obtained by repeatedly generating 10^8

Table 1. Approximate memory footprints, periods and speed on two devices for 32-bit generators

Generator	State-Space	Period	GTX 480 RN/s	GTX 295 RN/s
xorgensGP	129 words	2^{4128}	17.7×10^9	9.1×10^9
MTGP	1024 words	2^{11213}	17.5×10^9	10.7×10^9
CURAND	6 words	2^{192}	18.5×10^9	7.1×10^9

random numbers and timing the duration to produce the sequence of that length. We found that the performance of each generator was roughly the same, with no significant speed advantage for any generator. On the newer GTX 480, the CURAND generator was the fastest, and the MTGP was the slowest. On the older architecture of the GTX 295 the ordering was reversed: the CURAND generator was the slowest and the MTGP was fastest. These results can be explained in part by the fact that the CURAND generator was designed with the current generation of "Fermi" cards like the GTX 480, and the MTGP was designed and tested initially on a card very similar to the GTX 295. In any event, the speed differences are small and implementation/platform-dependent.

Finally, to compare the quality of the sequences produced, each of the generators was subjected to the SmallCrush, Crush, and BigCrush batteries of tests from the TestU01 Library. The xorgensGP generator did not fail any of the tests in any of the benchmarks. Only the MTGP failed in the Crush benchmark, where it failed two separate tests. This was expected as the generator is based on the Mersenne Twister, and the tests are designed to expose the problem of linearity over GF(2). The MTGP failed the corresponding, more rigorous tests in BigCrush. Interestingly, the CURAND generator failed one of these two tests in BigCrush.

Table 2. Tests failed in each standard TestU01 benchmark

Generator	SmallCrush	Crush	BigCrush
xorgensGP	None	None	None
MTGP	None	#71,#72	#80,#81
CURAND	None	None	#81

4 Discussion

We briefly discuss the results of the statistical tests, along with some design considerations for the xorgensGP generator.

CURAND fails one of the TestU01 tests. This test checks for linearity and exposes this flaw in the Mersenne Twister. However, like the xorgensGP, CURAND combines the output of an xorshift generator with a Weyl generator to avoid linearity over GF(2), so it was expected to pass the test. The period $2^{192} - 2^{32}$ of the CURAND generator is much smaller than that of the other two generators. The BigCrush test consumes approximately 2^{38} random numbers, which is still only a small fraction of the period.

A more probable explanation relates to the initialisation of the generators at the block level. In xorgensGP each block is provided with consecutive seed values (the id number of the block within the grid). Correlation between the resulting subsequences is avoided by the method xorgens uses to initialise the state space. It is unclear what steps CURAND takes in its initialisation.

The MTGP avoids this problem by providing each generator with different parameter sets for values such as the shift amounts. This approach was also explored in developing xorgensGP. It was found that the overhead of managing parameters increased the memory footprint of each generator enough to impact device occupancy, and reduced the optimisations available to the compiler. It should also be noted that alternative parameter sets do not allow for the same degree of thread level parallelisation that the most optimal parameter set does. The performance of this version of the generator was noticeably less, without any detectable improvement to the quality of the sequence, and was not developed further.

In conclusion, we presented a new PRNG, xorgensGP, for GPUs using CUDA. We showed that it performs with comparable speed to existing solutions and with better statistical qualities. The proposed generator has a period that is sufficiently large for Monte Carlo applications, while not requiring too much state space, giving good performance on different devices.

References

1. Andrieu, C., Doucet, A., Holenstein, R.: Particle Markov chain Monte Carlo methods. Journal of the Royal Statistical Society Series B 72, 269–302 (2010)
2. Brent, R.P.: xorgens version 3.05 (2008), http://maths.anu.edu.au/~brent/random.html
3. Brent, R.P.: Some long-period random number generators using shifts and xors. ANZIAM Journal 48 (2007)
4. Doucet, A., de Freitas, N., Gordon, N. (eds.): Sequential Monte Carlo Methods in Practice. Springer (2001)
5. Gilks, W., Richardson, S., Spiegelhalter, D. (eds.): Markov chain Monte Carlo in practice. Chapman and Hall (1995)
6. Hoberock, J., Bell, N.: Thrust: A parallel template library (2010), http://thrust.googlecode.com
7. Howes, L., Thomas, D.: Efficient Random Number Generation and Application Using CUDA. GPU Gems 3. Addison-Wesley (2007)
8. Langdon, W.: A fast high quality pseudo random number generator for NVIDIA CUDA. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers. pp. 2511–2514. ACM (2009)
9. L'Ecuyer, P., Simard, R.: TestU01: A C library for empirical testing of random number generators. ACM Transactions on Mathematical Software 33 (2007)

10. Leopardi, P.: Testing the tests: using random number generators to improve empirical tests. In: Monte Carlo and Quasi-Monte Carlo Methods 2008, pp. 501–512 (2009)
11. Marsaglia, G.: DIEHARD: a battery of tests of randomness (1996), http://stat.fsu.edu/~geo/diehard.html
12. Marsaglia, G.: Xorshift RNGs. Journal of Statistical Software 8(14), 1–6 (2003)
13. Matsumoto, M., Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Transactions on Modeling and Computer Simulation 8, 3–30 (1998)
14. Murray, L.M.: GPU acceleration of Runge-Kutta integrators. IEEE Transactions on Parallel and Distributed Systems 23, 94–101 (2012)
15. Murray, L.M.: GPU acceleration of the particle filter: The Metropolis resampler. In: DMMD: Distributed Machine Learning and Sparse Representation with Massive Data Sets (2011)
16. NVIDIA Corp: CUDA Compute Unified Device Architecture Programming Guide Version 3.2. NVIDIA Corp., Santa Clara, CA 95050 (2010)
17. NVIDIA Corp: CUDA CURAND Library. NVIDIA Corporation (2010)
18. Phillips, C.L., Anderson, J.A., Glotzer, S.C.: Pseudo-random number generation for Brownian dynamics and dissipative particle dynamics simulations on GPU devices. Journal of Computational Physics 230(19), 7191–7201 (2011), http://www.sciencedirect.com/science/article/pii/S0021999111003329
19. Saito, M.: A variant of Mersenne Twister suitable for graphic processors (2011), http://arxiv.org/abs/1005.4973
20. Sussman, M., Crutchfield, W., Papakipos, M.: Pseudorandom number generation on the GPU. In: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pp. 87–94. ACM, New York (2006), http://dl.acm.org/citation.cfm?id=1283900.1283914
21. Tzeng, S., Wei, L.: Parallel white noise generation on a GPU via cryptographic hash. In: Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games, pp. 79–87. ACM (2008)
22. Zafar, F., Olano, M., Curtis, A.: GPU random numbers via the tiny encryption algorithm. In: Proceedings of the Conference on High Performance Graphics, pp. 133–141. Eurographics Association, Aire-la-Ville (2010), http://dl.acm.org/citation.cfm?id=1921479.1921500

Auto-tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs

Hans Henrik Brandenborg Sørensen

Informatics and Mathematical Modelling,
Technical University of Denmark, Bldg. 321, DK-2800 Lyngby, Denmark
hhs@imm.dtu.dk
http://www.gpulab.imm.dtu.dk

Abstract. In this paper, we consider the automatic performance tuning of dense vector and matrix-vector operations on GPUs. Such operations form the backbone of level 1 and level 2 routines in the Basic Linear Algebra Subroutines (BLAS) library and are therefore of great importance in many scientific applications. As examples, we develop single-precision CUDA kernels for the Euclidian norm (SNRM2) and the matrix-vector multiplication (SGEMV). The target hardware is the most recent Nvidia Tesla 20-series (Fermi architecture). We show that auto-tuning can be successfully applied to achieve high performance for dense vector and matrix-vector operations by appropriately utilizing the fine-grained parallelism of the GPU. Our tuned kernels display between 25-100% better performance than the current CUBLAS 3.2 library.

Keywords: GPU, BLAS, Dense linear algebra, Parallel algorithms.

1 Introduction

Graphical processing units (GPUs) have already become an integral part of many high performance computing systems, since they offer dedicated parallel hardware that can potentially accelerate the execution of many scientific applications. Currently, in order to exploit the computing potential of GPUs, the programmer has to use either Nvidia's Compute Unified Device Architecture (CUDA) [1] or the Open Compute Language (OpenCL) [2]. Recent years have shown that many applications written in these languages, which fully utilize the hardware of the target GPU, show impressive performance speed-ups compared to the CPU. However, developers working in these languages are also facing serious challenges. Most importantly, the time and effort required to program an optimized routine or application has also increased tremendously.

An essential prerequisite for many scientific applications is therefore the availability of high performance numerical libraries for linear algebra on dense matrices, such as the BLAS library [3] and the Linear Algebra Package (LAPACK) [4]. Several such libraries targeting Nvidia's GPUs have emerged over the past few years. Apart from Nvidia's own CUBLAS, which is part of the CUDA Toolkit [1], other prominent libraries are the open source Matrix Algebra on GPU and Multicore Architectures (MAGMA) library [5] and the commercial CUDA GPU-accelerated linear algebra (CULA) library [6]. These provide subsets of the functionality of BLAS/LAPACK for GPUs but are not mature in their current versions.

Automatic performance tuning, or auto-tuning, has been widely used to automatically create near-optimal numerical libraries for CPUs [7], e.g., in the famous the ATLAS package [8]. Modern GPUs offer the same complex and diverse architectural features as CPUs, which require nontrivial optimization strategies that often change from one chip generation to the next. Therefore, as already demonstrated in MAGMA [9], auto-tuning is also very compelling on GPUs.

In this paper, our goal is to design vector and matrix-vector GPU kernels based on template parameters and auto-tune them for given problem sizes. All the kernels, which may be candidates as the best kernel for a particular problem size, are thereby generated automatically by the compiler as templates, and do not need to be hand-coded by the programmer.

In this work, we target Nvidia GPUs, specifically the Tesla C2050 (Fermi architecture), which is designed from the outset for scientific computations. All kernels are made in the CUDA programming model (Toolkit 3.2), where the hardware is controlled by using blocks (3D objects of sizes $1024 \times 1024 \times 64$ containing up to 1024 threads) and grids (2D objects containing up to 65535×65535 blocks). Groups of 32 threads are called warps. We consider only the single-precision case and present the double-precision results in another work.

This paper is organized as follows. Sect. 2 states the performance considerations for our kernels. Next we describe the implementations. In Sect. 4 we describe the auto-tuning process. The experimental results are presented in Sect. 5.

2 Performance Considerations

When implementing a GPU kernel in CUDA, the path to high performance follows mainly two directions. The first is to maximize the instructions throughput and the second is to optimize the memory access patterns. Often one needs to focus attention only on one of these directions, depending on whether the maximum performance is bounded by the first or the latter.

2.1 Memory Bound Kernels

For our target GPU the single-precision peak performance is 1.03 Tflops and the theoretical memory bandwidth is 144 GB/s. On such hardware, the kernels we develop for vector and matrix-vector operations will consistently fall under the memory bound category. For example, a matrix-vector multiplication requires N^2+2N memory accesses and $2N^2$ floating point operations. Since the resulting arithmetic intensity is much less than the perfect balance (~ 4.5 flops per byte) for the target GPU [10], the corresponding kernel is inherently memory bound for all N. This means that the arithmetic operations are well hidden by the latency of memory accesses, and we will concentrate on optimizing the memory access pattern in order to reach the maximum memory bandwidth of the GPU.

2.2 Coalesced Memory Access

In general, two requirements must be fulfilled for a kernel's memory access to be efficient. First, the access must be contiguous so that consecutive threads within a warp (32 threads) always read contiguous memory locations. Second, the memory must be properly aligned so that the first data element accessed by any warp is always aligned on 128 bytes segments. This allows the kernel to read 32 memory locations in parallel as a single 128 byte memory access, a so-called coalesced access. If by design an algorithm is required to read data in a non-coalesced fashion, one can use the shared memory available on the graphics card to circumvent such access patterns. Shared memory should also be used if data is to be re-used or communicated between threads within a block.

2.3 Registers

A key performance parameter in CUDA is the number of registers used per thread. The fewer registers a kernel uses, the more threads and blocks are likely to reside on a multiprocessor, which can lead to higher occupancy (the ratio of the resident warps on the hardware to the maximum possible number of resident warps). A large number of warps is often required to hide the memory access latencies well. However, since registers represents the major part of the per-multiprocessor memory and have the shortest access latency, it is advantageous to implement high-performance kernels for exhaustive register usage, if possible.

2.4 Loop Unrolling

It is important to design memory bound kernels with enough fine-grained thread-level parallelism (TLP) to allow for the occupancy to be high and latencies well hidden. Alternatively, having many independent instructions, i.e. high instruction level parallelism (ILP), can also hide the latencies [11]. In our kernels we allow the compiler to unroll inner loops (using the keyword `#pragma unroll`), which will increase the instruction level parallelism of the kernels and facilitate some degree of latency hiding. Unfortunately, this may also increase the register usage which may lower the occupancy and subsequently performance. The key is to find the best compromise between ILP, TLP, the number of threads per block, the register usage, and the shared memory usage to achieve the best performance of a given kernel. To this end, we will employ auto-tuning.

3 Vector and Matrix-Vector Operations on Fermi GPUs

The Tesla C2050 Fermi architecture provides 14 multiprocessors of 32 cores each that can execute threads in parallel. In CUDA, we utilize this parallel hardware by distributing the work of an operation to the individual threads via a grid of blocks. During execution, the blocks are assigned to multiprocessors, which further split each block into sets of 32 threads known as warps, that execute the same instructions on the multiprocessor synchronously.

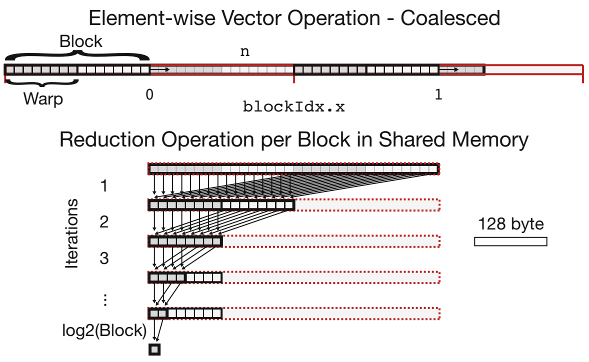

Element-wise Vector Operation - Coalesced
Block
n
Warp
0
blockIdx.x
1
Reduction Operation per Block in Shared Memory
Iterations
1
2
3
⋮
log2(Block)
128 byte

Fig. 1. Coalesced access pattern for the element-wise operation on a vector and the subsequent parallel reduction operation per block in shared memory

3.1 Operations on a Vector

In Fig. 1, we illustrate the two main cases of operations on vectors, where the first corresponds to merely reading and writing a vector as required for an operation on the individual elements of the vector, e.g., a vector scale or vector add (SAXPY), and the second corresponds to the subsequent reduction operation in shared memory required for, e.g., a sum or an Euclidian norm (SNRM2).

In the first case, the operation is embarrassingly parallel and the memory access pattern for reading the vector in a CUDA kernel is made fully coalesced by having a block size given by a multiple of the warp size (assuming the vector is stored at an aligned address). Each thread can be assigned to handle one or more elements. Whether the values should be stored in shared memory for optimal usage depends on the operation to be performed on its elements.

In the second case, the illustrated access pattern for the reduction is designed to avoid shared memory bank conflicts [12]. It requires log_2(`BLOCKSIZE`) iterations, where the last 5 are executed synchronously per warp. Although, the reduction operation suggested here suffers from performance inhibiters like the use of explicit synchronizations and leaving threads idle in the last 5 iterations, this technique is currently the optimal for reducing a result on Fermi GPUs.

3.2 Operations on a Matrix

An important access pattern for operations on a matrix is when each thread transverses elements of a given row in order to operate on the individual elements of the row or to reduce a result or part of a result. E.g., the typical parallel implementation of a matrix-vector multiplication (SGEMV), where each thread performs a dot product between one row of $\mathbf{A}$ and $\mathbf{x}$ to produce one element of the result $\mathbf{y}$. In the common case, where the matrix is stored in column major memory layout, this access pattern can be achieved by dividing the matrix into slices of `BLOCKSIZE` rows and launching a block for each of them. Each thread

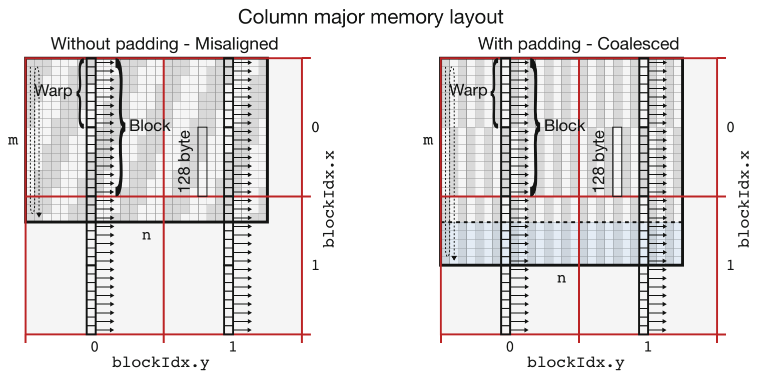


Fig. 2. Access pattern for the row-wise reading of matrices having column major memory layout. Left; the misaligned case of arbitrary number of rows. Right; the coalesced case occurring when the number of rows is padded to a multiple of the warp size.

might only take care of part of a row, which is accomplished by dividing the matrix into tiles instead of slices and using a 2D grid of blocks.

As illustrated in the left part of Fig. 2, the described memory access pattern for an arbitrary $m \times n$ matrix is contiguous but misaligned for each warp (except for every 32nd column). In single precision, each warp of 32 threads will request $32 \times 4 = 128$ bytes of memory per memory access in the kernel. On a Fermi GPU with compute capability 2.0, the misalignment breaks the memory access of the required 128 bytes per warp into two aligned 128 byte segment transactions [13].

As illustrated in the right part of Fig. 2, for the matrix memory accesses to be coalesced, both the number of threads per block and the height of the matrix must be a multiple of the warp size. In particular, this means that a matrix whose height is not a multiple of the warp size will be accessed more efficiently if it is actually allocated with a height rounded up to the closest multiple of this size and its columns padded accordingly.

3.3 Operations on a Transposed Matrix

In BLAS, all level 2 routines for matrix-vector multiplication and linear solvers are available for ordinary as well as transposed matrices (by specifying the `TRANS` argument to `'T'`). The operations on transposed matrices can be advantageously implemented without explicit transpositions. Moreover, one can view the operations on transposed matrices stored in column major layout as equivalent to operations on ordinary matrices stored in row major layout.

In Fig. 3, we illustrate an access pattern for an operation that requires the row-wise transversal of matrix elements in row major memory layout, e.g., $\mathbf{y} = \mathbf{A}^T\mathbf{x}$ (SGEMV with `'T'`). The threads in a block are distributed along the rows of the matrix in order for the memory access to be contiguous for each warp. For the case of arbitrary number of columns n (left part of the figure) the access will be misaligned. This can only be avoided if the width of the matrix is padded to a multiple of the warp size (right part of the figure).

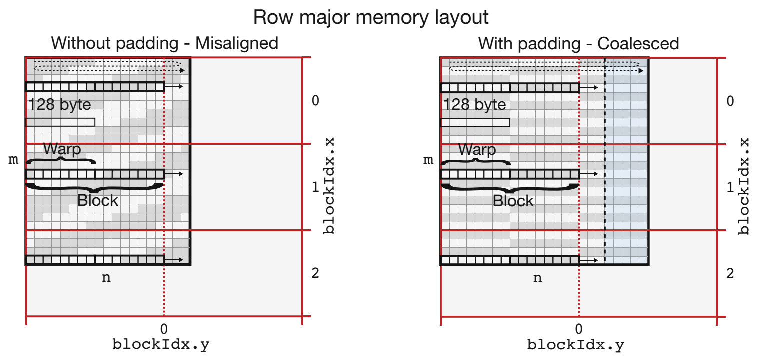


Fig. 3. Access pattern for the row-wise reading of matrices having row major memory layout. Left; the misaligned case of arbitrary number of columns. Right; the coalesced case occurring when the number of columns is padded to a multiple of the warp size.

In the access pattern shown, we designate one block per row and the threads have to work together if a reduction result is required, i.e., the row-wise transversal might be followed by a reduction in shared memory as discussed above. We allow each thread to take care of more elements on the same row. In addition, we allow each block to take care of more than one row by dividing the matrix into slices of a given number of rows and launching a block for each of them.

4 Auto-tuning

In order to configure the vector and matrix-vector kernels with optimal parameters for the targeted GPU we will use auto-tuning. We have implemented an auto-tuning framework that can automate the performance tuning process by running a large set of empirical benchmarks. Our search strategy is to consider a sub-set of the possible configurations based on knowledge we have about the GPU hardware and find the optimal among them in a brute-force manner.

4.1 Using C++ Templates

GPU auto-tuners can be constructed in a variety of ways to test different kernel implementations, ranging from simply calling the kernel with different function arguments to run-time code generation of kernel candidates [14]. In this work, we implement the auto-tuner based on C++ function templates. The goal is to represent all tuning parameters as template values, which are then evaluated at compile time. This allows inner loops over these parameters to be completely unrolled and conditionals to be evaluated by the compiler at compile time. We are also able to set launch bounds depending on the tuning parameters using the CUDA keyword `__launch_bounds__`. All that is required is the declaration of the kernel as a template via `template <..>` and a large switch statement [12].

4.2 Tuning Parameters

The vector and matrix-vector operations implemented in this work incorporate three tuning parameters, which are built into the design of the kernels.

Parameter 1: Block Size. Commonly, the most important tuning parameter in the CUDA model is the number of threads per block. Since the smallest work entity to be scheduled and executed is a warp of 32 threads, we know that having BLOCKSIZE a multiple of 32 is the best choice for a high-performance kernel. We also know that to reach an occupancy of 1, at least 192 threads per block are needed [15], while to use the maximum 63 registers per thread at most 64 threads per block are allowed [15]. This trade-off leads us to search the parameter space

$$\texttt{BLOCKSIZE} \in \{32, 64, 96, 128, 160, 192, 224, 256\}, \tag{1}$$

for the optimal value (experiments confirm this to be appropriate for the C2050).

Parameter 2: Work Size per Thread. Another tuning parameter adresses the performance trade-off between launching many threads in order to utilize the fine-grained parallelism of the GPU and having each thread perform a reasonable amount of work before it retires. Empirically, we found that the parameter space

$$\texttt{WORKSIZE} \in \{1, 2, 3, 4, 5, 6, 7, 8\} \times \texttt{BLOCKSIZE}, \tag{2}$$

for the number of elements handled per thread, is adequate for the C2050.

Parameter 3: Unroll Level. A final tuning parameter built into the design of our kernels is related to the CUDA compiler's technique for unrolling inner loops, where a particular unroll level `x` can be specified by `#pragma unroll x`. Using a high level gives the smallest loop counter overhead and fewer instructions but it also requires more registers per thread. We found that the unroll levels

$$\texttt{UNROLL\_LEVEL} \in \{\texttt{FULL}, 2, 3, 4, 5, 6, 7, 8\}, \tag{3}$$

can lead to different performances and this space is therefore searched.

We note that this amounts to total of $8 \times 8 \times 8 = 512$ configurations of our vector and matrix-vector kernels to be auto-tuned for a given problem size.

5 Results

Our test platform is a Nvidia Tesla C2050 card having 3 GB device memory on a host with a quad-core Intel® Core™i7 CPU operating at 2.80 GHz. The GPU's peak performance is 1.03 Tflops and the theoretical bandwidth is 144 GB/s. The error correction code (ECC) is on. A simple read-only kernel that estimates the effective bandwidth gives 94.0 GB/s. Note that the performance timings shown do not include transfer of data between host and GPU unless stated otherwise.

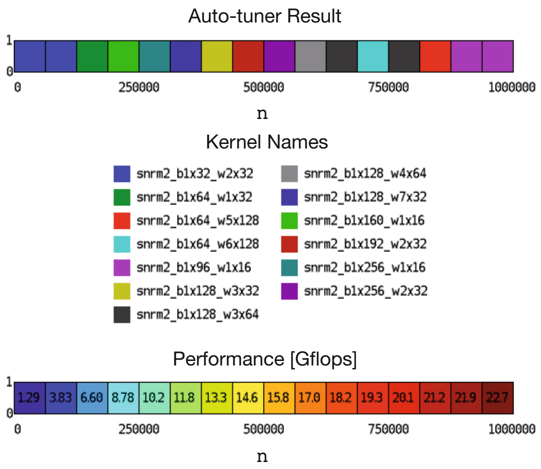


Fig. 4. Result of auto-tuning for the SNRM2 kernel on a 1×16 grid for vector sizes up to $n = 1000000$. Top; best kernel designated by color and name. Bottom; performance in Gflops.

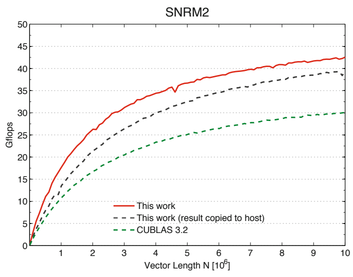


Fig. 5. Performance of the auto-tuned SNRM2 kernel on a Nvidia Tesla C2050 card. The curves show the average performance from ten subsequent calls to the kernel.

5.1 Euclidian Norm (SNRM2) on Fermi GPU

We show the result from auto-tuning the SNRM2 kernel in Fig. 4 for sizes up to 1000000. The top panel displays the selected best kernel designated by color and the bottom panel the corresponding performance achieved in Gflops. Also the names of the best kernels are listed in the middle in a form where the name of the operation is appended by "b1×{BLOCKSIZE}_w{UNROLL_LEVEL}×{WORKSIZE}".

In Fig. 5, we show the average performance of the auto-tuned SNRM2 kernel over a span of sizes of up to 10^7 elements. We compare the achieved results with the similar performance measurement for the SNRM2 kernel from CUBLAS 3.2 library. Note that the SNRM2 function in CUBLAS 3.2 gives the result on the host while our SNRM2 function by default gives the result on the GPU. For the sake of comparison, we make a version of our kernel that also copies the result to the host. As shown, the difference in performance because of this is relatively small and becomes smaller for larger sizes of n. On average, our auto-tuned SNRM2 kernel performs $> 30\%$ better than the current CUBLAS 3.2 kernel.

5.2 Matrix-Vector Multiplication (SGEMV) on Fermi GPU

We show the result of auto-tuning our SGEMV kernel in Fig. 6, where we have considered matrix sizes up to 10000 rows and 10000 columns on an 8×8 tuning

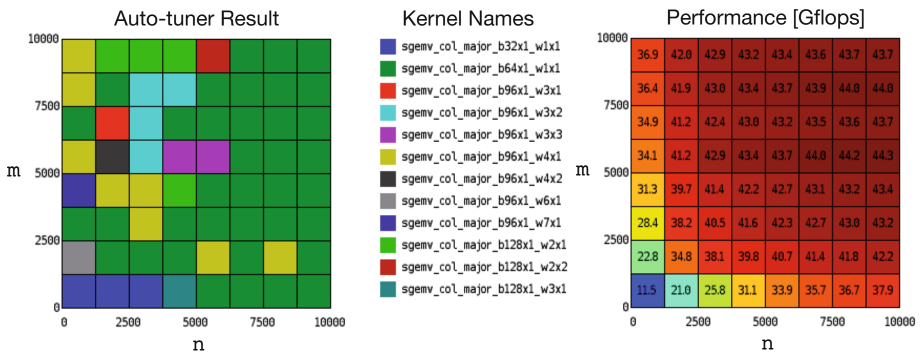


Fig. 6. Result of auto-tuning for the SGEMV kernel on a 8×8 grid for matrix sizes $m \times n$ up to 10000 Left; best kernel designated by color. Right; performance in Gflops.

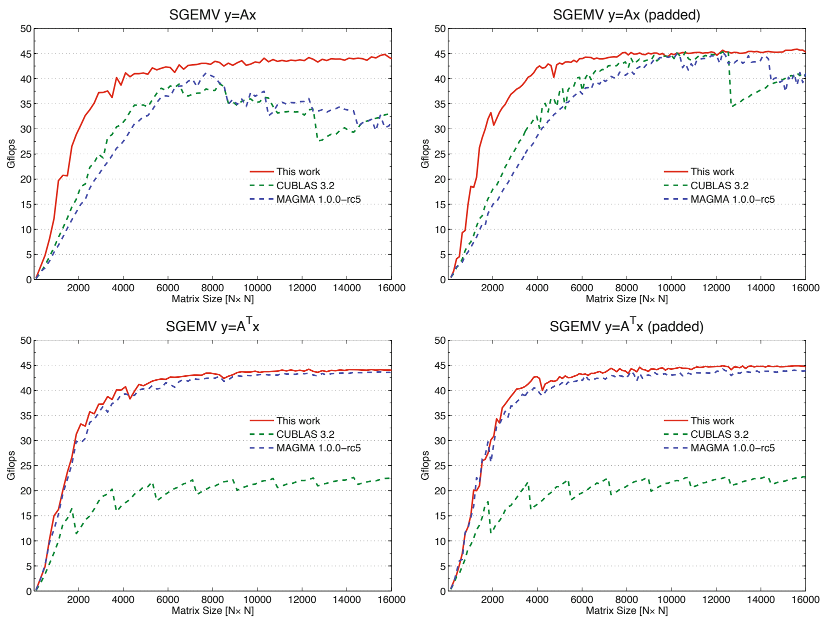


Fig. 7. Performance of the auto-tuned SGEMV kernel (with `TRANS` = 'N' and 'T') on a Nvidia Tesla C2050 card. The curves show the average performance from ten calls.

grid. The auto-tuner and performance results shown are obtained as averages from 25 samples within each tuning grid tile. A total of 12 different best kernels, designated by a unique color, are selected. The names of the best kernels have the extension "b{`BLOCKSIZE`}×1_w{`UNROLL_LEVEL`}×{`WORKSIZE`}".

In Fig. 7, we show the achieved performance of the auto-tuned SGEMV kernel in the cases of ordinary ($\mathbf{y} = \mathbf{Ax}$) and transposed ($\mathbf{y} = \mathbf{A}^{\mathrm{T}}\mathbf{x}$) square matrix-vector multiplication and for matrices with and without padding. We see a significant improvement in comparison with the corresponding kernel in the current CUBLAS 3.2 library (up to $\sim 100\%$ in the transposed case).

We also compare with the most recent MAGMA library [5] and see some improvement in the ordinary matrix-vector multiplication case. In the transposed matrix-multiplication case, our auto-tuned kernel confirms that MAGMA's kernel is already highly optimized for square matrices of the sizes considered here.

In addition, the performance results show that the padding of matrices is not necessary in order to achieve high performance on the C2050 card. In our kernel the increase in performance from padding to a multiple of the warp size is only a few percent. We credit this to the L1 and L2 caches available on Fermi GPUs.

6 Conclusion

In this work, we have implemented vector and matrix-vector operations as high-performance GPU kernels designed for auto-tuning. We used auto-tuning of the kernels in order to select the optimal kernel parameters on the Tesla C2050 card (Fermi GPU). The auto-tuning consisted of an exhaustive search of the tuning space containing key hardware dependent parameters that sets the number of threads per block, the work per thread, and the unroll level of the inner-most loop. An analysis of heuristics to reduce the search space is left as future work.

We have illustrated the approach for the Level 1 BLAS routines, with the example of the Euclidian norm, and for the Level 2 BLAS routines in the case of the matrix-vector product operations. We achieve significantly better performance compared to the CUBLAS 3.2 library. Two other basic Level 2 operations, the rank-1 and rank-2 updates and the triangular solve, are also left as future work.

References

1. NVIDIA Corp.: CUDA Toolkit Version 3.2. (2010)
2. Khronos Group: OpenCL Specification 1.1. (2010)
3. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. 16, 1–17 (1990)
4. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J.J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., Sorensen, D.: LAPACK Users' guide, 3rd edn. SIAM, Philadelphia (1999)
5. Tomov, S., Nath, R., Du, P., Dongarra, J.: MAGMA v0.2 Users' Guide (2009)
6. Humphrey, J.R., Price, D.K., Spagnoli, K.E., Paolini, A.L., Kelmelis, E.J.: CULA: hybrid GPU accelerated linear algebra routines. In: Proc. SPIE, vol. 7705 (2010)
7. Dongarra, J., Moore, S.: 12. In: Empirical Performance Tuning of Dense Linear Algebra Software, pp. 255–272. CRC Press (2010)
8. Whaley, R.C., Petitet, A., Clint, R., Antoine, W., Jack, P., Dongarra, J.J.: Automated Empirical Optimizations of Software and the ATLAS project (2000)
9. Li, Y., Dongarra, J., Tomov, S.: A note on auto-tuning gemm for gpus (2009)
10. Micikevicius, P.: Analysis-driven performance opt. GTC, Recorded Session (2010)
11. Volkov, V.: Better performance at lower occupancy. GTC, Recorded Session (2010)

12. Harris, M.: Optimizing parallel reduction in cuda. NVIDIA Dev. Tech. (2008)
13. NVIDIA Corp.: CUDA C Programming Guide Version 3.2. (2010)
14. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: PyCUDA: GPU Run-Time Code Generation for High-Performance Computing (2009)
15. NVIDIA Corp.: CUDA GPU Occupancy Calculator (2010)

GPGPU Implementation of Cellular Automata Model of Water Flow

Paweł Topa[1,2] and Paweł Młocek[1]

[1] AGH University of Science and Technology, Department of Computer Science, al. Mickiewicza 30, Kraków, Poland
topa@agh.edu.pl
[2] Institute of Geological Sciences, Polish Academy of Sciences, Research Centre in Kraków, Senacka St. 1, PL-31002 Kraków, Poland

Abstract. In this paper we present how Cellular Automata model can be implemented for processing on Graphics Processing Unit (GPU). Recently, graphics processors have gained a lot of interest as an efficient architecture for general-purpose computation. Cellular Automata algorithms that are inherently parallel give the opportunity to achieve very high efficiency when they are implemented on GPUs. We demonstrate how existing model of water flow can be ported to GPU environment with OpenCL programming framework. Sample simulation results and performance evaluations are included.

Keywords: Cellular Automata, GPGPU, OpenCL.

1 Introduction

Cellular Automata is a very efficient modelling paradigm for simulating variety of natural phenomena and physical processes [1], [2]. It models physical system as a lattice of cells that are characterized by a specific set of states that change according to certain rules of local interaction. The most important advantage of this approach is the ability to construct very fast and conceptually simple algorithms. CA based models overwhelm traditional approach based on solving equations by eliminating problems with numerical stability, round-off errors and truncation errors. Rules of local interaction that govern evolution of CA can be easily encoded as instructions of a programming language. Algorithms usually use relatively simple structure that gives the opportunity to provide fast access. CA is inherently parallel — the definition states that all cells should be processed simultaneously. The neighbourhood is usually limited to the nearest cells and remains invariant. As a result, only limited amount of data is exchanged between computational nodes. Therefore, the CA models can be very easily parallelized, just by simple static domain decomposition.

Cellular Automata models can be efficiently implemented on Graphics Processing Units (GPU) which are specialized for processing large amount of data with regular structure. For this reason, the CA models are readily implemented on the GPU. Gobron et al. ([3], [4]) applied cellular automata in computer graphics for visualisation and creating various surface effects like automatic texturing and creating surface imperfections. Their models were implemented for GPUs with GLSL language and gained performance up to 200 times better than CPU implementations. In [5] the same author demonstrated that the cellular automata model of retina implemented for GPUs was at least twenty times as fast as one for a classical CPU. Ferrando et al. [6] applied GPU for simulating evolving surfaces with continuous cellular automata and showed that the GPU-based algorithm is at least two orders of magnitude faster than the purely sequential code. Caux et al. [7] discussed various memory management scenarios in GP-GPU implementations of the cellular automata models applied in biology.

In recent years, it was noted that the computing power of GPUs increased rapidly surpassing, in some applications, the performance of general purpose processors. This has led to the conclusion that GPU can be also utilized for general purpose computing (GPGPU *General-Purpose Computing on Graphics Processing Units*) [10]. Initially, such computations were performed by programming vertex and pixel shaders [5]. In 2007, Nvidia announced CUDA (*Compute Unified Device Architecture*) [8] architecture with a dedicated programming language. Khronos Group in 2008 presented an open programming environment OpenCL (*Open Computing Language*) [9]. The goal of this project is to provide a unified programming environment for creating programs that use various computational resources available in modern computers: CPU units as well as GPU units. Both tools allow easy programming of the GPU for any type of computational problems.

This paper is organized as follows. In the next section we briefly describe the original algorithm as it was exploited in previous works. Next we present what modification has been introduced to the algorithm to make it more GPU-friendly. Implementation on the GPU is described in a very detailed way. Results of testing the algorithm efficiency are presented next. At the end we conclude our achievements.

2 Model

Cellular Automata algorithm presented in this paper, simulates flow of water through the terrain. It is based on a model of lava flow developed by Di Gregorio et al [11]. The algorithm was adapted to model water flow [12] and later it was applied to model anastomosing river phenomenon [13], [14], [15]. One of the main disadvantage of the model of anastomosing river was the need for a very accurate modelling of the terrain shape, which made it very time consuming. Application of GPU computing gives the opportunity to significantly speedup the algorithm.

We define Cellular Automata for modelling water flow as:

$$CA_{FLOW} = \langle Z^2, A_i, A_o, X, S, \delta \rangle \tag{1}$$

where:

- Z^2 — set of cells located on regular mesh and indexed by (i, j);
- $A_i \subset Z^2$ — set of inflows;
- $A_o \subset Z^2$ — set of sinks;
- X — Moore neighbourhood;
- $S = \{(g_{i,j}, w_{i,j}\}$ — set of parameters:
  - g — altitude,
  - w — amount of water;
- $\delta : (g^t_{i,j}, w^t_{i,j}) \to (g^{t+1}_{i,j}, w^{t+1}_{i,j})$ — set of rules of local interactions.

The key part of the model is a level minimization algorithm that calculates an exchange of water between neighbouring cells. The factor that drives the process of water exchange is a difference in water level, calculated as sum of g_{ij} (ground) and w_{ij} (water). The algorithm calculates an expected average level of water in cells. If there is a cell in neighbourhood for which the water level (or only the ground level) is higher than the calculated average level, it is eliminated from computation and the average level is calculated once again. Calculations are repeated until no cell is eliminated. Finally, the algorithm calculates a distribution of water between central cell and all not eliminated neighbours. The original algorithm is described in a very detailed way in [11], [12]. Here we only briefly outline it (see Fig. 1).

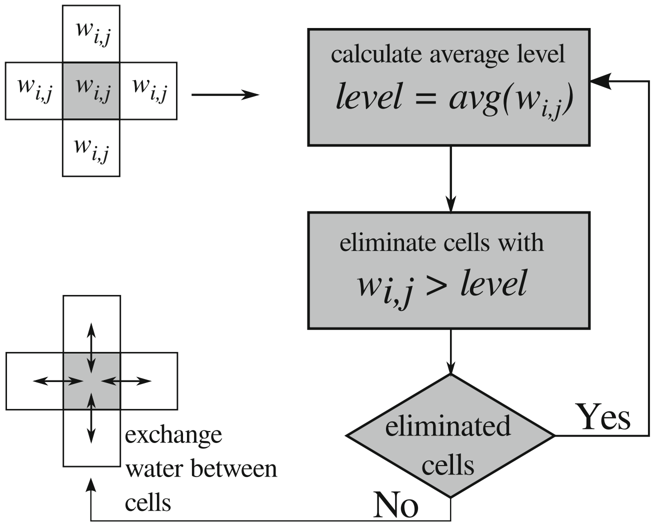


Fig. 1. Diagram demonstrating general idea of the original algorithm of water flow

For the purpose of implementing the model on GPU the algorithm has been redesigned. Unlike in original algorithm, the average level of water is calculated by including cells that can participate in exchanging water (see Algorithm 1.1). Cells are initially sorted according to their water level. Calculation of the expected average level includes only cells with the water level lower than the expected average level. The amount of water that has to be exchanged between cells is stored in a temporary "flow" table. The level minimization algorithm has to be executed for each cell in the mesh. Finally, the "flow" table is used to update water levels in all cells.

```
float best_level(float lvl[]) {
    sort(lvl, 5);
    float sum += lvl[0];

    for (int i = 1; i < 5; i++ ) {
        if ( lvl[i] > sum / i ) {
            return sum / i;
        } else {
            sum += lvl[i];
        }
    }
    return sum/5;
}
```

Listing 1.1. Source code of **best_level(...)** function that calculates the new average level of water in neighbouring cells: the `lvl` variable contains water levels in the central cell and all its neighbours

The level minimization algorithm in version presented above was used in an implementation of the reference model that runs on a single CPU.

2.1 Migrating from CPU to GPU

The algorithm of water distribution presented in previous section cannot be just rewritten in one of the GPU programming environment. GPUs have a completely different architecture compared to general purpose processors. They resemble the SIMD (*Single Instructions, Multiple Data*) architecture that was defined in Flynn's taxonomy. The algorithm for such type of processing should be deprived of branch instructions that can significantly decrease efficiency.

The mesh of cellular automata is directly mapped on a mesh of GPU threads. Coordinates of a particular cell can be obtained by querying the thread:
`int x = get_global_id(0)`
`int y = get_global_id(1)`

Each cell is independently processed by a separated GPU thread. Threads are scheduled by OpenCL device i.e. graphics processor. Data structures (tables) that store states of automata are copied to the global memory of a graphics card (function *clCreateBuffer*) before processing. The level minimization function is implemented as two OpenCL kernels (functions that are executed directly by OpenCL device):

- `gpu_best_levels` — calculates the expected average level of water in neighbouring cells (see Listing 1.2),
- `gpu_distribute` — calculates the exchange of water between cells according to the previously calculated average level (see Listing 1.4).

Kernels are send to execution by a function *clEnqueueNDRangeKernel*. The number of threads processed in parallel depends on the number of cores available on a graphics processor.

```
void gpu_best_levels(float level[][],
                     float terrain[][],
                     float best_levels[][]) {
    int x = get_global_id(0);
    int y = get_global_id(1);

    float a[5];
    a[0] = terrain[x][y];
    a[1] = level[x][y-1];      a[2] = level[x-1][y];
    a[3] = level[x+1][y];      a[4] = level[x][y+1];
    best_levels[x][y] = best_level(level[x][y], a);
}

float best_level(float p, float a[]) {
    float sum = p - a[0];

    sort5(a);
    sum += a[0];

    float i = 1, a_if;
    a_if = step(a[1], sum);
    s += a_if * a[1];
    i += a_if;

    a_if = step(a[2] * 2, sum);
    sum += a_if * a[2];
    i += a_if;

    a_if = step(a[3] * 3, sum);
    sum += a_if * a[3];
    i += a_if;

    a_if = step(a[4] * 4, sum);
    sum += a_if * a[4];
    i += a_if;

    return sum/i;
}
```

Listing 1.2. Source code of **gpu_best_level(...)** kernel functions

Kernels are constructed without any branch instructions — they were replaced by built-in function: `step`, `max`. Also a short loop was replaced by a sequence of instructions.

Sorting function **sort5** is also deprived of any conditional instructions. We use static sorting network (see Listing 1.3) for a five elements table.

```
void sort5(float a[]) {
#define SWAP(i, j) {float t=min(a[i], a[j]);
                    a[j]=max(a[i], a[j]);
                    a[i]=t;}
    SWAP(0, 1);    SWAP(3, 4);    SWAP(2, 4);
    SWAP(2, 3);    SWAP(0, 3);    SWAP(1, 4);
    SWAP(0, 2);    SWAP(1, 3);    SWAP(1, 2);
#undef SWAP
}
```

Listing 1.3. Sorting function

```
void gpu_distribute(float level[][],
                    float terrain[][],
                    float sources[][],
                    float best_levels[][]) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    float a_level = level[x][y];
    float a_terrain = terrain[x][y];
    float change = sources[x][y];
    change += max(0, best_levels[x][y-1] - a_level);
    change += max(0, best_levels[x-1][y] - a_level);
    change += max(0, best_levels[x+1][y] - a_level);
    change += max(0, best_levels[x][y+1] - a_level);
    change += max(a_terrain, best_levels[x][y]) - a_level;
    level[x][y] = max(change + a_level, a_terrain);
}
```

Listing 1.4. Source code of **gpu_distribute(...)** kernel function

Our implementation uses only global memory. Data organization in cellular automata is simple and the memory coalescing for store and read operations is achieved automatically. Exceptions are the cells at the edges of areas assigned to a group of threads. The use of faster local memory requires changes in the code of kernels (handling cells at the edges), which might destroy profits resulting from their simplicity.

3 Results

The algorithm was tested for its efficiency on several graphics processors. We focused on Nvidia processors as they have much better support for Linux platform.

- Nvidia Quadro NVS 140/ Intel Core2 duo 2.1GHz
- Nvidia Quadro FX 4800/Intel i7
- Nvidia GeForce 8800 GT (G92)/ AMD Athlon XP2 4000+
- Nvidia GTX460/Fermi/Intel Pentium D 2.8GHz

Table 1. Technical parameters of graphics cards used in tests

GPU	No of cores	Cores timing	Memory timing
GTX460/Fermi	336	1.3 GHz	3.4 GHz (256-bit)
Quadro FX4800	192	1.5 GHz	0.8 GHz (384-bit)
GeForce 8800 GT	112	1.5 GHz	0.9 GHz (256-bit)
Quadro NVS 140M	16	0.8 GHz	0.6 GHz (64-bit)

The model is implemented in two versions: “normal” with visualisation and “benchmark” which concentrates on computational efficiency only. The visualisation is implemented using OpenGL graphics library. SFML (*Simple and Fast Multimedia Library*) library [16] manages simulation and user interface. Terrain maps are supplied as pictures in grayscale where levels of luminance are transformed to altitudes. Additional grayscale picture allows for defining sources and sinks. Figure 2 demonstrates sample results of simulation for two different terrain maps: A) the undulating terrain (”land of lakes“) and B) the part of river embankment.

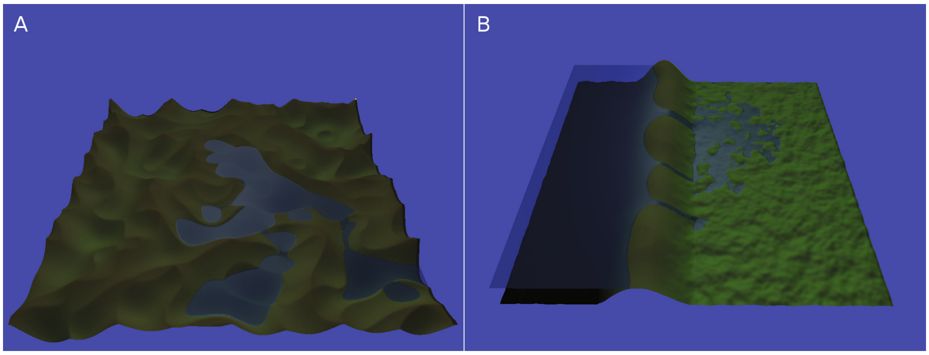


Fig. 2. Visualisation of sample results from simulations: A) ”land of lakes”, B) pouring water through the embankment

The ”Benchmark“ version of the model was used to perform efficiency tests. Figures 3 and 4 present results of the tests. Tests were run on various PC machines including laptop. There were very different combinations of graphics cards and CPUs, which is reflected in speedup charts. The most drastic example is a computer with a very old Pentium D processor combined with one of the most modern graphics cards Nvidia GTX 460 (Fermi core). The model that runs on

GPU is about 500 times faster than its equivalent for the CPU. The weakest GPU in these tests was Nvidia Quadro NVS 140 M. This is an old graphics card for laptops with small amount of relatively slow memory (GDDR3). However, even this card provides a bit better performance than one core of i7 — current Intel top processor.

Charts (Figs. 3 and 4) demonstrate that GPUs provide better efficiency for growing size of problems. Decrease of efficiency (number of simulation steps per second) for CPUs is almost linear. GPU implementation provides an execution of higher number of steps per second and their decrease is slower when the size of the problem grows. It is also worth to emphasize that for small lattices the GPUs are not fully utilized. This effect is observable for all GPUs except the weakest NVS 140M (see Fig. 4). For small lattices the performance remains almost at the same level.

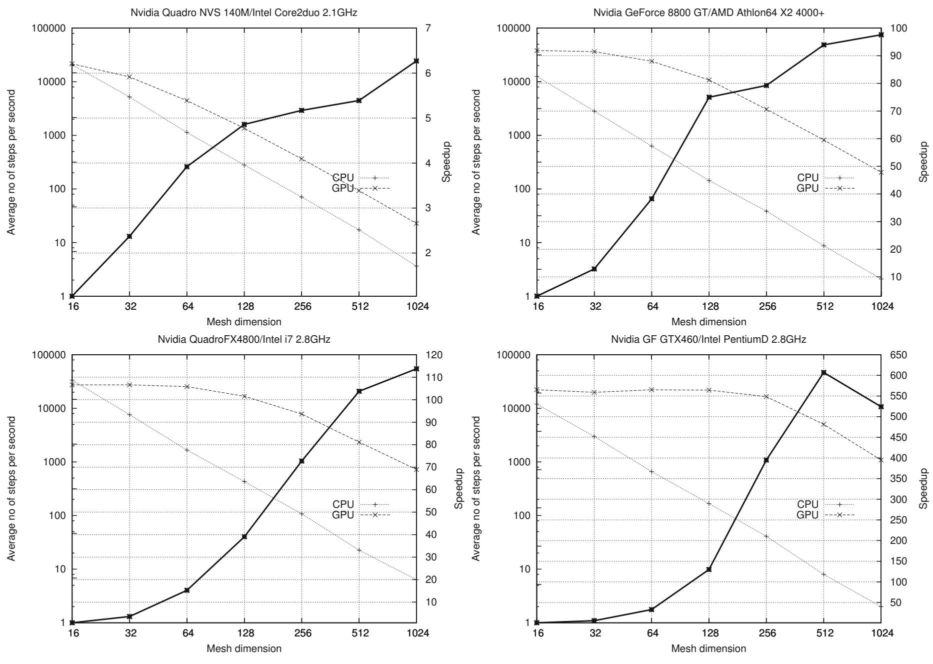


Fig. 3. Results of tests performed on various PC machines (identified by combination of CPU and GPU)

The implementation for CPU was written in the C99 language as a single threaded code and compiled using gcc 4.6 compiler with "-O3" option. No other optimizations have been applied.

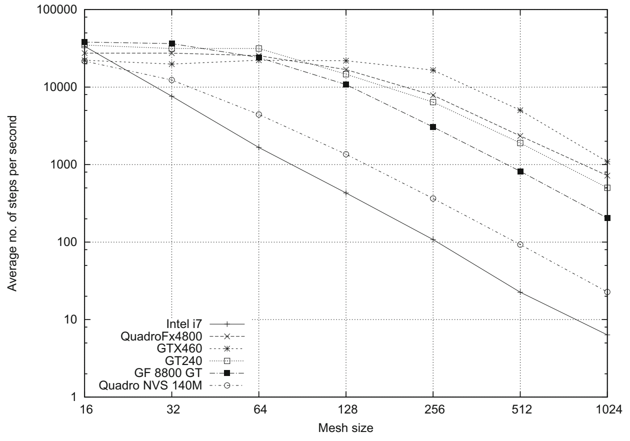


Fig. 4. Results of efficiency test for all cards compared to efficiency of Intel i7 processor

4 Conclusions

Our tests show that graphics processors can provide high efficiency for Cellular Automata models. The GPUs are very specialized processors designed for simultaneously processing huge amount of data with a single stream of instructions, as it is required in computer graphics. Ideal GPGPU applications have large data sets, high parallelism, and minimal dependency between data elements. Cellular Automata is a modelling paradigm that meets this requirements very well:

– modeled system is represented by an usually large regular mesh of cells,
– each cell should be processed simultaneously,
– changes in cells depend only on their very local neighbourhood.

GPGPU programming requires very deep understanding of how this calculations are preformed. The algorithms have to be carefully redesigned for the very specific architecture of these processors e.g. avoiding branching instructions. As a result we can obtain great improvements in efficiency as it is presented in this paper.

In this paper we concentrated only on algorithms issues. However, several preliminary tests performed with profiler showed that memory bandwidth is usually almost fully utilized. This suggests that the use of fast local memory may result even greater speedups. Our future works will focus on this issue.

We are aware that so significant supremacy of GPU implementation partially resulted from the fact that CPU code was not fully optimized. Careful optimization of that code could reduce the advantage of GPUs, but because of massive parallelism of Cellular Automata the similar performance could not be obtained.

Acknowledgements. This research is partially financed by the Polish Ministry of Higher Education and Science, project N519 579338 and partially by AGH grant No. 11.11.120.777.

References

1. Wolfram, S.: A New Kind of Science. Wolfram Media, Inc. (2002)
2. Chopard, B., Droz, M.: Cellular Automata Modeling of Physical Systems. Cambridge University Press (1998)
3. Gobron, S., Finck, D., Even, P., Kerautret, B.: Merging Cellular Automata for Simulating Surface Effects. In: El Yacoubi, S., Chopard, B., Bandini, S. (eds.) ACRI 2006. LNCS, vol. 4173, pp. 94–103. Springer, Heidelberg (2006)
4. Gobron, S., Coltekin, A., Bonafos, H., Thalmann, D.: GPGPU Computation and Visualization of Three-dimensional Cellular Automata. The Visual Computer 27(1), 67–81 (2011)
5. Gobron, S., Devillard, F., Heit, B.: Retina Simulation using Cellular Automata and GPU Programming. The Machine Vision and Applications Journal 18(6), 331–342 (2007)
6. Ferrando, N., Gosalvez, M.A., Cerda, J., Gadea, R., Sato, K.: Octree-based, GPU implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces. Computer Physics Communications 182(3), 628–640 (2011)
7. Caux, J., Siregar, P., Hill, D.: Accelerating 3D Cellular Automata Computation with GP-GPU in the Context of Integrative Biology. In: Salcido, A. (ed.) Cellular Automata - Innovative Modelling for Science and Engineering. InTech (2011) ISBN: 978-953-307-172-5
8. CUDA Zone, http://www.nvidia.com/object/cuda_home_new.html
9. Khronos Group, http://www.khronos.org/opencl/
10. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE 96(5), 879–899 (2008)
11. Di Gregorio, S., Serra, R.: An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata. Future Generation Computer Systems 16(2-3), 259–271 (1999)
12. Topa, P.: River Flows Modelled by Cellular Automata. In: Proceedings of 1st SGI Users Conference, Cracow, Poland, pp. 384–391. ACC Cyfronet UMM (October 2000)
13. Topa, P.: A Distributed Cellular Automata Simulations on Cluster of PCs. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J., Hoekstra, A.G. (eds.) ICCS-ComputSci 2002. LNCS, vol. 2329, pp. 783–792. Springer, Heidelberg (2002)
14. Topa, P., Paszkowski, M.: Anastomosing Transportation Networks. In: Wyrzykowski, R., Dongarra, J., Paprzycki, M., Wasniewski, J. (eds.) PPAM 2001. LNCS, vol. 2328, pp. 904–911. Springer, Heidelberg (2002)
15. Topa, P., Dzwinel, W., Yuen, D.: A multiscale cellular automata model for simulating complex transportation systems. International Journal of Modern Physics C 17(10), 1–23 (2006)
16. Simple and Fast Multimedia Library, http://www.sfml-dev.org/

A Multi-GPU Implementation of a D2Q37 Lattice Boltzmann Code

Luca Biferale[1], Filippo Mantovani[2], Marcello Pivanti[3], Fabio Pozzati[4], Mauro Sbragaglia[1], Andrea Scagliarini[5], Sebastiano Fabio Schifano[3], Federico Toschi[6], and Raffaele Tripiccione[3]

[1] University of Tor Vergata and INFN, Roma, Italy
[2] Deutsches Elektronen Synchrotron (DESY), Zeuthen, Germany
[3] University of Ferrara and INFN, Ferrara, Italy
[4] Fondazione Bruno Kessler Trento, Trento, Italy
[5] University of Barcelona, Barcelona, Spain
[6] Eindhoven University of Technology, The Netherlands and CNR-IAC, Rome, Italy

Abstract. We describe a parallel implementation of a compressible Lattice Boltzmann code on a multi-GPU cluster based on Nvidia *Fermi* processors. We analyze how to optimize the algorithm for GP-GPU architectures, describe the implementation choices that we have adopted and compare our performance results with an implementation optimized for latest generation multi-core CPUs. Our program runs at $\approx 30\%$ of the double-precision peak performance of one GPU and shows almost linear scaling when run on the multi-GPU cluster.

Keywords: Computational fluid-dynamics, Lattice Boltzmann methods, GP-GPUs computing.

1 Introduction

Computational techniques are ubiquitous today to compute reliable solutions to the highly non-linear equations of motion of fluids, in regimes interesting for physics or engineering. Over the years, many different numerical approaches have been proposed and implemented on virtually any computer architecture.

The Lattice Boltzmann (LB) method is a flexible approach, able to cope with many different fluid equations (e.g., multiphase, multicomponent and thermal fluids) and to consider complex geometries or boundary conditions. LB builds on the fact that the details of the interactions at microscopic level do not change the structure of the equations at the macroscopic level, but only modulate the values of their parameters; LB then relies on some simple synthetic dynamics of fictitious particle that evolve explicitly in time and, appropriately averaged, provide the correct values of the macroscopic quantities of the flow; see [1] for a complete introduction.

LB schemes are local (they do not require the computation of non local fields, such as pressure), so parallelization is in principle simple and efficient at all scales. In recent years, processing nodes have included more and more parallel features, such as many-core structures and/or vectorized data paths: the challenge now rests in combining effectively inter-node and intra-node parallelism.

In recent years, several methodologies for implementing efficient LB codes have been studied and developed for modern CPUs [2, 3]. Recently, GP-GPUs have drawn much attention to accelerate non-graphics applications, since the peak performance of one GPU system exceeds that of standard CPUs by roughly one order of magnitude; LB codes for GPUs have been recently considered in [4–6].

In this paper, we report on an efficient LB implementation on GP-GPUs that brings intra-node parallelism close to the highest level made possible by current technology. Our implementation builds on programming methodologies for GP-GPUs, like the data-layout and mapping of threads, studied, for example, in [4], but implements a more complex communication structure based on a D2Q37 model; our code is ready to run on a multi-GPU systems and enjoys high efficiency and scalability.

We have tested our codes on the *JUDGE* (JÜlich Dedicated Gpu Environment [8]) system at the Jülich Supercomputing Center (JSC). JUDGE is a cluster of 54 nodes, each with two multi-core processors (Intel Xeon X5650) and two Nvidia Tesla M2050 systems, that use the Fermi GPU. The Fermi GPU (see [7] for details) assembles 14 Symmetric Multiprocessors (SM), each containing a 32-way SIMD computing unit. In total 448 cores are available on Fermi. Each node has a peak double-precision performance of more that 1 Tflops.

This paper builds on previous work that addressed the same problem for massively parallel architectures based on multi-core processors, such as the IBM PowerXCell8i [9] or the Intel Xeon 5650 (Westmere) [10, 11].

2 Lattice Boltzmann Methods

In this work, we adopt a recently developed LB approach to the numerical solution of a class of Navier-Stokes equations in two dimensions, that correctly describes the behavior of a compressible gas, obeying the equation-of-state of an ideal gas ($p = \rho T$). The price to pay for this more accurate modelling is that this algorithm (see [12, 13] for details) is computationally more demanding and uses a more complex communication pattern than earlier approaches.

The model starts with a a thermal-kinetic description in the continuum of a compressible gas of variable density, ρ, local velocity $\boldsymbol{u}$, internal energy, $\mathcal{K}$ and subject to a local body force density, $\boldsymbol{g}$. The discretized counterpart of the continuum description (that we use in this paper) uses a set of fields $f_l(\boldsymbol{x}, t)$ associated to the so-called *populations*; the latter can be visualized as pseudo-particles moving in appropriate directions on a discrete mesh (see fig. 2); We use a set of velocities with 37 elements (a so-called D2Q37 model), significantly larger than earlier approaches. The master evolution equation in the discrete mesh is:

$$f_l(\boldsymbol{x} + \boldsymbol{c}_l \Delta t, t + \Delta t) - f_l(\boldsymbol{x}, t) = -\frac{\Delta t}{\tau} \left(f_l(\boldsymbol{x}, t) - f_l^{(eq)} \right) \tag{1}$$

where subscript l runs over the discrete set of velocities, $\boldsymbol{c}_l$ (see again fig. 2) and equilibrium is expressed in terms of hydrodynamical fields on the lattice, $f_l^{(eq)} = f_l^{(eq)}(\boldsymbol{x}, \rho, \bar{\boldsymbol{u}}, \bar{T})$. To first approximation, these fields are defined in terms of the LB populations: $\rho = \sum_l f_l$, $\rho \boldsymbol{u} = \sum_l \boldsymbol{c}_l f_l$, $D\rho T = \sum_l |\boldsymbol{c}_l - \boldsymbol{u}|^2 f_l$. When going into all mathematical details, one finds that shifts and renormalizations have to be applied to the averaged hydrodynamical quantities to correct for lattice discretization effects. It can be shown that with this approach one recovers the correct thermo-hydrodynamical equations of motion:

$$D_t \rho = -\rho \partial_i u_i^{(H)} \tag{2}$$

$$\rho D_t u_i^{(H)} = -\partial_i p - \rho g \delta_i + \nu \partial_{jj} u_i^{(H)} \tag{3}$$

$$\rho c_v D_t T^{(H)} + p \partial_i u_i^{(H)} = k \partial_{ii} T^{(H)}. \tag{4}$$

$D_t = \partial_t + u_j^{(H)} \partial_j$ is the material derivative and superscript H denotes lattice-corrected quantities; we neglect viscous dissipation in the heat equation (usually small), c_v is the specific heat at constant volume for an ideal gas and ν and k are the transport coefficients.

LB algorithms translate into very simple codes: we set $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{c}_l \Delta t$, and rewrite eq. 1 as

$$f_l(\boldsymbol{y}, t + \Delta t) = f_l(\boldsymbol{y} - \boldsymbol{c}_l \Delta t, t) - \frac{\Delta t}{\tau}\left(f_l(\boldsymbol{y} - \boldsymbol{c}_l \Delta t, t) - f_l^{(eq)}\right). \tag{5}$$

For each time step Δt, the algorithm sweeps the lattice and performs two tasks: a) it gathers population data from points in the lattice connected to $\boldsymbol{y}$ by a velocity vector $\boldsymbol{c}_l$; b) it evaluates the new value of all populations by performing the fully local computation of equation 5. Both the gather and computational steps have an available parallelism as large as the number of lattice sites; additional steps necessary in a real program, such as enforcing appropriate boundary conditions, do not modify this picture appreciably. In the following sections we discuss ways to exploit this parallelism on multi-GPU architectures.

3 Single GPU Implementation

In this section we consider a single-GPU implementation, and then describe, in a separate section, the implementation on a multi-GPU environment.

Following the programming model of CUDA [14], we associate one thread to the processing of one site of the lattice. Threads can be grouped in blocks, the latter being a batch of threads that cooperate, sharing data through a fast shared-memory and synchronizing their execution to coordinate memory accesses. We often use the possibility to define dynamically the size of the blocks, and the CUDA concept of *stream*, that identifies independent kernels that run concurrently on the GPU, performing different tasks (e.g., computation and communication). For the CPU-resident part of the code, we adopt the C language, but we carefully manage all details of parallelism using the standard POSIX

```
typedef struct {
  double p1 [NSITES]; // population  1 array
  double p2 [NSITES]; // population  2 array
  ...
  double p37[NSITES]; // population 37 array
} pop_type;
```

Fig. 1. Main data structures for population data, as seen by the GPU. Lattice data is stored in memory as a structure of arrays, to better exploit memory coalescing. Each member is an array, one for each population used by our model. NSITES is the number of lattice sites.

Linux *pthread* library (not to be confused with the thread concept of CUDA), while MPI supports communication between the nodes.

Each grid point has a set of 37 double-precision floating point values representing the *population*s of the D2Q37 LB code. We implement this data-structure as a structure of arrays as shown in figure 1. Each CUDA thread processes one site; threads processing adjacent sites address the same population-array at the same time. This makes it easy for the hardware to *coalesce* memory accesses by the threads into just one memory transaction. There are two copies of each population-array on GPU memory. Each step of the algorithm reads data from one copy and write results to the other. Each population-array includes a left and a right frame, containing a copy of (respectively) the three rightmost and leftmost columns of the lattice; this structure helps handle periodic boundary conditions (which we adopt in the x-direction).

At each time step, the host executes a loop that includes the following four computational phases:

– comm(): this step copies the left and right borders of the lattice grids onto the appropriate frames;
– move(): for each site, this phase gathers the population elements that will collide at the next computational phase (collide()), according to the scheme of figure 2. This implies accessing sparse blocks of memory locations, corresponding to populations of neighbor-cells at distance 1, 2 and 3 in the physical grid. This step is usually called stream, but we name it move to avoid confusion with *CUDA streams* that will be used later;
– bc(): this phase adjusts the values of the populations at the top and bottom edges of the lattice to enforce boundary conditions (e.g., a constant given temperature and zero velocity);
– collide(): this phase performs all the mathematical steps needed to compute the new population values at each grid site (called *collision* in LB jargon). Input data are the populations gathered by the move() phase. This is the floating point intensive kernel of the code; it uses only data available of the site on which it operates, making the processing of different sites fully uncorrelated.

As usual with CUDA, a computation is structured as a two-dimensional grid of blocks, and each block is a three-dimensional grid of threads. In our case all

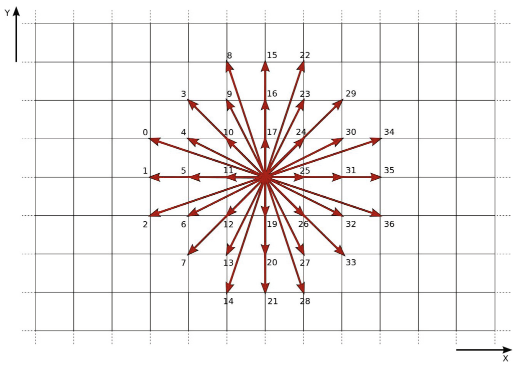


Fig. 2. Visualization of the move() phase. Populations from grid points at the edges of the arrows are gathered at the grid point at center, at a distance of 1, 2, 3 grid points.

threads execute the same code, processing a different lattice-site. Each phase corresponds to a separate CUDA kernel, as follows:

– comm is a copy operation. On a single GPU, a CUDA *memory-copy* function invoked by the host moves data from locations corresponding to the right border to locations corresponding to the left border and vice-versa. The device-to-device flag makes this operation an intra-GPU copy of data with no host involvement, and substantially increases performance. On a multi-GPU implementation this step implies network communications (see later);
– for move and collide operating on a lattice of NX $\times$ NY points, the layout of each block is (1 $\times$ N_THREAD $\times$ 1) and the grid of blocks is (NY/N_THREAD, NX) (N_THREAD is the number of thread per block), see figure 3 left. We have profiled the execution time of collide for several values of N_THREAD; no performance increase is found as N_THREAD ≥ 128.
– for bc, the layout of each block is changed to (NX $\times$ 1 $\times$ 1) and the grid of blocks is (1 $\times$ NY), see figure 3 right. bc runs only on the threads corresponding to lattice-sites with coordinate $y = 0, 1, 2$ and $y = NY - 1$, $NY - 2, NY - 3$.

A first version of the code (that we call V1), uses two separate kernels to perform the move and collide steps. We can further improve our implementation, noticing that we move data from/to memory two times, first during move, then during collide; however, move and collide can be merged in just one step applied to all cells of the grid (for more details see [15]). To do this we have to take into account that we must enforce boundary conditions (bc) after move but before collide, so we must structure the computation in a slightly different way (we call the resulting code version V2). Also in this case the host performs a loop over time; at each iteration, it launches GPU kernels performing the following steps:

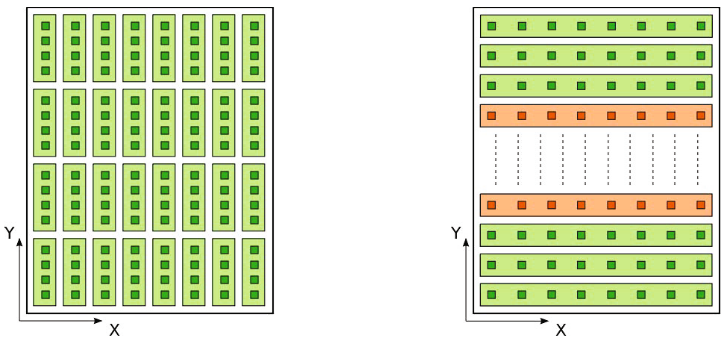


Fig. 3. (Left): Configuration used by the move() and collide() kernels, on a physical lattice of 8 × 16 points; the block-grid is 8 × 4, and the thread-grid is 1 × 4 × 1 (N_THREAD = 4). (Right): Configuration used by the bc() kernel for the same lattice; the block-grid is 1 × 16, and the thread-grid is 8 × 1 × 1 (N_THREAD = 8).

step 1: exchange border frames;
step 2: execute move over the three topmost and lowermost rows of the grid;
step 3: adjust boundary conditions on the cells at the three top and bottom rows of the grid; then run collide for those cells;
step 4: execute a kernel that jointly computes move and collide for all cells in the lattice bulk.

4 Multi-GPU Implementation

We now describe the parallelization of the code for a multi-GPU environment.

We split a lattice of size $L_x \times L_y$ on N_p GPUs along the X dimension. Each GPU allocates a *sub-lattice* of size $\frac{L_x}{N_p} \times L_y$. One could consider a different decomposition (e.g. $\frac{L_y}{N_p} \times L_x$ to reduce communication requirements if $L_y \geq L_x$); however, since we plan to use our code for physics simulations in a wide range of aspect-ratios (both $L_x > L_y$ and $L_x < L_y$) and the communication overhead is small (see later for details) we arbitrarily pick up only one of the two possibilities. This allocation scheme implies a virtual ordering of the GPUs along a ring, so each GPU is connected with a previous and a next one. Since each node of the cluster hosts two GPUs, each GPU has a neighbor installed on the same node while the other neighbor sits on a different node. GPUs must exchange data at the beginning of each time-step, before starting the move() phase, as cells close to the right and left edges of the sub-grid of each node need data allocated on the logically previous and next GPUs. We first copy the three left-most and right-most y columns of each sub-lattice from both GPUs to buffers on the host, using CUDA *memory-copy* instructions. We then perform the required buffer exchanges inside the node, and finally move buffered data from/to adjacent nodes. When this step completes, we copy the fresh buffers back into the GPUs and the

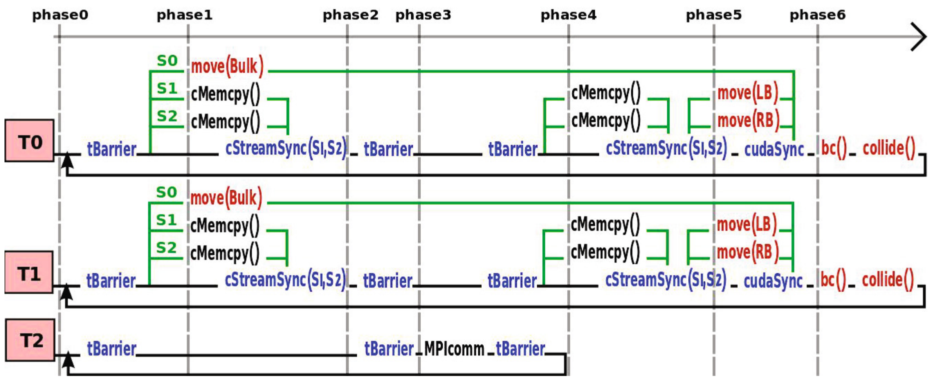


Fig. 4. Flow diagram of the code executed by each node. Threads T0 and T1 manage the GPUs, while thread T2 manages the exchange of border data and communication on the Infiniband network.

program continues with move(), bc() and collide(). This communication pattern uses standard MPI send and receive operations.

To improve performance, reduce the impact of communications and obtain better scaling performance, we have re-scheduled all tasks in order to overlap communication and processing. We use the CUDA concept of stream that allows the execution of concurrent kernels, so we exploit parallelism at various levels organizing the program on each node as a *multi-threaded* and *multi-streamed* code. We have re-scheduled the code as following. The node executes a multi-threaded code running three threads: T0, T1 and T2. Threads T0, T1 manage respectively GPU0 and GPU1, while T2 takes care of the operations associated to the comm() phase. To avoid conflicts in accessing GPU devices and memory, we have controlled the allocation of memory and threads by using the standard Linux NUMA library, and the lattice allocated on each node is split in two sub-lattices, each allocated on a separate memory bank. Threads T0 and T1 run respectively on CPU0 and CPU1 and execute the same program, while allocation of thread T2 is irrelevant since it has to perform copy operations from one device to the other, or node-to-node communications via MPI primitives. This allocation avoids or reduces memory and device access conflicts due to a thread running on a CPU and accessing memory or device physically attached to the other CPU [11]. The three threads run in parallel, and are synchronized by barriers shown in figure 4 as tBarrier(). The execution goes through the following phases for each time step, as shown in figure 4:

Phase 1: after an initial synchronization (phase0), threads T0 and T1 launch three CUDA streams which run on the GPU concurrently: S0, S1 and S2. S0 executes the move(Bulk) kernel over the bulk of the sub-lattice; streams S1 and S2 execute in parallel cMemcpy(), a memory copy of the borders from device to host;

Table 1. Performance comparison for the GPU and CPU codes, versions V1 and V2. Runs have been performed on a system of 252 × 16000 lattice points. We show performance in GFLOPs and as a fraction of peak (R_{max}).

	GPU code V1	CPU code V1
comm	0.20 ms	10.00 ms
stream	47.85 ms	140.00 ms
bc	0.60 ms	0.20 ms
collide	194.69 ms	360.00 ms
GFLOps	129.23	60.17
Rmax	25%	38%

	GPU code V2	CPU code V2
STEP 1	0.19 ms	7.00 ms
STEP 2	1.18 ms	0.64 ms
STEP 3	0.99 ms	0.62 ms
STEP 4	193.45 ms	410.00 ms
GFLOps	160.59	72.41
Rmax	31%	45%

Phase 2: threads T0 and T1 wait the end of streams S1 and S2 performing a CUDA StreamSync() call, and then make a synchronization with thread T2;

Phase 3: thread T2 exchanges border data with neighbor nodes (call MPI-comm). This proceeds in parallel with the move(Bulk) kernels run by streams S0;

Phase 4: as T2 has received data from neighbors nodes, it makes a synchronization with T0 and T1; the latter launches two streams (cMemcpy()) to copy data from host to GPUs;

Phase 5: as the copy operation ends, T0 and T1 runs kernels to apply the move() operation to the just updated borders;

Phase 6: T0 and T1 run sequentially kernels to compute bc() and collide().

5 Performance Results and Conclusions

To first approximation, performance results are either limited by the available memory bandwidth or computation throughput. A rough estimation of the computing time T of our code is:

$$T \geq \max(W/F,\ D/B) \tag{6}$$

where $W \approx 7800$ double-precision floating-point operations is the computation workload for each lattice point, F is the peak performance of the CPU, $D = 37 \times 2 \times 8 = 500$ is the amount of data in bytes exchanged with memory, and B is the peak memory bandwidth. Remembering that $F \approx 500$ Gflops and $B \approx 144$ Gbyte/sec for the GPU that we use, we obtain:

$$T \geq \max\left(\frac{7800}{500}, \frac{592}{144}\right) ns = \max(15.6, 4.1)\, ns. \tag{7}$$

In other words, if peak values for CPU performances and memory bandwidth apply, our code is strongly compute-bound, so one can hope to reach high efficiency. In presenting our performance result, we compare our GPU-based implementation with a CPU-code optimized for a commodity system based on two Intel six-core (*Westmere*) CPUs. Also for this system, equation 6 tells us that

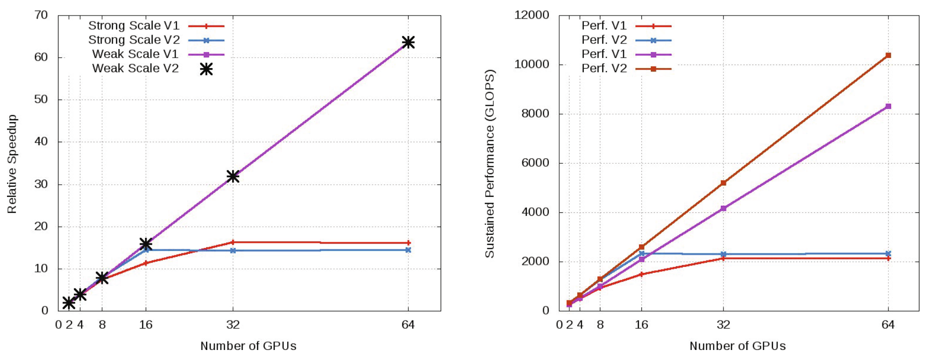


Fig. 5. Relative speedups in strong and weak regime (left), and sustained performance (right) for code versions V1 and V2, as function of the number of GPUs. Measurements in the strong regime have been done running the codes on a lattice size $L_x \times L_y = 1024 \times 7168$, while in the weak regime a sub-lattice of size $L_x \times L_y = 254 \times 14464$ have been allocated on each GPU.

the code is theoretically compute-bound. Our CPU implementation, described in detail in [10, 11], has been optimized at various levels exploiting core parallelism, vectorization and cache data re-use. Table 1 shows the results of this comparison. Figure 5 shows relative speedup and sustained performances (in GFLOPs) of the multi-GPU implementation as function of the number of the GPUs. Some final remarks are in order:

– A simple theoretical analysis shows that our code is strongly compute-bound, and could reach high efficiency. However, even if sustained performance is good from the point of view of physics application, the value measured is $\approx 25-30\%$ of peak, significantly lower than one would expect from equation 7. This is partially due to the fact that the the floating-point computation of the collide phase can be only partially mapped on fused-multiply-add instructions; it also uses many constants which have to be loaded from memory, causing stalls in the CPU pipeline. Moreover, the CPU registers are not enough to hold all the intermediate values of the computation, causing register-spilling, and introducing overheads. We are currently working to better characterize and reduce these overheads;
– The single GPU-code performs roughly a factor 2 better than the optimized code on multi-core CPUs. This result is due to the bad impact of overheads highlighted in the previous point;
– Even if efficiency on GPUs (as a fraction of peak) is lower than for multi-core CPUs, sustained performances are still remarkably high for a physics production-ready code;
– Fine tuning the CPU-program has required accurate programming efforts (see [10, 11]), while on GP-GPUs, the coders are forced to to (re-)write programs using the CUDA paradigm, which naturally exploits data-parallelism and performances of the GPUs;

- On the other hand, performance can be dramatically affected by communication overheads between the CPUs and GPUs. At least in this case, this problem can be swept under the carpet, but a non trivial subdivision in computing threads and an accurate schedule is necessary;
- Scalability of our code is good in the strong-regime as the size of the local lattice is large enough to hide communication and memory-copy overheads; in the weak-regime the scaling is linear as the communication time has been hidden with computation of move; same results apply to the CPU-code.

Performance improvements should be expected if communications are performed directly between GPUs with reduced involvement of the host, and workloads are balanced between GPUs and CPUs. Future works in this direction will explore these features. e.g. using Nvidia GPUDirect capabilities, and heterogenous programming.

Acknowledgements. We would like to thank the Jülich Supercomputing Center (JSC) for providing access to the JUDGE system; we acknowledge the precious support of W. Homberg and J. Kreutz during program development, debugging and test.

References

1. Succi, S.: The Lattice Boltzmann Equation for Fluid Dynamics and Beyond. Oxford University Press (2001)
2. Wellein, G., Zeiser, T., Hager, G., Donath, S.: On the Single Processor Performance of Simple Lattice Boltzmann Kernels. Computers & Fluids 35, 910–919 (2006)
3. Axner, L., et al.: Performance evaluation of a parallel sparse lattice Boltzmann solver. Journal of Computational Physics 227(10), 4895–4911 (2008)
4. Tölke, J.: Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. Comp. and Vis. in Science (2008)
5. Tölke, J., Krafczyk, M.: TeraFLOP computing on a desktop PC with GPUs for 3D CFD. Journal of Computational Fluid Dynamics 22(7), 443–456 (2008)
6. Habich, J., Zeiser, T., Hager, G., Wellein, G.: Speeding up a Lattice Boltzmann Kernel on nVIDIA GPUs. In: Proc. of PARENG09-S01, Pecs, Hungary (April 2009)
7. http://www.nvidia.com/object/fermi_architecture.html
8. http://www2.fz-juelich.de/jsc/judge
9. Biferale, L., et al.: Lattice Boltzmann fluid-dynamics on the QPACE supercomputer. In: ICCS Proc. 2010, Procedia Computer Science, vol. 1, pp. 1075–1082 (2010)
10. Biferale, L., et al.: Lattice Boltzmann Method Simulations on Massively Parallel Multi-core Architectures. In: HPC 2011 Proc. (2011)
11. Biferale, L., et al.: Optimization of Multi-Phase Compressible Lattice Boltzmann Codes on Massively Parallel Multi-Core Systems. In: ICCS 2011 Proc. 2011. Procedia Computer Science, vol. 4, pp. 994–1003 (2011)

12. Sbragaglia, M., et al.: Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria. J. Fluid Mech. 628, 299 (2009)
13. Scagliarini, A., et al.: Lattice Boltzmann Methods for thermal flows: continuum limit and applications to compressible Rayleigh-Taylor systems. Phys. Fluids 22, 055101 (2010)
14. NVIDIA, NVIDIA CUDA C Programming Guide
15. Pohl, T., et al.: Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes. Parallel Processing Letters 13(4), 549–560 (2003)

Combining Smoother and Residual Calculation in v-cycle AMG for Symmetric Problems

Maximilian Emans

Johann Radon Institute for Computational and Applied Mathematics (RICAM), and Industrial Mathematics Competence Center GmbH (IMCC), both 4040 Linz, Austria
maximilian.emans@ricam.oeaw.ac.at

Abstract. We examine a modified implementation of the v-cycle multigrid scheme which takes into account the memory traffic, i.e. the movement of data between the memory and the processing units. It is known that the relatively slow data transfer is responsible for the poor parallel performance of multigrid algorithms on certain shared memory architectures e.g. those with a front-side bus memory controller. The modification is simple but it speeds up computations by up to 15%.

1 Introduction

Multigrid algorithms have reached a particular maturity in the field of solvers for sparse linear systems. Such problems are often the core task of PDE (partial differential equation) solvers. In this case, AMG (algebraic multigrid) techniques are attractive since they exclusively require the information stored in the system matrix; due to this they have the potential to be used as a "black-box", whereas geometric multigrid methods always require further information that depends on the discretisation technique.

The memory access pattern, i.e. the order in which certain data in the memory is accessed for reading and writing, is crucial for the performance of any numerical algorithm, multigrid being no exception. The reason is that the rate at which data can be transferred from the main memory to the processing core is limited by the memory bandwidth; in many situations it is slower than the rate at which the data can be processed. The memory access therefore tends to limit the rate at which the processing unit can practically perform the arithmetic operations.

Emans and van der Meer [1] have reported the consequences of this on the performance of AMG algorithms running on modern cluster hardware. They found that the problem is particularly severe on shared memory machines where the memory access of the individual processes is slow since all processes require the data transfer at the same time. In this paper we examine the improvement through a combined execution of a smoothing step and the residual calculation that is applicable for symmetric problems; this idea has been mentioned among some other suggestions to enhance AMG implementations by Haase and Reitzinger [2], and a similar idea can be found in the paper by Douglas et al. [3].

However, the impact of this particular modification on the performance of AMG solvers has not been studied properly yet. The fact that still in many AMG implementations the conventional, modular approach seems to be preferred is the major motivation for this contribution.

2 AMG and Its Implementation

Suppose we want to use AMG to iteratively approach a solution of the system $A\boldsymbol{x} = \boldsymbol{b}$ where $A \in \mathbb{R}^{n \times n}$ is symmetric, regular, and sparse, $\boldsymbol{b} \in \mathbb{R}^n$ is some right-hand side vector and $\boldsymbol{x} \in \mathbb{R}^n$ the solution with n being the rank of A and thus the number of unknowns. Algorithm 1 represents a v-cycle which is one of the most commonly used cycling strategies. As with any multigrid algorithm, it requires the definition of a grid hierarchy with l_{max} levels where $A_l \in \mathbb{R}^{n_l \times n_l}$ $(l = 1, ...l_{max})$ is the system matrix on level l $(A_1 = A)$ with a system size n_l; $n_{l+1} < n_l$ holds for $l = 1, ..., l_{max}-1$. Moreover, a smoothers S_l and prolongation and restriction operators $P_l \in \mathbb{R}^{n_l \times n_{l+1}}$ and $R_l \in \mathbb{R}^{n_{l+1} \times n_l}$ for each level l need to be determined. As it is common practice in algebraic multigrid we choose $R_l = P_l^T$ $(l = 1, ..., l_{max} - 1)$ and define the coarse-grid hierarchy recursively according to Galerkin.

The matrices A_l are stored in the compressed row storage (CRS) format. This format is common in many linear solver packages, such as in hypre [4] and the algebraic multigrid package of Trilinos, ML [5]. Using this format, the matrix elements are stored row-wisely in a coherent vector; this format is favourable if several elements of one matrix row are accessed immediately after each other since on modern chip architectures with caches tiles, i.e. small coherent parts of the memory (instead of single floating-point numbers) are loaded at once to the cache. Once a single element of a row is in the cache, one can assume that, provided the CRS format is used, the following elements of this row have already been loaded to the cache and can be accessed very quickly. Since transferring the data from the main memory to the cache is limited by the memory bandwidth, using the elements of the same row more than once (while the same operations are done) can have a beneficial effect on the run time.

The smoother might be any algorithm that reduces high frequency error components on the respective grid level l. In most AMG implementations a Gauß-Seidel smoother, an ILU(0) smoother or a Jacobi smoother is employed. With regard to memory access of the matrix elements, these algorithms are similar; in the following we restrict ourselves to the Gauß-Seidel smoother.

In many multigrid algorithms, the application of the smoother and the computation of the residual contribute significantly to the total computing time (in serial as well as in parallel computations). Pseudocodes of an efficient parallel implementations of the Gauß-Seidel smoother and the computation of the residual are shown in algorithms 2 and 3. The following notation is used: p is the number of parallel processes, n_d is the total number of unknowns assigned to process d where $\sum_{d=1}^{p} n_d = n$, m_d number of internal unknowns without link to unknowns assigned to another process, $A_d \in \mathbb{R}^{n_d \times n_d}$ is the matrix that reflects the mutual influences between the unknowns assigned to process d; for

Algorithm 1. v-cycle AMG

```
    $\boldsymbol{x}_l^{(3)}$ = v-cycle($l$, $\boldsymbol{b}_l$, $\boldsymbol{x}_l^{(0)}$)
    **Input:** level $l$, right-hand side $\boldsymbol{b}_l$, initial guess $\boldsymbol{x}_l^{(0)}$
    **Output:** approximate solution $\boldsymbol{x}_l^{(3)}$
 1: pre-smoothing: $\boldsymbol{x}_l^{(1)} = S_l(\boldsymbol{b}_l, A_l, \boldsymbol{x}_l^{(0)})$
 2: compute residual: $\boldsymbol{r}_l = \boldsymbol{b}_l - A_l\boldsymbol{x}_l^{(1)}$
 3: restriction: $\boldsymbol{r}_{l+1} = P_l^T\boldsymbol{r}_l$
 4: if $l+1 = l_{max}$ then
 5:     direct solution of coarse-grid system: $\boldsymbol{x}_{l+1} = A_{l+1}^{-1}\boldsymbol{r}_{l+1}$
 6: else
 7:     recursive solution of coarse-grid system: $\boldsymbol{x}_{l+1}$ = v-cycle($l+1$, $\boldsymbol{r}_{l+1}$, $\boldsymbol{0}$)
 8: end if
 9: prolongation of coarse-grid solution and update: $\boldsymbol{x}_l^{(2)} = \boldsymbol{x}_l^{(1)} + P_l\boldsymbol{x}_{l+1}$
10: post-smoothing: $\boldsymbol{x}_l^{(3)} = S_l(\boldsymbol{b}_l, A_l, \boldsymbol{x}_l^{(2)})$
```

the elements of A_d, a_{ij}, the index d is dropped for the sake of simplicity. For parallel implementations it is also necessary to consider the influence of external unknowns (i.e. unknowns assigned to another process) onto unknowns assigned to the current process. This influence is reflected by $\tilde{A}_d \in \mathbb{R}^{n_d \times \tilde{n}_d}$ with elements $\tilde{a}_{ij}$; $\tilde{n}_d$ is the number of external unknowns that influence the calculation of process d. Note that $\tilde{a}_{ij} = 0$ for $i \leq m_d$ since the unknowns with index $i < m_d$ are not connected to unknowns assigned to another process. The notation is illustrated in figure 1. The sparsity pattern the matrices A_d and $\tilde{A}_d$ is expressed by the index sets $J_i = \{j|a_{ij} \neq 0\}$ and $\tilde{J}_i = \{j|\tilde{a}_{ij} \neq 0\}$, respectively, where again the index pointing to the process is skipped for simplicity.

The implementations that are expressed in algorithms 2 and 3 rely on an asynchronous data exchange mechanism. These implementations have the advantage that most of the computational work (the treatment of the internal unknowns, loop 3-5 in algorithm 2 and loop 2-4 in algorithm 3) and the actual data transfer can be done simultaneously so that the parallel overhead can be efficiently hidden as long as the internal tasks are sufficiently large.

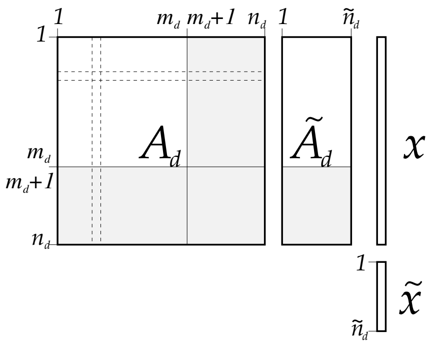


Fig. 1. Notation of parallelised matrix storage

Algorithm 2. parallel Gauß-Seidel smoother

```
    $\boldsymbol{x}_l^{(K)} = \mathrm{GS}(K, \boldsymbol{b}, \boldsymbol{x}^{(0)})$
    **Input:** number of sweeps $K$, right-hand side $\boldsymbol{b}$, initial value $\boldsymbol{x}^{(0)}$
    **Output:** smoothed vector $\boldsymbol{x}^{(K)}$
 1: for $k = 1$ to $k = K$ do
 2:   initialise and start exchange of $\boldsymbol{x}^{(k-1)}$
 3:   for $i = 1$ to $m_d$ do
 4:     update internal values $x_i^{(k)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \in J_i, j<i} a_{ij} x_j^{(k)} - \sum_{j \in J_i, j>i} a_{ij} x_j^{(k-1)} \right)$
 5:   end for
 6:   wait until $\tilde{x}_i^{(k-1)}$ for $i = 1, ..., \tilde{n}_d$ are obtained and terminate exchange
 7:   for $i = m_d + 1$ to $n_d$ do
 8:     $x_i^{(k)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \in J_i, j<i} a_{ij} x_j^{(k)} - \sum_{j \in J_i, j>i} a_{ij} x_j^{(k-1)} - \sum_{j \in \tilde{J}_i} \tilde{a}_{ij} \tilde{x}_j^{(k-1)} \right)$
 9:   end for
10: end for
```

Algorithm [1] suggests already how the v-cycle program can be organised: The application of the smoother and the matrix-vector product or the entire computation of the residual, but also that of the prolongation and the restriction operators, are implemented as separate program units, e.g. as functions or subroutines. This organisation of the program is found in most publicly available multigrid implementations such as hypre [4] and the algebraic multigrid package of Trilinos, ML [5]. The advantage is that it results in a clear and modular structure of the program.

Algorithm 3. parallel calculation of residual

```
    $\boldsymbol{r} = \mathrm{RS}(\boldsymbol{b}, \boldsymbol{x})$
    **Input:** right-hand side $\boldsymbol{b}$, current solution $\boldsymbol{x}$
    **Output:** residual $\boldsymbol{r}$
1: initialise and start exchange of $\boldsymbol{x}$.
2: for $i = 1$ to $m_d$ do
3:   compute internal values $r_i = b_i - \sum_{j \in J_i} a_{ij} x_j$
4: end for
5: wait until $x_i$ for $i = m_d + 1, ..., n_d$ are obtained and terminate exchange.
6: for $i = m_d + 1$ to $n_d$ do
7:   $r_i^{(k)} = b_i - \sum_{j \in J_i} a_{ij} x_j - \sum_{j \in \tilde{J}_i} \tilde{a}_{ij} \tilde{x}_j$
8: end for
```

2.1 Simultaneous Application of Smoother and Matrix

It has been observed that the data transfer on shared-memory architectures is a limiting factor for poor performance of parallel computations. The largest amount of data that needs to be transferred are the elements of the matrix; they need to be accessed in every sweep of the smoother and in the matrix-vector multiplication for the computation of the residual. Emans and van der Meer [1] have reduced the data transfer by replacing appropriate parts of an 8-byte arithmetic by a 4-byte implementation and report acceleration in the range of 25%. Note that this kind of data transfer is not relevant in the case of geometric multigrid since the implementation of matrix-related operations in this kind of algorithm is typically based on stencils and the matrix elements are not stored explicitly. Therefore our idea is restricted to algebraic multigrid.

A similar accelerating effect can be expected if the smoothing sweep and an immediately following computation of the residual (lines 1 and 2 of algorithm 1) can be implemented in a way that the matrix information needs to be loaded only once to the cache. Note that this loading is not relevant in the case of geometric multigrid since the implementation of matrix-related operations in geometric multigrid is typically based on stencils and the matrix elements are not stored explicitly. Therefore this idea is restricted to algebraic multigrid. Algorithm 4 shows the pseudocode for symmetric matrices where the smoothing and the residual calculation are combined into one program unit. Due to the symmetry of A we use a_{ij} instead of a_{ji} for the computation of the j-th component of $\boldsymbol{r}$; this way both accesses of each matrix element (one for the smoother and one for the residual calculation, see line 7 and 10 of algorithm 4, respectively) take place immediately one after the other such that each matrix row needs to be moved only once from the memory to the cache. The rest of the memory access pattern of this combined implementation of smoother and residual computation is not worse than that of the conventional implementation: In both algorithms, two random accesses to vectors occur: the unknown vector is randomly accessed — once for the update in line 4 of algorithm 2 and once again for the calculation of the residual in line 3 of algorithm 3. In algorithm 4, the unknown vector in line 7 and the residual vector in line 10 are accessed randomly.

In a parallel computation, however, it is possible that the parallel overhead is slightly larger: While in the conventional implementation the exchange of the unknowns at the boundaries and the treatment of the internal unknowns take place simultaneously, in the combined implementation of smoother and residual computation the time for the exchange of the smoothed solution can only be used to treat the internal part of the rows with links to external unknowns (lines 20-22 in algorithm 4).

3 Benchmark

In this section we describe a benchmark on different shared memory machines. The computers used are desktop workstations equipped with different Xeon processors by the manufacturer Intel. Table 1 compiles some information on the hardware. The most important differences between these processors are the number of cores, the organisation and the size of the cache (highest level in the cache hierarchy) as well as the type of memory bus. The processors X5470, X5365, and X5160 use a classical UMA architecture with a FSB (front-side bus) memory controller while X5670 has the QPI (QuickPath Interconnect), based on a NUMA

Algorithm 4. parallel Gauß-Seidel smoother with residual computation

```
$(\boldsymbol{x}_l^{(K)}, \boldsymbol{r}) = \text{GSRS}(K, \boldsymbol{b}, \boldsymbol{x}^{(0)})$
**Input:** number of sweeps $K$, right-hand side $\boldsymbol{b}$, initial value $\boldsymbol{x}^{(0)}$
**Output:** smoothed vector $\boldsymbol{x}^{(K)}$, residual $\boldsymbol{r}$
1: for $k = 1$ to $k = K$ do
2:    if $k = K$ then
3:       initialise residual: $\boldsymbol{r} = \boldsymbol{b}$;
4:    end if
5:    initialise and start exchange of $\boldsymbol{x}^{(k-1)}$;
6:    for $i = 1$ to $m_d$ do
7:       update internal values: $x_i^{(k)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \in J_i, j<i} a_{ij} x_j^{(k)} - \sum_{j \in J_i, j>i} a_{ij} x_j^{(k-1)} \right)$;
8:       if $k = K$ then
9:          for $j \in J_i$ do
10:            accumulate: $r_j \leftarrow r_j - a_{ij} x_i^{(k)}$;
11:         end for
12:      end if
13:   end for
14:   wait until $\tilde{x}_i^{(k-1)}$ for $i = 1, ..., \tilde{n}_d$ are obtained and terminate exchange;
15:   for $i = m_d + 1$ to $n_d$ do
16:      update boundary values:
         $x_i^{(k)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \in J_i, j<i} a_{ij} x_j^{(k)} - \sum_{j \in J_i, j>i} a_{ij} x_j^{(k-1)} - \sum_{j \in \tilde{J}_i} \tilde{a}_{ij} \tilde{x}_j^{(k-1)} \right)$;
17:   end for
18: end for
19: initialise and start exchange of $\boldsymbol{x}^{(K)}$;
20: for $i = m_d + 1$ to $n_d$ do
21:   for $j \in J_i$ do
22:      accumulate: $r_j \leftarrow r_j - a_{ij} x_i^{(K)}$;
23:   end for
24: end for
25: wait until $\tilde{x}_i^{(K)}$ for $i = 1, ..., \tilde{n}_d$ are obtained and terminate exchange;
26: for $i = m_d + 1$ to $n_d$ do
27:   accumulate: $r_j \leftarrow r_j + \sum_{j \in \tilde{J}} \tilde{a}_{ij} \tilde{x}_i^{(K)}$;
28: end for
```

memory layout. The new QPI is not only significantly faster, but also permits that the L3-cache is shared between all cores whereas only two nodes of the four nodes on the quad-cores processors share a common L2-cache. The transfer rate of the FSB systems would correspond theoretically to a transfer rate of 1.67 GT/s; since the specifications of the manufacturer refer to theoretical peak performance and the effective bandwidth does not only depend on the bus system, we added the measured bandwidth to the data in table 1.

Our benchmark is a symmetric linear system that has to be solved in a flow simulation of an engine calculation. It reflects the second order accurate finite volume discretisation of a pressure-correction equation in a SIMPLE scheme used to predict the unsteady compressible motion of the air in a cylinder of a combustion engine. The system is symmetric and positive definite; moreover, since each finite volume has six or less neighbours, the maximum number of elements per row is seven. For more information on the nature of this system we refer to Emans [7]. We would like to emphasis that in this contribution we are only interested in speeding up a particular part of the calculation; since we do not interfere with the numerical algorithm, the results are not changed by the suggested modification. Therefore, apart from the non-zero structure and the symmetry, the numerical properties of the matrix are irrelevant for this paper.

The parallel decomposition of the problem is done by the graph partitioning algorithm METIS, see Karypis and Kumar [8]. The matrices A_d and $\tilde{A}_d$ are stored separately, i.e. the indexing shown in figure 1 is used. The program is written in FORTRAN90 and it is compiled by the Intel-FORTRAN compiler 10.2. The communication is performed through calls to hp-MPI subroutines (C-binding). This MPI implementation invokes a shared memory mechanism whenever a shared memory is available. Since we consider only situations where all processes have access to a single physical memory, this is always the case. The consequence is that the data exchange between two processes is very fast. Relatively slow node-to-node communication does not occur. For an OpenMP implementation the same principal idea is valid since the data transfer from the memory to the processing core is the same; in this case the initialisation of the exchange should be replaced by a barrier and no waiting is necessary.

The linear system is solved by a conjugate gradient technique where AMG acts as a preconditioner. We have chosen two different AMG algorithms to demonstrate that the benefit of our modification is not restricted to a particular algorithm. The cycling strategy of both algorithms is the standard v-cycle of algorithm 1. The differences between both algorithms lie in the coarsening algorithm, i.e. in the definition of the restriction and prolongation operators; for both algorithms we apply two pre- and two post-smoothing sweeps.

The coarsening of algorithm ams1cg is an implementation of the Smoothed Aggregation scheme that has been suggested by Vaněk et al. [9]. This algorithm typically produces aggregates of between 10 and 50 fine-grid nodes and the structure of the prolongation operators is complex which entails that the computation

Table 1. Technical data of the employed Intel processors including bandwidth measured with the STREAM benchmark of McCalpin [6] (array size: $20 \cdot 10^6$ 8-byte variables, maximum number of cores employed in parallel using MPI)

processor	X5670	X5470	X5365	X5160
processors per machine	2	2	2	2
cores per processor	6	4	4	2
cache per processor	12MB (L3)	$2 \cdot 6$MB (L2)	$2 \cdot 4$ MB (L2)	$2 \cdot 2$MB (L2)
memory bus	QPI, 6.4GT/s	FSB, 1333MHz	FSB, 1333MHz	FSB, 1333MHz
bandwidth	26853 MB/s	7211 MB/s	6124 MB/s	7231 MB/s
clock speed	2.93GHz	3.33GHz	3.0GHz	3.0GHz
release	Q1/2010	Q3/2008	Q3/2007	Q2/2006

of the coarse-grid hierarchy is expensive. The advantage of the algorithm is that each coarse grid of level $l+1$ is significantly smaller than the fine grid of level l where it is constructed from; consequently the total number of grids is kept small. The coarsening of the algorithm amv1cg, on the other hand, is a modification of the pairwise aggregation described in the paper Emans [7] where the interpolation within the aggregates is constant. This makes the computation of the coarse-grid hierarchy particularly cheap since it is essentially reduced to an addition of rows of A_d. However, the number of grids will be rather large since each coarse-grid has only about half as many unknowns as the fine grid it is constructed from. The parallel implementation of both algorithms has been discussed comprehensively in Emans [7].

It is instructive to examine first the contribution of the smoother and the computation of the residual to the total computing time. In figure 2 the total computing time (of the linear solver, i.e. conjugate gradients and AMG preconditioner) and the computing times of the single tasks within the AMG algorithm are plotted: It turns out that the contribution of smoother and calculation of the residual (together) is around 60%. A more effective implementation of these two parts will therefore have a significant effect on the total computing time.

The parallel efficiency of the entire solution phase drops down to 30% for calculations on eight processes, see figure 2. Smoothing and calculation of the residual show the same poor values. A much better parallel efficiency of the same algorithm and the same implementation is obtained if the eight processes are distributed to two nodes, see Emans [7], although in the latter case a node-to-node communication through a network interconnect was necessary. Since the data transfer between memory and the cores is the limiting factor, it is promising to reduce this data transfer.

Figure 3 shows the ratio of the computing times measured for implementation with the combination of the smoother and residual calculation t_1 and the conventional implementation t_0 (using algorithms 3 and 2 separately). For the processors with FSB memory controller the proposed modification leads to an acceleration of the computations; for computations with four or eight parallel processes the gain lies in the range of up to 15% of the computing time for smoother and residual computation with translates to up to 10% of the total computation. The acceleration is larger if more cores are used in parallel: this leads to a higher load of the data bus and consequently to a slower transport of data to an individual core. It is noticeable, however, that the highest value of the curves of X5160 and of X5365 is reached in the calculation with two processes: Here the retardation of the computation through the slow data transfer is much less severe than in the calculations with four or eight processes, but it becomes noticeable that the parallel overhead of the combination of the smoothing sweep and the calculation of the residual cannot be hidden as efficiently as in the conventional implementation. Both effects are balanced in this example.

The processor X5670 with its QPI memory bus system can deliver the data required for the computation much faster to the cores. For calculations on these hardware the proposed modification is therefore not as efficient for calculations on hardware with FSB memory controller. Calculations with up to four parallel processes are slightly slowed down, but calculations involving more parallel processes can still be accelerated.

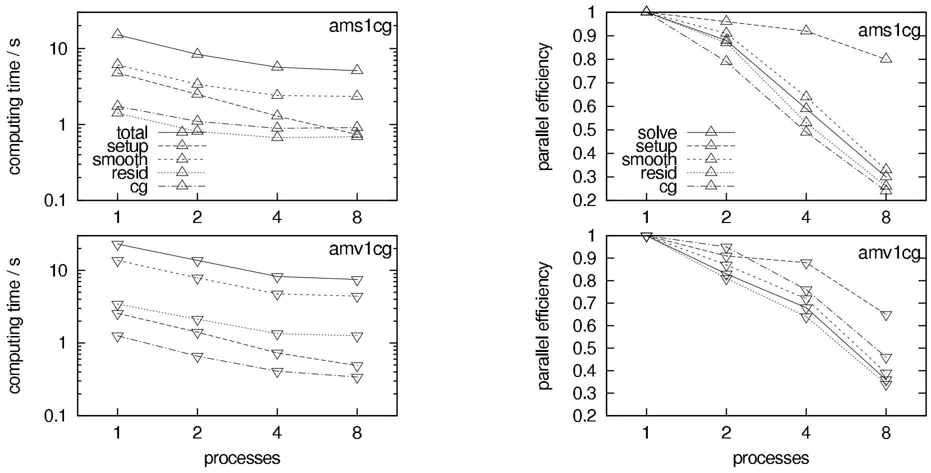


Fig. 2. Computing times of partial tasks on system X5470

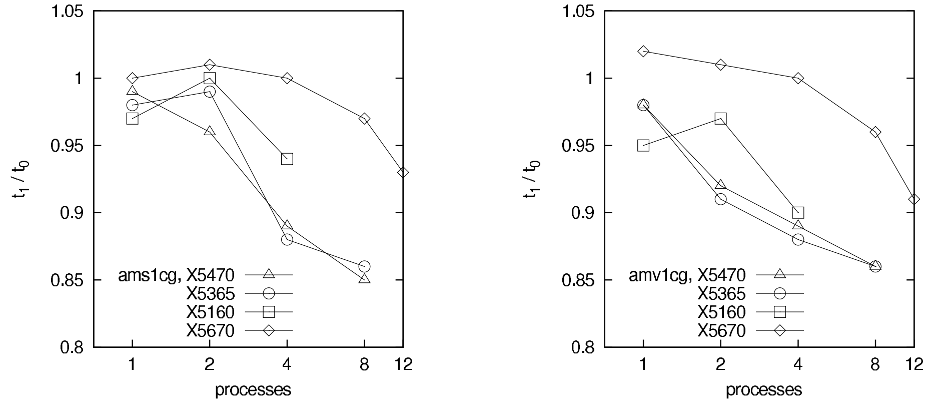


Fig. 3. Ratio of computing time of algorithms with combined smoother and residual calculation t_1 and conventional implementation t_0

4 Conclusions and Outlook

An implementation that avoids the double loading of the matrix for a smoothing step that is immediately followed by the calculation of the residual can be faster than a conventional implementation by up to 15%. The advantage depends on the capability of the hardware to transfer data from the main memory to the caches and the cores. The method is therefore most attractive for chips equipped with FSB memory connections. Future work will be focused on further exploiting the symmetry in the data structures.

Acknowledgement. Part of this work has been supported in the framework of "Industrielle Kompetenzzentren" by the Austrian Ministerium für Wirtschaft, Jugend und Familie and by the government of Upper Austria.

References

1. Emans, M., van der Meer, A.: Mixed-precision AMG as linear equation solver for definite systems. In: Sloot, P.M.A., Dongarra, G.V.A., Dongarra, J. (eds.) ICCS 2010, Part I. Procedia Computer Science, vol. 1, pp. 175–183. Elsevier Science, North Holland (2010)
2. Haase, G., Reitzinger, S.: Cache issues of algebraic multigrid methods for linear systems with multiple right-hand sides. SIAM Journal on Scientific Computing 27(1), 1–18 (2005)
3. Douglas, C., Hu, J., Ray, J., Thorne, D., Tuminaro, R.: Cache aware multigrid for variable coefficient elliptic problems on adaptive mesh refinement hierarchies. Numerical Linear Algebra with Applications 11, 173–187 (2004)
4. Falgout, R.D., Yang, U.M.: *hypre*: A Library of High Performance Preconditioners. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G. (eds.) ICCS-ComputSci 2002. LNCS, vol. 2331, pp. 632–641. Springer, Heidelberg (2002)
5. Gee, M., Siefert, C., Hu, J., Tuminaro, R., Sala, M.: ML 5.0 Smoothed Aggregation User's Guide. SAND 2006-2009 Unlimited Release (2006)
6. McCalpin, J.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 19–25 (December 1995)
7. Emans, M.: Efficient parallel AMG methods for approximate solutions of linear systems in CFD applications. SIAM Journal on Scientific Computing 32, 2235–2254 (2010)
8. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20, 359–392 (1998)
9. Vaněk, P., Mandel, J., Brezina, M.: Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Computing 56, 179–196 (1996)

Enhancing Parallelism of Tile Bidiagonal Transformation on Multicore Architectures Using Tree Reduction

Hatem Ltaief[1], Piotr Luszczek[2], and Jack Dongarra[2,3,4]

[1] KAUST Supercomputing Laboratory Thuwal, Saudi Arabia
[2] University of Tennessee, Knoxville, TN, USA
[3] Oak Ridge National Laboratory, USA
[4] University of Manchester, United Kingdom

Abstract. The objective of this paper is to enhance the parallelism of the tile bidiagonal transformation using tree reduction on multicore architectures. First introduced by Ltaief et. al [LAPACK Working Note #247, 2011], the bidiagonal transformation using tile algorithms with a two-stage approach has shown very promising results on square matrices. However, for tall and skinny matrices, the inherent problem of processing the panel in a domino-like fashion generates unnecessary sequential tasks. By using tree reduction, the panel is horizontally split, which creates another dimension of parallelism and engenders many concurrent tasks to be dynamically scheduled on the available cores. The results reported in this paper are very encouraging. The new tile bidiagonal transformation, targeting tall and skinny matrices, outperforms the state-of-the-art numerical linear algebra libraries LAPACK V3.2 and Intel MKL ver. 10.3 by up to 29-fold speedup and the standard two-stage PLASMA BRD by up to 20-fold speedup, on an eight socket hexa-core AMD Opteron multicore shared-memory system.

Keywords: Bidiagonal Transformation, Tree Reduction, High Performance Computing, Multicore Architecture, Dynamic Scheduling.

1 Introduction

This paper extends our prior work with one-sided factorizations and in particular, the tridiagonal reduction (TRD) [18] to the bidiagonal reduction (BRD) case, which presents more challenges due to its increased algorithmic complexity. BRD is an important first step when calculating the singular value decomposition (SVD). Two-stage reduction algorithms for two-sided factorizations are not new approaches but have recently enjoyed rekindled interest in the community. For instance, it has been used by Bischof et al. [6] for TRD (SBR toolbox) and Kågström et al. [16] in the context of Hessenberg and Triangular reductions for the generalized eigenvalue problem for dense matrices. The tile bidiagonal reduction for square matrices that was obtained in this way considerably outperforms the state-of-the-art open-source and commercial numerical libraries [17].

BRD for any rectangular dense matrix [11,9,22] is: $A = U\Sigma V^T$ with $A, \Sigma \in \mathbb{R}^{M\times N}$, $U \in \mathbb{R}^{M\times M}$, and $V \in \mathbb{R}^{N\times N}$. Following the decompositional approach to matrix computation [21], we transform the dense matrix A to an upper bidiagonal form B by applying successive distinct orthogonal transformations [15] from the left (X) as well as from the right (Y): $B = X^T AY \quad B,X,A,Y \in \mathbb{R}^{N \times N}$. This reduction step actually represents the most time consuming phase when computing the singular values. Our primary focus is the BRD portion of the computation which can easily consume over 99% of the time needed to obtain the singular values and roughly 75% if singular vectors are calculated [17].

The necessity of calculating SVDs emerges from various computational science areas, e.g., in statistics where it is directly related to the principal component analysis method [13,14], in signal processing and pattern recognition as an essential filtering tool and in analysis control systems [19]. However, the majority of the applications and especially data collected from large sensor systems involve rectangular matrices with the number of rows by far exceeding the number of columns [4,10]. We refer to such matrices as *tall and skinny*. For such matrices, the bulge chasing procedure (see Section 3) is no longer the bottleneck as it is the case for square matrices [17]. It is the reduction to the band form that poses a challenge which we address in this paper.

The remainder of this document is organized as follows: Sections 2 and 3 recalls the block BRD algorithm as implemented in LAPACK [1] as well as the two-stage BRD algorithm available in PLASMA [23] and explains their main deficiencies, especially in the context of tall and skinny matrices. Sections 4 gives a detailed overview of previous projects in this area and outlines the main contributions of the paper. Section 5 describes the implementation of the parallel two-stage tile BRD algorithm using a tree reduction for tall and skinny matrices. Section 6 presents the performance results. Finally, Section 7 summarizes the results of this paper and presents the ongoing work.

2 LAPACK Bidiagonal Transformation

LAPACK [1] implements a so called block variant of singular value decomposition algorithms. Block algorithms are characterized by two successive phases: a panel factorization and an update of the trailing submatrix. During the panel factorization, the orthogonal/unitary transformations are applied only within the panel one column at a time. As a result, the panel factorization is very rich in Level 2 BLAS operations. Once accumulated within the panel, the transformations are applied to the rest of the matrix (commonly called the trailing submatrix) in a blocking manner, which leads to an abundance of calls to Level 3 BLAS. While the update of the trailing submatrix is compute-bound and very efficient, the panel factorization is memory-bound and has mostly been a bottleneck for the majority of numerical linear algebra algorithms. Lastly, the parallelism within LAPACK occurs only at the level of the BLAS routines, which results in an expensive fork-join scheme of execution with synchronization around each call.

The use of tall-and-skinny matrices compounds the aforementioned inefficiencies. On one hand, the memory-bound panel factorization is now disproportionately expensive compared with the trailing matrix update. On the other hand, the fork-join parallelism does not benefit at all from the execution of the panel factorization because only Level 2 BLAS may be used – memory-bound operations that only marginally benefit from parallelization. Clearly, the parallelism needs to be migrated from the BLAS level up to the factorization algorithm itself.

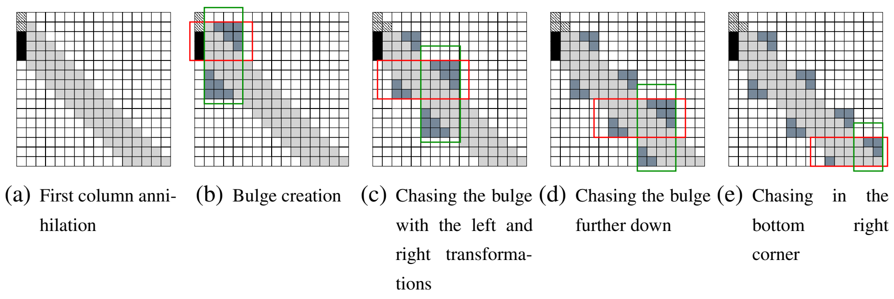

(a) First column annihilation (b) Bulge creation (c) Chasing the bulge with the left and right transformations (d) Chasing the bulge further down (e) Chasing in the bottom right corner

Fig. 1. Execution breakdown of the bulge chasing procedure for a band bidiagonal matrix of size N=16 with NB=4

3 PLASMA Bidiagonal Transformation Using Two-Stage Approach

In our last implementation described in [17], we fully utilize a two-stage approach – a technique that has recently proven its value as a viable solution for achieving high performance in the context of two-sided reductions [6,16,18]. The first stage consists of reducing the original matrix to a band form. The overhead of the Level 2 BLAS operations dramatically decreases and most of the computation is performed by the Level 3 BLAS, which makes this stage run closer to the theoretical peak of the machine. In fact, this stage has even enough computational load to benefit from offloading the work to GPU accelerators [5]. The second stage further reduces the band matrix to the corresponding compact form. A bulge chasing procedure, that uses orthogonal transformations annihilates the off-diagonal elements column-wise and eliminates the resulting fill-in elements that occur towards to the bottom right corner of the matrix. Figure 1 depicts the execution breakdown of chasing the first column (black elements) on a band bidiagonal matrix of size M=N= 16 and NB= 4.

The next section explains why this current implementation of the tile BRD using a two-stage approach is not appropriate for the case of tall and skinny matrices.

4 Related Work and Relevant Contributions

Numerical schemes based on tree reductions have been developed first for dense one-sided factorization algorithms [8]. It was done in the context of minimizing communication amount between the levels of the memory hierarchy as well as between remote parallel processors. Given the fact that such reduction schemes are numerically stable under a set of practical assumptions, we are able to apply similar schemes for the two-sided reductions.

Practical applications of these numerical reduction schemes on multicore architectures may indeed achieve very competitive results in terms of performance [12]. And the same apply equally to GPU-accelerated implementations [2] as well as the codes designed specifically for distributed memory clusters of multicore nodes [20].

To broaden the scope, Bouwmeester et al. [7] provide a taxonomy of QR variants based on, among others, the reduction algorithm for achieving unitary annihilation across the full matrix height. Accordingly, our implementation may be considered as the version named TT by the authors. And this includes both the left (QR) and the right (LQ) application of the Householder reflectors [15]. A more exhaustive study of various combinations of reduction algorithms is beyond the scope of this paper.

The communication optimality for eigenvalue and singular value decompositions (as well as other related decompositions) has been studied for the entire process of reduction to a condensed form [3]. Here, however, we only concern ourselves with the reduction to the bidiagonal form and focus primarily on the implementation aspects of a particular two-sided factorization algorithm.

Enhancing parallelism using tree reduction has already been performed in the context of one-sided factorizations for tall and skinny matrices, as mentioned earlier. However, we propose here the very first practical implementation that uses a tree reduction for BRD. We contribute the idea of splitting the sequential panel factorization step into independent subdomains, that can simultaneously be operated upon. Each subdomain computation proceeds locally in parallel. Once the local computation finishes, the reduction step is triggered using a binary tree, in which the contributions of neighbor pairwise subdomains are merged. With tile algorithms, the whole computation can be modeled as a directed acyclic graph (DAG), where nodes represent fine-grained tasks and edges correspond to data dependencies. Thanks to the dynamic scheduling framework QUARK [24], the different fine-grained tasks are processed as soon as their data dependencies are satisfied. Therefore, unnecessary synchronization points are completely removed between the steps of the local computations within each subdomain. The cores that are no longer active in the merging step do not have to wait until the end of the merging step before proceeding with the next panel. The whole computation then proceeds seamlessly by following a *producer-consumer* model.

5 Tile Bidiagonal Transformation Using Tree Reduction

5.1 Methodology

In the context of tall and skinny matrices, the first stage of the standard two-stage tile BRD is not suitable anymore. Indeed, when the number of rows is substantially larger than the number of columns, i.e. $M \gg N$, the first stage becomes now the bottleneck because of the panel being processed sequentially. The goal of the new implementation of this two-stage BRD is to horizontally split the matrix into subdomains, allowing independent computational tasks to concurrently execute. A similar algorithm has been used to improve the QR factorization of tall and skinny matrices in [12]. The second stage, which reduces the band matrix to the bidiagonal form, only operates on the top matrix of size $N \times N$ and is negligible compared to the overall execution time. Therefore, the authors will primarily focus on optimizing the first stage.

5.2 Description of the Computational Kernels

This section is only intended to make the paper self-contained as the description of the computational kernels have already been done in previous author's research papers

[12,18,17]. There are ten kernels overall, i.e. four kernels for the QR factorizations, four kernels for the LQ factorizations and two kernels in order to process the merging step. *CORE_DGEQRT* and *CORE_DGELQT* perform the QR/LQ factorization of a diagonal tile, respectively. It produces an upper (QR) or lower (LQ) triangular matrix. The upper triangular matrix is called reference tile because it will be eventually used to annihilate the subsequent tiles located below, on the same panel. *CORE_DTSQRT* and *CORE_DTSLQT* compute the QR/LQ factorization of a matrix built by coupling the upper/lower triangular matrices produced by *CORE_DGEQRT* (reference tile) and *CORE_DGELQT* with a tile located below (QR) or to the right of the diagonal (LQ), respectively. The transformations accumulated during the panel computation (characterized by the four kernels described above) are then applied to the trailing submatrix with *CORE_DORMQR* and *CORE_DORMLQ* using the Householder reflectors computed by *CORE_DGEQRT* and *CORE_DGELQT* and with *CORE_DTSMQR* and *CORE_DTSMLQ* using the Householder reflectors computed by *CORE_DTSQRT* and *CORE_DTSLQT*, respectively. The last two kernels, which perform the merging steps for tall and skinny matrices are *CORE_DTTQRT* and *CORE_DTTMQR*.

5.3 DAG Analysis

The dynamic runtime system QUARK [24] has the capability to generate DAGs of execution on the fly, which are critical in order to understand the performance numbers reported in this paper. For the next three figures, the yellow and blue nodes correspond to the tasks of the QR and LQ factorizations, respectively. The red nodes represent the tasks involved during the tree reduction, i.e. the merging step. The matrix size is defined to 10×2 in terms of number of row and column tiles. Figure 2 shows the DAG of the standard two-stage PLASMA BRD (first stage only). The bottleneck of sequentially computing the panel clearly appears. Here, the first stage would take 22 steps to achieve the desired band form. Figure 3 highlights the DAG of the two-stage PLASMA BRD using tree reduction with two subdomains. The two distinct entry points are identified, which allows the panel to proceed in parallel. Once computed, the merging phase (red nodes) can be initiated as soon as the data dependencies are satisfied. Here, the number of steps to obtain the band form has been significantly reduced to 15 steps. Finally, Figure 4 shows the DAG of the two-stage PLASMA BRD using tree reduction with eight subdomains. The eight distinct entry points are clearly distinguished. The DAG is now more populated with red nodes due to the high number of merging steps. The number of steps to obtain the band form has been further reduced to 13 steps, while the number of concurrent tasks has drastically increased.

6 Experimental Results

6.1 Environment Setting

We have performed our tests on a shared memory machine with the largest number of cores we could access. It is composed of eight AMD Opteron$^{\text{TM}}$ processors labelled 8439 SE. Each of the processors contains six processing cores each clocked at 2.8 GHz.

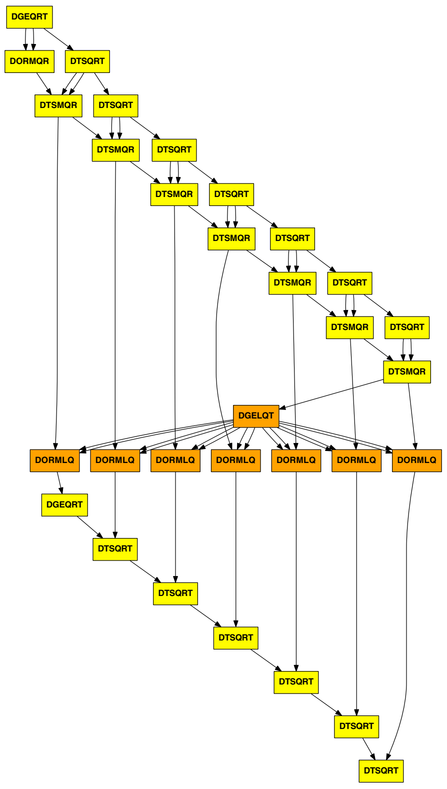


Fig. 2. DAG of the standard two-stage PLASMA BRD (first stage only) on a matrix with MT= 8 and NT= 2 tiles

The total number of cores is evenly spread among two physical boards. The theoretical peak for this machine for double precision floating-point operations is 537.6 Gflop/s (11.2 Gflop/s per core). And the total available memory is 128 GB which is spread among 8 NUMA nodes. On the software side, we used Intel Math Kernel Library MKL version 10.3 with an appropriate threading setting to force single-thread execution. The blocking parameters we used in our tests were NB of 144 (the external tile blocking) and IB of 48 (the internal tile blocking) for our double precisions runs. All experiments have been conducted on all 48 cores to stress not only the asymptotic performance but also scalability with the largest core count we could access.

6.2 Performance Comparisons

In the figures presented in this section, we refer to the standard two-stage tile BRD as *PLASMA* and to the optimized two-stage tile BRD for tall and skinny matrices using tree reduction as *PLASMA TR*. Figure 5 shows the performance of PLASMA TR with M = 57600 and N = 2880 (both sizes are fixed) and a varying number of subdomains. When the number of subdomains is one, the implementation of *PLASMA TR* is in fact

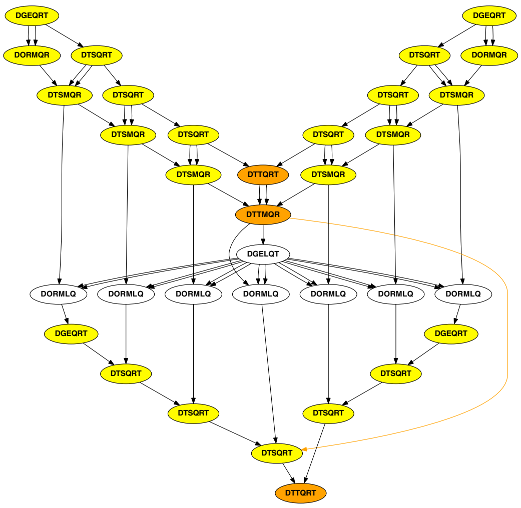


Fig. 3. DAG of the two-stage PLASMA BRD using tree reduction on a matrix with MT= 8 and NT= 2 tiles using two subdomains

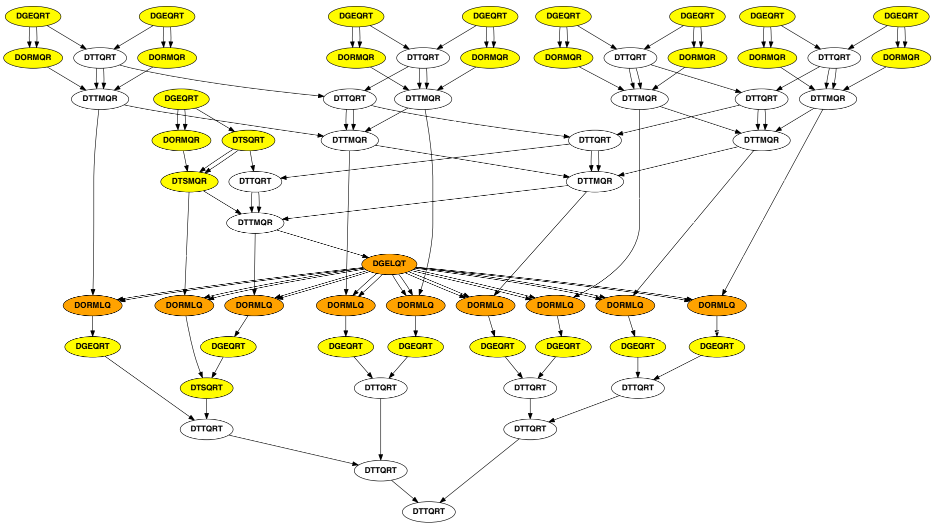


Fig. 4. DAG of the two-stage PLASMA BRD using tree reduction on a matrix with MT= 8 and NT= 2 tiles using eight subdomains

equivalent to the one of *PLASMA* and this is why they report the same performance number. However, when the number of subdomains increases, *PLASMA TR* rapidly outperforms *PLASMA*. Noteworthy to mention the very low rates of execution of LAPACK and MKL. This has been noticed for square matrices [17] and it is even worse for tall and skinny matrices. Figure 6 shows the performance of *PLASMA TR* with $M = 57600$

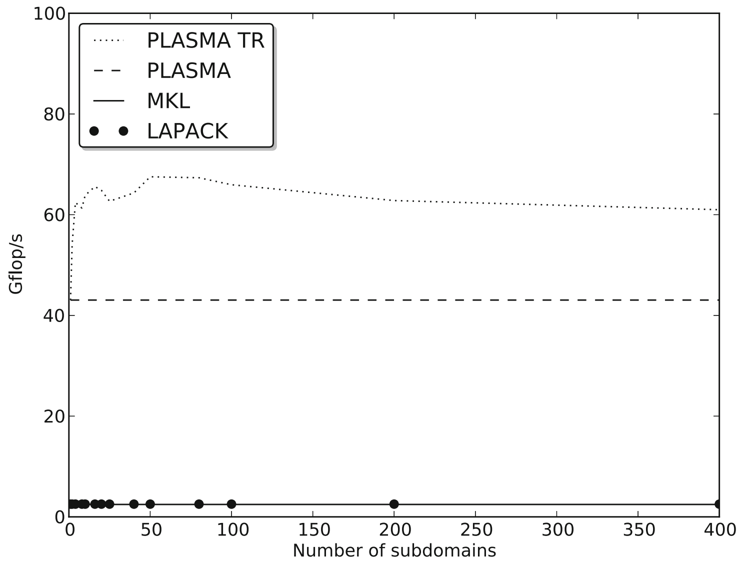


Fig. 5. Performance of *PLASMA TR* with M = 57600 and N = 2880 (fixed) and a varying number of subdomains on 48 AMD Opteron cores

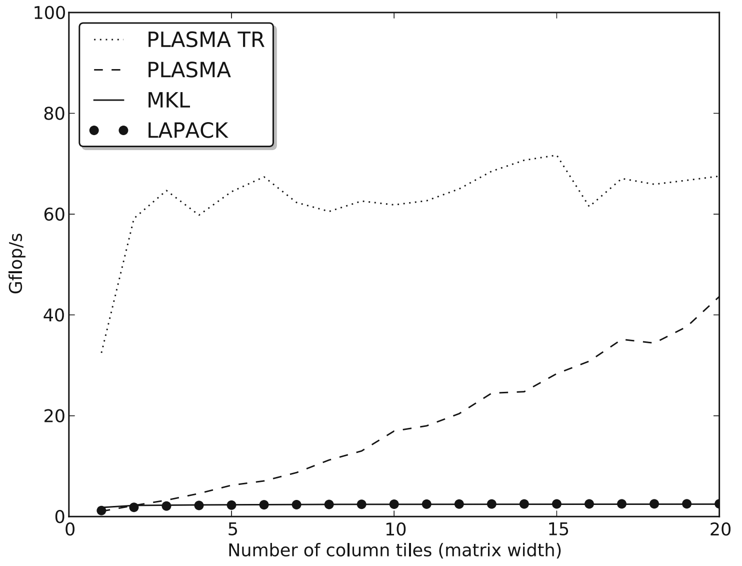


Fig. 6. Performance of *PLASMA TR* with M = 57600 (fixed) and a varying number of column tiles on 48 AMD Opteron cores

(fixed) and a varying number of column tiles. The subdomain sizes giving the best performance have been selected for *PLASMA TR*. When the matrix has only a small number of column tiles (i.e., skinny), this is where our implementation performs the best compared to the three other libraries. *PLASMA TR* achieves up to 20-fold speedup and up to 29-fold speedup compared to *PLASMA* (with $M = 57600$ and $N = 3 \times 144 = 432$) and LAPACK and MKL (with $M = 57600$ and $N = 15 \times 144 = 2160$), respectively. The clear advantage over LAPACK stems from exposing parallelism of reduction of tall matrix panels and the good locality and plentiful parallelism of the two-stage approach. As the number of column tiles or the matrix width increases, the performance of *PLASMA* implementation starts catching up *PLASMA TR*, since the matrix becomes square.

7 Conclusions and Future Work

In this paper, we presented a new parallel implementation of the tile two-stage BRD algorithm suitable for tall and skinny matrices on shared-memory multicore architectures. Our implementation is far superior to any functionally equivalent code that we are aware of. In fact, it outperforms LAPACK and Intel MKL nearly 29-fold and PLASMA – 20-fold for matrices it was designed for. Our ongoing and future work focuses on automatic selection of domains in the tree reduction stage as well as optimal interleaving strategies for QR and LQ application of the orthogonal reflectors. We would also like to use investigate ways of achieving larger fraction of peak performance by using static scheduling which has lower overhead.

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, S.L., Demmel, J.W., Dongarra, J.J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.C.: LAPACK User's Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia (1999)
2. Anderson, M., Ballard, G., Demmel, J., Keutzer, K.: Communication-avoiding QR decomposition for GPUs. In: Proceedings of IPDPS 2011, Anchorage, AK USA. ACM (2011); Also available as Technical Report UCB/EECS-2010-131, February 18, 2011 and LAWN 240
3. Ballard, G., Demmel, J., Dumitriu, I.: Minimizing communication for eigenproblems and the singular value decomposition (2010), arXiv:1011.3077
4. Balmino, G., Bruinsma, S., Marty, J.-C.: Numerical simulation of the gravity field recovery from GOCE mission data. In: Proceedings of the Second International GOCE User Workshop "GOCE, The Geoid and Oceanography", March 8-10, ESA-ESRIN, Frascati, Italy (2004)
5. Bientinesi, P., Igual, F., Kressner, D., Quintana-Orti, E.: Reduction to Condensed Forms for Symmetric Eigenvalue Problems on Multi-core Architectures. In: Parallel Processing and Applied Mathemetics, pp. 387–395 (2010)
6. Bischof, C.H., Lang, B., Sun, X.: Algorithm 807: The SBR Toolbox—software for successive band reduction. ACM Trans. Math. Softw. 26(4), 602–616 (2000)
7. Bouwmeester, H., Jacquelin, M., Langou, J., Robert, Y.: Tiled QR factorization algorithms. Technical Report RR-7601, INRIA (2011)
8. Demmel, J.W., Grigori, L., Hoemmen, M.F., Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations. Technical Report 204, LAPACK Working Note (August 2008)

9. Golub, G., Van Loan, C.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
10. Golub, G.H., Manneback, P., Toint, P.L.: A comparison between some direct and iterative methods for certain large scale geodetic least squares problems. SIAM J. Scientific Computing 7(3), 799–816 (1986)
11. Golub, G.H., Reinsch, C.: Singular value decomposition and least squares solutions. Numer. Math. 14, 403–420 (1970)
12. Hadri, B., Ltaief, H., Agullo, E., Dongarra, J.: Tile QR factorization with parallel panel processing for multicore architectures. In: IPDPS, pp. 1–10. IEEE (2010)
13. Hotelling, H.: Analysis of a complex of statistical variables into principal components. J. Educ. Psych. 24, 417–441, 498–520 (1933)
14. Hotelling, H.: Simplified calculation of principal components. Psychometrica 1, 27–35 (1935)
15. Householder, A.S.: Unitary triangularization of a nonsymmetric matrix. Journal of the ACM (JACM) 5(4) (October 1958), doi:10.1145/320941.320947
16. Kågström, B., Kressner, D., Quintana-Orti, E., Quintana-Orti, G.: Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited. BIT Numerical Mathematics 48, 563–584 (2008)
17. Ltaief, H., Luszczek, P., Dongarra, J.: High performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures. Technical report, LAPACK Working Note 247 (2011)
18. Luszczek, P., Ltaief, H., Dongarra, J.: Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In: Proceedings of IPDPS 2011, Anchorage, AK, USA. ACM (2011)
19. Moore, B.C.: Principal component analysis in linear systems: Controllability, observability, and model reduction. IEEE Transactions on Automatic Control AC-26(1) (February 1981)
20. Song, F., Ltaief, H., Hadri, B., Dongarra, J.: Scalable tile communication-avoiding QR factorization on multicore cluster systems. In: Proceedings of SC 2010, New Orleans, Louisiana. ACM (November 2010), Also available as Technical Report UT-CS-10-653 March, 2011 and LAWN 241
21. Stewart, G.W.: The decompositional approach to matrix computation. Computing in Science & Engineering 2(1), 50–59 (2000), doi:10.1109/5992.814658, ISSN: 1521-9615
22. Trefethen, L.N., Bau, D.: Numerical Linear Algebra. SIAM, Philadelphia (1997), `http://www.siam.org/books/OT50/Index.htm`
23. University of Tennessee. PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Archtectures, Version 2.2 (November 2009)
24. YarKhan, A., Kurzak, J., Dongarra, J.: QUARK users' guide: Queueing and runtime for kernels. Technical Report ICL-UT-11-02, University of Tennessee, Innovative Computing Laboratory (2011)

Autotuning of Adaptive Mesh Refinement PDE Solvers on Shared Memory Architectures

Svetlana Nogina, Kristof Unterweger, and Tobias Weinzierl

Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
{nogina,unterweg,weinzier}@in.tum.de

Abstract. Many multithreaded, grid-based, dynamically adaptive solvers for partial differential equations permanently have to traverse subgrids (patches) of different and changing sizes. The parallel efficiency of this traversal depends on the interplay of the patch size, the architecture used, the operations triggered throughout the traversal, and the grain size, i.e. the size of the subtasks the patch is broken into. We propose an oracle mechanism delivering grain sizes on-the-fly. It takes historical runtime measurements for different patch and grain sizes as well as the traverse's operations into account, and it yields reasonable speedups. Neither magic configuration settings nor an expensive pre-tuning phase are necessary. It is an autotuning approach.

1 Introduction

Grid-based solvers of partial differential equations (PDE) used for example in computational fluid dynamics (CFD) affect to this day the software landscape of high performance computing. In the zoo of evolutionary trends in the field we observe two paradigm shifts: On the one hand, the nodes of the supercomputers evolve from single core machines to multicore architectures. An increase in single-node performance not any longer stems from an improved clock rate but from multiple threads [12]. On the other hand, the PDE codes change from solvers working on regular grids to solvers working on dynamically adaptive grids due to adaptive mesh refinement (AMR). An increase of accuracy not any longer stems from globally refined grids but from grids that resolve the physical phenomena accurately where it is necessary. AMR codes hence have to be able to handle adaptive grids with a high parallel efficiency on multicore computers.

Both trends interfere with each other: Regular grids are convenient for shared memory parallelisation, as they can be traversed basically by one big parallel loop due to colouring techniques such as red-black Gauß-Seidel. Adaptive grids without a homogeneous regular structure are convenient with respect to their ratio of accuracy to computing workload. Hence, many AMR codes work with patches of regular grids. They then can distribute the patches among threads [7,10]. Or they can process the individual patches with several threads. While combinations of both approaches are possible, we focus on the latter [6] raising the question how to distribute the patch among the individual threads.

Multithreading libraries typically abstract from the number of threads and split up the work—in our case the patches—into work units, chunks, or tasks. These chunks then are distributed among the cores. Selecting a good chunk size, often also called grain size, for the patches is important: If the size is too small, the overhead to distribute and balance the chunks is significant. If the size is too big, the concurrency level suffers—some cores might run out of work while others still are busy. A good grain size depends on several factors: code location, size of total workload, characteristics of the hardware, state of the overall algorithm, synchronisation points, and so forth. Two different code fragments each comprising a parallel loop might require two different grain sizes. Two different patches might require two different grain sizes. Two different machines might require two different grain sizes. And two different algorithmic phases such as an equation system solver and a plot of the current solution might require two different grain sizes, too. All popular shared memory parallelisation environments allow the user to select a grain size individually for each parallel program statement. Intel's Threading Building Blocks (TBB) [11] for example enable the user to specify one grain size per parallel loop, and OpenMP also supports the idea of chunk sizes [3]. This paper discusses grain sizes with respect to loops. However, the reasoning also holds for approaches that model the parallel problem as task graph where interpreting every single work unit (stencil computation on the patch, e.g.) as task induces a tremendous overhead. Hence, one usually combines multiple work units into one task. Again there is the notion of a task size. While the developer can make the size depend on the influencing factors from above, this is cumbersome to do manually. We propose an autotuning approach trying to find good grain sizes on-the-fly and automatically for all parallel sections.

Such an approach is not new: Some authors suggest to run the application with different grain sizes for the individual parallel sections prior to any production run [1,11] to identify a reasonable application configuration. Some derive chunk models considering thread numbers, memory layout, and so forth [5,9]. Some hold several different implementations of parallel codes available [5,17] and use an elaborated run time estimator (typically based on historical data) as well as a performance model to select the best-suited implementation on-the-fly [4,14]. These techniques mirror feedback optimisation. We propose to combine historic data with a grain size search: Each time a parallel section is executed, we try to find a better grain size based upon the historic measurements. Different to other approaches, our approach explicitly takes into account both the code section running in parallel, the problem size, and the state of the algorithm, i.e. we do not study only one specific algorithmic phase such as stencil computations. Our approach is orthogonal to work stealing, self-scheduling, and factoring [2,8,11,13], as we do not discuss balancing but the minimal balancing granularity. Our results exhibit that a proper chunk choice depending on both the patch size and the algorithm's phase (which step of an CFD code is to be performed next, e.g.) is important no matter what work balancing approach is used. It is particularly important for high performance computing where the workload per patch does not change dynamically, and static load balancing often is sufficient.

The remainder is organised as follows: We first present some performance measurements and observations as well as some ideas what a good grain size selector should look like in Section 2. In Section 3, we then introduce an oracle telling the application on-the-fly what grain size to use for which loop. Afterwards, we present some results for this oracle and show the benefit for selected applications. A short summary and an outlook close the discussion.

2 Performance Observations and Model

Our discussion is led by two observations that we discuss by means of a multi-threaded AMR solver for the heat equation with an implicit Euler time stepping [16]. Three different algorithmic steps are subject to our study: The solver computes a solution to the equation system with a matrix-free multigrid solver based upon a Jacobi smoother. For this, it traverses the grid once per Jacobi iteration. We first study this smoother phase (smooth). Second, the solver switches to the subsequent time step as soon as the solution converges (step). For this, it traverses the grid once. Finally, the solver combines this transition in time with a stream to an output file every k time steps (plot). The observations discussed here for one Xeon with ten cores and TBB's work stealing are comparable to other multicore machines and also OpenMP though they differ in numbers, and they also resemble insights obtained from our CFD code that is based upon the same AMR grid management [15].

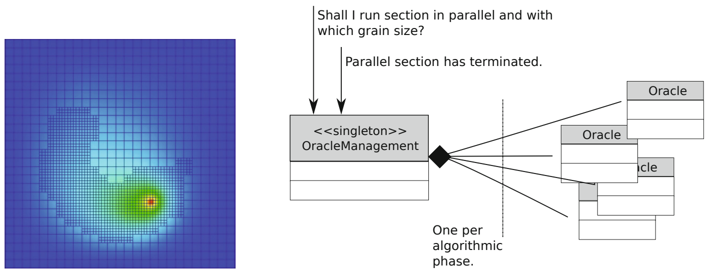

Shall I run section in parallel and with which grain size?
Parallel section has terminated.
<<singleton>> OracleManagement
Oracle
Oracle
cle
One per algorithmic phase.

Fig. 1. Typical grid of the heat equation solver (left). This experiment studies a rotating heat source, and, hence, the patches change in time permanently as the adaptivity structure "follows" the heat stimulus. Oracle architecture (right).

According to some measurements in Table 1 the three algorithmic phases exhibit different parallel behaviour. Switching to the subsequent time step scales. Computing the next iterate does scale up to a given number of threads, i.e. if we increase the number of computing threads beyond a magic number, the performance degenerates again. The plot operation does not scale at all. These different characteristics are due to the fact that the Jacobi smoother comprises some global updates. The global residual, e.g., is reduced from all the threads

Table 1. Averaged time ([t]=s) per grid traversal per patch for three different solver phases and two different patch sizes. The grain size, i.e. the size of individual subproblems deployed to threads, is fixed to 128 vertices or cells, respectively, per subproblem.

	82×82			162×162		
Threads	step	smooth	plot	step	smooth	plot
1	$4.83 \cdot 10^{-3}$	$1.64 \cdot 10^{-2}$	$5.41 \cdot 10^{-2}$	$1.83 \cdot 10^{-2}$	$6.14 \cdot 10^{-2}$	$1.84 \cdot 10^{-1}$
2	$2.70 \cdot \ldots$	$1.10 \cdot \ldots$	$8.18 \cdot \ldots$	$1.03 \cdot \ldots$	$4.23 \cdot \ldots$	$3.09 \cdot \ldots$
4	$1.66 \cdot \ldots$	$0.99 \cdot \ldots$	$10.21 \cdot \ldots$	$0.61 \cdot \ldots$	$3.99 \cdot \ldots$	$4.51 \cdot \ldots$
10	$1.07 \cdot \ldots$	$0.45 \cdot \ldots$	$7.05 \cdot \ldots$	$0.34 \cdot \ldots$	$1.72 \cdot \ldots$	$2.58 \cdot \ldots$
20	$1.35 \cdot \ldots$	$0.43 \cdot \ldots$	$7.51 \cdot \ldots$	$0.29 \cdot \ldots$	$1.55 \cdot \ldots$	$2.88 \cdot \ldots$

and induces a sequential section in the code. In the plot phase, the grid traversal writes to one output stream and this stream is protected by a semaphore. The lock acquisition is the more expensive the more threads are involved. As the grid traversal underlying all three phases is implemented only once—it triggers different behavioural routines—it is consequently important to know in which phase the algorithm is at the moment, i.e. to consider the invoked call graph.

If the grid traversal is not inherently sequential, we observe a second characteristics (Fig. 2): Dynamic load balancing for this kind of application where the load is directly proportional to the patch size is not thoroughly advantageous. Very small grain sizes allow the load balancer to distribute the workload with a very fine granularity. If the granularity is too fine, the effort spent on the load balancing however increases the wall clock time. If the grain size is equal to the problem size, only one thread works and the others run idle. The best performance often stems from a grain size in-between. Both observations make us draw the following conclusions:

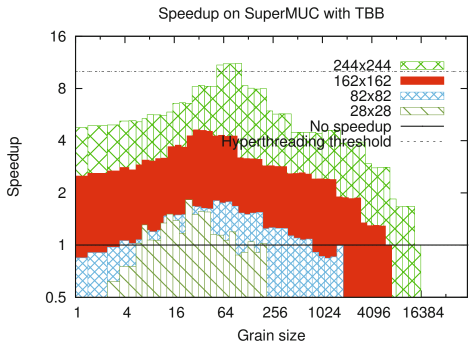


Fig. 2. The choice of a proper grain size for one specific loop (here the "run over all cells" for four 2d grid sizes) of the solver (the Jacobi smoother) determines the speedup

– For each individual location in the code where we enter a parallel section, we have to search for the best grain size.
– The bigger the problem handled by a parallel section, the bigger a good grain size might be chosen. Hence, it is important to search for good grain sizes for each problem size individually—in particular for AMR codes where the spectrum of grain sizes changes permanently (Fig. 1).
– Besides the location where the parallel code is invoked, we also have to take into account the phase of the algorithm, i.e. the call graph induced.
– The best grain size allows the load balancer to distribute the work equally among the threads. However, it might happen that the best grain size makes some processors run idle. A too small grain size implies a significant synchronisation overhead and slows down the overall application.

3 Grain-Size Oracle

Based upon the observations, we propose a very simple implementation pattern. An *Oracle* is a class providing two operations: `getGrainSize(`*problemSize*`)` and `terminate(`*elapsedTime*`)`. `getGrainSize` returns a well-suited grain size for a given problem. `terminate` informs the oracle how much time has elapsed since the last `getGrainSize` call. Besides several instances of *Oracle*, there is an *OracleManager*. The manager holds one oracle per calling code location (Fig. 1) and per algorithmic phase. Each time the solver enters an algorithmic phase, the manager exchanges the set of active oracles. It is a single-threaded implementation protected by a boolean semaphore, and it stores a time stamp each time `getGrainSize` is invoked. Then, it forwards the function call to the appropriate oracle. When `terminate` is called, it passes the elapsed time to the oracle that answered to `getGrainSize` before. If two `getGrainSize` calls for the same parallel section occur in a row, the second is answered with a "do not run in parallel" flag. This way, we forbid nested parallelisation for the same type of operations. With the history of grain sizes and elapsed times of one parallel code section for one algorithmic phase at hand, the individual oracles improve the selected grain sizes on-the-fly throughout the simulation run. In this paper, we study two different oracle implementations.

3.1 Oscillating Search

Our first oracle implementation is based upon two assumptions: If splitting up the problem into two pieces (i.e. to use two threads) doesn't accelerate the application, splitting up the problem into smaller subtasks is useless. And it assumes that the graph of the runtime is more or less convex.

Internally, the oracle is a mapping f of problem sizes N to tuples (g, s, t). The last case distinction in Algorithm 1 gives the oracle its name: For a given grain size g, the algorithm studies the impact of a new grain size increasing the current grain size by a search delta s. If this alternative size does not perform better than the original grain size, the search direction is inverted and divided by two.

Algorithm 1. `getGrainSize` and `terminate` of the oscillating search oracle.

```
 1: procedure GETGRAINSIZE($problemSize$)
 2:     Remember $problemSize$ locally
 3:     Let $f(problemSize) = (g, s, t)$
 4:     if $problemSize$ not studied before then
 5:         $(g, s, t) \leftarrow (0, \frac{problemSize}{2}, \perp))$
 6:         return 0                         ▷ Grain size 0 means "run sequential".
 7:     else return $g + s$
 8: procedure TERMINATE($elapsedTime$)
 9:     if $t = \perp$ then  $(g, s, t) \leftarrow (g, s, elapsedTime)$
10:     else if $elapsedTime < t$ then  $(g, s, t) \leftarrow (g + s, s, elapsedTime)$
11:     else
12:         if $g - b/2 < 0$ then  $(g, s, t) \leftarrow (0, 0, elapsedTime)$
13:         else $(g, s, t) \leftarrow (g, -s/2, t)$
```

The first case distinction ensures that whenever a function runs for the very first time, its sequential time is measured. And it also ensures that the next time it is called, the oracle makes two threads run in parallel. The last case distinction determines whether to parallelise at all. If the problem scales on two processors, g becomes a positive number and never underruns 0. If there is no speedup for two threads, the oracle immediately switches off the parallelisation.

3.2 Interval Search

The second oracle implementation (Algorithm [2]) assumes that the speedup graph is sufficiently smooth. It internally holds a grain-size interval represented by its minimal and maximal grain size. We start with the whole problem size being the interval, i.e. left and right boundary of the search interval represent sequential runs, and measure the runtime for the grain size represented by the mid of the interval. If this grain size yields a better runtime than the left and right interval boundary, we shrink the interval by a factor of two around the mid point and continue iteratively. Otherwise, we halve the interval and continue to search for a better grain size in the left or right subinterval.

4 Results

We tested our approach for the multiscale solver for the heat equation discussed before [16] and for a CFD code based upon Chorin's projection method with a Gauß-Seidel. Both solvers are based upon the same grid management and traversal routines [15] relying on OpenMP or TBB, and both support two- and three-dimensional simulations. The TBB variant applies dynamic load stealing dividing the problem further and further into smaller subproblems if necessary [2,8,11]. Here, the selected grain size is the minimum grain size allowed. The OpenMP variant splits up the whole problem into subproblems of fixed grain size and leaves it up to OpenMP's dynamic scheduling to distribute this sequence

Algorithm 2. The interval search oracle.

```
1: procedure GETGRAINSIZE($problemSize$)
2:     Remember $problemSize$ locally
3:     Let $f(problemSize) = (g_l, g_r, t_l, t_r)$
4:     if $problemSize$ not studied before then $(g_l, g_r, t_l, t_r) \leftarrow (0, problemSize, \perp, \perp)$
5:     if $t_l = \perp$ then return $g_l$
6:     else if $t_r = \perp$ then return $g_r$
7:     else return $\frac{g_l+g_r}{2}$
8: procedure TERMINATE($elapsedTime =: t$)
9:     if $t_l = \perp$ then $(g_l, g_r, t_l, t_r) \leftarrow (g_l, g_r, elapsedTime, t_r)$
10:    else if $t_r = \perp$ then $(g_l, g_r, t_l, t_r) \leftarrow (g_l, g_r, t_l, elapsedTime)$
11:    else
12:        if $t_l < t \wedge t_l \leq t_r$ then $(g_l, g_r, t_l, t_r) \leftarrow (g_l, \frac{g_l+g_r}{2}, t_l, t)$
13:        else if $t_r < t \wedge t_r < t_l$ then $(g_l, g_r, t_l, t_r) \leftarrow (\frac{g_l+g_r}{2}, g_r, t, t_r)$
14:        else $(g_l, g_r, t_l, t_r) \leftarrow (\frac{3g_l+g_r}{4}, \frac{g_l+3g_r}{4}, \perp, \perp)$
```

of tasks among the cores. We conducted the experiments on two different architectures: On an Intel Xeon Westmere-EX processor with four processors a ten cores and 20 threads at 2.4 GHz (called SuperMUC's fat node island), and on a BlueGene/P's quad core 850 MHz PowerPC (called Shaheen). Runtimes are always averaged over the whole simulation's execution time so far.

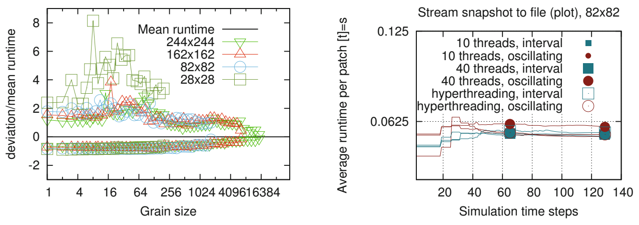


Fig. 3. The runtime measurements exhibit significant noise (left); the oracles identify sequential algorithm phases and switch off the multithreading (right)

For measurements as presented in Fig. 2, we observe that the data is instable (Fig. 3), and that the deviation is the bigger the smaller the grid. This holds for both codes and both machines. Consequently, it is important not to evaluate measurements directly in the oracle implementations but to work with averaged values. Our implementations consider a mean value to represent a valid runtime as soon as it does not change by more than 10^{-8} due to a new measurement. With this averaging, we observe, after a certain time, a good speedup on both the BlueGene/P machine and one Xeon, or we see that the oracle switches off the parallelisation if it does not yield a speedup due to a strictly sequential algorithm semantics (Fig. 3). Optimal speedup here is the best speedup obtained by a sampling over all grain sizes similar to Fig. 2. Hence, the following discussion concentrates on the startup phase and on more than one processor.

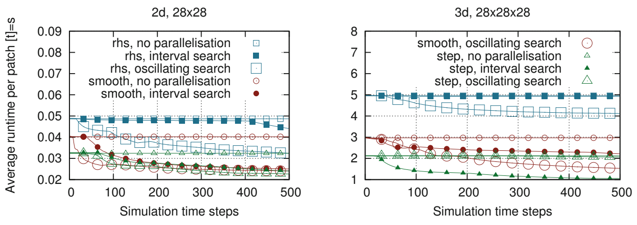


Fig. 4. Runtime evolvement for three phases of the CFD solver on Shaheen

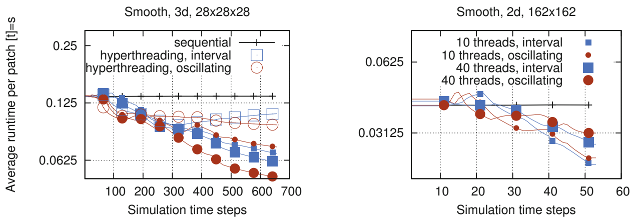


Fig. 5. Influence of hyperthreading and pinning on the performance of the first iterations of the diffusion equation solver's linear algebra phase (smooth)

When we study the CFD code and the diffusion equation solver on the BlueGene/P system running OpenMP, we observe that though the applications are different we can compare the individual algorithm phases with respect to their theoretical concurrency level. Plot operations, e.g., always are inherently sequential, while stencil evaluations on the same grid yield comparable results no matter what the stencil looks like. For the CFD code, we concentrate on three different scaling phases (rhs, smooth, plot). On the long term, both oracles converge to the same speedup. Both approaches are robust. However, the oracle behaviour differs at the application startup (Fig. 4). For most settings, the oscillating search identifies good grain sizes faster than the interval search analysing approximately two grain sizes per interval before the search continues on a subinterval. For 3d and a time-consuming non-linear operator evaluation (rhs) where the absolute runtime deviation also is rather big compared to the other measurements, the interval search does not even update the search grain size before the 500th solver time step. At the same time, the interval search proves to be more robust for phases which are embarrassingly parallel, i.e. where no sequential subparts of the code do exist (3d, phase step, around 50th time step).

On the Xeon, we are not able to exploit the hyperthreading facilities in our experiments (Fig. 5; left) for most problems. Furthermore, we observe issues with the thread pinning on this machine: As long as we restrict the codes to one processor (up to ten threads), the oracles identified good grain sizes after a startup phase (not illustrated here). If we use more than one processor, both the OpenMP and the TBB performance degenerate without a suitable pinning.

If the thread affinity is set to core (OpenMP) or an affinity partitioner is used (TBB), both shared memory paradigms yield comparable speedup. However, this speedup still suffers from the NUMA access and the work steeling overhead if the problems are too small (compare Fig. 5 left and right). Thread affinity issues cannot be resolved by the oracles (Fig. 5; left).

5 Summary and Outlook

Our approach selects reasonable grain sizes for all experiment settings. It also identifies regions where the parallelisation overhead eliminates any gain in performance—regions for which the code should fall back to a sequential implementation. The case study not only concentrate on individual parallel subproblems such as one data traversal with stencil computations but on the PDE solver as a whole. And it is totally autonomous, i.e. there are no magic configuration parameters. This makes the approach well-suited for heterogeneous supercomputers where the autotuning oracle runs on each computing node individually and for AMR codes where the spectrum of grain sizes alters permanently.

In future work, we will design better search algorithms based upon experience from other groups and upon performance models instead of observations. Furthermore, we will study the interplay of oracles, hardware, and PDE solvers in detail: it might be that the best results stem from search algorithms tailored to the PDE solver and the environment [4,17]. Also, the oracle mechanism should support implementational variants of one code fragment [5,14]. Of particular interest finally is the combination of shared with distributed memory parallelism and dynamic load balancing, i.e. cases where several cores handle several MPI processes. Here, randomised algorithms and algorithms noticing when the environment changes come into play.

Our approach can be interpreted as implementational pattern and could be integrated into standard shared memory toolkits such as TBB. An open issue however is to tackle the interplay of thread pinning or memory affinity, respectively, and the grain size. Our experiments reveal that the affinity of the threads is important, while TBB and OpenMP offer at most control over a static affinity or provide affinity as black box. Here, we see a need for more sophisticated and fine granular control for the high performance PDE solvers.

Acknowledgements. This publication is partially based on work supported by Award No. UK-c0020, made by the King Abdullah University of Science and Technology (KAUST). Computing resources for the present work have also been provided by the Gauss Centre for Supercomputing under grant pr63no.

References

1. Bilmes, J., Asanovic, K., Chin, C.-W., Demmel, J.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: Proceedings of the 11th International Conference on Supercomputing, pp. 340–347 (1997)

2. Cariño, R.L., Banicescu, I.: Dynamic Scheduling Parallel Loops With Variable Iterate Execution Times. In: 16th International Parallel and Distributed Processing Symposium (IPDPS 2002). IEEE (2002); electonical proceedings
3. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press (2007)
4. Cuenca, J., García, L.-P., Giménez, D.: A proposal for autotuning linear algebra routines on multicore platforms. Procedia CS 1(1), 515–523 (2010)
5. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 4:1–4:12. IEEE Press (2008)
6. Eckhardt, W., Weinzierl, T.: A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6067, pp. 567–575. Springer, Heidelberg (2010)
7. Gmeiner, B., Gradl, T., Köstler, H., Rüde, U.: Highly parallel geometric multigrid for hierarchical hybrid grids on Blue-Gene/P. Numer. Linear Algebr. (submitted)
8. Hummel, S.F., Schmidt, J., Uma, R.N., Wein, J.: Load-Sharing in Heterogeneous Systems via Weighted Factoring. In: Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 318–328. ACM (1997)
9. Kamil, S., Chan, C., Oliker, L., Shalf, J., Williams, S.: An auto-tuning framework for parallel multicore stencil computations. In: 24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, pp. 1–12. IEEE (2010)
10. Parker, S.G.: A component-based architecture for parallel multi-physics pde simulation. FGCS 22(1-2), 204–216 (2006)
11. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media (2007)
12. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 3(30), 202–210 (2005)
13. Tang, P., Yew, P.: Processor self-scheduling for multiple-nested parallel loops. In: Proceedings of the 1986 International Conference on Parallel Processing, pp. 528–535. IEEE (1986)
14. Vuduc, R.: Automatic assembly of highly tuned code fragments (1998), www.cs.berkeley.edu/~richie/stat242/project
15. Weinzierl, T.: A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids. Verlag Dr. Hut (2009)
16. Weinzierl, T., Köppl, T.: A geometric space-time multigrid algorithm for the heat equation. In: NMTMA (accepted)
17. Whaley, R.C., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. Parallel Comput. 27, 3–35 (2001)

GPU Acceleration of the Matrix-Free Interior Point Method

Edmund Smith, Jacek Gondzio, and Julian Hall

School of Mathematics and Maxwell Institute for Mathematical Sciences,
University of Edinburgh, JCMB, King's Buildings, Edinburgh,
EH9 3JZ, United Kingdom
J.A.J.Hall@ed.ac.uk

Abstract. The *matrix-free* technique is an iterative approach to interior point methods (IPM), so named because both the solution procedure and the computation of an appropriate preconditioner require only the results of the operations $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$, where A is the matrix of constraint coefficients. This paper demonstrates its overwhelmingly superior performance on two classes of linear programming (LP) problems relative to both the simplex method and to IPM with equations solved directly. It is shown that the reliance of this technique on sparse matrix-vector operations enables further, significant performance gains from the use of a GPU, and from multi-core processors.

Keywords: interior point methods, linear programming, matrix-free methods, parallel sparse linear algebra.

1 Introduction

Since they first appeared in 1984 [9], interior point methods (IPM) have been a viable alternative to the simplex method as a means of solving linear programming (LP) problems [14]. The major computational cost of IPM is the direct solution of symmetric positive definite systems of linear equations. However, the limitations of direct methods for some classes of problems have led to iterative techniques being considered [1,3,11]. The *matrix-free* method of Gondzio [5] is one such approach and is so named because the iterative solution procedure and the computation of a suitable preconditioner require only the results of products between the matrix of constraint coefficients and a (full) vector. This paper demonstrates how the performance of the matrix-free IPM may be accelerated significantly using a Graphical Processing Unit (GPU) via techniques for sparse matrix-vector products that exploit common structural features of LP constraint matrices. To the best of our knowledge this is the first GPU-based implementation of an interior point method.

Section 2 presents an outline of the matrix-free IPM that is sufficient to motivate its linear algebra requirements. Results for two classes of LP problems demonstrate its overwhelmingly superior performance relative to the simplex method and to IPM with equations solved directly. Further analysis shows that the computational cost of the matrix-free IPM on these problems is dominated by the iterative solution of linear systems of equations which, in turn, is dominated by the cost of matrix-vector products. Techniques for evaluating products with LP constraint matrices on multi-core CPU and many-core GPU are developed in Section 3. These techniques exploit commonly-occurring structural features of sparse LP constraint matrices. The results from an implementation with accelerated matrix-vector products show that significant speed-up in the overall solution time can be achieved for the LP problems considered, with the GPU implementation in particular providing large gains. Conclusions and suggestions for future work are offered in Section 4.

2 The Matrix-Free Interior Point Method

The theory of interior point methods [6,14] is founded on the following general primal-dual pair of linear programming (LP) problems.

$$\begin{array}{llll} & \text{Primal} & & \text{Dual} \\ & \min \boldsymbol{c}^T\boldsymbol{x} & & \max \boldsymbol{b}^T\boldsymbol{y} \\ & \text{s. t. } A\boldsymbol{x} = \boldsymbol{b} & & \text{s. t. } A^T\boldsymbol{y} + \boldsymbol{s} = \boldsymbol{c} \\ & \quad \boldsymbol{x} \geq \boldsymbol{0} & & \quad \boldsymbol{y} \text{ free}, \boldsymbol{s} \geq \boldsymbol{0}, \end{array} \tag{1}$$

where $A \in \mathbb{R}^{m \times n}$ has full row rank $m \leq n$, $\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{c} \in \mathbb{R}^n$ and $\boldsymbol{y}, \boldsymbol{b} \in \mathbb{R}^m$. IPMs employ logarithmic barrier functions to handle simple inequality constraints. The first order optimality conditions for the corresponding logarithmic barrier problems can be written as

$$\begin{aligned} A\boldsymbol{x} &= \boldsymbol{b} \\ A^T\boldsymbol{y} + \boldsymbol{s} &= \boldsymbol{c} \\ XS\boldsymbol{e} &= \mu\boldsymbol{e} \\ (\boldsymbol{x},\ \boldsymbol{s}) &\geq \boldsymbol{0} \end{aligned} \tag{2}$$

where X and S are diagonal matrices whose entries are the components of vectors $\boldsymbol{x}$ and $\boldsymbol{s}$ respectively and $\boldsymbol{e}$ is the vector of ones. The third equation $XS\boldsymbol{e} = \mu\boldsymbol{e}$ replaces the usual complementarity condition $XS\boldsymbol{e} = \boldsymbol{0}$ which holds at the optimal solution of (1). As μ is driven to zero in the course of a sequence of iterations, the vectors $\boldsymbol{x}$ and $\boldsymbol{s}$ partition into zero and nonzero components. In each IPM iteration, a search direction is computed by applying the Newton method to optimality conditions (2):

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^T & I_n \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \boldsymbol{\Delta x} \\ \boldsymbol{\Delta y} \\ \boldsymbol{\Delta s} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\xi_p} \\ \boldsymbol{\xi_d} \\ \boldsymbol{\xi_\mu} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b} - A\boldsymbol{x} \\ \boldsymbol{c} - A^T\boldsymbol{y} - \boldsymbol{s} \\ \mu\boldsymbol{e} - XS\boldsymbol{e} \end{bmatrix}. \tag{3}$$

By using the sets of equations in (3) to eliminate first $\boldsymbol{\Delta s}$, and then $\boldsymbol{\Delta x}$, the following symmetric positive definite **normal equations** system is obtained

$$(A\Theta A^T)\boldsymbol{\Delta y} = \boldsymbol{g}, \tag{4}$$

where $\Theta = XS^{-1}$ is a diagonal matrix. Since the normal equations matrix $A\Theta A^T$ is symmetric and positive definite, its LL^T Cholesky decomposition may be formed. In IPM, this is the usual means of solving directly for $\boldsymbol{\Delta y}$ and hence, by reversing the elimination process, $\boldsymbol{\Delta x}$ and $\boldsymbol{\Delta s}$. However, the density of $A\Theta A^T$ may be significantly higher than A, and the density of L may be higher still. For some large LP problems, the memory required to store L may be prohibitive. Following [6] test problems which exhibit this behaviour are given in Table 1. The first two problems are larger instances of quadratic assignment problems (QAP) [10] whose solution is one of the great challenges of combinatorial optimization. The remaining problems are part of a calculation from Quantum Physics of non-classicality thresholds for multiqubit states, and were provided to us by Jacek Gruca [7]. As problem size increases, the memory requirement of the Cholesky decomposition prevents them from being solved via standard IPM and the simplex method is seen not to be a viable alternative.

Table 1. Prohibitive cost of solving larger QAP problems and qubit problems using Cplex 11.0.1 IPM and dual simplex

	Dimensions			IPM		Simplex
Problem	Rows	Columns	Nonzeros	Cholesky Nonzeros	Time	Time
nug20	15,240	72,600	304,800	38×10^6	1034 s	79451 s
nug30	52,260	379,350	1,567,800	459×10^6	OoM	>28 days
1kx1k0	1,025	1,025	34,817	0.5×10^6	0.82 s	0.38 s
4kx4k0	4,097	4,097	270,337	8×10^6	89 s	11 s
16kx16k0	16,385	16,385	2,129,921	128×10^6	2351 s	924 s
64kx64k0	65,537	65,537	16,908,289	2048×10^6	OoM	111 h

For some LP problems the constraint matrix may not be known explicitly due to its size or the nature of the model, but it may nonetheless be possible to evaluate $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. Alternatively, for some problems there may be much more efficient means of obtaining these results than evaluating them as matrix-vector products. For such problems, Gondzio [5] is developing *matrix-free* IPM techniques in which systems of equations are solved by iterative methods using only the results of $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. However, the present work is concerned with LPs for which which A is known explicitly but solution via standard IPM and the simplex method is impractical. This is the case for the problems given in Table 1.

Since the normal equations matrix $A\Theta A^T$ is symmetric and positive definite, the method of conjugate gradients can, in theory, be applied. However, its convergence rate depends on the ratio between the largest and smallest eigenvalues of $A\Theta A^T$, as well as the clustering of its eigenvalues [8]. Recall that since there will be many indices j for which only one of x_j and s_j goes to zero as the optimal solution is approached, there will be a very large range of values in Θ. This ill-conditioning means that conjugate gradients is unlikely to converge. Within matrix-free IPM, the ill-conditioning of $A\Theta A^T$ is addressed in two ways: by modifying the standard IPM technique and by preconditioning the resulting normal equations coefficient matrix.

The optimization problem is regularized by adding quadratic terms $\boldsymbol{x}^T R_p \boldsymbol{x}$ and $\boldsymbol{y}^T R_d \boldsymbol{y}$ to the primal and dual objective in (1), respectively. Consequently, the matrix in the normal equations (4) is replaced by

$$G_R = A(\Theta^{-1} + R_p)^{-1} A^T + R_d, \tag{5}$$

in which R_p guarantees an upper bound on the largest eigenvalue of G_R and R_d guarantees that the spectrum of G_R is bounded away from zero. Therefore, for appropriate R_p and R_d the condition number of G_R is bounded regardless of the conditioning of Θ.

The convergence properties of the conjugate gradient method are improved by applying a preconditioner P which approximates the partial Cholesky decomposition of G_R

$$G_R = \begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D_L & \\ & S \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ & I \end{bmatrix} \approx P = \begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D_L & \\ & D_S \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ & I \end{bmatrix}. \tag{6}$$

Namely, in the preconditioner P, the Schur complement S is replaced with its diagonal D_S.

The number of nontrivial columns in the preconditioner is $k \ll m$ so, since only the diagonal entries of S are ever computed, the preconditioner is vastly cheaper to compute, store and apply than the complete Cholesky decomposition. Each iteration of the preconditioned conjugate gradient (PCG) method requires one operation with both P^{-1} and G_R. Since D_L, D_S, Θ, R_p and R_d are all diagonal matrices, the major computational costs are the operations with the nontrivial columns of P and the matrix-vector products with A and A^T. It is seen in Table 2 that the cost of PCG dominates the cost of solving the LP problem, and that PCG is dominated by the cost of operating with P^{-1} and calculating $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$. For the QAP problems the cost of applying the preconditioner is significant, but for the quantum physics problems the cost of the

Table 2. Proportion of solution time accounted for by preconditioned conjugate gradients, operations with P^{-1} and calculations of $A\boldsymbol{x}$ and $A^T\boldsymbol{y}$

	Percentage of solution time			
Problem	PCG	P^{-1}	$A\boldsymbol{x}$	$A^T\boldsymbol{y}$
nug20	89	55	17	15
nug30	90	54	18	17
1kx1k0	62	41	12	11
4kx4k0	89	42	19	28
16kx16k0	87	30	30	29
64kx64k0	87	19	37	34

matrix-vector products dominates the solution time for the LP problem, and this effect increases for larger instances. Section 3 considers how the calculation of $\boldsymbol{A x}$ and $A^T \boldsymbol{y}$ may be accelerated by exploiting a many-core GPU and multi-core CPU.

3 Accelerating Sparse Matrix-Vector Products

The constraint matrix of a large LP problem is usually sparse and structured, and any competitive routine must take advantage of this fact. Sparse arithmetic presents different challenges as compared to dense arithmetic. Although sparse matrices of similar size to dense matrices can be multiplied much more quickly, it is in general the case, on both contemporary CPUs and GPUs, that the rate at which floating point operations are performed is lower; there are simply far fewer such operations.

We will consider the acceleration of the following operations

$$\boldsymbol{y} = A\boldsymbol{x} \quad (\textsc{fsax}), \quad \boldsymbol{x} = A^T\boldsymbol{y} \quad (\textsc{fsaty}), \quad \boldsymbol{z} = (A\Theta A^T)\boldsymbol{y} \quad (\textsc{fsaat}) \tag{7}$$

where $A \in I\!R^{m\times n}$ is sparse, $\Theta \in I\!R^{n\times n}$ is a diagonal matrix and, and the vectors $\boldsymbol{x} \in I\!R^n$, $\boldsymbol{y} \in I\!R^m$ and $\boldsymbol{z} \in I\!R^m$ are dense.

Memory bandwidth is a critical bottleneck for current generation CPUs. To alleviate this, high-speed cache is available to store recently used, or soon to be used, data. Caching is most beneficial when data items are re-used multiple times in quick succession. For operations requiring $O(n^3)$ operations on $O(n^2)$ data, such as matrix-matrix multiply, correct use of cache can give significant gains. In the case of matrix-vector multiply, there are only $O(n^2)$ operations: each data item is used once. Thus raw bandwidth determines performance on a large dataset, and caching may be expected to be mostly ineffective.

Vectorisation can give a significant performance gain for dense arithmetic on modern CPUs: identical operations are performed in parallel on a bank of data items (two at once for SSE2; four at once for AVX). Unfortunately, in sparse matrix-vector products, the elements of the vector corresponding to the non-zeroes in a given matrix row are widely separated in memory, and this makes efficient vectorisation difficult (in our tests, the packing costs outweighed the gains).

It has become typical for a CPU to contain multiple cores, independent execution units with some mutual dependencies, for example shared memory bandwidth and some shared cache. For a task like sparse matrix-vector multiply, which can be easily divided into a small number of independent pieces, there are no great problems with exploiting multiple cores. The effectiveness of the result though is less certain, given the essentially bandwidth limited nature of the problem in the first place. It should be noted that we shall be using a dual CPU system, so that double the bandwidth is available when all cores are used.

Current GPUs have large numbers of execution contexts, called threads, each of which is significantly slower and less able than a CPU core. A GPU benefits from potentially better memory performance and an explicitly managed cache, but that memory performance depends critically on achieving *coalesced* accesses: adjacent threads must read adjacent memory locations. The challenge in mapping sparse matrix-vector products to a GPU, then, is predominantly in arranging for the task to be broken down into many small ones, and for those threads to access memory appropriately to preserve performance.

3.1 GPU Kernels

A number of kernels have been proposed in the literature for sparse matrix-vector products [2,4,12]. We summarise some of the key ideas below.

Threads on a GPU can only synchronise in a limited context - synchronisation within a warp is guaranteed, synchronisation within a block can be arranged. This means that any one element of the final result must be calculated by no more than a single block if the answer is to achieved without running the kernel multiple times. In the context of sparse matrix-vector products, this means each entry of the result vector may be calculated by at most one block.

If each thread calculates a row, then the data must be laid out so that the data for neighbouring rows are interleaved: this will mean adjacent threads read adjacent memory locations at each time step. This organisation has been called ELLPACK [2]. In practice, having each thread calculate a row under-utilises the device: insufficient parallelism is being identified.

An entire warp (thirty-two threads) can be used to calculate a row by first performing all the multiplications and storing the result in cache (shared memory), then performing a parallel reduce. A warp is automatically synchronised so there is no synchronisation overhead in this algorithm. If this is done with variable run-length storage of the rows, the result is vector CSR [2].

When the lengths of the rows vary considerably, it can be a problem both for load balancing, and for memory requirements. Pure ELLPACK is not practical for matrices with dense rows. ELLR-T [12] can eliminate some of this inefficiency by storing the length of each row, but the need to reserve enough memory for the matrix to be stored as fully dense remains. The HYB [2] kernel overcomes this limitation by storing a core of the problem as ELL, and any extra elements in long rows as COO (unstructured, sparse). Unfortunately, COO is not an especially fast kernel.

If the constraint matrix has dense blocks which can be identified, data blocking can give significant speedup [4].

Our target problems have rows and columns of mostly identical length, barring a fully dense row and a fully dense column. They do not lend themselves to data blocking. The kernels discussed below were optimized for these problems.

We considered three families of kernel. Firstly, dense-hybrid ELL (DHELL) in which dense rows are extracted for treatment by a block of their own, and the remainder are stored in ELL format. Secondly, vector CSR as discussed by [2]. Finally, dense-hybrid transpose ELL (DHTELL), in which the matrix of indices and coefficients is transposed relative to that encountered in ELL. It can be seen also as CSR with a fixed row length. This kernel is novel.

As for [12], we considered using different numbers of threads per row. The ELL format (equivalently, ELLR-T) hampers such explorations, because an entire half-warp must read adjacent memory locations. Thus the maximum number of threads per row is the same as the number of half-warps per block, usually sixteen. For both CSR and TELL formats, an entire block can be brought to bear on a row (256 threads), but the minimum number of threads per row falls to sixteen.

The best performing kernel was DHTELL with a half-warp per row (sixteen threads) and a block devoted to any dense part. Although this level of parallelism could be matched in the DHELL kernel, the former was marginally faster. There is little to choose between any of the vectorised kernels, when compared to the basic CSR or ELL kernels.

Note that the constraint matrix is stored twice on the GPU, once row-wise and once column-wise, to allow operations with the transpose. Any naïve implementation of the alternative would require impractical amounts of memory in which to accumulate partial results.

The only significant optimization for this platform which has not been considered, that we are aware of, is use of the texture cache to store the input vector. Results presented for band diagonal matrices in [2] suggest this as a possible future enhancement.

3.2 Results

The following results are obtained from a test system having two AMD Opteron 2378 (Shanghai) quad-core processors, 16 GiB of RAM and a Tesla C2070 GPU with 6 GiB of RAM. Note that the processors are relatively slow in serial, though the NUMA configuration of the memory bus gives high parallel memory performance. The GPU is a significantly more highly powered unit, making raw speed characterisations of less interest than the potential for improvement with a given investment.

Table 3. Comparison of accelerated matrix-free IPM codes. All times include data transfer.

	Solve time (s)			SpMV time (s)		
Problem	Serial	8 core	GPU	Serial	8 core	GPU
nug20	2.19	1.18	1.60	1.49	0.495	0.945
nug30	20.5	15.8	15.4	15.1	9.69	9.45
1kx1k0	0.244	0.177	0.217	0.0360	0.0128	0.0506
4kx4k0	3.03	2.06	2.15	1.06	0.218	0.336
16kx16k0	24.9	18.4	13.5	13.1	6.70	1.72
64kx64k0	170.0	109.0	74.0	115.0	47.4	12.2
96kx128-0	137.0	71.1	58.8	93.4	28.6	15.3
256x256-0	866.0	283.0	222.0	699.0	119.0	56.4

Speed-up of sparse matrix-vector kernels using all eight cores of the test system is between two and six times, giving at most a threefold speed-up of the IPM solution time. Using the high powered GPU, speed-up of these same kernels can approach ten times, though overall solution time is reduced by no more than a factor of four. Clearly significant speed-up of matrix-free interior point, whether by many-core or multi-core parallelism, is possible.

4 Conclusions

The matrix-free approach shows promise in making some of the most difficult classes of problem tractable by interior point methods. Its focus on a small core of sparse operations makes highly optimized implementations using state of the art hardware possible without excessive difficulty.

The particular choice of many-core or multi-core acceleration depends on the hardware available. As has been noted elsewhere [13], a GPU can provide performance essentially equivalent to a small number of multi-core processors in the context of sparse problems.

References

1. Al-Jeiroudi, G., Gondzio, J., Hall, J.: Preconditioning indefinite systems in interior point methods for large scale linear optimization. Optimization Methods and Software 23(3), 345–363 (2008)
2. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. Tech. Rep. NVR-2008-004, NVIDIA Corporation (2008)
3. Bergamaschi, L., Gondzio, J., Zilli, G.: Preconditioning indefinite systems in interior point methods for optimization. Computational Optimization and Applications 28, 149–171 (2004)
4. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 115–126. ACM (2010)
5. Gondzio, J.: Matrix-free interior point method. Computational Optimization and Applications, published online October 14 (2010), doi:10.1007/s10589-010-9361-3
6. Gondzio, J.: Interior point methods 25 years later. European Journal of Operational Research, published online October 8 (2011), doi:10.1016/j.ejor.2011.09.017
7. Gruca, J., Wiesław, L., Żukowski, M., Kiesel, N., Wieczorek, W., Schmid, C., Weinfurter, H.: Nonclassicality thresholds for multiqubit states: Numerical analysis. Physical Review A 82 (2010)
8. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. J. Res. Natl. Bur. Stand 49, 409–436 (1952)
9. Karmarkar, N.K.: A new polynomial–time algorithm for linear programming. Combinatorica 4, 373–395 (1984)
10. Nugent, C.E., Vollmann, T.E., Ruml, J.: An experimental comparison of techniques for the assignment of facilities to locations. Operations Research 16, 150–173 (1968)
11. Oliveira, A.R.L., Sorensen, D.C.: A new class of preconditioners for large-scale linear systems from interior point methods for linear programming. Linear Algebra and its Applications 394, 1–24 (2005)

12. Vázquez, F., Ortega, G., Fernández, J., Garzón, E.: Improving the performance of the sparse matrix vector product with GPUs. In: 2010 10th IEEE Conference on Computer and Information Technology (CIT 2010), pp. 1146–1151 (2010)
13. Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M., Shringapure, A.: On the limits of GPU acceleration. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism. USENIX Association (2010)
14. Wright, S.J.: Primal-Dual Interior-Point Methods. SIAM, Philadelphia (1997)

Deconvolution of 3D Fluorescence Microscopy Images Using Graphics Processing Units

Luisa D'Amore[1], Livia Marcellino[2], Valeria Mele[3], and Diego Romano[4]

[1] University of Naples Federico II,
Complesso Universitario M.S. Angelo, Via Cintia, 80126 Naples, Italy
luisa.damore@unina.it

[2] University of Naples Parthenope,
Centro Direzionale, Isola C4, 80143 Naples, Italy
marcellino@uniparthenope.it

[3] University of Naples Federico II,
Complesso Universitario M.S. Angelo, Via Cintia, 80126 Naples, Italy
valeria.mele@unina.it

[4] ICAR - CNR, Via Pietro Castellino 111, Naples, Italy
diego.romano@na.icar.cnr.it

Abstract. We consider the deconvolution of 3D Fluorescence Microscopy RGB images, describing the benefits arising from facing medical imaging problems on modern graphics processing units (GPUs), that are non expensive parallel processing devices available on many up-to-date personal computers. We found that execution time of CUDA version is about 2 orders of magnitude less than the one of sequential algorithm. Anyway, the experiments lead some reflections upon the best setting for the CUDA-based algorithm. That is, we notice the need to model the GPUs architectures and their characteristics to better describe the performance of GPU-algorithms and what we can expect of them.

1 Introduction

Graphics Processing Units (GPUs) emerge nowadays like a solid and compelling alternative to traditional computing, delivering extremely high floating point performance at a very low cost. In this paper we focus on medical imaging applications, namely the deconvolution of 3D Fluorescence Microscopy images.

A fluorescence microscope is a light microscope used to study properties of organic or inorganic substances. Deconvolution is an operation that mitigates the distortion created by the microscope. Indeed, the application of image deconvolution to microscopy was stimulated by the introduction of 3D fluorescence microscopy [1].Deconvolution problem is a inverse and ill-posed, hence regularization methods are due to provide a solution within a reasonable accuracy [6,7,12,13]. A recent review of deconvolution methods in 3D fluorescence microscopy is given in [11].

We obtain good performance using GPUs and our experiments lead reflections upon the execution time expression and the opportunity of further work about its modelling.

The paper is organized as follows: in section 2 we describe the inverse problem of deconvolution and the Lucy Richardson algorithm while in section 3 we introduce the GPU computing environment, discuss the CUDA-based algorithm, and make some considerations about the performance analysis from a theoretical point of view. Section 4 concludes the paper.

2 Inverse Problem and Numerical Approach

2.1 The Problem and Its Discretization

We represent images by a lowercase letter, e.g., $f(x, y, x)$, where (x, y, z) is a position vector locating an arbitrary point in image space, and it is:

$$f : (x, y, z) \in \Omega \subset \Re^3 \to \Re$$

Given the Fourier transform of f, $\Im(u, v, w)$, knowing the related *Convolution Theorem* and *Correlation Theorem* , and assuming isoplanatism (spatial invariance), the fluorescence microscope is modeled as a linear shift-invariant system, due to the incoherent nature of the emitted fluorescence light. In that case, the 3-D image is represented by the convolution operation between the PSF (Point Spread Function) $h(x, y, z)$ of the fluorescence microscope and the 3-D specimen of interest $f(x, y, z)$. This means that the imaging process can be expressed as:

$$g(x, y, z) = f(x, y, z) \otimes h(x, y, z) \equiv$$

$$\equiv \int\int\int f(x', y', z')h(x - x', y - y', z - z')dx'dy'dz' \tag{1}$$

where $g(x, y, z)$ is the noisy measured data and $\otimes$ is the convolution product. In reality, our observed data $g(x, y, z)$ are given by

$$g(x, y, z) = \widetilde{g}(x, y, z) + \eta(x, y, z)$$

where a noise term $\eta(x, y)$ has been added to the underlying convolution to represent the errors present in real world measurements. The problem of deconvolution is the inverse problem of recovering $f(x, y, z)$ from $g(x, y, z)$ and $h(x, y, z)$.

Sampling the three-dimensional function $f(x, y, z)$ via an uniform grid in space Ω, it results in the P vector $f(i, j, k)$, where: $i = 0, ..., L-1$, $j = 0, ..., M-1$ and $k = 0, ...N - 1$, $P = L \times M \times N$. Concerning the PSF function, h, as is usual we assume that it is normalized to 1, that is the matrix H, discretization of h, is such that:

$$\sum_{i,j} H_{i,j} = 1$$

From these notations, we can consider the three-dimensional discrete Fourier transform (DFT) and inverse discrete Fourier transform (IDFT). Sampling both sides of (1) we obtain:

$$g = Hf \tag{2}$$

where g is the P vector of the measured data, H is the $P \times P$ blurring matrix, i.e., sampled PSF, and f is the P vector form of the discrete object. In general, the operator H is normalized to 1. Assuming fluorescence microscope to be shift-invariant for the deconvolution operation, H has a block-circulant form, and its eigenvalues are given by the discrete Fourier transform (DFT) of the sampled PSF. In addition, the matrix multiplication can be efficiently represented by point-by-point multiplication in the Fourier domain.

2.2 The FFT-ARL Algorithm

Due to the ill posedness of deconvolution problem, let us rewrite the discrete problem (2) as a least square approximation problem, where, taking into account that data are affected by photon noise described as a Poisson process on g, we use the Csiszar [5] distance (or I-divergence) to replace the metric distance that appears in the standard least square approach used in presence of Gaussian noise. This lead to the Expectation-Minimization (EM) algorithm[14] or, equivalently, the Lucy -Richardson (LR) algorithm [8,10]:

$$f^{k+1} = f^k H^T \frac{g}{Hf^k}, \quad k = 0, 1, ...$$

starting from $f_0 > 0$. LR can be interpreted as a **scaled gradient projected** (SGP) algorithm with constant step length and projection matrix and a special scaling matrix:

$$f^{k+1} = P_k(f^k - \alpha_k D_k \nabla J(f^k))$$

where[1] $\alpha_k = 1, \quad D_k = X_k \quad P_k = 1.$

α_k is the step length and D_k is the scaling diagonal matrix. Gradient-projection-type methods seem appealing approaches for these problems because the nonnegativity constraint, that characterizes most image restoration applications, makes the projection a non expensive operation [4].

The Accelerated LR algorithm (ARL) is the following:

$$f_{k+1} = [f_k + \alpha_k(f_k - f_{k-1})] \left[h \star \frac{g}{h \otimes [f_k + \alpha_k(f_k - f_{k-1})]} \right] \tag{3}$$

where there are two basic operations at each iteration: convolution and correlation. Therefore, by applying the Fourier transform to both sides, the FFT-based ARL can be written as:

$$f_{k+1} = \overline{f}_k \cdot \Im^{-1} \left[H^\star \cdot \Im \left(\frac{g}{\Im^{-1}[H \cdot \overline{F}_k]} \right) \right] \tag{4}$$

[1] A gradient descent algorithm computes a zero of a function f as the limit of $f^{k+1} = f^k + \alpha_k p_k \quad (p_k = -\nabla J(f_k))$ moving along the steepest direction, i.e. along the negative gradient direction. If D is a symmetric positive definite matrix and Ω is a subset of $\Re^n$ the projection $P_{\Omega,D}$ of $f \in \Re^n$ onto Ω is:

$$P_{\Omega,D}(x) = argmin_{y \in \Omega} \|y - x\|_D = argmin_{y \in \Omega} (y - x)^T D(y - x)$$

where $\overline{f}_k = f_k + \alpha_k(f_k - f_{k-1})$, $\overline{F}_k$ is the Fourier transform of function $\overline{f}_k$. Finally, $\Im$ and $\Im^{-1}$ are the *Fourier transform* operator and *Inverse Fourier transform* operator, respectively.

Moreover, we employ the *zero padding* technique to, as explained before, reduce the errors on the deconvolved image and to decompose the vectors among the computing nodes of the GPU: in such a way we may use radix-2 FFT algorithms.

We use the software library FFTW (*Fastest Fourier Transform in the West*), as proposed in [9]. FFTW is known as the fastest software implementation of the Fast Fourier transform (FFT) algorithm, written in the C language: it can be used to compute DFT and IDFT in only $O(PlogP)$ operations [3]. So, using FFT algorithm the computational cost of accelerate EM-RL method is $O(P+PlogP)$. As stopping criterium we use both the maximum number of iterations, that is used just to control the algorithm's robustness, and the minimum residual.

3 The GPU Algorithm

We now describe the implementation of ARL algorithm an a Tesla C1060. Our algorithm has been implemented in C with CUDA extension, so it was possible to run it on several NVIDIA cards to analyse the efficiency.

In table 1 specifications of hardware development/testing environment.

Table 1. Hardware and Software specifications

Hardware

Processor	Intel(R) Core(TM) i7 CPU 950 (3.07GHz)
RAM	6 2 GB DRR3 1333 DIM
Hard disk	2 500 GB SATA,16 MB cache, 7.200 RPM
GPU	1 Quadro FX5800 4 GB RAM 2 Tesla C1060 4 GB RAM

Tesla C1060 details

Streaming Multiprocessors (SM)	30
Streaming Processor (SP) cores	240
SP core Frequency	1300 MHz
Memory Bandwidth	102 GB/2

Let suppose we have enough resources on the GPU to store the input image, the PSF, two consequent iteration estimates, and all the temporary arrays. We can implement a di fferent parallel kernel (each followed by a synchronization barrier) for each step of the ARL algorithm: when the parallel execution of a kernel ends, the overall control returns to the CPU. The macro-operations that can be implemented in parallel are:

– Entry-wise matrix product
– Entry-wise matrix ratio
– Conjugate of complex matrix

– Multiply-add combination of scalar-matrix and matrix-matrix (Entry-wise) operations
– Calculation of acceleration parameter
– DFT and Inverse DFT.

Drawing apart the DFT, all the other macro-operations have a straight-forward implementation in parallel on GPUs. In fact they consist in entry-wise calculations that can be easily separated in different independent threads. About the DFT calculation, we utilize the CUDA optimized CUFFT library [15], that has been modelled after FFTW [9]. It exploits a configuration mechanism called plan which completely specifies the optimal configuration for a specific FFT. Once the plan is created, it can be used for multiple FFT calculations of the same type without requiring new configurations. This approach perfectly suits our ARL algorithm, where several plans can be created once and then used at each iteration. In this framework, the CPU works as a high level macroscopic control unit which submit kernel executions, synchronizes the macro-operations, executes the control flow directives, while data persist on device memory during the overall execution.

3.1 Experiments

The ALR algorithm was stopped when the I-divergence reaches the minimum value. This stopping criterium is usual for those iterative algorithms employed to solve ill posed problems, because they suffer from the so-called *semiconvergence* behavior. We report results on a real 3D RGB images.

Let us consider the image of an embryo C. Elegans. In fig. 1 we show on the first row three slices of the image to be restored, on the second row the restored images, obtained after 9 itrations of GPU-based ARL algorithm, are shown. Fig. 2 show the residuals behavior. Concerning the PSF, it was defined following characteristics of fluorescence microscopy.

Then, when looking at fig.3 it's easy to notice that most of the execution time is devoted to DFT and IDFT calculation.

But looking at this numbers, we want to understand what is the best setting for our algorithms. That is, we have to define some performance analysis parameters that can fit the GPU-enhanced enviroments peculiarities.

3.2 Performance Analysis Considerations

We notice the need to model the GPUs architectures and their characteristics to describe the behavior of GPU-algorithms and what we can expect of them. Let us introduce some preliminaries (see [16] for more details).

Definition 1. *Given any algorithm A, its execution time on a processing unit is $T_s(A, N)$, where N is the problem dimension, and it is always decomposable as*

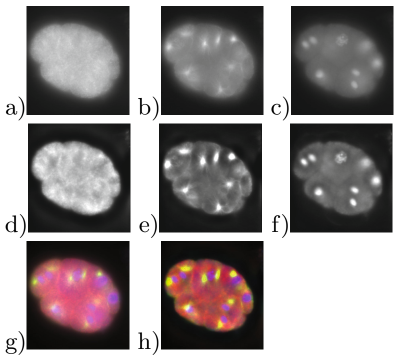


Fig. 1. Degraded and restored images. Images size is 256 × 271 × 103. a), b) e c) show three black and white bi dimensional degraded slices at $z = 49$, related to the three florescence emissions (CY3, FITC, DAPI) respectively, d) e) f) show the restored images, g) is the RGB sum of a) b) and c), while h) is the RGB sum of d), e) and f).

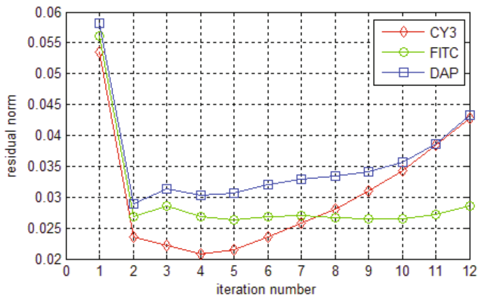


Fig. 2. Test1: ARL residuals behavior corresponding to CY3, FITC, DAPI, respectively. According to the semiconvergence of ARL algorithm, the minimum is reached at iterations 4, 5 and 4, respectively.

– *T_{seq} execution time of A_{seq} that is the part that must be executed sequentially*
– *T_{par} execution time of A_{par} that is the part that can be decomposed in a number of components to execute concurrently*

So $T_s(A, N) = T_{seq} + T_{par}$.

Definition 2. *If the parallel system has P processing units, and the work of A_{par} is welll balanced among its P components, the execution time of A on that system is*

$$T_c(A, P, N) = T_{seq}(A, N) + \frac{T_{par}(A, N)}{P} + T_O(A, P, N) \tag{5}$$

where $T_O(A, P, N)$ is the total overhead, $T_O(A, 1, N) = 0$ and $T_c(A, 1, N) = T_s(A, N)$.

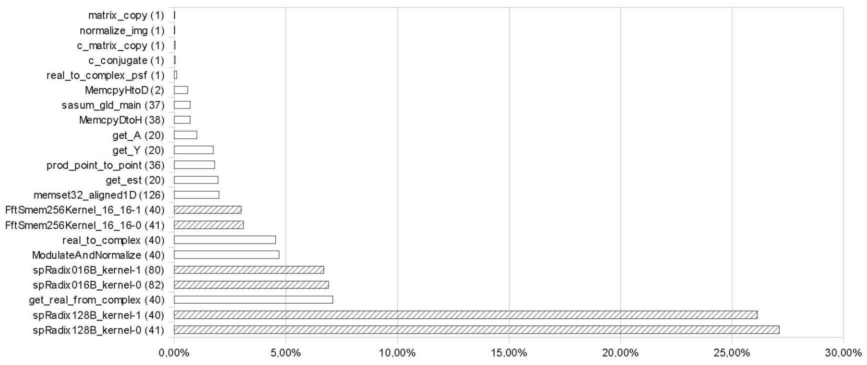


Fig. 3. Percentage of kernel execution time over total time, per kernel. Kernels using CUFFT are marked with stripes. In brackets the number of executions for a deconvolution with a 20 iterations loop.

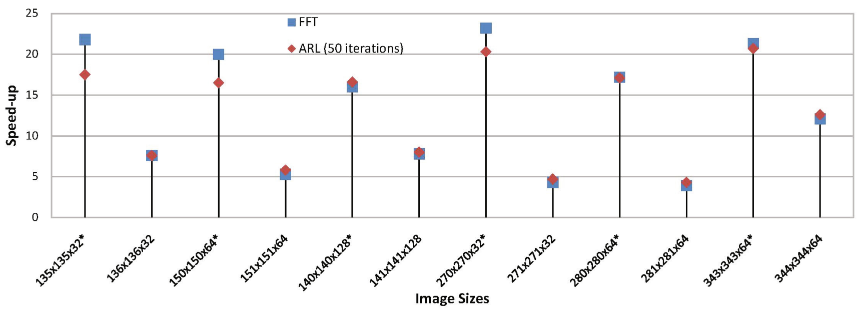


Fig. 4. Speed-up of ARL algorithm, as defined by Definition 4. Sequential FFT has been implemented utilizing FFTW. Image sizes marked with "*" are factorizable as $2^a \cdot 3^b \cdot 4^c \cdot 5^d$.

Of course the work of any algorithm includes mainly two kind of operations: *floating point operations* and *memory accesses*. So we can give also the next:

Definition 3. *Given any algorithm A, its sequential execution time can be written as*

$$T_s(A,N) = T_{s[flop]}(A,N) + T_{s[mem]}(A,N) \tag{6}$$

Given any algorithm A, its parallel execution time can be written as

$$T_c(A,P,N) = T_{c[flop]}(A,P,N) + T_{c[mem]}(A,P,N) + T_O(A,P,N) \tag{7}$$

where

- $T_{*[flop]}$, *is the time spent in floating point operation, with the related latency,*
- $T_{*[mem]}$, *is the time spent in memory accesses, with the related latency.*

Let's now suppose that the parallel part of A, A_{par}, is executed by p sets of q thread, on a GPU-based computing architecture as described in [16], made of P Multiprocessors (MP), and Q ALUs per MP. Let's call *warp* the execution unit on that machine, that is the fixed number of threads running simultaneously on the ALUs of each MP at the same time. Let be $dimW$ the dimension of the warp and q multiple of $dimW$. On that kind of architecture we have:

Definition 4 (Occupancy). *At a given instant, the occupancy of each MP is a function of the number of thread running concurrently on that MP, say $p \cdot q$ and is defined as*

$$\vartheta(A_{par},q,p) = \frac{\#active\_warps\_block}{\#max\_warps\_per\_MP} =$$

$$= \frac{q}{dimW} \cdot \frac{1}{\#max\_warps\_per\_MP} \cdot \#active\_blocks\_MP(A_{par},q,p) \tag{8}$$

where

- *$\#max\_warps\_per\_MP$ depends on hardware*
- *$\#threads\_per\_warp$ depends on hardware*
- *$\#active\_blocks\_MP = min(p, \#max\_blocks\_MP(A_{par})$*
- *$\#max\_blocks\_MP(A_{par},q)$ depends on hardware but also on the algorithm characteristic, as the local memory management (*shared memory *and* registers*) and on the number of threads q.*

Then, we get the following result [16]:

Proposition 1. *The expected total execution time of a parallel algorithm Par_A designed to run on the above architecture by p sets of q thread, posed $nw = \lceil \frac{p}{P} \rceil \cdot \lceil \frac{q}{dimW} \rceil = \#active\_warps\_MP$, could be written as follows:*

$$T_c(A_{par},p,q,N) = \frac{\sum_{i=0}^{nw-1} \left(T_{c[flop]}(warp_i, dimW, N_i)\right)}{\varphi_1(\vartheta(A_{par},q,p))} +$$

$$+ \frac{\sum_{i=0}^{nw-1} \left(T_{c[mem]}(warp_i, dimW, N_i)\right)}{\varphi_2(\vartheta(A_{par},q,p))} + T_O(A_{par},p,q,N) \tag{9}$$

where

- $T_{c[flop]}(warp_i, dimW, N_i)$ *is the execution time spent in floating point operations by the* i^{th} *warp, on each MP,*
- $T_{c[mem]}(warp_i, dimW, N_i)$ *is the execution time spent in memory accesses by the* i^{th} *warp, on each MP,*
- $\varphi_1(\vartheta(A_{par}, q, p))$ *is a function of the occupancy that expresses the hiding of the aritmetic pipeline latency by by concurrent work,*
- $\varphi_2(\vartheta(A_{par}, q, p))$ *is a function of the occupancy that expresses the hiding of the memory latencies by concurrent work,*
- $T_O(A_{par}, p, q, N)$ *is the cost of kernel launch, host/device data transfers, synchronization, divergences and data non-coalescence.*

So, on the above GPU-based computing architecture, the key concepts seem to be the *occupancy* and the amount of indipendent operations (of each kind) of each thread: they are both quantities that we can control, given the hardware characteristics.

In case of the FFT-ARL algorithm, suitable strategies have been implemented to keep the overhead T_O low for each kernel, also keeping memory coalescence under control (using one dimensional vectors and CUDA optimized algorithms for vector operation, i.e. CUBLAS [15] routines).

Fig.4 shows that running parallel ARL on GPU leads to smaller speed-ups if the image sizes are not factorizable with primes, that is the condition to obtain the best accuracy and performance in FFT algorithm of the CUFFT package.

4 Conclusions

We described the benefits arising from facing medical imaging problems on GPUs. The algorithm implemented reaches good performance on GPUs because many of the steps in the sequential algorithm consist in entry-wise matrix operations (embarrassingly parallel): in our experiments we found that execution time of CUDA version is about 2 orders of magnitude less than the one of sequential algorithm. Analysing the execution time from a theoretical point of view we note that there are many different levels of parallelism to exploit and we can say that the right setting for the algorithm heavily depends on the hiding of the aritmetic pipeline latency, of the memory latencies and on the cost of kernel launch, host/device data transfers, synchronization, divergences and data non-coalescence. The experiments lead reflections upon the execution time expression and the opportunity of further work about its modelling.

Acknowledgement. We would like to thank Prof. Giovanni Paolella, Dipartimento di Biochimica e Biotecnologie Avanzate, University of Naples Federico II, and CEINGE, Biotecnologie Avanzate (Naples, Italy) for posing us the deconvolution problem and providing the images.

References

1. Agard, D.A., Hiraoki, Y., Sedat, J.W.: Three-dimensional microscopy: image processing for high-resolution subcellular imaging. In: Proc. SPIE, vol. 1161, pp. 24–30 (1989)
2. Biggs, D.S.C., Andrews, M.: Acceleration of iterative image restoration algorithms. Applied Optics 36(8), 1766–1775 (1997)
3. Brigham, E.O.: The Fast Fourier Transform and its application. Prentice Hall Signal Processing Series Alan V. Oppenheim, Series Editor
4. Bonettini, S., Zanella, R., Zanni, L.: A scaled gradient projection method for constrained image deblurring. Inverse Problems 25, 015002 (2009)
5. Csiszar, I.: Why least squares and maximum entropy? An axiomatic approach to inference for linear inverse problems. The Annals of Statistics 19(4), 2031–2066 (1991)
6. Hadamard, J.: Sur les problemes aux derivees partielles et leur signification physique. Bull. Univ. Princeton 13, 49–52 (1902)
7. Hadamard, J.: Lectures on Cauchy's problem in Linear Partial Differential Equations. Yale Univ. Press, New Haven (1923)
8. Lucy, L.B.: An iterative technique for the rectification of observed images. The Astronomical Journal 79(6), 745–754 (1974)
9. Johnson, S.G., Frigo, M.: Implementing FFTs in practice in Fast Fourier Transforms. In: Burrus, C.S. (ed.), ch. 11, Rice University, Houston TX: Connexions (September 2008)
10. Richardson, W.H.: Bayesian-based iterative method of image restoration. Journal of the Optical Society of America 62(1), 55–59 (1972)
11. Sarder, P., Nehorai, A.: Deconvolution methods for 3-D fluorescence microscopy images. IEEE Signal Process. Mag. 23, 32–45
12. Tikhonov, A.N., Arsenin, V.Y.: Solutions of ill-posed problems. Wiley, Chichester (1977)
13. Tikhonov, A.N.: On the stability of inverse problems. Dokl. Akad. Nauk. SSSR 39, 195–200 (1943)
14. Vardi, V., Shepp, L.A., Kauffman, L.: A statistical model for positron emission tomography. Journal of the American Statistical Association 80(389), 8–37 (1985)
15. NVIDIA Corporation, Documentation for CUDA FFT (CUFFT) Library (2008), http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf
16. Mele, V., Murli, A., Romano, D.: Some remarks on performance evaluation in parallel GPU computing. Preprint del Dipartimento di Matematica e applicazioni. Univerity of Naples Federico II (2011)

HADAB: Enabling Fault Tolerance in Parallel Applications Running in Distributed Environments

Vania Boccia[1], Luisa Carracciuolo[2], Giuliano Laccetti[1], Marco Lapegna[1], and Valeria Mele[1]

[1] Dept. of Applied Mathematics, University of Naples Federico II, Naples. 80126, Complesso Universitario Monte S. Angelo, Via Cintia, Italy
{vania.boccia,giuliano.laccetti,marco.lapegna,valeria.mele}@unina.it
[2] Italian National Research Council, Italy
luisa.carracciuolo@cnr.it

Abstract. The development of scientific software, reliable and efficient, in distributed computing environments, requires the identification and the analysis of issues related to the design and the deployment of algorithms for high-performance computing architectures and their integration in distributed contexts. In these environments, indeed, resources efficiency and availability can change unexpectedly because of overloading or failure i.e. of both computing nodes and interconnection network. The scenario described above, requires the design of mechanisms enabling the software to "survive" to such unexpected events by ensuring, at the same time, an effective use of the computing resources. Although many researchers are working on these problems for years, fault tolerance, for some classes of applications is an open matter still today. Here we focus on the design and the deployment of a checkpointing/migration system to enable fault tolerance in parallel applications running in distributed environments. In particular we describe details about HADAB, a new hybrid checkpointing strategy, and its deployment in a meaningful case study: the PETSc Conjugate Gradient algortithm implementation. The related testing phase has been performed on the University of Naples distributed infrastructure (S.Co.P.E. infrastructure).

Keywords: Fault tolerance, checkpointing, PETSc library, HPC and distributed environments.

1 Introduction

In recent decades the focus of the scientific community moved from the traditional parallel computing systems to high performance computing systems for distributed environments, generally consisting of a set of HPC resources (clusters) geographically scattered. They provide increasing computing power and are characterized by a great resources availability (typical of distributed systems) and a high local efficiency (typical of traditional parallel systems). These systems may be used to solve the so-called “challenge problems”. However, employing a distributed infrastructure, where HPC resources are geographically scattered and are not dedicated to a specific application, is not priceless. Indeed such kind of environments are characterized by high dynamicity in resources load and by a high failure rate, thus applications fault tolerance and efficiency are key issues [14].

For many years by now researchers are working to identify standard methods to solve the problem of fault tolerance and efficiency of software designed for distributed environments. However, today this is still an open issue.

In this paper we focus on the design and deployment of a checkpointing/ migration system, in order to enable fault tolerance in parallel applications running in distributed environments.

In Sec. 2 is presented a short overview on fault tolerance and checkpointing mechanisms. In Sec. 3 is presented our checkpointing strategy, HADAB (**H**ybrid, **A**daptive, **D**istributed and **A**lgorithm-**B**ased), underlining criteria to enhance strategy robustness and to narrow the overhead. In Sec. 4 the HADAB deployment in a case study (the parallel version of PETSc library Conjugate Gradient) is shown. In Sec. 5, are described some tests and results, and finally, in Sec. 6, are presented some conclusions and a preview on future works.

2 Fault Tolerance and Checkpointing: State of the Art

Fault tolerance is the ability of a system to react to unexpected events such as sudden overload, temporary or persistent failure of resources [13]. An application is called fault tolerant if it is able to complete its execution in spite of the occurrence of faults. In “complex” computing systems, application survival to failures during execution, depends on the proper behavior of all software layers that the application uses and on the integrity and coherence of the execution environment. There are applications that, because of special properties of the algorithms on which they are based, are “naturally fault tolerant” (i.e. “super scalable” applications)[5,6]. For all the others, it is necessary to provide “mechanisms” [9] to detect and report the presence of faults (detect/notify), making it possible “to take a snapshot” of the current execution state (checkpointing) and allow the application to resume its execution from the point where the fault occurred (rolling back/migration) [10].

Detect/notify mechanisms are generally in charge to the runtime environment, while checkpointing/migration mechanisms are in charge to the application (or to its runtime environment) and consist of:

– procedures to store data (checkpointing) that, in case of fault, enable the process restart from the point of execution where the fault occurred (checkpoint)
– procedures to resume the execution (rolling back), from where it left off because of the fault, on alternative resources, by recovering and using checkpointing data.

In literature there are different approaches that can be followed to realize checkpointing mechanisms: algorithm-based vs. transparent, disk-based vs. diskless and an exhaustive description of these can be found in [3,7,8,15,16].

The approaches used to implement such mechanisms are numerous and each one has advantages and disadvantages in relation to checkpointing technique efficiency and robustness [3]. Sometimes, it is not sufficient to use a single strategy to realize a robust checkpointing strategy, but new strategies can arise from the combination of multiple methodologies (strategies for hybrid checkpointing [12]). In general, strategies combination can be used to increase the robustness with respect to the single methodologies, but it is necessary to limit checkpointing overhead.

3 HADAB: The Hybrid, Adaptive, Distributed, Algorithm-Based Checkpointing

The strategy described in this work is:

- hybrid, because combines two strategies: a variant of diskless parity-based [11] and coordinated [12] checkpointing;
- adaptive, because different checkpointing techniques are performed each with different frequency, with the aim to reduce the total overhead;
- distributed, because checkpointing data are periodically saved on a remote storage resource;
- algorithm-based because, although hard to implement, this approach is still the safest method to select and reduce the checkpointing data amount.

We focus on parallel applications based on Message Passing paradigm, in particular based on MPI standard. Currently FT-MPI [2] is the only existing message passing library, implementing MPI standard, that providing the software tools to identify and manage faults, makes applications able to use a diskless approach. Indeed, FT-MPI allows to re-spawn failed processes redefining the MPI context. Unfortunately, its development has been stopped in 2003 and so, on some new architectures, the library seems to be not stable.

Other implementations of MPI standard, as Open MPI [4], promise to introduce the important features of FT-MPI, but developers are waiting for the MPI3 standard to implement these in the production release.

In absence of the software tools needful to use a diskless approach, we chose a disk-based approach and a stop/restart method to implement our checkpointing strategy. In the rest of this section we describe how we built the HADAB strategy.

First, we considered a disk-based variant of the parity-based checkpoint, where the checkpointing phase can be divided into two sub-phases:

1. each application processor [11] saves its checkpointing data locally and sends them to the checkpoint processor
2. the checkpoint processor [11] calculates the bitwise-XOR of the received data and stores it on its local storage device.

In a similar way, the rolling back phase can be divided into two sub-phases:

1. survived processors recover their data from local disks
2. the processor, that took the place of the failed processor, reconstructs its portion of data, by both local data coming from other processors and encrypted data sent by the checkpoint processor.

The strategy described above is already quite robust because it ensures the checkpointing integrity (the last successfully saved checkpointing data are removed only after that new coherent checkpointing data are saved) and allows the application to survive to the fault of one processor at a time (application or checkpoint processor).

Moreover, the use of encryption offers advantages in terms of efficiency, because the amount of data that checkpoint processor has to store is drastically reduced (for a problem of dimension N, using p processors, checkpoint processor data dimension is equal to N/p). So it is clear that this strategy has a lower I/O overhead than a non-coding one, but it can tolerate only one fault at a time.

Thus, to improve checkpointing robustness, we decided to add also "few" phases of coordinated checkpointing, realizing an hybrid strategy.

In general checkpointing frequency cannot be high (to avoid the increasing of the total checkpointing overhead) and, anyway, the time for data saving should not be relevant in relation to the total execution time. So we developed our hybrid strategy using an automatic choice of the checkpointing rate that depends on the extimated execution time.

The advantage introduced by the hybrid strategy is that, if p is the number of processors, the application can tolerate up to $p-1$ simultaneous faults, except the fault of the checkpoint processor. So, paying a not so relevant price in terms of total overhead, we gain in terms of checkpointing strategy robustness.

In case of checkpoint processor unavailability, the hybrid strategy is not able to recover the application, because all checkpointing data are lost. Hence the idea to build a distributed version of the hybrid strategy (HADAB) that periodically saves, in an asynchronous way, checkpointing data on an "external" storage resource.

HADAB is able to guarantee up to p faults at a time:

– up to $p-1$ faults there always is a local copy of checkpointing data available (from coordinated or parity-based strategies) to recover from the fault;
– if all processes fail, a remote saved copy of all checkpointing data is available for application resumption: an external stop/restart system migrates application execution on a new set of computational resources, using the remote copy of all checkpointing data (Fig. 1).

The next section describes the work done to deploy the HADAB checkpointing strategy in a case study.

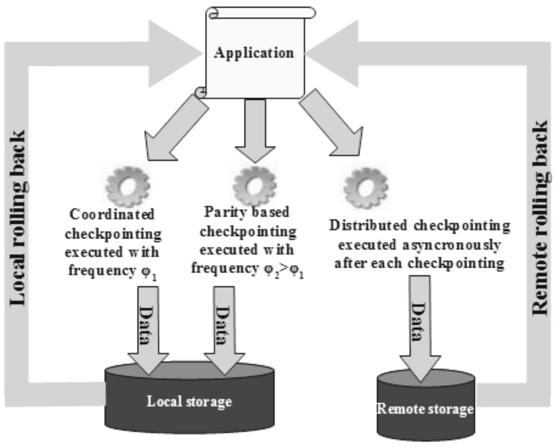

Application
Local rolling back
Remote rolling back
Coordinated checkpointing executed with frequency φ_1
Parity based checkpointing executed with frequency $\varphi_2 > \varphi_1$
Distributed checkpointing executed asyncronously after each checkpointing
Data
Data
Data
Local storage
Remote storage

Fig. 1. Migration system schema

4 HADAB Deployment on the PETSc Conjugate Gradient (CG)

We deployed HADAB into the parallel version of Conjugate Gradient (CG) algorithm implemented in PETSc library [1] with the objective to realize a fault tolerant version of such a procedure (CGFT).

In order to develop a fault tolerant version of CG algorithm (Fig. 2), we followed an algorithm-based approach [3]:

– first we identified the data needed for the checkpointing in the CG algorithm: four vectors and four scalars
– then we added to the PETSc CG routine, the code needed to implement checkpointing phases (see lines 19-32 in Fig. 2) and rolling back phases (see lines 5-14 in Fig. 2) of the HADAB strategy.

Application starts with checkpointing frequency chosen by the user (i.e., parity-based checkpointing is performed at each iteration while the coordinated one is performed every k iterations); during execution the `PetscCheckFreq` routine modifies the checkpointing frequency on the bases of both:

– the average of previous iterations execution time and
– the real time spent to save data (for each checkpointing type).

During the recovery phase, the `CheckCheckpoint` routine selects the most "convenient"[1] checkpointing type to be used in application resumption: if it is parity-based, the `PetscRollbackCodif` routine is called, otherwise the `PetscRoll` backCoord routine is executed.

[1] Metric for convenience is the total cost of the recovery phase that is related to both the data "freshness" and to the overhead in data reading.

```
Fault tolerant version of Conjugate Gradient with hybrid adaptive distributed checkpointing:
code fragment.
```

```
1   PetscErrorCode KSPSolve_CGFT (KSP ksp)
2   PetscFunctionBegin;
3   /* Initialization phase */
4   ...
5   IF (restart)
6           rt =  CheckCheckpoint(...);
7           IF ( rt == 1 )
8                   ierr =  PetscRollbackCoord(...);
9           ELSE IF ( rt == 0 )
10                  ierr =  PetscRollbackCodif(...);
11          ELSE
12                  printf("It is not possible to recover locally from the fault");
13          ENDIF
14  ENDIF
15  REPEAT
16          ...
17          /* repeat-until loop of the CG algorithm */
18          ...
19          IF (chkenable)
20                  /* ck_coord is the iteration number when
21                     coordinated checkpointing is performed */
22                  /* ck_codif is the iteration number when
23                     coded checkpointing is performed */
24                  IF (i % ck_coord == 0 )
25                          ierr =  PetscCheckpointingCoord(...);
26                          ierr =  PetscStartCopyThreads(...);
27                  ELSE /* case  i % ck_codif == 0  */
28                          ierr =  PetscCheckpointingCodif(...);
29                          ierr =  PetscStartCollectThreads(...);
30                  ENDIF
31                  ierr =  PetscCheckFreq(...);
32          ENDIF
33  UNTIL (i < max_it && r > tol)
34  /* finalization phase */
35  PetscFunctionReturn(0);
```

Fig. 2. PETSc CG fault tolerant version

During the checkpointing phase `PetscCheckpointingCodif` routine (for parity-based checkpointing) and `PetscCheckpointingCoord` routine (for coordinated checkpointing) are called respectively with a frequency equal to `ck_codif` and `ck_coord` (φ_2 and φ_1 respectively, see Fig. 1).

Finally, `PetscStartCopyThreads` and `PetscStartCollectThreads` routines perform the asynchronous distributed checkpointing data saving on external storage resources. Distributed checkpointing phase does not add any overhead bacause of the use of threads.

When ever the local rolling back phase is impossible (see line 14 in fig. 2), application stops and the migration system migrates the execution on a new set of computational resources. The remote rolling back phase, included in the migration task, introduces an overhead that may significantly change on the basis of several parameters depending on distributed environment characteristics.

In the next section we report some tests performed at the University of Naples Federico II on the HPC computational resources available in the S.Co.P.E. GRID infrastructure. Tests results provided a first validation of HADAB checkpointing strategy and some useful information about migration system overheads.

5 Tests and Remarks

The first tests have been carried out with the aim to verify the behavior of both the checkpointing strategies: coordinated and parity-based. Tests are related to the solution, by means of CGFT, of a linear system where sparse matrix has $N = 3.9 * 10^{17}$ non-zero elements[2]. The linear system is solved after 6892 iterations.

Indeed, depending on when the fault occurs, the overhead introduced by the checkpointing/rolling back phases becomes more or less relevant in comparison to the total execution time of the application.

Table 1. Execution times in seconds: to execute one CG iteration (T_{iter}), to save checkpoinitng data with parity-based strategy ($T_{checkCodif}$) and to save checkpoinitng data with coordinated strategy ($T_{checkCoord}$). p is the number of processors. Checkpointing data are written on a shared area based on Lustre File System. Data dimension is $2.5 * 10^9$. $T_{checkCoord}$ value is indipendent from p.

p	T_{iter}	$T_{checkCodif}$	$T_{checkCoord}$
8	8.76	362.68	417
12	5.65	206.81	417
16	5.46	196.19	417
20	4.67	192.85	417
24	4.16	189.72	417
28	2.31	188.12	417

Table 1 is useful to understand the optimal value for checkpointing frequencies. Focusing on the test performed with 16 processors, the total execution time for the application, in absence of faults, is about 10 hours and 45 minutes. The `PetscCheckFreq` routine chooses to execute a coordinated checkpointing every 196 iterations and a parity-based one every 14 iterations.

In the following tables, when HADAB is enabled, we consider checkpointing frequencies defined on the bases of results reported in table 1.

Looking at the tables 2, 3 and 4 it is possible to evaluate the benefits, if any, due to the use of HADAB checkpointing in the following scenarios:

- Case 1: failure free execution (see table 2)
- Case 2: a single fault during execution (see tables 3 and 4)

From table 2 we can observe that HADAB adds about the 33% of overhead on the total execution time in absence of faults.

However, if we consider execution with faults, the use of HADAB checkpointing becomes ever more affordable when the iteration number, where the fault occurs, increases (see table 4). Indeed, in the last three rows of the table 4, the $Overhead_{chkp}$ is negative, because the T_{tot}^{NoC} is greater than T_{tot}^{C}. Thus HADAB

[2] Checkpointing data are $M = 2.5 * 10^9$ and their amount is of about $18GB$.

Table 2. Application execution with HADAB checkpointing enabled: T_{comp} is the time related to the computational phase, T_{check} is the time due to HADAB checkpointing, $T_{tot} = T_{comp} + T_{check}$ and $Overhead_{check} = T_{check}/T_{tot}$ is the overhead introduced by HADAB in a failure free execution. All times values are expressed in seconds.

N	T_{comp}	T_{check}	T_{tot}	$Overhead_{check}$ (%)
$3.9 * 10^{17}$	37630	18666	56296	33.1%

Table 3. Application execution with HADAB checkpointing disabled in an execution with one fault occurring at It_{fault}. $T_{it-lost}$ is the time spent to re-execute It_{fault} iterations where It_{fault} is: 1000, 2000, 3000, 4000, 5000, 6000. $T_{tot}^{NoC} = T_{comp} + T_{it-lost}$. All times values are expressed in seconds.

It_{fault}	$T_{it-lost}$	T_{tot}^{NoC}
1000	5460	37630+5460 = 43090
2000	10920	37630+10920 = 48550
3000	16380	37630+16380 = 54010
4000	21840	37630+21840 = 59470
5000	27300	37630+27300 = 64930
6000	32760	37630+32760 = 70390

Table 4. Application execution with HADAB checkpointing enabled: $T_{it-lost}$ is the time spent to execute again only the It_{lost} iterations from the last saved checkpointing to It_{fault}. In last column we report overhead, $Overhead_{chkp}$, introduced by HADAB where $Overhead_{chkp} = (T_{tot}^{C} - T_{tot}^{NoC})/T_{tot}^{NoC}$. All times values are expressed in seconds.

It_{fault}	It_{lost}	$T_{it-lost}$	T_{tot}^{C}	$Overhead_{chkp}$
1000	6	32.76	56296+32.76 = 56328.76	31%
2000	12	65.52	56296+65.52 = 56361.52	16%
3000	4	21.84	56296+21.84 = 56317.84	0%
4000	10	54.60	56296+54.60 = 56350.60	-1%
5000	2	10.92	56296+10.92 = 56306.92	-13%
6000	8	43.68	56296+43.68 = 56339.68	-20%

use is even profitable for the application: i.e. if the fault occurs at iteration 6000, we gain about the 20% on the time T_{tot}^{NoC}.

Looking at all the tables above it is possible to evaluate the benefits due to the use of HADAB checkpointing also in an execution where more than a fault occurs. Indeed, in case where a fault occurs twice, i.e. one at iteration 2000 and the other at iteration 5000, the application recovers twice from the fault, re-executing only 14 iterations. In this case, thanks to HADAB checkpointing we gain about 26% on the time T_{tot}^{NoC}[3].

[3] $T_{tot}^{C} = T_{tot} + T_{it-lost}(2000) + T_{it-lost}(5000)$, by using data in tables 2 and 4; $T_{tot}^{NoC} = T_{tot} + T_{it-lost}(2000) + T_{it-lost}(5000)$, by using data in tables 2 and 3.

Finally table 5 reports the case when all processors p fail at the same time. Here, a migration phase is needed. HADAB checkpointing saved asyncronously data on external storage resource during application execution. Thus when p fault occurr at the same time, migration system selects a new cluster for execution, moves checkpointing data from external storage to the new cluster and restarts application execution from the point when it left out.

Table 5. Application execution with HADAB checkpointing enabled and all processors fail at the same time: $T_{Data-trans}$ is the time to move checkpointing data from storage external resource to the new execution cluster (remote rolling back phase).

N	$Check_{Data-dim}$ (GB)	$T_{Data-trans}$ (secs.)
$3.9 * 10^{17}$	18	134

$T_{Data-trans}$ are related only to the rolling back phase[4], but we have to consider, as overhead, also times due i.e. to new computational resource recruitment and to the queue time on the scheduling system before application restarts. These times are not fixed but depend on the distributed infrastructure characteristics.

6 Conclusion and Future Works

Checkpointing mechanisms deployment in scientific libraries, as PETSc, always is a "good investment". Indeed, if a library is fault tolerant so are all applications that use it. The work here reported, gave us the opportunity to evaluate the benefit in using hybrid strategies for the implementation of checkpointing mechanisms even when disk-based approaches are used.

The implemented system is robust and efficient enaugh; further improvements in efficiency can arise from the use of diskless strategies, currently not feasible, while more improvements in robustness can arise i.e. from the use of virtual resources, fault tolerant networks and fault tolerant message passing libraries.

The remarks made in Sec. 3 and 5 about the overhead introduced by the HADAB checkpointing, are related to an application that, on a big amount of data, performs a "small" amount of computations.

Thus the utility of the checkpointing mechanisms is much more evident in other contexts as i.e.:

- applications handling the same amount of data, but using algorithms with more complexity than that here considered;
- computer centers where it is permitted the use of computing resources for a time not adequate to terminate the application execution.

[4] Data have been moved among two clusters of the S.Co.P.E. distributed infrastructure, by using grid protocols.

References

1. Balay, S., et al.: PETSc Users Manual. ANL-95/11 - Revision 3.1, Argonne National Laboratory (2010)
2. Chen, Z., Fagg, G.E., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., Dongarra, J.: Building Fault Survivable MPI Programs with FT MPI Using Diskless Checkpointing. In: Proceedings for ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 213–223 (2005)
3. Dongarra, J., Bosilca, B., Delmas, R., Langou, J.: Algorithmic Based Fault Tolerance Applied to High Performance Computing. Journal of Parallel and Distributed Computing 69, 410–416 (2009)
4. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B.W., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004)
5. Geist, A., Engelmann, C.: Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors (2002)
6. Engelmann, C., Geist, A.: Super-Scalable Algorithms for Computing on 100,000 Processors. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005. LNCS, vol. 3514, pp. 313–321. Springer, Heidelberg (2005)
7. Hung, E., Student, M.P.: Fault Tolerance and Checkpointing Schemes for Clusters of Workstations (2008)
8. Kofahi, N.A., Al-Bokhitan, S., Journal, A.A.: On Disk-based and Diskless Checkpointing for Parallel and Distributed Systems: An Empirical Analysis. Information Technology Journal 4, 367–376 (2005)
9. Lee, K., Sha, L.: Process resurrection: A fast recovery mechanism for real-time embedded systems. In: Real-Time and Embedded Technology and Applications Symposium, pp. 292–301. IEEE (2005)
10. Murli, A., Boccia, V., Carracciuolo, L., D Amore, L., Lapegna, M.: Monitoring and Migration of a PETSc-based Parallel Application for Medical Imaging in a Grid computing PSE. In: Proceedings of IFIP 2.5 WoCo9, vol. 239, pp. 421–432. Springer (2007)
11. Plank, J.S., Li, K., Puening, M.A.: Diskless Checkpointing. Technical Report CS-97-380, University of Tennessee (December 1997)
12. Silva, L.M., Silva, G.J.: The Performance of Coordinated and Independent Checkpointing. In: Proceedings of the 13th International Symposium on Parallel Processing, pp. 280–284. IEEE Computer Society, Washington, DC (1999)
13. Simon, H.D., Heroux, M.A., Raghavan, P.: Faul Tolerance in Large Scale Scientific Computing, ch. 11, pp. 203–220. SIAM Press (2006)
14. Song, H., Leangsuksun, C., Nassar, R.: Availability Modeling and Analysis on High Performance Cluster Computing Systems. In: First International Conference on Availability, Reliability and Security, pp. 305–313 (2006)
15. Vadhiyar, S.S., Dongarra, J.: SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. In: Parallel Processing Letters, pp. 291–312 (2002)
16. Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In: Parallel and Distributed Processing Symposium (2007)

Increasing the Efficiency of the DaCS Programming Model for Heterogeneous Systems

Maciej Cytowski and Marek Niezgódka

Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw, Poland

Abstract. Efficient programming of hybrid systems is usually done with the use of new programming models. It creates a unique opportunity to increase the performance of scientific applications and is also especially interesting in the context of future exascale applications development where extreme number of MPI processes tend to be a limitation. Future scientific codes will make use of hierarchical parallel programming models with message passing techniques used between nodes and optimized computational kernels used within multicore, multithreaded or accelerator nodes. In this article we consider the x86 and PowerXCell8i heterogeneous environment introduced in the High Performance Computing (HPC) sites like Roadrunner [6] or Nautilus [5]. Programming techniques for this environment are usually based on the IBM Data Communication and Synchronization library (DaCS). We describe our effort to increase the hybrid efficiency of the DaCS library and show how it affects the performance of scientific computations based on FFT kernels. The results are very promising especially for computational models that involve large three dimensional Fourier transformations.

Keywords: hybrid computing, Cell processor, FFT, parallel computing.

1 Introduction

Novel computer systems from desktops to world's biggest supercomputers are very often based on new architectures or hardware accelerators. Efficient computations on such systems can be achieved with the use of new programming models. The performance of scientific applications can be increased by the functional decomposition of computations and offloading chosen computational kernels on accelerators. However the final performance of those applications depend on their feasibility to the architectures in use and on the performance of the programming tools. Therefore developers have to examine the efficiency of both applications and programming environments in order to produce the fastest solutions for specific scientific problems. In this work we present a performance benchmark of a novel heterogeneous programming model together with its specific scientific application - Fast Fourier Transform (FFT) computations. The outcomes and measurements presented in this work are important not only for FFT computations but for many other computational algorithms and scientific disciplines. The work is accompanied with libraries and example codes.

In this paper we are looking at the heterogeneous programming techniques for high performance systems based on the PowerXCell8i architecture. The PowerXCell8i was released in 2008 as an enhanced Cell Broadband Engine processor with improved double-precision floating point performance. It is a multi-core chip composed of one Power Processor Unit (PPU) and eight Synergistic Processing Units (SPU). The architecture itself was already extensively described e.g. in [11], [14], [15] and [6]. The PowerXCell8i was designed to bridge the gap between general purpose processors and specialized computer architectures like GPUs. Applications can be compiled and executed in a standard Linux environment on the PPU. Furthermore specific computational kernels can be implemented and executed on the SPUs playing a role of hardware accelerators. Most of the applications achieve poor performance on the PPU since it is not designed for computations. The computations that are not optimized for execution on the SPUs are very often 3 to 5 times slower when compared with their performance on modern x86 compute cores. One of the techniques used to overcome those issues is to use a heterogeneous environment. Therefore HPC architectures like Roadrunner [6] or Nautilus [5] utilize the PowerXCell8i chip as an accelerator for calculations running on x86 cores. Both systems are composed of the IBM LS21 and IBM QS22 blades but they differ in a type of interconnect. In the case of the Roadrunner system [6] the interconnect is based on the PCIe x8 and HyperTransport technology. In the case of the Nautilus system [5] nodes are connected with the DDR Infiniband 4x network. Both systems are important milestones in the development of future HPC systems. Roadrunner is well known to be the world's first TOP500 Linpack sustained 1.0 petaflops system (November 2008). On the other hand Nautilus has been ranked on the 1'st place of the Green500 list twice (November 2008 and June 2009).

Most important programming technique available for such heterogeneous architectures is the IBM Data Communication and Synchronization library (DaCS) used in several scientific codes already developed for the Roadrunner and Nautilus supercomputers ([16], [8], [12]). One of the main advantage of DaCS is that it creates a very interesting hierarchical programming model together with the message passing techniques like the MPI library. This is very important in the context of future exascale applications development where extreme number of MPI processes tend to be a limitation.

We decided to take a closer look at DaCS performance on the Roadrunner like systems. Especially we decided to measure the data transfer rate since it is one of the fundamental factor for application optimization on the accelerator based systems. We describe DaCS functionality and our benchmark results in Chapter 2. The effort we have made to increase DaCS performance is extensively covered in Chapter 3. Finally in Chapter 4 we discuss one of the potential usage scenarios of our implementation, the FFT computations. We describe how to use DaCS for offloading the FFTW library computations on the PowerXCell8i processor.

2 Data Communication and Synchronization Library

2.1 Overview

The DaCS [1] library was designed to support development of applications for the heterogeneous systems based on the PowerXCell8i and x86 architectures. It contains two main components: the application programming interface (API) and the runtime environment. The DaCS API provides an architecturally neutral layer for application developers. It serves as a resource and process manager for applications that use different computing devices. With the use of specific DaCS functions we can execute remote processes and initiate data transfers or synchronization between them.

One of the main concepts of DaCS is a hierarchical topology which enables application developers to choose between a variety of hybrid configurations. First of all it can be used for programming applications for the Cell processor by exploiting its specific hybrid design. In such a model developers use DaCS to create and execute processes on the PPU and SPUs and to initiate data transfers or synchronization between those processes. However developers can choose between few other programming concepts for the Cell processor and the DaCS model is for sure not the most productive and efficient one. The DaCS library is much more interesting as a tool for creating hybrid applications that use two different processor architectures. In such a model DaCS can support the execution, data transfers, synchronization and error handling of processes on three different architectural levels (i.e. the x86, PPU and SPU levels). Additionally programmer can decide to use DaCS with any other Cell programming language on the PPU level. The PPU process can execute the SPU kernels provided by optimized libraries or created originally by developers with the use of programming tools like the Libspe2 [13], Cell SuperScalar [9] or OpenMP [7].

The DaCS library has a much wider impact on high performance computing since it was designed to support highly parallel applications where the MPI library is used between heterogeneous nodes and the DaCS library is used within those nodes. Such programming model was used for applications development on the Roadrunner and Nautilus systems ([16], [8], [12]).

2.2 Performance Benchmarking

A common feature of heterogeneous systems is the bottleneck introduced by the data transfer crossing the accelerator boundary. The computational granularity of the optimized compute kernels must be carefully measured and compared with the data transfers performance in order to make a decision on offloading specific calculations on the accelerator.

The performance measurements presented within this work were prepared on two systems: the Roadrunner-like system located in IBM Laboratories in Rochester and the Nautilus system located at ICM in Warsaw. The best data transfers rate was achieved on the Roadrunner-like system with the use of the PCI x8 interconnect. The measurements on the Nautilus system were performed with the use of the Gigabit Ethernet and Infiniband interconnect. Unfortunately the DaCS library does not support RDMA over Infiniband mechanism which results in a very poor data transfer's rate. In this work we will mainly concentrate on the results achieved on the Roadrunner-like system.

We have prepared a performance benchmark for the DaCS library and evaluated it on available heterogeneous systems. It is a simple ping-pong program similar to the one used for benchmarking of MPI point-to-point communication. The sending host process sends a message of a given data size to the accelerator process and waits for a reply. The average time of such communication is measured for different message sizes. The data transfers are initiated with the use of `dacs_put` and `dacs_get` functions.

Developing applications on the heterogeneous systems based on the x86 and PowerXCell8i architectures introduces an additional byte-swapping step related to different endianness of the host and accelerator nodes. The DaCS library addresses this issue by providing a byte swapping mechanism that can be switched on and off by setting corresponding parameters of the DaCS data transfers functions.

The data transfers implemented in our benchmark program were used for transferring of the double precision floating point numbers and were executed with byte swapping mechanism turned on and off for comparison. In this way the influence of the DaCS byte swapping step on the overall performance of the data transfers was measured. We will refer to those two different setups by using two shortcuts for simplicity: BS and NBS will stand for byte swapping and no byte swapping version respectively.

The performance results obtained on the Gigabit Ethernet for medium and large data transfers (between 2^{18} and 2^{30} bytes) were reaching 70 MB/s for NBS version. The average difference between BS and NBS versions for those data transfers is around 9 MB/s. This difference is of minor importance for computations. The performance results obtained on the Infiniband network were reaching the level of 100 MB/s. This weak performance is related to the lack of support for the Infiniband over RDMA transfers in the DaCS library. Much more interesting results were those obtained on the Roadrunner-like system where data transfers are handled by dedicated PCI based interconnect. Although the maximum performance for the NBS version reached more than 1090 MB/s, the BS version was much slower. The difference for large data transfers exceeds 800 MB/s. The results of the benchmark on PCI interconnect are depicted on Figure 2.

3 Optimized Byte Swapping

The benchmark results described in the previous chapter present an undesirable dependence of the computational performance of DaCS applications on the performance of byte swapping step. The byte swapping mechanism implemented in DaCS seems to be very unoptimal. Unfortunately almost every application implemented on the described heterogeneous system have to make use of it.

Byte swapping is a very simple operation and can be implemented with the use of permutation of bytes which transforms the bit representation of a given number from one endianness format to the other. Our implementation on the PowerXCell8i processor is based on two very important observations. First of all for large data sizes byte swapping can be easily parallelized in a data parallel mode. Secondly byte swapping can be performed with the use of SIMD vector operations. For large data sizes we decided to make use of the available vector SPUs. For smaller data sizes we propose a SIMD-ized optimal implementation on PPU. The final PowerXCell8i byte swapping library (PXCBS) is a mix of both described implementations based on their performance measured for different data sizes.

Table 1. Time measurement of byte swapping optimized kernels

Kernel version	2^{14} **bytes**	2^{20} **bytes**	2^{30} **bytes**
1 threaded PPU	9 usec	1672 usec	1677745 usec
4 SPU threads	11 usec	1520 usec	99159 usec
AMD Opteron	29 usec	1696 usec	1716653 usec

3.1 Key Optimization Steps

Here we describe the consecutive optimization steps of byte swapping for the PowerXCell8i architecture.

SPU Implementation. For large data sizes we can parallelize byte swapping simply by dividing the data into blocks. For double precision floating point numbers we choose a block of size 1024 since the maximal transfer size for the DMA operations is 16kB. We use a double buffering scheme to overlap computations and communication. Moreover we exploit the SIMD operations by using the `spu_shuffle` instruction. We've measured the performance achieved with the use of 1,2,4,8 and 16 concurrent SPU threads. Although the final implementation presents very good performance for large data sizes it is not optimal for smaller sizes due to the overhead introduced by creation and maintaining parallel SPU threads during execution.

PPU Implementation. For smaller data sizes we decided to implement byte swapping on the PPU. The main optimization technique used here is SIMD-ization. Here we use `vec_perm` Altivec operation instead of `spu_shuffle`. Performance results are summarized in Table 1. Since the PPU is a two-way multithreaded core we decided to create a dual threaded version of byte swapping with the use of POSIX threads. We've measured the performance achieved with the use of 1 and 2 concurrent PPU threads.

Performance Measurements. The final version of optimized byte swapping is a mix of developed kernels used interchangeably dependent on the data size. The decision which kernel should be executed is statically implemented in the code based on two comparisons. First of all we compare the performance of those kernels on small, medium and large data sizes (see Table 1). In this comparison we measure only the time needed to perform single byte swapping step.

We have also performed measurements of the full DaCS data transfer process that includes data movement and byte swapping. This was done with the use of the previously described DaCS ping-pong test. The results are presented in Figure 1 for varying data sizes and show that each of the implemented kernels should be considered for usage in the final solution.

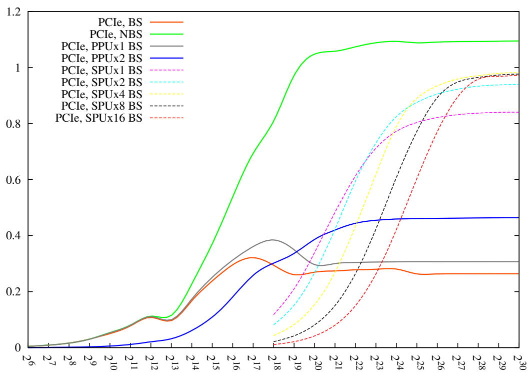


Fig. 1. Measurements of data transfers performance for different version of optimized byte swapping (SPU and PPU kernels). Data size in bytes is depicted on the x-axis. Performance measured in GB/s is depicted on the y-axis.

3.2 Result and Usage Details

Based on the results presented in Table 1 and Figure 1 we have prepared a final implementation which make use of the optimized kernels in a following way:

- the single threaded PPU version is used for sizes smaller than 2^{19} bytes,
- the dual threaded PPU version is used for sizes between 2^{19} and 2^{20} bytes,
- the single SPU version is used for sizes between 2^{20} and 2^{22} bytes,
- the two SPUs version is used for sizes between 2^{22} and 2^{24} bytes,
- the four SPUs version is used for sizes bigger than 2^{24} bytes.

The overall performance of our implementation is presented in Figure 2.

Note that the shape of presented curve is similar to the original BS version for small and medium sizes (growth, local minimas and maximas). However the performance for large sizes achieves a stable level of ~980 MB/s.
The PowerXCell8i optimized byte swapping (PXCBS) is available for download and licensed on the basis of GPL. It is a lightweight library that enables byte

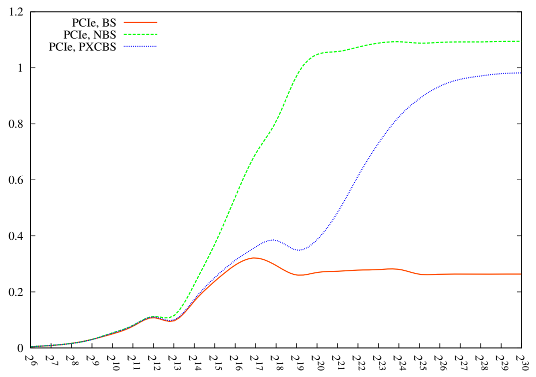


Fig. 2. DaCS data transfers performance with optimized byte swapping library compared to previous versions (ping-pong benchmark). Data size in bytes is depicted on the x-axis. Performance measured in GB/s is depicted on the y-axis.

swapping for 32- and 64-bit data. The library functions are accessible from PowerXCell8i accelerator code. Basic usage is simple and straightforward. In order to transfer a double precision floating point table T of size N programmer needs to perform the DaCS data transfers with byte swapping turned off and call the `bswap64_pxc(N,T)` function. The program needs to be linked with provided PXCBS library.

4 Usage Scenario: FFTW Library

The usage scenario presented in this chapter addresses applications that involve FFT computations. One of the most popular tools used for Fourier transform computations in scientific codes is the FFTW library [10]. The FFTW library was ported and optimized for execution on the PowerXCell8i architecture by IBM Austin Research Laboratories [2]. The performance of the optimized FFTW library on the PowerXCell8i architecture was reported for many different benchmark settings in [3] and [4]. Applications that use FFTW can be compiled and executed on the PowerXCell8i architecture. There are some important caveats that need to be taken into account. First of all the data must be stored in contiguous arrays aligned at 16-byte boundary. Secondly as noted by the developers the `FFTW_ESTIMATE` mode may produce unoptimal plans and the user is encouraged to use `FFTW_MEASURE` instead. However the latter is much more time consuming on the PowerXCell8i processor and for many applications `FFTW_ESTIMATE` is still a better choice. All performance measurements presented in this article were obtained with the use of `FFTW_ESTIMATE` mode.

We present the results of performance analysis of the FFTW library on heterogeneous systems based on the PowerXCell8i architecture. The benchmark problem was designed to evaluate the possible advantages of using such heterogenous approach. We compare the reference x86 performance (AMD Opteron

2216, 2.4 GHz) with corresponding hybrid implementation. Time measurements include the data transfers, byte swapping steps and FFT computations.

4.1 Computational Model

We have developed a set of simple heterogeneous programs that serve as a benchmark suite for the FFTW library. The main purpose was to show how the PowerXCell8i optimized library could be used to accelerate scientific applications on heterogeneous architectures. In a very first step of the program the decision on type, size and direction of the transform is made within application running on AMD Opteron core. These informations are then sent to the accelerator process. In the next step accelerator process allocates tables for transform and prepares the FFT plan with the use of the FFTW library interface. At the same time transformation datas are being prepared on the host process and are sent to the accelerator process when ready. The FFT computations on the accelerator process are preceded and followed by the byte swapping steps. A reverse data transfer is carried out and the FFT plan created for computations is destroyed. The performance comparison presented here is made between reference x86 program and two heterogeneous programs: the one that uses the PXCBS library and the one that uses the DaCS built-in byte swapping mechanism.

4.2 Performance Measurements

The performance measurements for 1D, 2D and 3D FFT transforms are presented in Table 2. All performed FFTs are double-precision complex forward transforms. In the case of heterogeneous programs we present the communication time and walltime measured during execution. The heterogeneous programs based on the DaCS library built-in byte swapping mechanism do not achieve a significantly better performance than their x86 equivalents. However the use of optimized byte swapping mechanism gives much better overall performance. The best speedup measured was 4.3x, 4.8x and 7.4x for 1D, 2D and 3D cases respectively.

5 Summary

In this work we have presented the optimized byte swapping mechanism that replaces its unoptimal equivalent in the DaCS library and can be used for implementation of heterogeneous applications that involve movement of large data between host and accelerator. The performance rate of our solution measured with the use of DaCS ping-pong test is up to 3.7x more efficient in terms of MB/s. This result was achieved by exploiting parallel and SIMD processing features of the PowerXCell8i chip. The PXCBS library can be used together with the IBM DaCS library to support heterogeneous computations. Usually it significantly increases the overall performance and in our opinion it should be always considered as a tool for byte swapping on PowerXCell8i processor.

Table 2. Performance comparison of FFT transforms on heterogeneous architecture

		DaCS PCI			**DaCS PCI + PXCBS**		
1D FFT size	**x86**	**Comm.**	**Walltime**	**Speedup**	**Comm.**	**Walltime**	**Speedup**
131072	0.025s	0.016s	0.018s	1.38x	0.009s	0.011s	2.27x
262144	0.076s	0.032s	0.035s	2.17x	0.014s	0.018s	4.22x
524288	0.102s	0.064s	0.067s	1.52x	0.024s	0.030s	3.4x
1048576	0.210s	0.127s	0.141s	1.48x	0.043s	0.057s	3.68x
2097152	0.446s	0.254s	0.288s	1.54x	0.079s	0.113s	3.94x
4194304	0.924s	0.503s	0.704s	1.31x	0.146s	0.347s	2.66x
8388608	1.838s	1.007s	1.153s	1.59x	0.282s	0.425s	4.32x
		DaCS PCI			**DaCS PCI + PXCBS**		
2D FFT size	**x86**	**Comm.**	**Walltime**	**Speedup**	**Comm.**	**Walltime**	**Speedup**
256x256	0.009s	0.009s	0.009s	1.0x	0.006s	0.006s	1.5x
512x512	0.073s	0.033s	0.047s	1.55x	0.015s	0.029s	2.51x
1024x1024	0.345s	0.128s	0.169s	2.04x	0.044s	0.086s	4.01x
2048x2048	1.674s	0.512s	0.701s	2.38x	0.156s	0.346s	4.83x
		DaCS PCI			**DaCS PCI + PXCBS**		
3D FFT size	**x86**	**Comm.**	**Walltime**	**Speedup**	**Comm.**	**Walltime**	**Speedup**
64x64x46	0.021s	0.033s	0.036s	0.58x	0.016s	0.018s	1.16x
128x128x28	0.583s	0.261s	0.279s	2.08x	0.088s	0.105s	5.55x
256x256x256	6.062s	2.083s	2.246s	2.69x	0.643s	0.812s	7.46x

We have also described how the proposed solution can be directly and efficiently used for accelerated FFT computations based on the FFTW library. One of the key results here is the one reported for 3D Fast Fourier transforms. For large transform sizes like 256x256x256 we've achieved approximately 7.4x speedup over reference x86 implementation. The same hybrid FFT computations based on the DaCS library built-in byte swapping mechanism achieved only 2.6x speedup. The presented FFT performance result based on optimized byte swapping mechanism can have a significant impact on the performance of many algorithms that involve Fourier transforms: convolution and correlation computations, spectral windowing, power spectra computations, periodicity searching algorithms, linear prediction, wavelet transforms and many more.

All the results presented in this work can be easily reproduced with the use of freely available tools and benchmarks available at: `http://www.icm.edu.pl/~sheed/dacs_performance`.

Acknowledgments. We would like to thank Peter Hofstee (IBM Systems and Technology Group, Austin), Minassie Tewoldebrhan (IBM Rochester) and Maciej Remiszewski (IBM Deep Computing, Central and Eastern Europe) for making IBM triblades available remotely for tests. This research was carried out with the support of the "HPC Infrastructure for Grand Challenges of Science and Engineering" Project, co-financed by the European Regional Development Fund under the Innovative Economy Operational Programme. Part of the calculations were performed in the Interdisciplinary Centre for Mathematical and Computational Modelling (ICM) at University of Warsaw under the computational grant G33-19.

References

1. Data Communication and Synchronization for Cell BE Programmer's Guide and API Reference IBM SC33-8408-01, Publication Number: v3.1
2. FFTW on Cell, http://www.fftw.org/cell
3. FFTW on the Cell Processor benchmarks, http://www.fftw.org/cell/cellblade/
4. JCCC FFTW benchmark suite, http://cell.icm.edu.pl/index.php/FFTW_on_Cell
5. Nautilus supercomputer site, http://cell.icm.edu.pl/index.php/Nautilus
6. Roadrunner supercomputer site, http://www.lanl.gov/roadrunner/
7. OpenMP Application Program Interface, Version 3.0 (2008)
8. Bowers, K.J., Albright, B.J., Yin, L., Bergen, B., Kwan, T.J.T.: Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. Phys. Plasmas 15 (2008)
9. BSC: Cell Superscalar (CellSs) User's Manual, Version 2.1 (2008)
10. Frigo, M., Johnson, S.G.: The Design and Implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005)
11. Gschwind, M., Hofstee, H., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic Processing in Cell's Multicore Architecture. IEEE Micro 26(22), 10–24 (2006)
12. Habib, S., Pope, A., Lukić, Z., Daniel, D., Fasel, P., Desai, N., Heitmann, K., Hsu, C.H., Ankeny, L., Mark, G., Bhattacharya, S., Ahrens, J.: Hybrid petacomputing meets cosmology: The Roadrunner Universe project. Journal of Physics: Conference Series 180(1), 012019 (2009), http://stacks.iop.org/1742-6596/180/i=1/a=012019
13. International Business Machines Corporation: Programming Tutorial. Technical document SC33-8410-00. Software Development Kit for Multicore Acceleration Version 3.1
14. Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., Shippy, D.: Introduction to the Cell Multiprocessor. IBM Journal of Research and Development 49(4), 589–604 (2005)
15. Kistler, M., Perrone, M., Petrini, F.: Cell Multiprocessor Communication Network: Build for Speed. IEEE Micro 26(3), 10–23 (2006)
16. Swaminarayan, S., Kadau, K., Germann, T.C., Fossum, G.C.: 369 Tflop/s molecular dynamics simulations on the Roadrunner general-purpose heterogeneous supercomputer. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 64:1–64:10. IEEE Press, Piscataway (2008), http://dl.acm.org/citation.cfm?id=1413370.1413436

A Software Architecture for Parallel List Processing on Grids

Apolo H. Hernández[1], Graciela Román-Alonso[1], Miguel A. Castro-García[1], Manuel Aguilar-Cornejo[1], Santiago Domínguez-Domínguez[2], and Jorge Buenabad-Chávez[2]

[1] Universidad Autónoma Metropolitana
Departamento de Ing. Eléctrica, México, DF., A.P. 55-534 México
apolo.h.s@gmail.com, {grac,mcas,mac}@xanum.uam.mx
[2] Centro de Investigación y de Estudios Avanzados del IPN
Departamento de Computación, México, DF., A.P. 14-740 México
{sdguez,jbuenabad}@cs.cinvestav.mx

Abstract. The Data List Management Library (DLML) processes data lists in parallel, balancing the workload transparently to programmers. Programmers only need to organise data into a list, use DLML functions to insert and get data items, and specify the sequential function(s) to process each data item according to the application logic. The first design of DLML was targeted for use at a single cluster.

This paper presents *DLML-Grid*, a software architecture for DLML to run in Grid environments composed of multiple distributed clusters. The architecture is hierarchical and tends to localise communication within clusters, thus reducing communication overhead. Using OpenVPN, we implemented a prototype version of *DLML-Grid* to gather empirical results on its performance using two clusters and two applications whose workload is static and dynamically generated. *DLML-Grid* performs much better than DLML overall.

Keywords: Cluster Computing, Parallel Computing, Grid Computing, Load Balancing, OpenVPN.

1 Introduction

Clusters have become the most widely used architecture for high performance computing for the cost-to-computation ratio they offer. Combined with Grid software technologies such as Globus [1], clusters can provide in principle limitless processing power in a flexible yet controlled manner. To some extent computing resources are not longer a problem; but how to use them efficiently is. Developing parallel applications for Grid environments must address both intra- and inter-cluster parallelism. In addition, good performance and efficient use of resources also depend on applications being adaptive, e.g., to load imbalance and hardware failure. To cope with this complexity, software tools and middleware have been proposed whose aim is to simplify parallel programming, such as Skeletons [6,9] and Mapreduce [4].

The Data List Management Library (DLML) is a middleware to process data lists in parallel. Users only need to organise their data into items to *insert* and *get* from a list using DLML functions. DLML applications run under the SPMD (Single Program Multiple Data) model: all processors run the same program but operate on distinct data lists. When a list becomes empty, it is refilled by DLML through fetching data items from another list transparently to the programmer. Only when *DLML_get* does not return a data item the processing in all nodes is over. Thus DLML functions hide synchronisation communication from users, while automatic list refilling tends to balance the workload according to the processing capacity of each processor, which is essential for good performance. The first version of DLML [8] was targeted for use at a single cluster.

This paper presents *DLML-Grid*, a software architecture for DLML designed to make better use of resources in Grid environments. The architecture is hierarchical and its purpose is to localise communication within clusters as much as possible, and thus reduce communication overhead, i.e.: intra-cluster load balancing takes precedence over inter-cluster load balancing. Only when the workload in a cluster is exhausted, inter-cluster load balancing takes place. Using OpenVPN [5], we implemented a prototype version of *DLML-Grid* to gather empirical results on its performance using two clusters and two applications whose workload is static and dynamically generated. The clusters used are located at the north and the east of Mexico City. *DLML-Grid* performs much better than the original DLML. The latter could *see* and access both clusters through VPN as a single cluster, but not being aware of (i.e., designed to handle) their long-distance distributed location would not give preference to intra-cluster load balancing.

Section 2 provides background material to DLML: its architecture and load balancing algorithm. Sections 3 and 4 present the architecture of *DLML-Grid* and its load balancing algorithm. Section 5 presents our experimental evaluation and results. Section 6 presents related work and Section 7 our conclusion.

2 DLML

DLML is written in the C language and uses MPI library functions to implement the protocol to automatically fetch remote data, various reduce operations, and the synchronisation needed at the start and end of computation.

DLML Architecture

The DLML architecture consists of two processes running on each processor/core: an *Application* process and a *DLML* process, as shown in Figure 1. The former runs the application code while the latter is in charge of: i) making data requests to remote nodes when the *local* list becomes empty, and ii) serving data requests from remote nodes whose list has become empty.

In making a data request to remote nodes, a *DLML* process can follow one of the following policies which will load balance, re-distribute data, in the system:

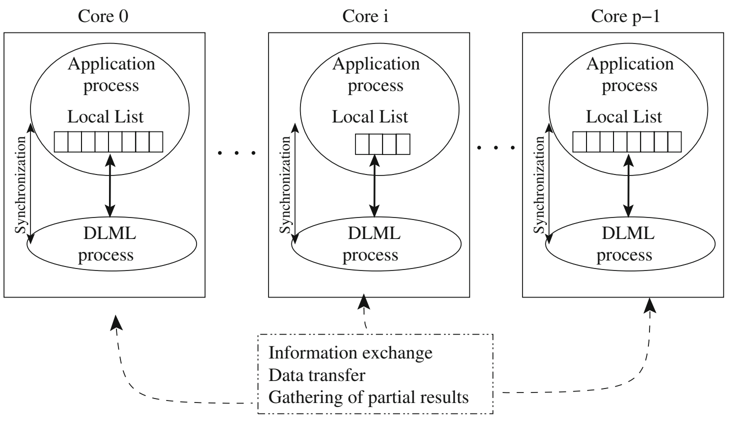


Fig. 1. DLML architecture

Global Auction - when the local list of a processor/core M becomes empty, M sends a message to *all* other system cores requesting their load level. All the cores respond to M sending back a message with the size of their local list. Then M chooses the core with the largest list, say R, and sends R a data request message. Finally, R sends M half of its list.

Torus-Based Partial Auction - as above, but in each auction only participate *neighbour* cores as specified by a logical Torus topology in order to reduce communication overhead and thus improve DLML scalability [8].

3 DLML-Grid Architecture

Our design of the DLML-Grid architecture was mostly based on the hierarchical model described in [10]; the main differences are in the load balancing strategy and are discussed in Section 6 Related Work. In that model, a Grid is a system of interconnected clusters managed through three different kinds of cores (processes), see Figure 2. The levels comprising the hierarchical architecture are described below.

Level 0: A Grid Manager (GM) core with the following responsibilities:

- To collect load information on each cluster.
- To make load balancing decisions at inter-cluster level: what cluster should transfer load to an underloaded cluster.
- To interact with Cluster Managers (CMs) of level 1 to orchestrate data transfer between them.
- To collect final partial results from CMs.

Level 1: Cluster Managers, one per cluster, whose responsibilities are:

- To gather load information from worker cores (WC) of level 2.
- To communicate with the GM to send the amount of total load in their clusters, to ask for more data, or to transfer required load.

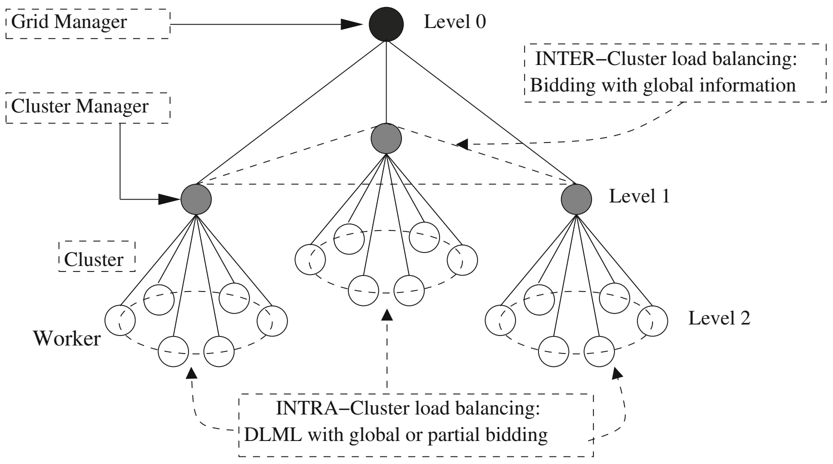

Grid Manager
Level 0
INTER–Cluster load balancing:
Bidding with global information
Cluster Manager
Level 1
Cluster
Worker
Level 2
INTRA–Cluster load balancing:
DLML with global or partial bidding

Fig. 2. DLML load balancing model for a Grid environment

– To communicate with other *CM* to send or receive load, following a decision made by the *GM*.
– To communicate with worker cores *WC* in level 2 to collect and send load in case of a transfer.
– To collect the final partial results from *WCs* and sent them to the *GM*.

Level 2: Worker Cores (*WC*) - Each *WC* runs an *Application* process and a *DLML* process and is also in charge of doing the following tasks:

– To send their load information to its *CM* when is requested.
– To run the application code.
– To participate in intra-cluster load balancing actions. In contrast to the model in [10], where load balancing in each cluster is managed by the *CM* in a centralized way, in our design each *WCs* can start load balancing actions with sibling *WCs*.
– To send their partial results to the *CM* at the end of execution.

Thus *intra-cluster* load balance precedes *inter-cluster* (or *intra-grid*) load balance, which is only activated when there is no work left within a cluster. Communication thus tends to be localised within each cluster.

Figure 3 shows the architecture DLML-Grid for a system comprising four clusters. Each *WC* communicates only with its associated *CM* and sibling *WCs* in the same cluster. *CMs* maintain communication with each other and with the *GM* through Internet. Communication between *CMs* is restricted to load transfers as requested by the *GM*.

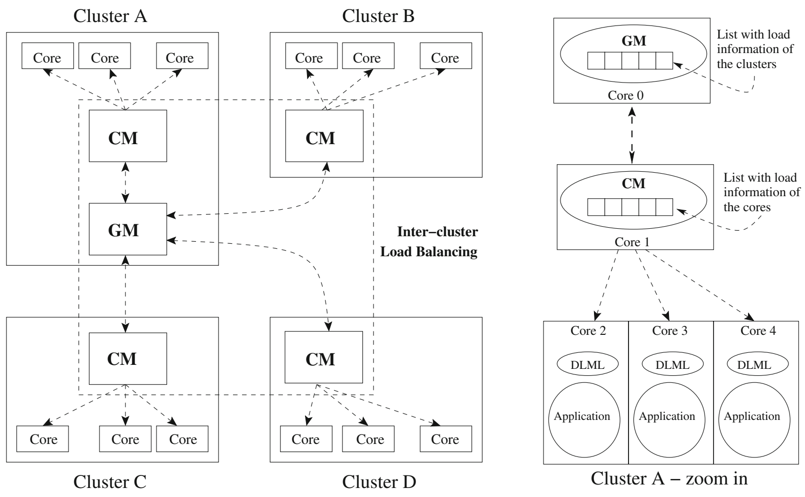

Cluster A
Cluster B
Core
CM
GM
Inter−cluster Load Balancing
Cluster C
Cluster D
List with load information of the clusters
Core 0
List with load information of the cores
Core 1
Core 2
Core 3
Core 4
DLML
Application
Cluster A − zoom in

Fig. 3. DLML-Grid architecture with cluster A zoomed in

4 DLML-Grid Load Balancing Algorithm

In DLML-Grid, load balancing between clusters follows the global auction policy described above, i.e., considering the load information in all clusters. It is initiated by a *receptor* cluster and consists of 4 stages: local load search, external load search, selection of the sender cluster and load redistribution.

Local Load Search. When a cluster has no more data to process, one *WC* sends a request to its *CM*. Which *WC* can make this request depends on the algorithm used for *intra*-cluster load balancing. If the global auction is used, any *WC* can make the request. If the Torus-based partial auction is used, only the first *WC* can make the request [8]. When a *CM* receives the request, it in turn sends a data request to the *GM*, who initiates an external load search.

External Load Search. When the *GM* receives a data request from an underloaded cluster, it requests load information from all *CM*s (except the *CM* that requested data). Each *CM* then gathers load information asking the list size from each *WC* under its control; all sizes are added and sent to the *GM*.

Sender Cluster Selection. *GM* selects the cluster with the largest amount of data as the *sender* cluster, and checks if the load information reported by the *sender* cluster comes from at least 50% of the cores. If not, the *sender* cluster is likely to be close to finish processing its workload, and therefore, a distribution to the *receiver* cluster would increase the processing time rather than reduce it.

Load Redistribution. *GM* requests the *CM* in the *sender* cluster a data transfer to the *receiver* cluster. The *CM* in the *sender* cluster requests *workload* from its *WCs*, each of which sends back a percentage of their local list. *CM* gathers all the items into a temporary list and sends it to the *CM* in the *receiver* cluster through the Internet. Finally, the *CM* in the *receiver* cluster receives the list and distributes it among its *WCs*.

If the *receiver* and *sender* have the same processing power, the *sender* sends 50% of its load. If the *receiver* has less or more processing power than the *sender*, the *sender* sends less or more than 50% of its load, respectively. How much less or more of its load a sender sends is a fraction computed based on the processing power of both sender and receiver, as follows. Suppose cluster A has 32 2.4GHz cores and cluster B has 120 2.6 GHz cores. The processing power of A is 76.8 GHz (32×2.4), and of B 312 GHz (120×2.6); the processing power of A and B combined is 388.8 GHz ($76.8 + 312.0$). Thus the fraction of its load that cluster A sends to B is 0.802 (312/388.8), while the fraction of its load that cluster B sends to A is 0.197 (76.8/388.8).

5 Experimental Evaluation

5.1 Platform Setting and Applications

To compare the performance of the original DLML and DLML-Grid we need to use at least two distributed clusters so that the hierarchical management of DLML-Grid is exerted. Using more than one cluster can be made through a grid middleware such as Globus [1]. However, Globus is rather complex out of the many functions it supports (resource discovery, localisation and management, among others). As we only need to be able to use more than one cluster, we interconnected two clusters, C1 and C2, through VPN, which makes them appear to software as a single cluster. C1 has 6 nodes, each node with an Intel Core2 Quad 2.4GHz, 4G RAM, located at the UAM-I (East of Mexico City); C2 has 30 nodes, each node with an Intel Core i7 2.67GHz, 4G RAM, located at CINVESTAV (North of Mexico City). C1 nodes are interconnected through Fast Ethernet, while C2 nodes through Gigabit Ethernet; the (inter-cluster) link between C1 and C2 is about 5 MB/seg.

We used 140 cores in both clusters combined, running the non-attacking N-Queens problem [2] and a Matrix Multiplication (MM) algorithm with 4 DLML configurations: 1) Global-VPN, 2) Torus-VPN, 3) Global-Grid and 4) Torus-Grid. Configurations 1 and 2 run the original DLML with global and partial auction, respectively. Configurations 3 and 4 run the new DLML-Grid architecture described in Section 3, using both clusters through VPN too. All versions were tested with regard to response time and data distribution. N-Queens was run with N = 16, 17 and 18 (average), and Matrix Multiplication with size 400x400 and 600x600 (average).

5.2 Response Time and Load Distribution under N-Queens

Figure 4 (top) shows that Global-Grid and Torus-Grid have better response time than corresponding VPN versions. This is because the DLML-Grid architecture gives preference to local communication over Internet communication. Table 1

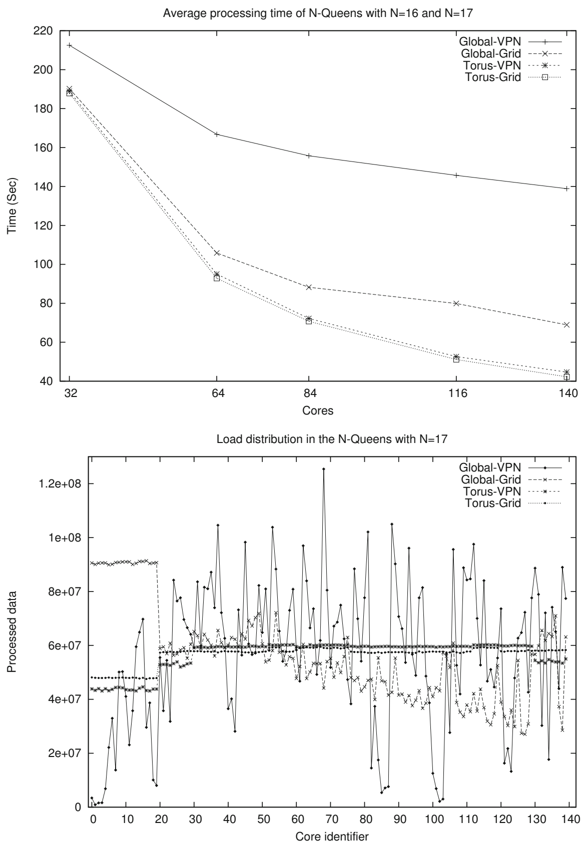


Fig. 4. N-Queen: response time (top) and load distribution (bottom)

shows the number of Requests and Load Transfers (# list items) made by each configuration over the Internet. The auction type used also has a saying on performance. Global-Grid improved 50.35% over Global-VPN in the best case, while Torus-Grid only 5.58% over Torus-VPN. This is because Torus connexions are limited to 4 cores, thereby reducing communication over the Internet. Figure 4 (bottom) shows the load distribution under each configuration. The best load distribution corresponds to Torus-Grid, as it has the lowest standard deviation of items processed in each of the 140 cores, see Table 1.

5.3 Response Time and Load Distribution under MM

Figure 5 shows response time (top) and load distribution (bottom) for MM under each configuration. The response times of all configurations are very similar. This is because MM is a static application (all data to be processed is known in advance and is distributed equally among processors at start of computation),

Table 1. N-Queens: number of Internet messages and standard deviation of items processed by each node under each configuration

DLML version	*# Requests*	*# Load Transfer*	*Standard Deviation*
Global-VPN	2,508,684	22,476	27,467,836.85
Global-Grid	17	1,767	17,386,869.94
Torus-VPN	20,319	14,467	5,698,175.98
Torus-Grid	28	2,352	3,593,140.42

and there is no imbalance. This experiment shows that the overhead of *DLML-Grid* hierarchical organisation is low.

Load distribution under MM (bottom in figure) is different to that under N-Queens. Global-Grid distributes better the load, while Torus-VPN distributes it the worst, but the spread between both is small (standard deviation in Table 2) and the effect on performance is not significant because it corresponds to the initial load distribution, and there is no imbalance. N-queens does incur imbalance out of dynamically generating data. Table 2 shows the number of messages over Internet; the number of requests is better for the Grid versions. In MM, load transfers are more expensive than in N-Queens, because in MM each list item is of size 2424 bytes, while in N-Queens is 96 bytes.

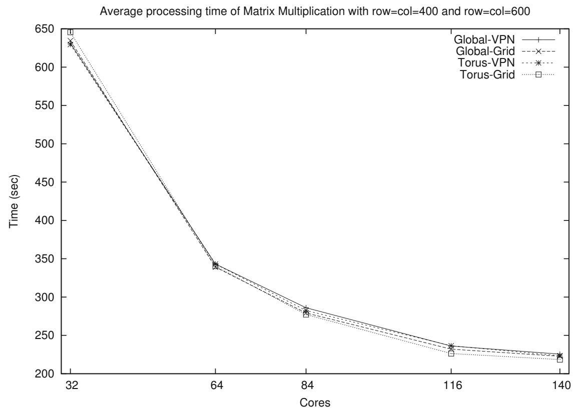
Table 2. Matrix Multiplication: number of Internet messages and standard deviation of items processed by each node under each configuration

DLML version	*# Requests*	*# Load Transfers*	*Standard Deviation*
Global-VPN	86,708	325,426	533.54
Global-Grid	6	296,552	512.99
Torus-VPN	8,196	326,681	540.65
Torus-Grid	9	297,520	532.96

6 Related Work

The architecture of *DLML-Grid* is based on the hierarchical model for load balancing in grid environments proposed in [10], but our load balancing algorithm is different to that proposed therein. Our algorithm relies on the underlying list processing model which triggers load balancing actions when a list (or set of lists in a cluster) becomes empty. The algorithm in [10] is more general in that it monitors the workload periodically in order to decide whether or not to balance the workload. The evaluation reported in [10] is based on simulation and thus our work complements it with similar results in a real platform.

Globus [1] is the *de facto* standard middleware to deploy Grid infrastructures. It offers mechanisms for resource discovery, localisation and management, and secure communication among others. A fully-fledged *DLML-Grid* would map its architecture onto Globus grid services; we are working on this.



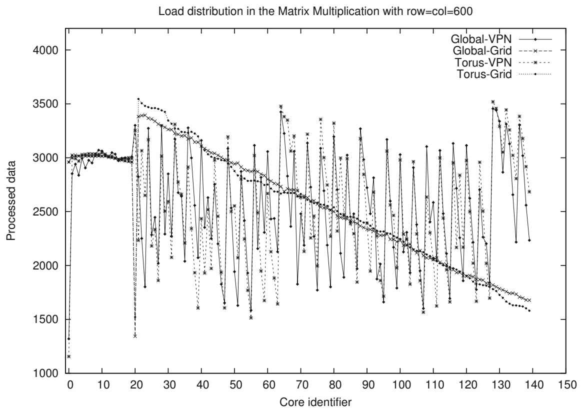


Fig. 5. MM: response time (top) and load distribution (bottom)

Skandium [6] is a library of skeletons (widely used parallel operations/patterns) written in Java that includes: farm, pipe, for, fork, divide and conquer, among others. Similar to DLML-Grid, Skandium distributes tasks, but does so only among cores within a single node (shared memory system).

Proactive [3] is a middleware written in Java that facilitates developing and running applications on Grids and P2P systems. It includes load balancing and fault tolerance mechanisms, and file transfer. Similar to DLML-Grid, this middleware makes load balancing based on partial information, but the load balance policy is initiated by the sender.

Probability Work-Stealing (PWS) and Hierarchical Work-Stealing (HWS) [7] have some similarity to the load balancing algorithm of DLML-Grid. PWS gives nearer processors more probability of being stolen work than more distant ones, while HWS manages group leaders and slaves and load balancing at two levels: (leader $\Rightarrow$ slaves) and (leader $\Leftrightarrow$ leader). Based on a threshold (chosen by the user), tasks are classified into *light* and *heavy* (in processing). The DLML-Grid hierarchy gives preference to intra-cluster load balancing which may be considered an inherent higher probability, and also as being of level (*slave* $\Leftrightarrow$ *slave*). However, DLML-Grid makes no distinction between light and heavy tasks. In HWS, a leader chooses at random another leader to steal work from it, while DLML-Grid follows an auction; we are working on trying the random approach which seems less costly. HWS has been tested on clusters (LAN) only.

7 Conclusions and Future Work

DLML-Grid is a hierarchical design of DLML that better uses resources on clusters of clusters typical of grid environments. The hierarchy tends to keep communication localised which is essential for good performance. Under both types of auctions used DLML-Grid performs much better than the counterpart configurations. The experiments with our static application show that the overhead of the hierarchical organisation is low. We are investigating further improvements to reduce communication overhead, such as overlapping communication and computation by making data requests to remote nodes before a local list becomes empty, and adding load monitoring to promote load balancing from overloaded to underloaded nodes by CMs, or between clusters by the GM.

References

1. Globus toolkit (February 2010), `http://www.globus.org/`
2. Bruen, A., Dixon, R.: The n-queens problem. Discrete Mathematics 12, 393–395 (1975)
3. Caromel, D., Delbé, C., di Costanzo, A., Leyton, M.: ProActive: an integrated platform for programming and running applications on grids and P2P systems. Computational Methods in Science and Technology 12(1), 69–77 (2006)
4. Dean, J., Ghemawat, S.: Mapreduce: Simplifed data processing on large clusters. In: Operating Systems Design and Implementation, pp. 137–149 (2004)
5. Feilner, M., Graf, N.: Beginning OpenVPN 2.0.9. Build and Integrate Virtual Private Networks Using OpenVPN. Packt Publishing (December 2009)
6. Leyton, M., Piquer, J.M.: Skandium: Multi-core programming with algorithmic skeletons. In: PDP, pp. 289–296 (2010)
7. Quintin, J.-N., Wagner, F.: Hierarchical Work-Stealing. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 217–229. Springer, Heidelberg (2010)
8. Santana-Santana, J., Castro-García, M.A., Aguilar-Cornejo, M., Roman-Alonso, G.: Load balancing algorithms with partial information management for the dlml library. In: 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 64–68 (2010)
9. Tanno, H., Iwasaki, H.: Parallel Skeletons for Variable-Length Lists in SkeTo Skeleton Library. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 666–677. Springer, Heidelberg (2009)
10. Yagoubi, B., Medebber, M.: A load balancing model for grid environment. In: 22nd International Symposium on Computer and Information Sciences, ISCIS 2007, pp. 268–274. IEEE (2007)

Reducing the Time to Tune Parallel Dense Linear Algebra Routines with Partial Execution and Performance Modeling*

Piotr Luszczek[1] and Jack Dongarra[1,2,3]

[1] University of Tennessee, Knoxville, TN, USA
{dongarra,luszczek}@eecs.utk.edu
[2] Oak Ridge National Laboratory, USA
[3] University of Manchester, United Kingdom

Abstract. We present a modeling framework to accurately predict time to run dense linear algebra calculation. We report the framework's accuracy in a number of varied computational environments such as shared memory multicore systems, clusters, and large supercomputing installations with tens of thousands of cores. We also test the accuracy for various algorithms, each of which having a different scaling properties and tolerance to low-bandwidth/high-latency interconnects. The predictive accuracy is very good and on the order of measurement accuracy which makes the method suitable for both dedicated and non-dedicated environments. We also present a practical application of our model to reduce the time required to tune and optimize large parallel runs whose time is dominated by linear algebra computations. We show practical examples of how to apply the methodology to avoid common pitfalls and reduce the influence of measurement errors and the inherent performance variability.

Keywords: Linear systems, parallel algorithms, modeling techniques.

1 Introduction

Dense systems of linear equations are found in numerous applications, including: airplane wing design; flow around ships and other off-shore constructions; diffusion of solid bodies in a liquid; noise reduction; and diffusion of light through small particles. Of particular interest is the solution of the so-called radar cross-section problem – the principal equation is the Helmholtz equation. To solve this equation, researchers use the method of moments [19,29]. In the case of fluid flowthe boundary integral solution is known as the panel methods [20,21].A typical approach to solving such systems is to use LU factorization. Another major source of large dense linear systems is problems involving the solution of boundary integral equations [14]. A recent example of the use of dense linear algebra at a very large scale is physics plasma calculation in double-precision complex arithmetic based on Helmholtz equations [2].

* This research was supported by DARPA through ORNL subcontract 4000075916 as well as NSF through award number 1038814. We would like to also thank Patrick Worley from ORNL for facilitating the large scale runs on Jaguar's Cray XT4 partition.

Finally, virtually all large supercomputer sites do run the High Performance LINPACK (HPL) benchmark [13] which is primarily based on dense linear algebra. The reduction of time to run the benchmark is of paramount importance. The first machine on the 34th TOP500 list took over 20 hours to complete the HPL run [17]. In 2011, this time goes up to 30 hours [18]. And this is only a single run not counting the time spent in optimizing the parameters for the run. In this paper we address the problem of increasing execution time with performance modeling. This is an extension of our previous work [10] with more comprehensive data sets.

2 Related Work

Execution model for HPL based on Self-Similarity Theory and the Π-Theorem models floating-point performance on a p by q process grid [24]: $r_{\text{fp}} = \gamma\, p^{\alpha} q^{\beta}$. The values for α, β, and γ need to be determined experimentally. Because of the nonlinear relation between these coefficients, we need to use a nonlinear optimization to fit the model to the experimental data.

A more complex models for HPL and other parallel linear algebra routines resort to modeling each individual period of time spent in every non-trivial computational or communication routine involved in the factorization and back-solve [15,9]. They tend to give accurate results provided that they are populated with accurate software and hardware parameters such as computational rate of execution as well as bandwidth and latency of both memory and the interconnect network.

An attempt to model HPL using memory traces [30] resulted in scoring HPL on par if not worse with an FFT implementation in terms of memory locality. This was due to the use of reference BLAS implementation and lack of accounting for register reuse. Our model captures the efficiency of register reuse.

Partial execution was successfully used to perform cross-platform performance predictions of scientific simulations [33]. The study relied on the iterative nature of the simulated codes. Dense linear algebra computations, as opposed to iterative methods [11], are not iterative in nature and commonly exhibit non-linear variation in performance.

Performance prediction in the context of grid environments focuses work load characterization and its use in effective scheduling and meta-scheduling algorithms[28]. The techniques used in such characterization tend to have a higher error rates (“between 29 and 59 percent of mean application run times”).

3 Performance Prediction by Correlation

One of the building blocks of dense linear algebra solvers and, by far, the main source if their high performance is a dense matrix-matrix multiply routine – a Level 3 BLAS [8,7] called DGEMM. Naturally then, the sequential version of the routine may be used to estimate the time to run as well as the performance of a parallel dense solve. By utilizing the information from the publicly available[1] results of the HPC Challenge benchmark [23]: during a single execution both matrix-matrix multiplication and HPL are

[1] For more details please visit `http://icl.cs.utk.edu/hpcc/`

benchmarked which should provided a consistent experimental setup. DGEMM and HPL are indeed correlated based on over 250 entries in the HPCC database. In fact, the Pearson product-moment correlation coefficient [27] exceeds 99%. This is a somewhat deceptive achievement though. If we use DGEMM as a predictor for HPL then the median relative prediction error will be just over 15% and the smallest one will be 1.4%. Even if we generously dismiss all the results with greater-than-median error then we are still left with 1% to 15% variability in prediction accuracy.

4 Execution Model of HPL

Our goal is to come up with a comprehensive model for HPL without resorting to counting complexities of each and every routine involved in the factorization and back-solve as was done by Cuenca et al. [15] and Emmanuel Jeannot and Julien Langou [9]. The model for the HPL's floating-point execution rate is influenced by the operation count and the time to perform the solve. The operation count is fixed regardless of the underlying algorithm to facilitate performance comparisons:

$$\mathrm{op}_{\mathrm{count}} = {}^{2}/_{3}\, n^3 + {}^{3}/_{2}\, n^2 \tag{1}$$

The time to do the solve has three components:

$$t = Fn^3 + Bn^2 + Ln + C \tag{2}$$

where F represents the inverse of the actual floating-point rate of the update phase of the LU factorization commonly refered to as the Schur's complement [16]. Values of F differ with the algorithm of choice: left-looking, right-looking, top-looking, Crout, recursive, etc. [12]. The B term corresponds to $O(n^2)$ floating-point operations (primarily panel factorizations) and various bandwidth levels such as between the cache and the main memory as well as the interconnect bandwidth. As B embodies execution rate of second-order terms, its value changes with the peak performance and bandwidth imbalance: the execution rate included in B may be an order of magnitude lower than the one represented by F. L mainly corresponds to both the memory hierarchy as well as the interconnect latency. Finally, C represents constant overheads such as populating caches with initial values and initializing network's communication layer. The floating-point rate is obtained as a ratio of operation count and the time to solution: $r_{\mathrm{fp}} = \frac{\mathrm{op}_{\mathrm{count}}}{t}$.

To an extent, the above model takes into account non-linear dependence of the running time on some algorithmic constants such as blocking factor NB. These constants may be hidden inside F, B, L, and C as long as they don't change with n. The algorithmic parameters[2] that may be hidden in this manner include: blocking factor NB, logical process grid order and so on. The reason why these parameters may be accounted for by properly selecting F, B, L, and C, is that they can be fixed and don't change with number of cores or the problem size N.

From the data analysis standpoint, performance is the inverse of time multiplied by the matrix size cubed: this translates to amplification of the experimental and measurement errors by a large quantity. Naturally then, time to run HPL is less sensitive to

[2] For more details see http://www.netlib.org/benchmark/hpl/

the errors and outlying data points. The time is then chosen for modeling. By making this choice, we alleviate the influence of outliers on the data and thus we avoid the necessity of using non-linear statistical methods that involve medians. The model fitting may be performed with the standard least-squares formulation rather than its non-linear counterparts that are less developed and suffer from the inaccuracies of non-smooth optimization [3]. As described above, the time to run HPL can be written as:

$$t = f_3 n^3 + f_2 n^2 + f_1 n + f_0 \tag{3}$$

Given a k number of experiments with varying problem sizes $n_1, n_2, n_3, \ldots, n_k$ we obtain the actual running times $t_1, t_2, t_3, \ldots, t_k$. Using the results of these experiments, we formulate the problem as a linear least-squares problem:

$$\begin{bmatrix} n_1^3 & n_1^2 & n_1 & 1 \\ n_2^3 & n_2^2 & n_2 & 1 \\ & & \ldots & \\ n_k^3 & n_k^2 & n_k & 1 \end{bmatrix} \begin{bmatrix} f_3 \\ f_2 \\ f_1 \\ f_0 \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ \ldots \\ t_k \end{bmatrix} \tag{4}$$

or more compactly:

$$Af = t \tag{5}$$

with $A \in \mathbb{R}^{k\times 4}$, $f \in \mathbb{R}^4$, and $t \in \mathbb{R}^k$. The absolute modeling error can then be defined as

$$M_{\text{err}} = \|Af - t\|_\infty \tag{6}$$

with the assumption that the entries of t are relatively accurate. Such is the case when we use the median of multiple time measurements.

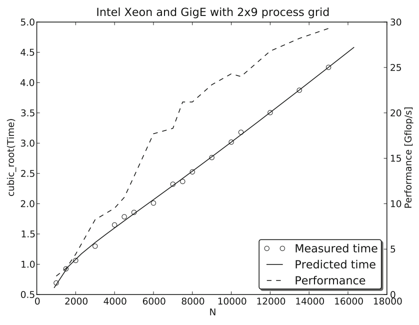
The system matrix A from equation (5) is a Vandermode matrix and tends to be badly conditioned [16]. Matrices from typical experiments can have a norm-2 condition number as high as 10^{11} which means that the model fitting needs to be performed in double-precision arithmetic even though the data itself has only a handful of significant digits worth of accuracy. Equilibration [1] may reduce the condition number by nearly ten orders of magnitude but the resulting modeling error gets reduced only slightly. A similar reduction of the modeling error may be achieved with a simple row scaling that results from dividing each row of the linear system (4) by the respective problem size:

$$a_3 n_i^2 + a_2 n_i + a_1 + a_0/n_i = t_i/n_i \tag{7}$$

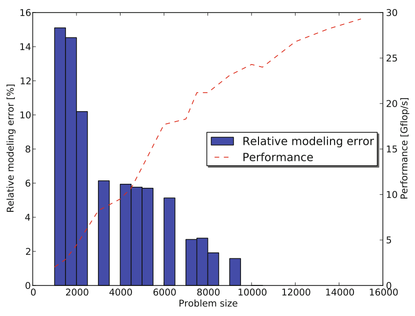
The reduction of error may be as high 20 percentage points.

Figure 1(a) shows how the model performs on a non-dedicated cluster[3] comprised of commodity hardware components. The modeling error is 14% when all the data points are accounted for. The most problematic are measurements for small n values. If a handful of these initial data points is removed then the modeling error drops to just over 2% which is within the noise levels of a non-dedicated system. Thus, the method is sensitive to the measuring error but (in numerical sense) is stable because it delivers the answer whose quality is close to the quality of the input data – a property that is an established standard for properly implemenented numerical libraries [31,32].

[3] Dual-core 1.6 GHz Intel Core 2 with Gigabit Ethernet interconnect.



(a) Time and performance



(b) Error and performance

Fig. 1. Modeled versus measured time to run HPL on a common cluster. The cubic root of time is plotted to increase graph's clarity. Performance numbers are shown for reference only.

Figure 1(b) shows how the error is reduced (the farther to the right the shorter the bars) as the left-most points are eliminated. The explanation is that the leftmost data points do not represent the asymptotic performance rate of HPL: they cannot be modeled with Equation (3) because the attained performance varies significantly with the problem size (coefficients a_i are no longer constant – instead they are a function of n).

5 Modeling ScaLAPACK: A Generic Linear Algebra Library

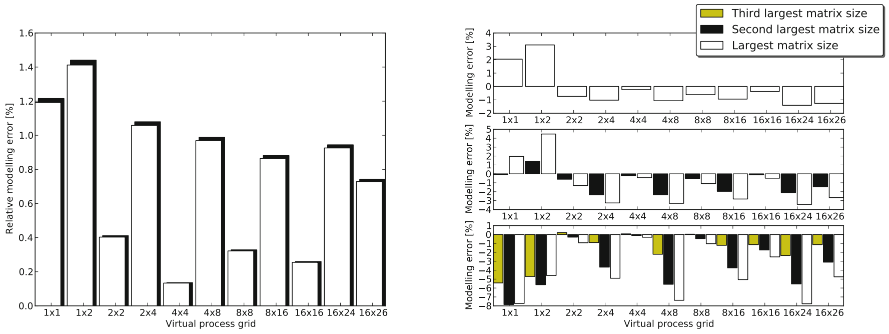
As mentioned earlier, our modeling methodology is applicable to more than just HPL. For example, Table 1 shows the modeling errors achieved on a dedicated cluster[4] running LU factorization available in ScaLAPACK [4,6]. The table indicates very high accuracy (mostly around 1% and not exceeding 3%) provided that the measurements with different virtual process grids are not modeled together but rather treated separately as they exhibit different scalability properties [26,5]. Similarily, the size of the matrix blocking factor influences the tradeoff between the local performance and the ability to tolerate low-bandwidth/high-latency at the interconnect level [25].

Table 1. Various logical process grids and the corresponding modeling error for LU factorization as implemented in ScaLAPACK

Virtual process grid		Modeling error
rows	columns	[%]
1	30	0.7
2	15	1.0
3	10	0.8
5	6	2.2

Virtual process grid		Modeling error
rows	columns	[%]
6	10	1.6
1	60	0.5
3	20	0.7
2	30	0.5
4	15	2.2
5	12	2.7

[4] Intel 2.4 GHz Pentium 4 cluster with Gigabit Ethernet interconnect.



(a) Overall relative modeling error

(b) Relative prediction error

Fig. 2. Relative modeling and prediction error. On the right figure: the largest matrix size (top), the largest and the second largest matrix size (middle), as well as the first, second, and third largest matrix size (bottom) on a single-core dual-processor Intel Xeon 3.2 GHz cluster with 416 processors connected with InfiBand interconnect.

For ScaLAPACK's main three one-sided factorizations on a shared-memory multi-core machine[5], the error is larger (around 15%) then in previous dedicated runs. This experiment shows how our modeling framework performs in an environment with noise in the collected data. In this case, the noise comes from a stock Linux kernel installation without necessary optimizations for large shared memory installation. The standard deviation relative to the median for 16 time measurements for a relatively small problem size ($n = 2000$, median time under half a second) is around 15%. As shown previously, the modeling error is on the order of the standard deviation of time measurement which indicates to us that the method is as accurate as the quality of its input data.

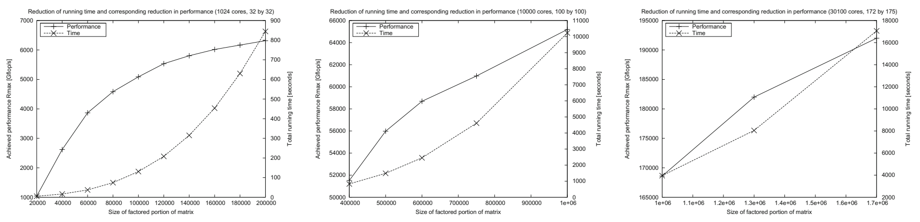
6 Quality of Prediction for Extrapolation

Arguably, the most useful aspect of modeling is extrapolation of acquired data. High quality models are able to provide information about runs with larger data sets based on the results from runs with the smaller ones. To evaluate our approach we chose a publicly availble data set of the HPC Challenge (HPCC) benchmark results[6]. Figure 2(a) shows a relative modeling error for all the available results sorted by the virtual process grids from 1-by-1 to 16-by-26. As before, the model is accurate and the relative error stays around 1% and often drops to a fraction of a percent.

A more interesting application of our model is to compute the time to run without actually running the code. Figure 2(b) illustrates this kind of experiment using the same data as was modeled in Figure 2(a). There are eleven process grid in the modeled data

[5] The machine has 8 NUMA nodes with an AMD Instanbul 2.8 GHz 6-core processor for each node and a stock Linux kernel installation.

[6] The data set comes from runs performed in 2007 and more information on the used hardware is available at `http://icl.cs.utk.edu/hpcc/custom/index.html?lid=111&slid=218`



(a) Size: 0.2×10^6; 1024 cores (b) Size: 1×10^6; 10,000 cores (c) Size: 1.7×10^6; 30,100 cores

Fig. 3. Performance of end section factorization with various matrix sizes on Cray XT 4 with various core counts

set and each grid was used seven times to execute the HPCC benchmark with increasing problem sizes. However, for any given process grid, the first problem size corresponded to the same percentage of the memory or (in other words) the same amount of data per process was used. Then, the second process size corresponded to twice as large amount of data per process and so on. If we look at this data set from the perspective of our model, we may treat first six data points as measurements and use them for obtaining the coefficients of the model and then we can predict the runtime for the remaining seventh data point. The modeling error for each process grid is presented in the top graph of Figure 2(b). The middle graph shows the scenario where 5 data points establish the coefficients and the remaining two data points are predicted. Finally, the bottom of the figure shows the situation where 4 data points are used to compute the coefficients and the remaining 3 data points are being predicted. Since our model has 4 coefficients we need at least 4 data points to calculate them. By looking at Figure 2(b) we see that the modeling error is small, always below 8% and often within or below 1%. What the figure doesn't show is the reduction of time that a prediction would afford. With 6 data points for modeling and 1 data point predicted (top of the figure) there are no savings in time, in fact the time to run the first 6 experiments is about 20% longer then the time it takes to run the remaining seventh experiment. In the case of 5 data points for modeling (middle of the figure), there is already a substantial reduction of time: the last two experiments take about 70% of the total time. And finally, with 4 experiments predicting the 3 longest runs, the savings in time exceed 90%.

7 Combined Application of Modeling and Sampled Factorization

To test our modeling framework together with the sampled factorization approach, we performed a series of experiments at the Oak Ridge National Laboratory on a large scale supercomputer: Cray XT 4 with quad-core AMD Opteron 1345 processor clocked at 2.6 GHz. The whole installation consists of 31328 cores. Figures 3(a), 3(b), and 3(c) show performance and running time for varying number of cores, matrix sizes, and portions of fully factored matrix. The code used for runs was the HPL code modified to allow for partial execution as described earlier. Despite the order of magnitude

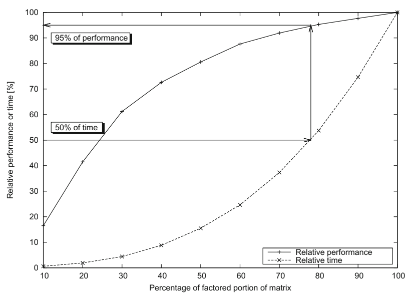


Fig. 4. Relative running time and achieved performance with relation to fraction of factored portion of matrix (1024 cores, with 32 by 32 virtual process grid.)

difference in core counts and matrix sizes, the model is accurate to about 1% or less. The *performance-time curves* from the figures show that in order to attain comparable fraction of the peak performance the time to run will increase faster than linearly with the number of cores.

Figure 4 shows the performance-time curve from Figure 3(a) but this time it is recast in relative terms as a fraction of the maximum attained performance. In this manner both time and performance may coexist on the same Y-axis because they both vary between 0% and 100%. In this setting it is now easy to perform a what-if analysis of the data as it is indicated in the Figure 4 with arrows. The question being answered by the Figure is this: if the time to run the benchmark is reduced by 50% how much will the performance result be reduced. The answer is encouraging: the resulting performance drop will only be 5%. The making of such predictions is simplified with our model based on a handful of runs that accurately determine the modeling coefficients. Without a model, it would be necessary to make many more additional runs which would diminish the benefits of reduced running time. And even with the extra runs the exact point of 50% reduction of time is unlikely to be found experimentally as is the case in Figure 4 and a curve fitting or an approximation technique would be necessary to provide more refined guidance for finding the right problem size. These issues are already accounted for in our model. The sampled factorization provides the added benefit of stressing the entire system: the system matrix occupies the entire memory. Also, the result of the partial execution can be verfied as rigorously as is the case for the standard factorization.

8 Conclusions and Future Work

We presented a modeling framework and its applications to prediction of performance across very diverse hardware platforms ranging from commodity clusters to large scale supercomputer installations. The accuracy is as good as the quality of the input data which has been a golden standard for numerical linear algebra for years [31,32]. Thus, we consider the model to be validated by experiments with varying measurement errors due to both multi-user environment and operating system noise. We applied our methodology to facilitate what-if analysis that can inform decisions regarding the tradeoff between higher performance and shorter running time. We plan to increate the detail of our model including sensitivity analysis and the ability to indicate measurement errors, hardware problems, or software misconfiguration [22].

References

1. Anderson, E., Bai, Z., Bischof, C., Blackford, S.L., Demmel, J.W., Dongarra, J.J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.C.: LAPACK User's Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia (1999)
2. Barrett, R.F., Chan, T.H.F., D'Azevedo, E.F., Jaeger, E.F., Wong, K., Wong, R.Y.: Complex version of high performance computing LINPACK benchmark (HPL). Concurrency and Computation: Practice and Experience 22(5), 573–587 (2010)
3. Björk, Å.: Numerical methods for Least Squares Problems. SIAM (1996) ISBN 0-89871-360-9
4. Suzan Blackford, L., Choi, J., Cleary, A., D'Azevedo, E.F., Demmel, J.W., Dhillon, I.S., Dongarra, J.J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D.W., Clint Whaley, R.: ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia (1997)
5. Chen, Z., Dongarra, J., Luszczek, P., Roche, K.: Self-adapting software for numerical linear algebra and LAPACK for Clusters. Parallel Computing 29(11-12), 1723–1743 (2003)
6. Choi, J., Dongarra, J.J., Ostrouchov, S., Petitet, A., Walker, D.W., Clint Whaley, R.: The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Scientific Programming 5, 173–184 (1996)
7. Dongarra, J., Du Croz, J., Duff, I., Hammarling, S.: Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Soft. 16(1), 18–28 (1990)
8. Dongarra, J., Du Croz, J., Duff, I., Hammarling, S.: A set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Soft. 16(1), 1–17 (1990)
9. Dongarra, J., Jeannot, E., Langou, J.: Modeling the LU factorization for SMP clusters. In: Proceeedings of Parallel Matrix Algorithms and Applications (PMAA 2006), September 7-9. IRISA, Rennes, France (2006)
10. Dongarra, J., Luszczek, P.: Reducing the time to tune parallel dense linear algebra routines with partial execution and performance modelling. In: Poster Session of SC 2010, New Orleans, Louisianna, USA, November 13-19 (2010), Also: Technical Report UT-CS-10-661, University of Tennessee, Computer Science Department
11. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. Society for Industrial and Applied Mathematics, Philadelphia (1998)
12. Dongarra, J.J., Gustavson, F.G., Karp, A.: Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. SIAM Review 26(1), 91–112 (1984)
13. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience 15, 1–18 (2003)
14. Edelman, A.: Large dense numerical linear algebra in 1993: the parallel computing influence. International Journal of High Performance Computing Applications 7(2), 113–128 (1993)
15. García, L.-P., Cuenca, J., Giménez, D.: Using Experimental Data to Improve the Performance Modelling of Parallel Linear Algebra Routines. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 1150–1159. Springer, Heidelberg (2008) ISSN 0302-9743 (Print) 1611-3349 (Online), doi:10.1007/978-3-540-68111-3

16. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. The Johns Hopkins University Press, Baltimore and London (1996)
17. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: TOP500 Supercomputer Sites, 34th edn. (November 2009), http://www.netlib.org/benchmark/top500.html and http://www.top500.org/
18. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: TOP500 Supercomputer Sites, Hambug, Germany, 37th edn. (June 2011), http://www.netlib.org/benchmark/top500.html and http://www.top500.org/
19. Harrington, R.: Origin and development of the method of moments for field computation. IEEE Antennas and Propagation Magazine (June 1990)
20. Hess, J.L.: Panel methods in computational fluid dynamics. Annual Reviews of Fluid Mechanics 22, 255–274 (1990)
21. Hess, L., Smith, M.O.: Calculation of potential flows about arbitrary bodies. In: Kuchemann, D. (ed.) Progress in Aeronautical Sciences, vol. 8. Pergamon Press (1967)
22. Kerbyson, D.J., Hoisie, A., Wasserman, H.J.: Verifying Large-Scale System Performance During Installation using Modeling. In: High Performance Scientific and Engineering Computing, Hardware/Software Support. Kluwer (October 2003)
23. Luszczek, P., Dongarra, J., Kepner, J.: Design and implementation of the HPCC benchmark suite. CT Watch Quarterly 2(4A) (November 2006)
24. Numerich, R.W.: Computational forces in the Linpack benchmark. Concurrency Practice and Experience (2007)
25. Oram, A., Wilson, G. (eds.): Beautiful Code. O'Reilly (2007), Chapter 14: How Elegant Code Evolves with Hardware: The Case of Gaussian Elimination
26. Roche, K.J., Dongarra, J.J.: Deploying parallel numerical library routines to cluster computing in a self adapting fashion. In: Parallel Computing: Advances and Current Issues. Imperial College Press, London (2002)
27. Rodgers, J.L., Nicewander, W.A.: Thirteen ways to look at the correlation coefficient. The American Statistician 42, 59–66 (1988)
28. Smith, W., Foster, I., Taylor, V.: Predicting application runt times with historical information. In: Proceedings of IPPS Workshop on Job Scheduling Strtegies for Parallel Processing. Elsevier Inc. (1998), doi:10.1016/j.jpdc.2004.06.2008
29. Wang, J.J.H.: Generalized Moment Methods in Electromagnetics. John Wiley & Sons, New York (1991)
30. Weinberg, J., McCracken, M.O., Strohmaier, E., Snavely, A.: Quantifying locality in the memory access patterns of HPC applications. In: Proceedings of SC 2005, Seattle, Washington. IEEE Computer Society Washington, DC (2005)
31. Wilkinson, J.H.: Rounding Errors in Algebraic Processes. Prentice Hall, Englewood Cliffs (1963)
32. Wilkinson, J.H.: The Algebraic Eigenvalue Problem. Oxford University Press, Oxford (1965)
33. Yang, L.T., Ma, X., Mueller, F.: Cross-platform performance prediction of parallel applications using partial execution. In: Proceedings of the ACM/IEEE SC 2005 Conference (SC 2005). IEEE (2005)

A General-Purpose Virtualization Service for HPC on Cloud Computing: An Application to GPUs

Raffaele Montella[1], Giuseppe Coviello[1], Giulio Giunta[1], Giuliano Laccetti[2], Florin Isaila[3], and Javier Garcia Blas[3]

[1] Department of Applied Science, University of Napoli Parthenope, Italy
[2] Department of Mathematics and Applications
University of Napoli Federico II, Italy
[3] Department of Computer Science, University of Madrid Carlos III, Spain

Abstract. This paper describes the generic virtualization service GVirtuS (Generic Virtualization Service), a framework for development of split-drivers for cloud virtualization solutions. The main goal of GVirtuS is to provide tools for developing elastic computing abstractions for high-performance private and public computing clouds. In this paper we focus our attention on GPU virtualization. However, GVirtuS is not limited to accelerator-based architectures: a virtual high performance parallel file system and a MPI channel are ongoing projects based on our split driver virtualization technology.

Keywords: HPC, Cloud Computing, Virtualization, Split Driver, GPGPU.

1 Introduction

In recent years the general purpose graphics processing units (GPGPUs) have become attractive cost-efficient platforms for scientific computing community. GPGPUs have a shared-memory massive multicore architecture, which can be efficiently leveraged in order to accelerate data-parallel computing tasks [1]. An active research field currently focuses on exploiting special purpose processors as accelerators for general-purpose scientific computations. In addition to high-performance, GPGPUs have the advantage of being energy-efficient, as the high number of processing elements operating at low frequency dissipate less heat than one core with the same aggregate frequency [2].

Cloud computing offers an appealing elastic distributed infrastructure, but whether and how it can efficiently provide the high performance required by most e-science applications is still a research issue [9]. Especially in the field of parallel computing applications, virtual clusters deployed on cloud infrastructures suffer from the low performance of message passing communication among virtual machine instances and from the difficulties to access hardware specific accelerators such as GPUs [3].

In this paper we describe a general-purpose virtualization service (GVirtuS) for high performance computing applications on cloud computing environments, focusing on GPU virtualization and distributed memory virtual clusters. The rest of this work has the following structure. Section 2 introduces GVirtuS and describes the evolution of its architecture and implementation. Section 3 discusses GPU virtualization. Section 5 presents some preliminary experimental results. Section 5 overviews related work. Finally, Section 6 concludes and outlines future work directions.

2 GVirtuS: General Virtualization Service

GVirtuS extends and generalizes gVirtus (with lower case g), a GPU virtualization solution proposed in our previous work [4]. The main motivation of gVirtuS was to address the limitations of transparently employing accelerators such as CUDA-based GPUs in virtualization environments. Before gVirtuS, using GPUs required explicit programming of the communication between virtual and physical machines and between guest and host operating systems. Furthermore, the solution had to deal with issues related to the vendor specific interfaces on the virtual machine side. These limitations drastically reduced the productivity of virtualization solutions based on GPUs and hindered the employment of accelerators as on-demand resources in cloud computing infrastructures.

The generic virtualization service presented in this work, GVirtuS (Generic Virtualization Service) is a framework for facilitating the development of split-drivers for virtualization solutions (as shown in Figure 1). As the previous gVirtus, the brightest GVirtuS feature is the independence from all involved technologies: the hypervisor, the communicator and the target of the virtualization (general purpose graphics processing units for computing acceleration, high performance network interface cards, distributed parallel file systems, measurement instruments and data acquisition interfaces).

GVirtuS, the software component presented in this paper is different from his ancestor gVirtus in the following ways. While gVirtuS proposes a virtualization solution for CUDA, GVirtuS offers virtualization support for generic libraries such as accelerator libraries (OpenCL, OpenGL and CUDA as well), parallel file systems, communication libraries (MPI). gVirtuS required explicit porting on virtualization technologies of frontends, bbackends, and communicators. Further, GVirtuS targets to facilitate independence of virtualization technology by offering generic interfaces, which simplifies porting virtualization solutions across platforms. Finally, GVirtuS could be seen as an abstraction for generic virtualization in HPC on cloud infrastructures.

In GVirtuS the split-drivers are abstracted away, while offering the developers abstractions of common mechanisms, which can be shared for implementing the desired functionality. In this way, developing a new virtualization driver is simplified, as it is based on common utilities and communication abstractions. GVirtuS abstracts away frontends, bbackends, and communicators. The GVirtuS software stack is designed in a modular fashion: the frontend, the communicator, and the backend are implemented as plug-ins. For each virtualized device the frontend and the backend are cooperating, while both of them are completely independent from the communicator. Developers can focus their efforts on virtual device and resource implementation without taking care of the communication technology.

2.1 Frontends

The frontend is the component used by the applications running in the virtual machine to access services and resources provided by the virtualized device. The frontend provides applications the routines for requesting physical device services in a transparent way and using the same API offered by device driver manufacturers (device library and/or driver interface). This choice permits to run applications in virtual environments without modifications.

In GVirtuS the frontend is implemented as a stub library. A stub library is a virtualization of the physical driver library on the guest operating system. The stub library implements the driver functionality in the guest operating system in cooperation with the backend running on the host operating system. The communication between the frontend and backend is done via abstract communicators.

2.2 Backends

The backend is a component serving frontend requests through the direct access to the driver of the physical device. This component is implemented as a server application waiting for connections and responding to the requests submitted by frontends. In an environment requiring shared resource access (as it is very common in cloud computing), the back-end must offer a form of resource multiplexing. Another source of complexity is the need to manage multi-threading at the guest application level.

A daemon runs on the host operating system in the user space or super user space depending on the specifics of applications and security policies. The daemon implements the back-end functionality dealing with the physical device driver and performing the host-side virtualization.

2.3 Communicators

A communicator is a key piece of software in GVirtuS stack because it connects the guest and host operating systems. The communicators have strict high-performance requirements, as they are used in system-critical components such as split-drivers. Additionally, in a virtual machine environment the isolation between host and guest and among virtual machines is a design requirement. Consequently, the communicator main goal is to provide secure high-performance direct communication mechanisms between guest and host operating systems.

In GVirtuS the communicators are independent of hypervisor and virtualized technology. Additionally, novel communicator implementations can be provided independently from the cooperation protocols between frontend and back-end.

GVirtuS provides several communicator implementations including a TCP/IP communicator. The TCP/IP communicator is used for supporting virtualized and distributed resources. In this way a virtual machine running on a local host could access a virtual resource physically connected to a remote host in a transparent way.

However, in the some application scenarios the TCP/IP based communicator is not feasible because of the following limitations:

- The performance is strongly impacted by the protocol stack overhead.
- In a large size public or private cloud computing environment the use of the network could be restricted for security and performances reasons.
- For protection reasons a virtual machine may be unaware of the network address of its hosting machine.

For addressing these potential limitations, GVirtuS open solution allows for future protocols to be simply integrated into the architecture without any frontend or backend modification.

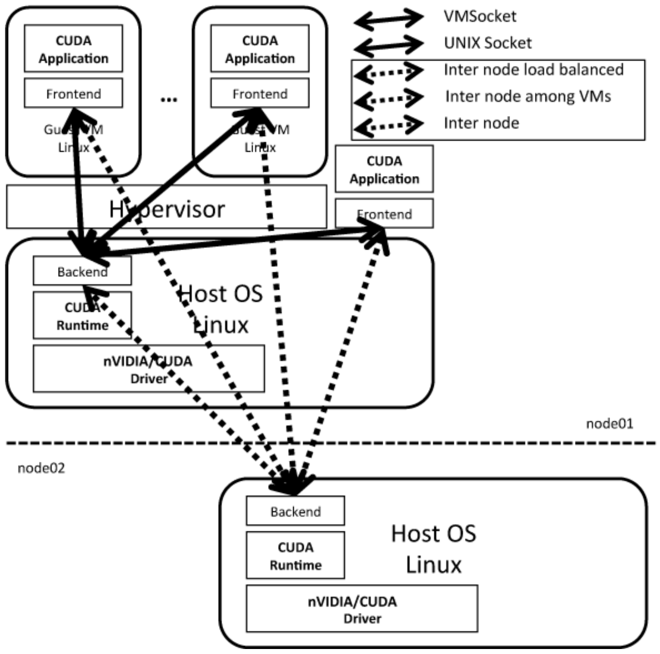


Fig. 1. GVirtuS architecture in a distributed virtualized GPU scenario

3 Implementing the GPU Virtualization Plug-in

As stated above, GVirtuS is a generalization of the previously developed gVirtuS. In this paper we focus on acceleration services provided by the nVIDIA in order to support the CUDA programming model. More exactly, we target two main goals: to provide a fully transparent virtualization solution (CUDA enabled software has to be executed in a virtual environment without any further modification of binaries or source code) and to reduce the overhead of virtualization so that the performance of the virtualized solution is as close as possible to the one of the bare metal execution.

The GVirtuS framework is implemented as a collection of C++ classes. The utility library contains the common software components used by both frontend and back-end. This library provides the support for communication between the frontend and the backend. The communicator component is implemented behind an interface called Communicator. GVirtuS already provides several Communicator subclasses such as TCP/IP, Unix sockets, VMSocket (high performance communicator for KVM based virtualization) [5], and VMCI (VMWare efficient and effective communication channel for VMWare based virtualization) [6,7].

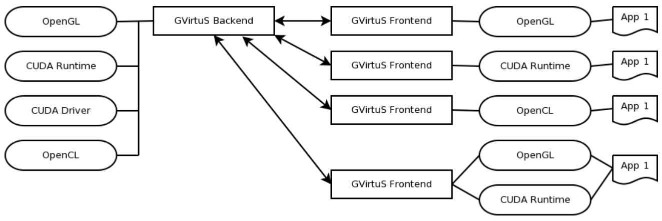


Fig. 2. GVirtuS nVIDIA/CUDA plug-in schema

The frontend library allows the development of the frontend component and contains a single class, called Frontend. There is only one instance of this class for each application using the virtualized resource. This instance is in charge of the backend connection, which is based on an implementation of a Communication interface. This is a critical issue especially when the virtualized resources have to be thread-safe as in the case of GPUs providing CUDA support. The methods implemented in this class support request preparation, input parameters management, request execution, error checking, and output data recovery.
The backend is executed on the host machine. It waits for connections from frontends. As a new connection is incoming it spawns a new thread for serving the frontend requests. The application running on the virtual machine requests services to the virtualized device via the stub library. Each function in the stub library follows these steps: 1) obtains a reference to the single frontend instance, 2) uses frontend class methods for setting the parameters, 3) invokes the frontend handler method specifying the remote procedure name, and 4) checks the remote procedure call results and handles output data.

As an improvement over the nVIDIA/CUDA virtualization offered by gVirtuS, we used the general-purpose virtualization service GVirtuS to provide virtualization support for CUDA, openCL and openGL. The CUDA driver implementation is similar to the CUDA runtime except for the low-level ELF binary management for CUDA kernels. A slightly different strategy has been used for openCL and openGL support. The openCL library provided by nVIDIA is a custom implementation of a public specification. That means that the openCL GVirtuS wrap is independent of the openCL implementation. Based on the implemented split driver a software using openCL running on a virtual machine could use nVIDIA or ATI accelerated devices physically connected to other remote machines in a transparent and dynamic way. As in the case of openCL, openGL wrap is a custom implementation of a public specification. In order to support remote visualization in a high performance way a VNC based screen virtualization had to be implemented (Figure 2).

Each GPU physical devices executes kernels from VMs as in first come first served fashion. If multiple GPU devices are available a round robin schema is applied. More complex GPU scheduling policies could be implemented as pluggable components. In a high performance on demand cloud computing scenario, different QoS levels could be enforced enabling remote GPU sharing.

4 Experimental Results

We have designed and implemented the GVirtuS virtualization system with the primary goal of using GPUs in a private cloud setup for high performance scientific computing. We have set up a prototype IaaS cloud system based on open source software for cloud management [8], [9], [10] based on the same interface as Amazon Web Services [11]. The system gives users the ability to run and control entire virtual machine instances deployed across a variety of physical resources. Our objective was to set up a high-performance computing cluster provisioning on-demand resources of an already existing infrastructure [12]. Our experimental evaluation aims at assessing the performance of GVirtuS in a high performance computing cloud system. The experiments have been performed in cluster equipped with Intel-based 4-core processors with hyperthreading (i7-940 2,93 133 GHz, Quad Core 8 MB cache, 12 GB of RAM) workstation imitating a computing nodes in a bigger cluster infrastructure. Each compute node is equipped with a Tesla C1060 with 240 CUDA cores, 4 GB of on-board memory and 1.3 compute capabilities. The software stack is based on Ubuntu 10.04 Linux operating system, nVIDIA CUDA drivers and Tools/SDK 3.x, and KVM-QEMU (kvm-88).

4.1 GVirtuS/CUDA Performance Analysis

The Table 1 reports some experimental results carried out running tree nVIDIA SDK sofware in different scenarios:

- 0: No virtualization, no accleration (blank)
- 1: Acceleration without virtualization (target)
- 2,3: Virtualization with no acceleration
- 4...6: GPU acceleraion, Tcp/Ip communication
- 7: GPU acceleration using GVirtuS, Unix Socket based communication
- 8,9: GVirtuS virtualization

This benchmarks have been performed in order to understand the impact of the CPU virtualization and the overhead introduced by GVirtuS/CUDA stack. The scenario defined as 0 is the experiment blank where none acceleration has been used. With the number 1 is pointed out the target as the maximum performance achievable. Comparing the results labelled as 0 with ones labelled as 2 and 3 is evaluated the CPU virtualization overhead (averagely about the 6%). Scenarios number 4 to 6 has similar performances due to communication overhead. The scenario 7 depict the weight of the GVirtuS stack without considering virtualization and communication between VMs and the physical host. Finally 8 and 9 represent performances using GVirtuS and CUDA virtualization comparing KVM+vmSocket (8) and VMWare+VMCI (9): the open source stack presented in this paper performs in the best way (8).

Table 1. CUDA/SDK benchmarks: computing times as Host-CPU rate

n	Hypervisor	Comm.	Hystogram	matrixMul	scalarProd
0	Host	CPU	100.00%	100.00%	100.00%
1	Host	GPU	9.50%	9.24%	8.30%
2	KVM	CPU	105.57%	99.48%	106.75%
2	Kvm	CPU	105.57%	99.48%	106.75%
3	VM-Ware	CPU	103.63%	105.34%	106.58%
4	Host	Tcp/Ip	67.07%	52.73%	40.87%
5	Kvm	Tcp/Ip	67.54%	50.43%	42.95%
6	VM-Ware	Tcp/Ip	67.73%	50.37%	41.54%
7	Host	AfUnix	11.72%	16.73%	9.09%
8	Kvm	vmSocket	15.23%	31.21%	10.33%
9	VM-Ware	vmcl	28.38%	42.63%	18.03%

4.2 Analyzing GVirtuS/CUDA in a Virtual Cluster Environment

In the high performance cloud computing scenario, our evaluation is based on a benchmark implementing a parallel matrix multiplication algorithm. The algorithm decomposes the domain by distributed the first matrix by rows and the second matrix by columns over all processes. Subsequently, each process performs a local matrix multiplication. The final results are stored in RAM memory. For remote machine message passing we have used the Massage Passing Interface (MPI) distribution MPICH2 version 1.2.1p1. Each process uses the CUDA library to perform the local matrix multiplication.

Figure 3 shows the results obtained by running the MPI-based algorithm on a virtual cluster with 8 virtual computing nodes (VCNs). Each VCN is an instance of a virtual machine based on a lightweight Ubuntu 10.04 Linux installation. Each presented result is the mean of 100 runs. We have evaluated two scenarios:

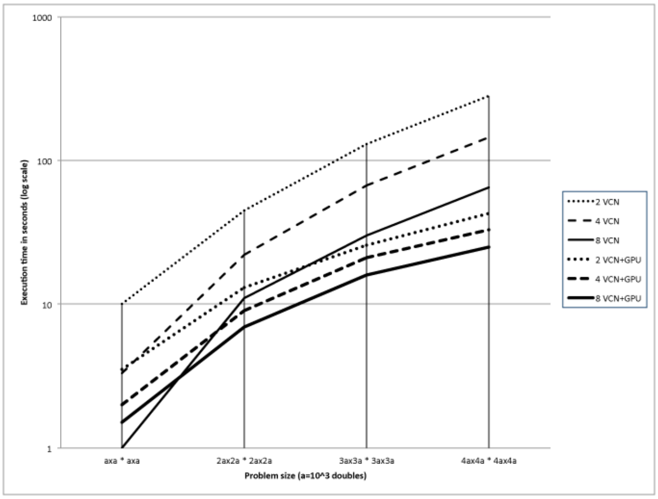


Fig. 3. Matrix multiplication algorithm performance on GVirtuS for 2, 4 and 8 nodes

VCN and VCN-GPU. The VCN scenario runs only on virtual CPUs. The VCN-GPU scenario runs on virtual machine instances with GPUs enabled thanks to GVirtuS/CUDA [5]. The problem size ranges from matrix sizes of $10^3 x 10^3$ doubles to $4 * 10^3 x 4 * 10^3$ doubles. The number of VCNs ranged from 2 to 8 increasing by a factor of 2.

The results show that virtualized GPUs provide substantial performance benefits in all cases but one. The only case when the VCN scenario outperforms the VCN-GPU is for the smallest matrix size ($10^3 x 10^3$) and 8 VCN. This can be attributed to the fact that the small amount of computations does not compensate the cost of transferring the data from virtual machine memory to GPU memory and back. However, in all other cases for the same number of VCNs, the GPU virtualization solution substantially outperforms the non-GPU solution. Even more, for matrix sizes larger than $3 * 10^3 x 3 * 10^3$ doubles, the time for using VCN with GPUs is considerably lower than using 8 VCN without GPU. Increasing the number of VCNs with GPUs further improves the performance.

5 Related Works

GViM (GPU-accelerated Virtual Machines) [13] and vCUDA [14] are Xen-based system that provide GPU virtualization solutions [15]. GVirtuS, gVirtuS, GViM, and vCUDA use a similar split-driver approach for GPU virtualization. The main difference lies in the use of the hypervisor and in the communication mechanisms. Both GViM and vCUDA employ the Xen hypervisor and execute CUDA drivers in the Domain 0. GViM uses the XenStore component to provide communication between the two components of the CUDA split driver. The GViM performance is mainly affected by the XenStore behavior and in particular by memory copy operations, as shown in [16]. vCUDA transforms the CUDA calls in the frontend into XML-RPCs, which are served by the backend. This is a commonly used approach, but which requires a considerable overhead as shown in [14].

Our solution differs from the latter two solutions because GVirtuS is completely independent of the hypervisor. We have proved it by implementing GVirtuS in a VMWare [16] setup and in a KVM setup. In our operational GVirtuS setup, we have chosen the KVM open source hypervisor because it is a Linux loadable kernel module included in mainline Linux [5], [17]. The fact that KVM is not a modified Linux kernel (as in the case of Xen) ensures future compatibility between the hypervisor and the CUDA drivers. GVirtuS can run on Xen using the VGA pass-through feature: a thin Linux virtual machine runs on Xen DomainU with the CUDA device directly attached to it. This virtual machine instance is provided by nVIDIA/CUDA drivers and acts as a backend for other Xen virtual machines executing CUDA code via GVirtuS. In this case a XenStore based communicator component is used in order to increase performance. Finally, GVirtuS relies on a high performance communication channel between the virtual and the physical machines. The communicator interface is open and new communicators could be developed and plugged in. We also emphasize that neither GViM nor vCUDA use a TCP based communicator for the deployment on asymmetric computing clusters.

gVirtuS, the predecessor of GVirtuS, has been successfully used in a framework to enable applications executing within virtual machines to transparently share one or more GPUs by Ravi et al [8]. They extend gVirtuS to include efficient GPU sharing, and propose solutions to the conceptual problem of GPU kernel consolidation. Finally, they develop an algorithm to efficiently map a set of kernels on GPUs and find that even when contention is high, their consolidation algorithm is effective for improving the throughput with a low overhead.

Our General Virtualization Service has been used as GPU virtualization solution for virtual machines hosted by the OpenStack cloud management software [18] providing a cloud-based access model to clusters containing heterogeneous architectures and accelerators [19].

6 Conclusions and Future Directions

In this work we presented GVirtuS, a general-purpose virtualization service, which is a direct evolution and generalization of the previously developed gVirtuS. GVirtuS brightest feature is its modularity, which allows for the development of split-drivers in a straightforward fashion by requiring to only implement stub routines in the frontend and service routines in the backend.

GVirtuS has been implemented so far to offer transparent virtualization support for CUDA library, openCL and openGL on top of nVIDIA/CUDA virtualization. We show that GVirtuS can be efficiently used for leveraging acceleration devices in scientific clouds. Based on GVirtuS architecture we are working on two ambitious projects: a virtual parallel high performance file system and a MPI channel for low-latency communication among virtual machines running in a virtual HPC cluster environment [20].

References

1. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program gpus for general-purpose uses. SIGOPS Oper. Syst. Rev. 40, 325–335 (2006)
2. Wang, L., Tao, J., von Laszewski, G., Marten, H.: Multicores in cloud computing: Research challenges for applications. JCP 5(6), 958–964 (2010)
3. Vecchiola, C., Pandey, S., Buyya, R.: High-performance cloud computing: A view of scientific applications. In: Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks, ISPAN 2009, pp. 4–16. IEEE Computer Society, Washington, DC (2009)
4. Giunta, G., Montella, R., Agrillo, G., Coviello, G.: A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 379–391. Springer, Heidelberg (2010)
5. KVM website, http://www.linux-kvm.org
6. VMWare website, http://www.vmware.com
7. VMCI website, http://pubs.vmware.com/vmci-sdk
8. Ravi, V.T., Becchi, M., Agrawal, G., Chakradhar, S.: Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC 2011, pp. 217–228. ACM, New York (2011)
9. OpenNebula website, http://www.opennebula.org
10. Eucalyptus website, http://open.eucalyptus.com
11. Amazon Web Services, http://aws.amazon.com
12. Sotomayor, B., Keahey, K., Foster, I.: Combining batch execution and leasing using virtual machines. In: Proceedings of the 17th HPDC
13. Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., Ranganathan, P.: Gvim: Gpu-accelerated virtual machines. In: Proceedings of the 3rd ACM Workshop on System-level Virtualization for HPC, HPCVirt 2009, pp. 17–24. ACM, New York (2009)
14. Shi, L., Chen, H., Sun, J.: vcuda: Gpu accelerated high performance computing in virtual machines. In: Proceedings of the 2009 IEEE IPDPS (2009)
15. XEN website, http://www.xen.org
16. Wang, J., Wright, K.L., Gopalan, K.: Xenloop: a transparent high performance inter-vm network loopback. In: Proceedings of the 17th HPDC
17. vmChannel web, http://www.linux-kvm.org/page/VMchannel_Requirements
18. OpenStack website, http://www.openstack.org
19. Crago, S.P., et al.: Heterogeneous cloud computing. In: CLUSTER, pp. 378–385 (2011)
20. Giunta, G., Montella, R., Coviello, G., Laccetti, G., Isaila, F., Garcia Blas, J.: A gpu accelerated high performance cloud computing infrastructure for grid computing based virtual environmental laboratory | intechopen

A Simulated Annealing Algorithm for GPU Clusters

Maciej Zbierski

Institute of Computer Science, Warsaw University of Technology,
ul. Nowowiejska 15/19, Warsaw 00-665, Poland
m.zbierski@ii.pw.edu.pl

Abstract. Simulated Annealing (SA) is a powerful global optimization technique that is frequently used for solving many practical problems from various scientific and technical fields. In this article we present a novel approach to parallelization of SA and propose an algorithm optimized for execution in GPU clusters. Our technique exploits the basic characteristics of such environments by using hierarchical problem decomposition. The proposed algorithm performs especially well for complex problems with large number of variables. We compare our approach with traditional parallel Simulated Annealing, both in terms of speed and result accuracy.

Keywords: Simulated Annealing, GPU clusters, global optimization, distributed systems, GPGPU, Monte Carlo methods.

1 Introduction

Parallelization has recently become a popular approach to solving complex computational problems, mainly as a result of high availability and relatively low cost of computers with many processing cores. This process can usually be performed using either powerful parallel computers or distributed systems, i.e. groups of interconnected machines. Over the years, many parallel computation models, such as farm processing [17], have emerged. Nowadays, GPU (graphics processing unit) computing is also considered a powerful technique for obtaining inexpensive, high performance parallelism [16]. Grouping a considerable number of machines designed to perform computations using their graphic processing units in so called GPU clusters might enable solving even more complex problems.

Unfortunately, regardless of the effort put into expansion of parallel processing, there still exist problems that cannot be solved using conventional numerical methods. Perhaps the most well known example is the travelling salesman problem. One of the techniques that is often used when dealing with computationally hard, global optimization tasks is Simulated Annealing (SA) [9]. The main concept of this approach is the probabilistic acceptance of suboptimal solutions in order to prevent from locking in local minima. Simulated Annealing is popular in many scientific and technical areas, including physics, chemistry and biology; several specific applications have been described in [8].

While the Simulated Annealing is an inherently sequential algorithm, much work has been put into its parallel adaptations. Different approaches are discussed for instance in [3], [1], [6], [8] and [4]. Verhoeven and Aarts [19] provide a further overview of parallel implementations of local search techniques. An extensive survey on various parallel Simulated Annealing algorithms and their applications in global optimization can also be found in [15].

To the best of our knowledge, the GPU-based Simulated Annealing has not yet achieved much attention. The majority of existing work in this field lacks generality and is limited to very specialized applications. These include for instance FPGA placement [2] and integrated circuit floorplanning [7]. Moreover, all these techniques consider single GPU environments only.

In this paper we present a novel approach to parallelization of Simulated Annealing and demonstrate that it can be successfully used in GPU clusters. We contribute with the design of a two-level algorithm, called TLSA, that can be used in such environments. We also compare its efficiency with previously developed methods, both in terms of speed and result accuracy.

The remainder of this article is constructed as follows: section 2 presents the architecture of the distributed GPU-based system we will use. In section 3 we introduce the proposed two-level algorithm and explain how it differs from the traditional approach. The description and results of the experiments we have performed are presented in section 4. Finally, section 5 contains conclusion and discussion on the work presented in the article.

2 Basic Concepts of the Processing Platform

Let us consider a simple distributed system, i.e. a set of computers connected by a common network. Regardless of the choice of physical representation, which lies beyond the scope of this article, there exist a wide selection of decomposition and mapping techniques that can be applied. We propose the usage of the master-slave model [5,20], as it is the most straightforward and versatile for our purpose. In this method selected nodes perform the role of managers. These machines, sometimes also referred to as masters, are responsible for generating the work, allocating it to the workers and gathering their results. Throughout this article we assume that the set of managers is limited to one machine only.

It is worth noting that the model itself does not limit in any way the type of computations, nor the way they are performed by the workers. While in the most typical case the workers would run sequential algorithms, we focus on the case, where each worker performs parallel computations using its graphics processing unit. Such architecture can be treated as a two-level distributed system and is sometimes called a GPU cluster. From now on we will refer to any tasks performed by workers, including the GPU-based computations, as the work performed on the lower level. By the upper level tasks will on the other hand denote all actions taken by the master, including the communication with workers.

In order to maximize the efficiency of the whole system, we assume that each machine performs its computations independently from the others. While this implies the impossibility of cooperation between GPU threads of different machines, it also creates the opportunity of using the asynchronous communication on the upper level. This on the other hand allows for more flexibility with regard to the system architecture and its components. For instance, a slow machine or network interconnection does not have to drastically limit the throughput of the whole system. Furthermore, as the system can be entirely heterogeneous, we can imagine a situation where some machines, instead of carrying out the computations on graphic processing units, might rather use their CPUs.

While the distributed system presented above can be used as a basis for different types of algorithms, we will focus specifically on Simulated Annealing. In order to produce accurate results, it requires fast and high quality random number generators. This usually does not pose any problems for CPU computations, as RNGs are already implemented in many well-known libraries, such as *boost* (`http://www.boost.org/`) or *gsl* (`http://www.gnu.org/software/gsl/`) to name only a few. However, random number generation is still missing from most GPU programming environments. In our previous study we have performed a comparison of GPU-based implementations of RNGs and have converged on the Warp Standard generator [18], which has demonstrated both high quality output and good speed [20].

The most recent version of CUDA GPU programming environment supplies XORWOW generator [12], which has not been included in our previous research. While both Warp Standard and XORWOW provide acceptable amount of randomness, in a quick test we have performed on GeForce GTX 260, XORWOW has turned out to be around 1.5 times slower. As a result, the Warp Standard will be used as the random number generator in the remainder of this article. Note that we have previously demonstrated in [20] that when applying this generator in GPU clusters, the probability of obtaining overlapping random number series throughout the cluster is negligible. This is true as long as those generators are initialized with a high quality CPU RNG, such as Mersenne Twister [10].

3 The Proposed Algorithm

Simulated Annealing is a single point stochastic search technique proposed by Kirkpatrick et al. [9]. In each iteration of the algorithm the current optimal solution undergoes slight modification. If this new solution is better, it is accepted; otherwise it is accepted probabilistically. The probability of acceptance is varied accordingly to a selected cooling scheme, which can be chosen differently for specific needs. We will describe the approach presented in [14], which has previously provided good result.

Let us consider the global optimization problem, that can be stated as $min\ f(\mathbf{x})$, $\mathbf{L} \leq \mathbf{x} \leq \mathbf{U}$, where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{L}$ and $\mathbf{U}$ are the lower and upper boundary vectors respectively. A random point from $\mathbb{R}^n$ located between $\mathbf{L}$ and $\mathbf{U}$ is chosen as the initial solution $\mathbf{x}^{(0)}$. In each iteration a random value $d \in \{1..n\}$ is selected. This number represents the dimension of the solution that will be modified. The d-th dimension of the solution in the step $(i + 1)$ is computed as follows:

$$\mathbf{x}_d^{(i+1)} = \begin{cases} \mathbf{x}_d^{(i)} + u \cdot (\mathbf{U}_d - \mathbf{x}_d^{(i)}) & \text{if } u < 0.5 \\ \mathbf{x}_d^{(i)} - u \cdot (\mathbf{x}_d^{(i)} - \mathbf{L}_d) & \text{if } u \geq 0.5 \end{cases},$$

where u is a uniformly distributed random variable on $[0, 1]$. All the other dimensions of the solution are copied from the previous step.

The probability of accepting the suboptimal solution is controlled according to a geometric cooling scheme, and can be computed using the following formula:

$$PA^{(i+1)} = exp\left(-\frac{f(\mathbf{x}^{(i+1)}) - f(\mathbf{x}^{(i)})}{f(\mathbf{x}^{(i)}) \cdot T^{(i+1)}}\right).$$

The T factor represents the temperature of the system, initially set to $T^{(0)} = 1$. In each iteration its value is recalculated using the algorithm:

$$T^{(i+1)} = \begin{cases} \frac{T^{(i)}}{1+0.1 \cdot T^{(i)}} & \text{if a worse solution has been chosen in } i\text{-th iteration} \\ T^{(i)} & \text{otherwise} \end{cases},$$

Additionally, if at some point the temperature drops below 0.01, it is re-assigned the initial value. This process, called the re-annealing, prevents the algorithm from blocking in function's local minima.

In the remainder of this chapter the introduced sequential algorithm will be adapted to parallel computations according to the two-level concept described earlier. The functions and cooperation of the lower and upper levels are presented in the following sections.

3.1 The Lower Level

As mentioned before, the lower level comprises all the computations performed by each node independently. Namely, as we are proposing the usage of the GPU, it will be responsible for supervising the work of all GPU threads, gathering their results and presenting them in a form acceptable by the upper level.

We have chosen to base on the MHCS (Modified Highly Coupled Synchronous) technique, as it has demonstrated very good results in previous studies and has shown to outperform many other parallel approaches [15]. However, as it was originally designed to be used on the CPU, we have introduced several modifications in order to adapt it to the GPU-based architecture.

The initial solution $\mathbf{x}^{(0)}$ is obtained from the upper level and is identical for all the threads. Each GPU thread selects a value d, which represents the dimension of the solution it will modify. We propose that this value is connected with the thread identifier ID in a way that $d \equiv ID \bmod n$, where n is the number of dimensions of the minimized function. The assigned d value therefore remains unchanged for the whole run of the algorithm. This allows to limit the total amount of memory required to store the intermediate solution. As a result, each thread needs only to keep the value of the dimension it will modify, while the remainder of the solution might be stored in the shared or global memory.

Each thread performs m steps of sequential Simulated Annealing, altering only the d-th dimension of the solution. After this is done, for each dimension $i \in \{1..n\}$ we choose the most optimal solution among those generated by the threads altering this dimension. We will denote this value as $\mathbf{X}_i$. Note that $\mathbf{X} = \{\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_n\}$ is a set of solutions that differ from $\mathbf{x}^{(0)}$ on one dimension only. All these results are then copied to the CPU memory and sorted accordingly to $f(\mathbf{X}_i)$.

The obtained results are then combined into one output solution using the following algorithm. Let $\widehat{\mathbf{X}}$ be the output solution, initially $\widehat{\mathbf{X}} = \mathbf{x}^{(0)}$. In each step the next element is acquired from our sorted list and the value of the modified dimension is used to replace the value on corresponding dimension in $\widehat{\mathbf{X}}$. If the new solution is better, the substitution is accepted, otherwise it is rejected and a rollback is performed on $\widehat{\mathbf{X}}$. When all the elements of the list have been considered, the $\widehat{\mathbf{X}}$ is treated as a temporary result that will be used as the initial solution in the next run of the algorithm. After performing the desired number of runs, the temporary result is transmitted to the master.

3.2 The Upper Level

Let F_b represent the current best solution, corresponding to the vector of parameters $\mathbf{X}_b$, so that $F_b = f(\mathbf{X}_b)$. Each time the worker reports its solution F_w and $\mathbf{X}_w$, the temporary optimal result is calculated as the smaller of the two, that is $FL = min(F_b, F_w)$. The vector of function parameters $\mathbf{XL}$ is defined as:

$$\mathbf{XL} = \begin{cases} \mathbf{X}_b & \text{if } FL = F_b \\ \mathbf{X}_w & \text{if } FL = F_w \end{cases}$$

Similarly, we denote FG and $\mathbf{XG}$ as those corresponding to the less optimal of the two, i.e. $FG = F_w$ and $\mathbf{XG} = \mathbf{X}_w$ if $FL = F_b$, and vice versa.

The next step is to combine these two solutions. For each dimension i, if $\mathbf{X}_{bi} \neq \mathbf{X}_{wi}$ we perform a substitution $\mathbf{XL}_i = \mathbf{XG}_i$. If $f(\mathbf{XL}) < FL$, we accept the new move and recalculate FL, otherwise we retrieve the old value of $\mathbf{XL}_i$. After all the dimensions have been considered, we store the FL and $\mathbf{XL}$ values and treat them as the current best solution. Each time the worker requests a new set of input data, it receives the current best result, which it then uses as the initial solution on the lower level of the algorithm.

It is worth noting that the quality of final result depends on the sequence of the abovementioned substitutions. For instance, iterating through the dimensions in the reverse order $(n, n-1, ...1)$ might sometimes produce better results than in the natural one $(1, 2, ..., n)$, as different points are being checked in each case. If $\mathbf{XL}$ and $\mathbf{XG}$ are entirely different, it gives the total of $n!$ combinations to check, which might prove impossible for big n values. However, as workers provide their results relatively rarely, several different step sequences might be examined to determine the best result.

The algorithm proposed for the upper level diverges greatly from the traditional SA approach, mainly because it does not introduce the possibility of accepting the non-optimal solutions. We have rejected this idea mostly because the upper level algorithm is executed relatively low number of times in each simulation. As a result, accepting worse solutions might lead to decrease in the rate of convergence of the whole approach. However, for very long simulations, applying SA-like pattern might be considered.

4 Experiments and Results

In order to verify our approach, we have performed a series of comparison tests between TLSA and a widely used MHCS implementation of parallel SA. The latter was run on a 16-core Sun Fire X4600M2 machine, equipped with 8 AMD Opteron 8218 CPUs and 32GB of DDR2-667 memory. The execution environment of the former consisted of 16 PCs, each fitted with one Intel Core2Duo E8400 CPU and one NVIDIA GeForce 130 GT graphic card.

The goal of our experiment was to compare the execution times and result accuracy of both algorithms. For this purpose we have chosen several well-known, demanding test functions. The more detailed description of these functions, including their formulae and domain, can be found for instance in [11]. All the tests have been performed for $n \in \{10, 100\}$ dimensions. We have set a time limit for all our experiments to four minutes in case of 10 dimensional functions and four hours for 100 dimensional ones.

Our test has been divided into three parts. Firstly, we have converged on the total number of iterations to be performed in the GPU cluster and carried out five simulations for each test function. Then we have computed mean values of these results and iterated the MHCS until it achieved at least the same result precision. Finally, we have calculated the mean runtime achieved in the GPU cluster and executed the MHCS algorithm for at least the same amount of time.

The mean execution times required to obtain the desired result precision for five runs of both algorithms are presented in table 1. For 10 dimensional problems a variety of results can be observed. In most cases the MHCS algorithm either converged very quickly on the desired solution or did not converge at all before reaching the time limit. After increasing the number of dimensions, the results have become more uniform and TLSA has proven to be faster than MHCS for all the test functions. The obtained mean runtimes of TLSA are from 1.5 to roughly 70 times shorter. Additionally, none of the MHCS simulations for the Michalewicz function has converged on the result before reaching the time limit. The standard deviation values of the MHCS algorithm are higher as a result of the selected test strategy.

Table 2 contains the observed result errors in the situation when both algorithms were run for roughly the same amount of time. In the ideal case, these results should be as close to zero as possible.

In case of 10 dimensional functions, both techniques have generated relatively accurate solutions. While for some problems the MHCS algorithm has provided more precise results (De Jong and Rosenbrock's Valley functions), in most cases our algorithm has been more accurate. The greatest difference between the results of both methods can be observed for the Schwefel function.

Table 1. Execution times required to achieve the same result accuracy for a 16-node GPU cluster running TLSA and a 16-core Sun Fire machine running MHCS

	Function	**GPU cluster**		**Sun Fire**	
		mean [s]	**STDV** [s]	**mean** [s]	**STDV** [s]
$n = 10$	Ackley	26.36	0.31	> 4min	0
	De Jong	14.65	0.46	1.11	0.55
	Griewank	26.38	0.29	> 4min	0
	Michalewicz	26.29	0.35	32.94	10.39
	Rastrigin	26.28	0.22	> 4min	0
	Rosenbrock	34.92	0.29	1.32	0.92
	Schwefel	26.42	0.25	46.47	18.01
$n = 100$	Ackley	109.82	0.1	704.56	302.57
	De Jong	55.15	0.33	113.98	5.61
	Griewank	110.64	0.33	7886.96	7522.44
	Michalewicz	113.41	0.27	> 4h	0
	Rastrigin	109.22	0.12	174.93	19.17
	Rosenbrock	165.14	0.27	1284.03	1479.14
	Schwefel	111.81	0.26	1027.48	70.75

Table 2. Mean result errors for a 16-node GPU cluster running TLSA and a 16-core Sun Fire machine running MHCS, both executed for the same amount of time

	Function	**GPU cluster**		**Sun Fire**	
		mean	**STDV**	**mean**	**STDV**
$n = 10$	Ackley	5×10^{-4}	2×10^{-4}	3×10^{-3}	8×10^{-4}
	De Jong	1×10^{-5}	1×10^{-5}	8.6×10^{-6}	4×10^{-6}
	Griewank	8×10^{-6}	1×10^{-5}	0.05	0.02
	Rastrigin	2×10^{-5}	3×10^{-5}	7×10^{-4}	3×10^{-4}
	Rosenbrock	0.38	0.44	0.11	0.12
	Schwefel	6.73	3.4	48.55	23.65
$n = 100$	Ackley	0.006	0.0009	2.57	0.14
	De Jong	0.0003	0.0002	0.03	0.01
	Griewank	0.009	0.008	3.66	0.79
	Rastrigin	0.003	0.002	0.005	0.0006
	Rosenbrock	107.65	26.12	91.92	56.92
	Schwefel	0.33	0.05	3054.05	290.62

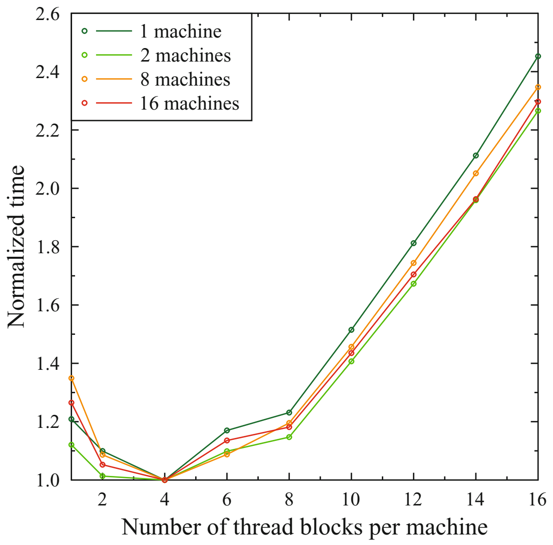


Fig. 1. Execution times for various cluster sizes. The results were normalized to the shortest time in each run. Each block contained 64 threads.

For all 100 dimensional problems, apart from the Rosenbrock's Valley, TLSA has provided more accurate results than MHCS. For this function, however, the minimum estimated by both techniques lies relatively far from the optimal one. Furthermore, high standard deviation values indicate significant divergence of solutions obtained in the consecutive runs of the algorithm. Therefore, before assessing the results for this function, the number of iterations should be increased until the solutions are more accurate.

The tests with different number of machines in the GPU cluster have revealed almost ideal scalability of the upper level, which confirms the expected theoretical results. Multiple runs of the algorithm on the 16 node cluster achieved a mean speedup of around 15.3 compared to a single-GPU environment.

A more interesting aspect is the scalability with regard to the number of GPU threads in each machine. The obtained results, presented in figure 1, demonstrate that the number of threads can be increased only up to some point without further decrease in the achieved speedup. This optimal number of threads is tightly connected to the number of stream processors of the underlying graphic processing unit. An important observation is that the general trend of the increase in runtimes presented in figure 1 is similar regardless of the number of machines in the cluster.

The performed experiments show that in case of computationally hard problems our approach will with high probability perform better than the standard MHCS technique. This however depends greatly on the complexity of the optimized function and the number of its variables. The main strength of the proposed two-level architecture is the greater number of threads that can be run in parallel. For instance, in our test roughly several thousand threads were used in the whole GPU cluster, compared to sixteen in case of the SUN16 machine. This transfers into the ability to alter many more dimensions in parallel, rather than consider them sequentially, as it would be the case with the traditional approach. However, when the number of function dimensions is close to the number of threads of the one-level SA algorithm, this advantage might disappear.

Our technique might also be preferable with less complex functions when the ultimate result precision is desirable. As the experiment has shown, our algorithm can sometimes provide more accurate solutions than MHCS even for functions with small number of variables. However, it cannot be determined a priori whether it would be the case for the given problem. We have presented sample functions (De Jong and Rosenbrock's Valley) where the traditional approach had been more effective.

5 Conclusion

We have presented a new approach to Simulated Annealing by developing a two-level parallel algorithm designed to perform especially well in GPU clusters. The main advantage of TLSA over traditional methods is that it is able to perform significantly more independent SA steps in parallel. This is vital especially for problems with a considerable number of variables. We have demonstrated that in some cases our algorithm can achieve the same result precision as other parallel SA methods even several dozen times faster.

The accuracy of the obtained results is however strictly related to the characteristics of the optimized functions. As we have mentioned earlier, the performance of our approach might be reduced for some less complicated problems, mostly because of the low number of variables. Furthermore, we need to remember that some problems are not suited for the GPU architectures, for instance due to requirements on the memory size, inconsistent memory access patterns or frequent data exchange between CPU and GPU memory. In such cases it is therefore still better to use the CPU-based techniques, such as MHCS or other parallel Simulated Annealing algorithm.

We believe that GPU clusters will become even more popular in the future, which will result in increased interest in efficient and practical algorithms designed for execution in such environments. Considering other approaches to Simulated Annealing rather than the MHCS we have used, might lead to obtaining even better results. The future work might also include the adaptation of other Monte Carlo methods for GPU clusters.

References

1. Boissin, N., Lutton, J.-L.: A parallel simulated annealing algorithm. Parallel Computing 19(8), 859–872 (1993)
2. Choong, A., Beidas, R., Zhu, J.: Parallelizing Simulated Annealing-Based Placement Using GPGPU. In: Proceedings of the 2010 International Conference on Field Programmable Logic and Applications, pp. 31–34 (2010)

3. Debudaj-Grabysz, A., Czech, Z.: Theoretical and Practical Issues of Parallel Simulated Annealing. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 189–198. Springer, Heidelberg (2008)
4. Frost, R., Heineman, P.: Simulated annealing: A heuristic for parallel stochastic optimization. Tech. rep., San Diego Supercomputer Center (1997)
5. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison Wesley, Harlow (2003)
6. Greening, D.R.: Parallel simulated annealing techniques. Physica 42, 293–306 (1990)
7. Han, Y., Roy, S., Chakraborty, K.: Optimizing simulated annealing on GPU: A case study with IC floorplanning. In: Proceedings of the 12th International Symposium on Quality Electronic Design, pp. 1–7 (2011)
8. Ingber, L.: Simulated annealing: Practice versus theory. Mathematical Computer Modelling 18(11), 29–57 (1993)
9. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
10. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. 8(1), 3–30 (1998)
11. Molga, M., Smutnicki, C.: Test functions for optimization needs (2005), http://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf
12. NVIDIA: CUDA C programming guide (2010), http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf
13. NVIDIA: CUDA CURAND library (2010), http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CURAND_Library.pdf
14. Özdamar, L., Demirhan, M.: Experiments with new stochastic global optimization search techniques. Comput. Oper. Res. 27, 841–865 (2000)
15. Onbaşoğlu, E., Özdamar, L.: Parallel simulated annealing algorithms in global optimization. Journal of Global Optimization 19, 27–50 (2001)
16. Ryoo, S., Rodrigues, C., Stone, S., et al.: Program optimization carving for GPU computing. Journal of Parallel and Distributed Computing 68(10), 1389–1401 (2008)
17. Sosnowski, J., Tymoczko, A., Gawkowski, P.: An Approach to Distributed Fault Injection Experiments. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 361–370. Springer, Heidelberg (2008)
18. Thomas, D.B., Luk, W.: GPU optimised uniform random number generation, http://www.doc.ic.ac.uk/~dt10/research/gpu_rng/gpu_warp_rng.pdf
19. Verhoeven, M., Aarts, E.: Parallel local search. Journal of Heuristics 1, 43–65 (1995)
20. Zbierski, M.: Analysis of a CUDA-based distributed system in the context of selected Monte Carlo methods. Master's thesis, Warsaw University of Technology (2011)

Author Index