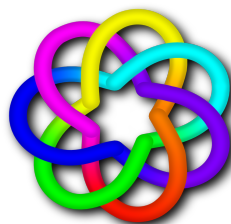


CONFERENCES IN RESEARCH AND PRACTICE IN
INFORMATION TECHNOLOGY

VOLUME 127

PARALLEL AND DISTRIBUTED COMPUTING 2012

AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS, VOLUME 34, NUMBER 6



PARALLEL AND DISTRIBUTED COMPUTING 2012

Proceedings of the Tenth Australasian Symposium on
Parallel and Distributed Computing
(AusPDC 2012), Melbourne, Australia,
31 January – 3 February 2012

Jinjun Chen and Rajiv Ranjan, Eds.

Volume 127 in the Conferences in Research and Practice in Information Technology Series.
Published by the Australian Computer Society Inc.



Published in association with the ACM Digital Library.

Parallel and Distributed Computing 2012. Proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), Melbourne, Australia, 31 January – 3 February 2012

Conferences in Research and Practice in Information Technology, Volume 127.

Copyright ©2012, Australian Computer Society. Reproduction for academic, not-for-profit purposes permitted provided the copyright text at the foot of the first page of each paper is included.

Editors:

Jinjun Chen

Faculty of Engineering and Information Technology
University of Technology, Sydney
Broadway, NSW 2007
Australia
Email: Jinjun.Chen@uts.edu.au

Rajiv Ranjan

Information Engineering Lab
CSIRO ICT Centre
Acton, ACT, 2601
Australia
Email: rajiv.ranjan@csiro.au

Series Editors:

Vladimir Estivill-Castro, Griffith University, Queensland
Simeon J. Simoff, University of Western Sydney, NSW
Email: crpit@scm.uws.edu.au

Publisher: Australian Computer Society Inc.
PO Box Q534, QVB Post Office
Sydney 1230
New South Wales
Australia.

Conferences in Research and Practice in Information Technology, Volume 127.
ISSN 1445-1336.
ISBN 978-1-921770-08-1.

Printed, January 2012 by University of Western Sydney, on-line proceedings
Printed, January 2012 by RMIT, electronic media
Document engineering by CRPIT

The *Conferences in Research and Practice in Information Technology* series disseminates the results of peer-reviewed research in all areas of Information Technology. Further details can be found at <http://crpit.com/>.

Table of Contents

Proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), Melbourne, Australia, 31 January – 3 February 2012

Preface	vii
Programme Committee	viii
Organising Committee	x
Welcome from the Organising Committee	xi
CORE - Computing Research & Education	xiii
ACSW Conferences and the Australian Computer Science Communications	xiv
ACSW and AusPDC 2012 Sponsors	xvi

Contributed Papers

On Supernode Transformations and Multithreading for the Longest Common Subsequence Problem . . . <i>Johann Steinbrecher and Weijia Shang</i>	3
Hard-Sphere Collision Simulations with Multiple GPUs, PCIe Extension Buses and GPU-GPU Communications	13
<i>Ken Hawick and Daniel Playne</i>	
A Comparative Study of Parallel Algorithms for the Girth Problem	23
<i>Michael J. Dinneen, Masoud Khosravani and Kuai Wei</i>	
The Use of Fast Approximate Graph Coloring to Enhance Exact Parallel Algorithm Performance ...	31
<i>John Eblen, Gary Rogers, Charles Phillips and Michael A. Langston</i>	
Managing Large Numbers of Business Processes with Cloud Workflow Systems	33
<i>Xiao Liu, Yun Yang, Dahai Cao, Dong Yuan and Jinjun Chen</i>	
Use of Multiple GPUs on Shared Memory Multiprocessors for Ultrasound Propagation Simulations ..	43
<i>Jiri Jaros, Bradley E. Treeby and Alistair P. Rendell.</i>	
Scaling Up Transit Priority Modeling using High Throughput Computing	53
<i>Mahmoud Mesbah, Jefferson Tan, Majid Sarvi and Fatemeh Karimirad</i>	
Author Index	63

Preface

These proceedings contain the papers presented at the 10th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), held between 31 January – 3 February 2012 in Melbourne, Australia in conjunction with the Australasian Computer Science Week (ASCW). Over the years, previously known as Australasian Symposium on Grid Computing and e-Research (AusGrid), and starting 2010, it is being referred to as AusPDC, has become the flagship symposium for Grid, Cloud, Cluster, and Distributed Computing research in Australia. Submissions were received, mostly from Australia, but also from New Zealand, United States, Asia and Europe. The full version of each paper was carefully reviewed by at least two referees, and evaluated according to its originality, correctness, readability and relevance. A total of 7 papers were accepted. The accepted papers cover topics from Cloud resource management, grid inter-operation, multi-processing systems, trusted brokering, performance models, operating systems, and networking protocols.

We are very thankful to the Program Committee members, and external reviewers for their outstanding and timely work, which was invaluable for taking the quality of this year's program to such a high level. We also wish to acknowledge the efforts of the authors who submitted their papers and without whom this conference would have not been possible. Due to the very competitive selection process, several strong papers could not be included in the program. We sincerely hope that prospective authors will continue to view the AusPDC symposium series as the premiere venue in the field for disseminating their work and results. We would also like to thank the ASCW Organising Committee, those that submitted papers and those that attended the conference their work and contributions have made the symposium a great success.

Rajiv Ranjan
CSIRO ICT Centre

Jinjun Chen
University of Technology, Sydney
AusPDC 2012 Programme Chairs
January 2012

Programme Committee

Chairs

Jinjun Chen, University of Technology, Sydney, Australia
Rajiv Ranjan, CSIRO ICT Centre, Australia

Members

Jemal Abawajy, Deakin of University, Australia
David Abramson, Monash University, Australia
David Bannon, Victoria Partnership for Advanced Computing, Australia
Peter Bertok, RMIT, Australia
Rajkumar Buyya, University of Melbourne, Australia
Phoebe Chen, University of Melbourne, Australia Geoffrey Fox, Indiana University, USA Andrzej Goscinski, Deakin University, Australia
Kenneth Hawick, Massey University, New Zealand
John Hine, Victoria University of Wellington, New Zealand
Michael Hobbs, Deakin University, Australia
Jiankun Hu, RMIT University, Australia
Zhiyi Huang, Otago University, New Zealand
Nick Jones, University of Auckland, New Zealand
Wayne Kelly, Queensland University of Technology, Australia
Kevin Lee, Murdoch University, Australia
Young Choon Lee, University of Sydney, Australia
Laurent Lefevre, University of Lyon, France
Andrew Lewis, Griffith University, Australia
Jiuyong Li, University of South Australia, Australia
Weifa Liang, Australian National University, Australia
Teo Yong Meng, National University of Singapore, Singapore
Lin Padgham, RMIT University, Australia
Judy Qiu, Indiana University, USA
Paul Roe, Queensland University of Technology, Australia
Justin Rough, Deakin University, Australia
Hong Shen, University of Adelaide, Australia
Jun Shen, University of Wollongong, Australia
Michael Sheng, University of Adelaide, Australia
Gaurav Singh, CSIRO Mathematical and Information Sciences, Australia
Peter Strazdins, Australian National University, Australia
Srikumar Venugopal, University of New South Wales, Australia
Yan Wang, Macquarie University, Australia
Andrew Wendelborn, University of Adelaide, Australia
Yang Xiang, Deakin University, Australia
Jingling Xue, University of New South Wales, Australia
Jun Yan, University of Wollongong, Australia
Yun Yang, Swinburne University of Technology, Australia
Yanchun Zhang, Victoria University, Australia
Rui Zhang, University of Melbourne, Australia
Albert Zomaya, University of Sydney, Australia

Steering Committee

Prof. David Abramson, Monash University, Australia
Prof. Rajkumar Buyya, University of Melbourne, Australia
Dr. Jinjun Chen (Vice Chair), University of Technology Sydney, Australia
Dr. Paul Coddington, University of Adelaide, Australia

Prof. Andrzej Goscinski (Chair), Deakin University, Australia
Prof. Kenneth Hawick, Massey University, New Zealand
Prof. John Hine, Victoria University of Wellington, New Zealand
Dr. Rajiv Ranjan, CSIRO ICT Centre, Australia
Dr. Wayne Kelly, Queensland University of Technology, Australia
Prof. Paul Roe, Queensland University of Technology, Australia
Dr. Andrew Wendelborn, University of Adelaide, Australia

Organising Committee

Members

Dr. Daryl D'Souza
Assoc. Prof. James Harland (Chair)
Dr. Falk Scholer
Dr. John Thangarajah
Assoc. Prof. James Thom
Dr. Jenny Zhang

Welcome from the Organising Committee

On behalf of the Australasian Computer Science Week 2012 (ACSW2012) Organising Committee, we welcome you to this year's event hosted by RMIT University. RMIT is a global university of technology and design and Australia's largest tertiary institution. The University enjoys an international reputation for excellence in practical education and outcome-oriented research. RMIT is a leader in technology, design, global business, communication, global communities, health solutions and urban sustainable futures. RMIT was ranked in the top 100 universities in the world for engineering and technology in the 2011 QS World University Rankings. RMIT has three campuses in Melbourne, Australia, and two in Vietnam, and offers programs through partners in Singapore, Hong Kong, mainland China, Malaysia, India and Europe. The University's student population of 74,000 includes 30,000 international students, of whom more than 17,000 are taught offshore (almost 6,000 at RMIT Vietnam).

We welcome delegates from a number of different countries, including Australia, New Zealand, Austria, Canada, China, the Czech Republic, Denmark, Germany, Hong Kong, Japan, Luxembourg, Malaysia, South Korea, Sweden, the United Arab Emirates, the United Kingdom, and the United States of America.

We hope you will enjoy ACSW2012, and also to experience the city of Melbourne.,

Melbourne is amongst the world's most liveable cities for its safe and multicultural environment as well as well-developed infrastructure. Melbourne's skyline is a mix of cutting-edge designs and heritage architecture. The city is famous for its restaurants, fashion boutiques, café-filled laneways, bars, art galleries, and parks.

RMIT's city campus, the venue of ACSW2012, is right in the heart of the Melbourne CBD, and can be easily accessed by train or tram.

ACSW2012 consists of the following conferences:

- Australasian Computer Science Conference (ACSC) (Chaired by Mark Reynolds and Bruce Thomas)
- Australasian Database Conference (ADC) (Chaired by Rui Zhang and Yanchun Zhang)
- Australasian Computer Education Conference (ACE) (Chaired by Michael de Raadt and Angela Carbone)
- Australasian Information Security Conference (AISC) (Chaired by Josef Pieprzyk and Clark Thorburn)
- Australasian User Interface Conference (AUIC) (Chaired by Haifeng Shen and Ross Smith)
- Computing: Australasian Theory Symposium (CATS) (Chaired by Julián Mestre)
- Australasian Symposium on Parallel and Distributed Computing (AusPDC) (Chaired by Jinjun Chen and Rajiv Ranjan)
- Australasian Workshop on Health Informatics and Knowledge Management (HIKM) (Chaired by Keryn Butler-Henderson and Kathleen Gray)
- Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Aditya Ghose and Flavio Ferrarotti)
- Australasian Computing Doctoral Consortium (ACDC) (Chaired by Falk Scholer and Helen Ashman)

ACSW is an event that requires a great deal of co-operation from a number of people, and this year has been no exception. We thank all who have worked for the success of ACSE 2012, including the Organising Committee, the Conference Chairs and Programme Committees, the RMIT School of Computer Science and IT, the RMIT Events Office, our sponsors, our keynote and invited speakers, and the attendees.

Special thanks go to Alex Potanin, the CORE Conference Coordinator, for his extensive expertise, knowledge and encouragement, and to organisers of previous ACSW meetings, who have provided us with a great deal of information and advice. We hope that ACSW2012 will be as successful as its predecessors.

Assoc. Prof. James Harland

School of Computer Science and Information Technology, RMIT University

ACSW2012 Chair

January, 2012

CORE - Computing Research & Education

CORE welcomes all delegates to ACSW2012 in Melbourne. CORE, the peak body representing academic computer science in Australia and New Zealand, is responsible for the annual ACSW series of meetings, which are a unique opportunity for our community to network and to discuss research and topics of mutual interest. The original component conferences - ACSC, ADC, and CATS, which formed the basis of ACSW in the mid 1990s - now share this week with seven other events - ACE, AISC, AUIC, AusPDC, HIKM, ACDC, and APCCM, which build on the diversity of the Australasian computing community.

In 2012, we have again chosen to feature a small number of keynote speakers from across the discipline: Michael Kölling (ACE), Timo Ropinski (ACSC), and Manish Parashar (AusPDC). I thank them for their contributions to ACSW2012. I also thank invited speakers in some of the individual conferences, and the two CORE award winners Warwish Irwin (CORE Teaching Award) and Daniel Frampton (CORE PhD Award). The efforts of the conference chairs and their program committees have led to strong programs in all the conferences, thanks very much for all your efforts. Thanks are particularly due to James Harland and his colleagues for organising what promises to be a strong event.

The past year has been very turbulent for our disciplines. We tried to convince the ARC that refereed conference publications should be included in ERA2012 in evaluations – it was partially successful. We ran a small pilot which demonstrated that conference citations behave similarly to but not exactly the same as journal citations - so the latter can not be scaled to estimate the former. So they moved all of Field of Research Code 08 “Information and Computing Sciences” to peer review for ERA2012. The effect of this will be that most Universities will be evaluated at least at the two digit 08 level, as refereed conference papers count towards the 50 threshold for evaluation. CORE’s position is to return 08 to a citation measured discipline as soon as possible.

ACSW will feature a joint CORE and ACDICT discussion on Research Challenges in ICT, which I hope will identify a national research agenda as well as priority application areas to which our disciplines can contribute, and perhaps opportunity to find international multi-disciplinary successes which could work in our region.

Beyond research issues, in 2012 CORE will also need to focus on education issues, including in Schools. The likelihood that the future will have less computers is small, yet where are the numbers of students we need?

CORE’s existence is due to the support of the member departments in Australia and New Zealand, and I thank them for their ongoing contributions, in commitment and in financial support. Finally, I am grateful to all those who gave their time to CORE in 2011; in particular, I thank Alex Potanin, Alan Fekete, Aditya Ghose, Justin Zobel, and those of you who contribute to the discussions on the CORE mailing lists. There are three main lists: csprofs, cshods and members. You are all eligible for the members list if your department is a member. Please do sign up via <http://lists.core.edu.au/mailman/listinfo> - we try to keep the volume low but relevance high in the mailing lists.

Tom Gedeon

President, CORE
January, 2012

ACSW Conferences and the Australian Computer Science Communications

The Australasian Computer Science Week of conferences has been running in some form continuously since 1978. This makes it one of the longest running conferences in computer science. The proceedings of the week have been published as the *Australian Computer Science Communications* since 1979 (with the 1978 proceedings often referred to as *Volume 0*). Thus the sequence number of the Australasian Computer Science Conference is always one greater than the volume of the Communications. Below is a list of the conferences, their locations and hosts.

2013. Volume 35. Host and Venue - University of South Australia, Adelaide, SA.

2012. Volume 34. Host and Venue - RMIT University, Melbourne, VIC.

2011. Volume 33. Host and Venue - Curtin University of Technology, Perth, WA.

2010. Volume 32. Host and Venue - Queensland University of Technology, Brisbane, QLD.

2009. Volume 31. Host and Venue - Victoria University, Wellington, New Zealand.

2008. Volume 30. Host and Venue - University of Wollongong, NSW.

2007. Volume 29. Host and Venue - University of Ballarat, VIC. First running of HDKM.

2006. Volume 28. Host and Venue - University of Tasmania, TAS.

2005. Volume 27. Host - University of Newcastle, NSW. APBC held separately from 2005.

2004. Volume 26. Host and Venue - University of Otago, Dunedin, New Zealand. First running of APCCM.

2003. Volume 25. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Adelaide Convention Centre, Adelaide, SA. First running of APBC. Incorporation of ACE. ACSAC held separately from 2003.

2002. Volume 24. Host and Venue - Monash University, Melbourne, VIC.

2001. Volume 23. Hosts - Bond University and Griffith University (Gold Coast). Venue - Gold Coast, QLD.

2000. Volume 22. Hosts - Australian National University and University of Canberra. Venue - ANU, Canberra, ACT. First running of AUIC.

1999. Volume 21. Host and Venue - University of Auckland, New Zealand.

1998. Volume 20. Hosts - University of Western Australia, Murdoch University, Edith Cowan University and Curtin University. Venue - Perth, WA.

1997. Volume 19. Hosts - Macquarie University and University of Technology, Sydney. Venue - Sydney, NSW. ADC held with DASFAA (rather than ACSW) in 1997.

1996. Volume 18. Host - University of Melbourne and RMIT University. Venue - Melbourne, Australia. CATS joins ACSW.

1995. Volume 17. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Glenelg, SA.

1994. Volume 16. Host and Venue - University of Canterbury, Christchurch, New Zealand. CATS run for the first time separately in Sydney.

1993. Volume 15. Hosts - Griffith University and Queensland University of Technology. Venue - Nathan, QLD.

1992. Volume 14. Host and Venue - University of Tasmania, TAS. (ADC held separately at La Trobe University).

1991. Volume 13. Host and Venue - University of New South Wales, NSW.

1990. Volume 12. Host and Venue - Monash University, Melbourne, VIC. Joined by Database and Information Systems Conference which in 1992 became ADC (which stayed with ACSW) and ACIS (which now operates independently).

1989. Volume 11. Host and Venue - University of Wollongong, NSW.

1988. Volume 10. Host and Venue - University of Queensland, QLD.

1987. Volume 9. Host and Venue - Deakin University, VIC.

1986. Volume 8. Host and Venue - Australian National University, Canberra, ACT.

1985. Volume 7. Hosts - University of Melbourne and Monash University. Venue - Melbourne, VIC.

1984. Volume 6. Host and Venue - University of Adelaide, SA.

1983. Volume 5. Host and Venue - University of Sydney, NSW.

1982. Volume 4. Host and Venue - University of Western Australia, WA.

1981. Volume 3. Host and Venue - University of Queensland, QLD.

1980. Volume 2. Host and Venue - Australian National University, Canberra, ACT.

1979. Volume 1. Host and Venue - University of Tasmania, TAS.

1978. Volume 0. Host and Venue - University of New South Wales, NSW.

Conference Acronyms

ACDC	Australasian Computing Doctoral Consortium
ACE	Australasian Computer Education Conference
ACSC	Australasian Computer Science Conference
ACSW	Australasian Computer Science Week
ADC	Australasian Database Conference
AISC	Australasian Information Security Conference
AUIC	Australasian User Interface Conference
APCCM	Asia-Pacific Conference on Conceptual Modelling
AusPDC	Australasian Symposium on Parallel and Distributed Computing (replaces AusGrid)
CATS	Computing: Australasian Theory Symposium
HIKM	Australasian Workshop on Health Informatics and Knowledge Management

Note that various name changes have occurred, which have been indicated in the Conference Acronyms sections in respective CRPIT volumes.

ACSW and AusPDC 2012 Sponsors

We wish to thank the following sponsors for their contribution towards this conference.



CORE - Computing Research and Education,
www.core.edu.au



RMIT University,
www.rmit.edu.au/



AUSTRALIAN
COMPUTER
SOCIETY

Australian Computer Society,
www.acs.org.au

CONTRIBUTED PAPERS

On Supernode Transformations And Multithreading For The Longest Common Subsequence Problem

Johann Steinbrecher and Weijia Shang
Santa Clara University
500 El Camino Real
Santa Clara, CA 95053

j1steinbrecher@scu.edu, wshang@scu.edu

Abstract

The longest common subsequence (LCS) problem is an important algorithm in computer science with many applications such as DNA matching (bio-engineering) and file comparison (UNIX diff). While there has been a lot of research for finding an efficient solution to this problem, the research emphasis has shifted with the advent of multi-core architectures towards multithreaded implementations. This paper applies supernode transformations to partition the dynamic programming solution of the LCS problem into multiple threads. Then, we enhance this method by presenting a transformation matrix that skews the loop nest such that loop carried dependencies of the inner loop are eliminated in each supernode. We find that this technique performs well on microarchitectures supporting out-of-order execution while in-order execution machines do not benefit from it. Furthermore, we present a variation of the supernode transformations and multithreading strategy which groups entire rows of the index set to form a supernode. The inter thread synchronization is performed by an array of mutexes. We find that this scheme reduces the amount of thread management overhead and improves the data locality. A formula for the total execution time of each method is presented. The techniques are benchmarked on a 12 core and a four core machine. At the 12 core machine the traditional supernode transformation speeds up the original loop nest 16.7 times. We enhance this technique to score a 42.6 speedup and apply our new method scoring a 59.5 speedup. We experience the phenomenon of super-linear speedup as the the performance gain is larger than the number of processing cores. Concepts presented and discussed in this paper on the LCS problem are generally applicable to regular dependence algorithms.

I. Introduction

The LCS algorithm solves the problem of finding the LCS shared by two strings. It is well-known for its application at DNA matching in bio-engineering. Here, two DNA's are represented as strings of characters 'ACGT'. Finding out how similar two DNA's are is important when researching the properties of new DNA's or for criminal evidence. Furthermore, the LCS problem has its application in file comparison utilities such as the UNIX *diff* program, data compression, editing error correction and syntactic pattern recognition as well as the evidence of plagiarism ([7], [15]). The dynamic programming solution to the LCS problem is asymptotically bounded by $O(N^2)$ for N character inputs. Considering the human DNA which is organized in 23 chromosomes each holding $50 \cdot 10^6$ to $220 \cdot 10^6$ base pairs, where each base pair is represented by a character, this generates a significant amount of computation. This paper discusses multithreaded implementation of the dynamic programming solution of the LCS problem to utilize the resources of multiple instruction multiple data machines (MIMD) as efficient as possible. We apply previous work on thread partitioning and supernode transformations to this problem, enhance these methods and present a variation of the supernode transformations and multithreading strategy. Concepts, ideas and observations made and presented in this paper on the LCS problem are generally applicable to regular dependence algorithms.

With the advent of multi-core systems in high performance computing as well as in embedded systems it is important to understand how to take advantage of the available resources. In high performance computing, single

Copyright 2012, Australian Computer Society, Inc. This paper appeared at the 10th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 127. J. Chen and R. Ranjan, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

instruction multiple data (*SIMD*) architectures such as most general purpose graphics processing units (*GPGPU*) and MIMD architectures such as many integrated cores (*MIC*) provide up to hundreds of cores. In embedded computing the trend clearly is towards parallel architectures, either by providing multiple cores inside the host processor or by building a heterogeneous system where the host processor is supported by a multi-core graphics processing unit. While supercomputers usually have physically distributed-memory, *GPGPU*'s and *MIC*'s have a centralized main memory shared by all cores. Cache memories may be distributed to different cores. For optimizing the processor utilization of each core, *hyperthreading* is used. Hyperthreading is a hardware technique supporting thread execution switch if the functional units are stalled by the currently running thread. These technological facts and trends motivate to understand how the LCS problem can be implemented to saturate the resources of a multi-core system.

This paper performs supernode transformations on the LCS problem in order to utilize the resources of a parallel system as efficient as possible. Hodzic and Shang presented in [10] supernode transformations for loop nests with regular dependencies. They group iterations in the index set to form supernodes. Those supernodes are then scheduled according to a linear schedule, which is respecting the data dependence, to the processing units. While this paper exploits parallelism amongst supernodes, it does not consider how to form supernodes in order to exploit the resources of each processing core. Therefore, we enhance this method by skewing the loop. We present a transformation matrix that eliminates the loop carried dependencies of the inner loop. This enables processing cores (hardware) or compilers (software) to reorder the instructions for hiding memory access latencies. Furthermore, we present a variation of the supernode transformations and multithreading strategy which groups entire rows of the index set to form a supernode. The inter thread synchronization is performed by an array of mutexes. We find that this scheme reduces the amount of thread management overhead and improves the data locality. We apply this method then to the original and skewed loop nest. The total execution time of each method is presented as a function of the computation time and communication time assuming systems with infinite resources and limited resources. The techniques presented and discussed in this paper are benchmarked on two multi-core MIMD machines with four cores and 12 cores, respectively.

The rest of the paper is organized as follows. In section II related work and our contribution is discussed. Section III presents architecture, programming and algorithm models and the concepts of linear scheduling. Section IV gives an overview over the basic ideas and implementation

strategies. The total execution time at systems with limited resources is discussed in section V. In section VI our implementations are benchmarked on multi-core machines. Section VII concludes this paper.

II. Related work and our contribution

The LCS problem has been studied over the previous decades from different point of views. The dynamic programming solution was originally presented by Hirschberg in [9] performing the algorithm in an $O(N^2)$ complexity, where N equals the size of the input strings. Based on this sequential method there have been many efforts for optimizing the time complexity, the space complexity, implement strategies using special data structures and design algorithms for special input properties targeting various systems as summarized in [4]. Our paper investigates on how to execute the problem on multi-core systems. Mabrouk presents in [3] a survey on the parallel complexity of the LCS problem on various target machines referring to papers in the time range from 1990 to 2006. Furthermore, this paper contributes a greedy strategy to schedule iterations on available machines in a parallel system. A summary of systolic algorithms and an efficient hardware-implementable systolic algorithm for the LCS is presented in [12]. While most commonly two strings are analyzed for their LCS, paper [13] discusses an optimal implementation on how to find the LCS of multiple input sequences (MLCS). It presents an efficient parallel algorithm for the MLCS based on a dominant points method. A coarse grained multithreaded implementation of the LCS is discussed in [7]. Supernode transformations for optimal running time on loop nests with regular dependencies are discussed in [10]. The supernode shape, expressed by the relative side length is discussed in [11] for optimal running time and discussed in [8] for minimizing the communication volume.

Having discussed related work we summarize here our contributions. While previous work on parallel implementations of the LCS problem did not consider supernode transformations we incorporate the concepts from [8], [10] and [11]. Techniques presented in this paper are based on several important observations we made. First, we find that skewing the loop nest, which eliminates inner loop dependencies, speeds up the algorithm on out-of-order execution cores while it does not gain a speedup on in-order architectures. Therefore, we enhance Hodzic's method ([10]) by loop skewing. By benchmarking the technique on a 12 core MIMD machine with out-of-order pipelines we have found that the original supernode transformations speedup the LCS problem 16.7 times while our enhancement achieves a 42.6 times speedup over the original loop nest. As these performance gains are larger than the number of processing cores, we experience the

```

for (i=0; i < M; i++)
  for (j=0; j < N; j++) {
    if (i == 0 || j == 0)
      c[i][j] = 0;
    else if (x[i] == y[j])
      c[i][j] = c[i-1][j-1] + 1;
    else
      c[i][j] = max(c[i][j-1], c[i-1][j]);
  }
    
```

Fig. 1. The longest common subsequence algorithm [6].

phenomenon of super-linear speedup. This is mainly due to fortunate memory sharing among threads. The data that is loaded into cache memory due to one thread can be reused by other threads, which reduces cache misses. As the LCS problem is a very data intense algorithm, reducing cache misses results in a large performance gain. This observation motivated us to design and present a variation of the supernode transformations and multithreading strategy scoring a speedup of 59.5 on the 12 core machine. While paper [10] presents an expression for the total execution on systems with infinite resources we contribute a formula for the total execution time on systems with limited resources for Hodzic's method and our new technique. While we find that the methods presented in this paper score phenomenal speedups, how to incorporate the concepts and ideas of paper [7] will be conducted in future work.

III. Algorithm, architecture and programming models

In this section we present an overview on our algorithm model for the LCS problem, architectural model, programming model and linear scheduling.

A. Algorithm model

The LCS algorithm takes as inputs two sequences of characters: $x[1...M]$ and $y[1...N]$. The program then attempts to find the longest subsequence that is shared by these two strings, and returns the length of that subsequence. For example, for the input strings 'ABCBDAB' and 'BDCABA', the program will output four, because the longest common subsequence is 'BCBA', which has a length of four. The dynamic programming version of the LCS algorithm can be found in [6] and presented in figure 1.

The LCS problem is an algorithm with regular dependencies [17]. An algorithm with regular dependencies is one containing array references such that the dependencies remain constant from one iteration to the next. Such algorithms may be described by two parameters, the dependence matrix D and the iteration space J . For the

```

for (i=0; i < N; i++)
  c[i][0] = c[0][i] = 0;
for (i=0; i < N; i++)
  for (j=0; j < N; j++) {
    if (x[i] == y[j])
      c[i][j] = c[i-1][j-1] + 1;
    else
      c[i][j] = max(c[i][j-1], c[i-1][j]);
  }
    
```

Fig. 2. The peeled original loop nest.

LCS, the dependence matrix equals:

$$D = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

The iteration space is the set of all iteration indexing vectors and shown in figure 3. The process of establishing the dependence matrix and the iteration space is beyond the scope of this paper and can be found in [17]. Intuitively, each point in the iteration space corresponds to an iteration in the loop. An arrow between two points indicates a dependence between the two iterations. The dependence matrix D corresponds to the loop carried write after read (WAR) in figure 1. For the correctness these dependencies must be respected.

This paper has an emphasis on the overall performance of an implementation of the LCS problem. Before discussing optimal thread partitioning and loop transformations we analyze the loop body. The code as shown in figure 1 consists of three if-statements. If-statements generate branch instructions which reduce the size of basic blocks in the compiled code. This has a negative impact on the performance as branches usually require the hardware pipelines to stall. Hardware techniques such as speculation and branch prediction reduce this effect but can't eliminate it. Therefore, we try to reduce the number of branches in the loop body. By analyzing the data dependencies we find the second and third if-statement to hold loop carried dependencies while the first if-statement does not. Therefore, we move the first if-statement out to a separate loop. The resulting code is shown in figure 2 and used throughout this paper. Furthermore, all codes and examples used in this paper assume a squared index set.

B. Linear schedule and wavefronts

A linear schedule $f(\Pi)$ is a linear or affine function that maps multiple dimensional iteration vectors in the iteration space onto a set of execution times represented by integers. Multiple independent iterations are assigned the same execution time for parallel processing. How to identify linear schedules that respect the data dependencies and minimize the total execution time is discussed in [17]. For

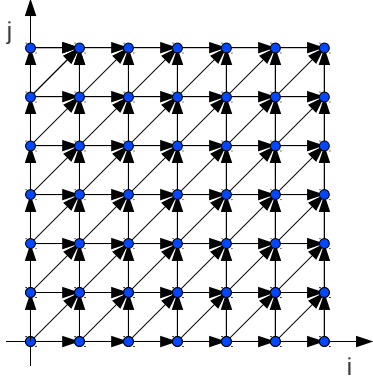


Fig. 3. Iteration space for the LCS algorithm.

the dependence structure of the LCS problem in figure 3, one *feasible* linear schedule that respects dependencies is

$$f((i \ j)^t) = (1 \ 1)(i \ j)^t = i + j$$

The vector $(1 \ 1)$ specifying the linear schedule is called *linear schedule vector* and is denoted as Π . The iteration space is partitioned by the linear schedule into a series of hyperplanes $(1 \ 1)(i \ j)^t = \text{constant}$. These hyperplanes are called *wavefronts* that are perpendicular to Π (see [17]). Such wavefronts may be generated using CLoog [2], a loop transformation and code generation tool based on the polyhedral model. One observation is that all iterations in the same wavefront are independent.

C. Architecture and programming model

For the purpose of this paper, a computer system is modeled by the parameters p and t where p is the number of cores in the system and t is the number of hyper threads each core accommodates. We assume multiple instruction multiple data architectures (MIMD) with shared memory. MIMD architectures can be programmed amongst others with Open MP [5], Intel®Cilk™Plus or Posix Threads. In this paper the Posix Thread programming model [1] is used. The main memory is shared, while each core may have its own cache. This stands in many aspects in contrast to the CUDA programming model and architectural model, where a single instruction multiple data (SIMD) architecture is programmed. One major limitation of CUDA is that only threads within the same thread block can communicate with each other efficiently. One of the advantages of CUDA is an extremely low thread creation overhead and cache access latency. The programming model in this paper is especially applicable to future MIC systems ([14] and [16]). MIC's can be classified as MIMD systems with SIMD and superscalar properties. The system consists of multiple cores running independent instruction streams (MIMD). Each core accommodates several functional units (superscalar) as well as a SIMD unit for fast floating point calculations.

IV. Basic ideas and concepts

The basic ideas and concepts for speeding up the dynamic programming algorithm of the LCS problem are illustrated in this section. Subsection IV-A shows how to partition the algorithm according to Hodzic's method. In subsection IV-B, we enhance Hodzic's method by skewing the loop. In section IV-C we propose a variation of the supernode transformations and multithreading strategy. Subsection IV-D discusses the proposed techniques.

A. Traditional supernode transformation and thread partitioning

Hodzic presents in paper [10] a supernode transformation for algorithms with regular dependencies. It discusses the supernode transformation and optimal grain size to minimize the total running time. Furthermore, in [8], [10] and [11] the supernode shape is discussed for minimum running time and minimum communication volume. The total running time is the sum of the computation time and the communication cost. Hodzic groups a number of computation nodes to form a supernode and assigns each supernode to a processor as an unit for execution. Applying this concept to our programming model we spawn and terminate a thread for each supernode. Hodzic's paper uses a loop nest in Example 2.1 with the dependence matrix D :

$$D = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

As we have shown in section III-A this is the dependence matrix of the LCS problem. Therefore, we can apply the method as presented by Hodzic where the index set is partitioned into rectangular supernodes as shown in figure 4. Supernodes are formed by rectangles with the side length w and h . The wavefronts for linear schedule $\Pi = (1 \ 1)$ are indicated by transparent diagonal lines. In this technique, each thread executes one supernode. Assuming a system with infinite resources, all superiterations that share the same wavefront can be executed simultaneously on different cores. Therefore, a closed form expression for the total execution time is presented in section 3 of paper [10]:

$$T = P \cdot (T_{comp} + T_{comm})$$

Where P , the number of computation phases, equals the linear schedule length or the number of wavefronts, T_{comp} equals the computation time per supernode and T_{comm} equals the communication time. Starting with this technique we will exploit further optimizations. While the actual computation time per supernode is strongly dependent on the input strings, this equation assumes that each computation phase takes the same time to execute. Furthermore, the aspect of smaller supernodes at the borders of the index set is not considered. We keep these

assumptions in all equations presented in this paper.

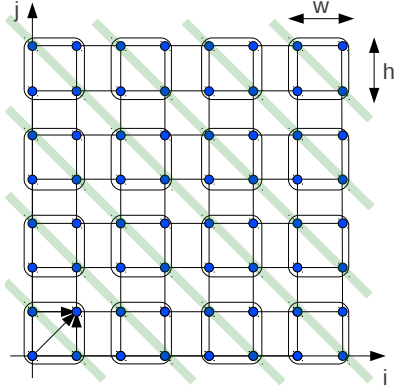


Fig. 4. The loop nest according to Hodzic’s supernode transformations (paper [10]).

One observation from this method is that while this linear schedule exploits parallelism among superiterations, the iterations within a supernode may hold dependencies. On out-of-order microarchitectures such as most recent x86 and ARM CPU’s, this may not perform best as instructions can not be reordered for hiding memory access latencies because of dependence. This observation is the basis for subsection IV-B. Also, this method requires in our programming model to spawn and terminate a thread for each supernode. We will present a multithreading strategy that requires a reduced amount of thread management overhead, with an improved data access pattern for reducing cache misses, scoring the same amount of parallelism in subsection IV-C.

B. Eliminating inner loop carried dependencies

Mabrouk presents in section 2 of [3] how to transform the index set of the LCS in order to eliminate loop carried dependencies of the inner loop. Using the loop transformation and code generation tool ClooG, we experimented with this idea. The LCS is a very data intense algorithm. Each computation node does at least three loads ($x[i], y[j], c[]$) and one store. Therefore, the bottleneck of the algorithm are the memory access latencies. For minimizing stalls in the pipelines due to memory access latencies, two approaches are possible: hiding the latencies by loading several data in a pipelined fashion and reducing memory access latencies by exploring data locality for minimizing cache misses. In this section we investigate on hiding the memory access latencies by reordering the instructions. When skewing the loop nest as presented in [3] and shown in figure 5, there are no loop carried dependencies along the inner loop. This enables the software, i.e., the compiler or the hardware to reorder the instructions. One advantage is that multiple data can be loaded in a pipelined fashion to hide their latencies.

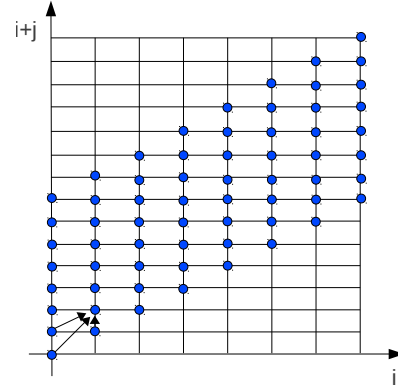


Fig. 5. The skewed index set.

Figure 5 shows the skewed loop nest. This transformation can be formally described by the transformation matrix TR :

$$TR = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

The new dependence matrix is indicated by arrows in figure 5 and can be calculated by

$$D1 = TR \cdot D = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

To support this observation we do a preliminary experiment. In figure 6 and figure 7 we show the results of executing the original and skewed loop nest on two out-of-order microarchitectures (x86-64) and one in-order microarchitecture (x86-64): an Intel®Atom™D510 (Atom microarchitecture; in-order execution; 1.66GHz), an Intel®Core™2 U7300 (Core microarchitecture; out-of-order execution; 1.30GHz) and an Intel®Core™i5-2500 (Sandy Bridge microarchitecture; out-of-order execution; 3.3GHz). Using GCC 4.6.1, we compiled the original loop nest and the skewed loop nest with an input size of 20000 characters per string performing no compiler optimizations (figure 6) and performing compiler optimizations (figure 7). Overall the skewed loop nest performs better on out-of-order execution microarchitectures while the in-order machine didn’t show any speedup. Noticeable is also that compiler optimization performed well on the in-order execution machine while it increases the execution time on the out-of-order architectures.

Based on this observation we enhance the strategy of Hodzic. Applying Hodzic’s supernode transformations on the skewed loop nest keeps the communication time constant while the computation time decreases due to the fact discovered in the experiment. This strategy is visualized in figure 8. Here the inner loop which iterates along the horizontal axis doesn’t have loop carried dependencies. Skewing the loop nest transforms also the dependence vectors as indicated by arrows in figure 8. Assuming a

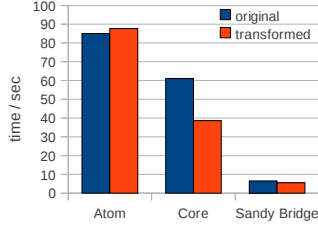


Fig. 6. The sequential implementation of the LCS problem performing no compiler optimizations (-O0).

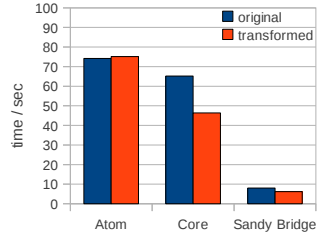


Fig. 7. The sequential implementation of the LCS problem performing compiler optimizations (-O3).

speed-up due to loop skewing of s , the total execution time results in:

$$T = P \cdot \left(\frac{T_{comp}}{s} + T_{comm} \right)$$

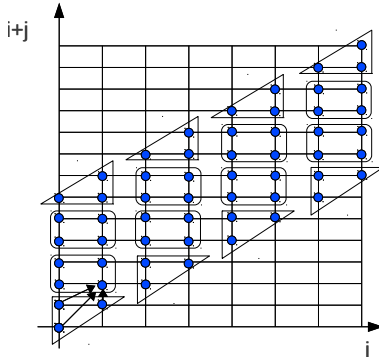


Fig. 8. Enhancing Hodzic's method by loop skewing.

C. Variation of the supernode transformation and multithreading strategy

The original implementation according to Hodzic has the disadvantage of grouping iterations with loop carried dependencies along the inner loop. We eliminated this by the method presented in section IV-B. One remaining downside of this technique is the high amount of thread creation overhead. For each supernode a thread is spawned and terminated. Generally, this can be reduced by partitioning the loop nest into as many supernodes as cores available in the system. This results in a minimum amount of thread management overhead but may not score a high

degree of parallelism. Furthermore, the original supernode transformation does not optimize cache sharing among threads. The method presented in this subsection evolves from considering these aspects.

This method partitions the index set in vertical and horizontal zones of size w and h . While previous methods scheduled each supernode in terms of a thread to a processor, this method executes one horizontal zone per thread. Each group of $w \times h$ index nodes is dedicated to a mutex. Therefore, the mutexes form a $\lceil N/w \rceil \times \lceil N/h \rceil$ matrix M where N is the dimension of the index set. This idea is illustrated in figure 9.

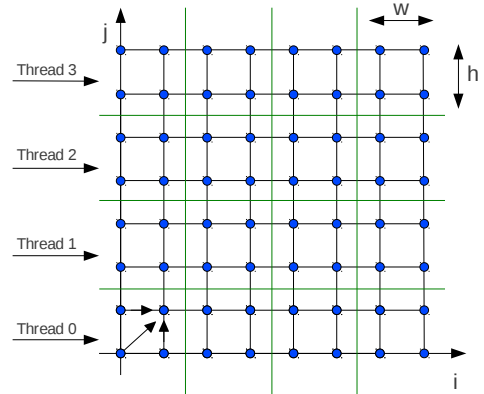


Fig. 9. Variation of the supernode transformation and multithreading strategy

Here, the index set is divided in four horizontal zones $0, \dots, 3$ and four vertical zones $0, \dots, 3$ of size $w = 2$ and $h = 2$. The inter thread synchronization is performed by locking and unlocking the mutexes, where $M[k][l]$ corresponds to the mutex for vertical zone k and horizontal zone l . Using the example from figure 9 we illustrate the method thoroughly. Initially, $Thread_0$ is created. Having finished with vertical zone 0 it creates $Thread_1$. Every thread is executing the same code. Therefore, in general $Thread_i$ creates $Thread_{i+1}$. To enforce the dependence structure of the loop nest, $Thread_k$ performing vertical zone k locks $M[k][l]$ when entering horizontal zone l and unlocks $M[k][l]$ when leaving horizontal zone l . Furthermore, after its creation $Thread_k$ executing vertical zone k , locks all mutexes $M[k+1][i]$ with $i = 0, \dots, 3$. Having finished horizontal zone l , $Thread_k$ unlocks mutex $M[k+1][l]$. To control the number of active threads $Thread_i$ waits until $Thread_{i-n}$ terminates, where n is the number of threads that should be active at any time. According to this scheme the total execution time can be expressed as:

$$T = \left(\left\lceil \frac{N}{w} \right\rceil - 1 \right) \cdot T_{zone} + \left\lceil \frac{N}{h} \right\rceil \cdot T_{zone},$$

where T_{zone} is the amount of time required for each thread to execute one $w \cdot h$ group. Depending on h , this scheme may require a minimum number of thread creation.

Furthermore, a large number of horizontal zones may make it possible that cache lines, loaded by $Thread_i$, can be reused by $Thread_{i+1}$.

The technique as presented in figure 9 did not eliminate the loop carried dependencies. Therefore, we present the same idea on the skewed loop nest in figure 10. By this method we expect to decrease T_{zone} , one parameter of the total execution time.

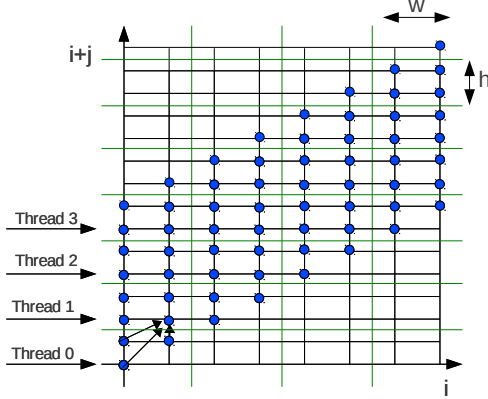


Fig. 10. Variation of the supernode transformation and multi-threading strategy on the skewed loop nest

D. Analysis and discussion

In this subsection we analyze the advantages and challenges of the three methods presented. Applying Hodzic's method to the LCS problem gave a general idea on how to partition the loop nest and how to schedule the supernodes to the processing units of a multi-core machine. We enhanced this method by loop skewing for reducing the computation time of each supernode (subsection IV-B). While this method improves the performance on architectures supporting out-of-order execution, in-order execution machines do not benefit from it. Furthermore, both methods generate an unnecessary high amount of thread control overhead as each supernode requires to spawn and terminate a thread. Our supernode transformation and threading strategy presented in subsection IV-C incorporates these observations. Furthermore, it provides an improved data access pattern. As $Thread_i$ executes the horizontal zone j of vertical zone i the data of this zone is loaded into the cache. Once $Thread_i$ has left zone j , $Thread_{i+1}$ can enter it and read some of the required data from the cache instead of paying the penalty for accessing the main memory. Therefore, the sizes of the horizontal and vertical zones w and h are a function of the cache size and cache associativity. This method can be applied to the skewed loop nest as well. The technique is suitable for in-order and out-of-order execution architectures. We especially see potential for this method at future MIC architectures where a significant amount of processors and

hardware threads are provided [14]. Such architectures usually consist of in-order execution architectures such as the Larrabee architecture [16].

V. The total execution time on systems with p processing cores

The total execution times used and presented in section IV assumed an infinite number of computation cores. This section enhances Hodzic's formula used in sections IV-A and IV-B and the formula for our new methods presented in section IV-C for the case of p processors.

A. Hodzic's total execution time

Hodzic presented the total execution time as the number of phases P , which he defines as the number of wavefronts, multiplied with the sum of the computation time T_{comp} and communication time T_{comm} .

$$T = P \cdot (T_{comp} + T_{comm})$$

While this remains generally true, the number of computation phases needs to be redefined. If all supernodes of a wavefront can be executed simultaneously, each wavefront requires one computation phase. If there are not enough resources in the system, a wavefront may require more than one computation phase. The total number of computation phases for this case can be estimated as:

$$P = \sum_{i=1}^W (\max(\frac{w_i}{p}, 1))$$

where p equals the number of processors, W equals the number of wavefronts and w_i equals the number of supernodes on wavefront i .

B. Total execution time of the new supernode transformation

In section IV-C we presented the total execution time as

$$T = (\lceil \frac{N}{w} \rceil - 1) \cdot T_{zone} + \lceil \frac{N}{h} \rceil \cdot T_{zone}$$

We construct the execution time for p processors by considering the example in figure 11. Assuming $p = 4$ we execute four threads $Thread_i$ with $i = 0, \dots, 3$. The execution order is illustrated in figure 12 by staggered horizontal lines.

Here the thread executing the very last horizontal zone is $Thread_1$, which requires $((\lceil \frac{N}{h} \rceil \bmod 4) - 1) \cdot T_{zone}$ due to dependencies to start and executes in $(\lceil \frac{\lceil \frac{N}{h} \rceil}{4} \rceil \cdot \lceil \frac{N}{w} \rceil) \cdot T_{zone}$. Therefore, the total execution time results for the general case in

$$T = ((\lceil \frac{N}{h} \rceil \bmod(p)) - 1 + \lceil \frac{\lceil \frac{N}{h} \rceil}{p} \rceil \cdot \lceil \frac{N}{w} \rceil) \cdot T_{zone}$$

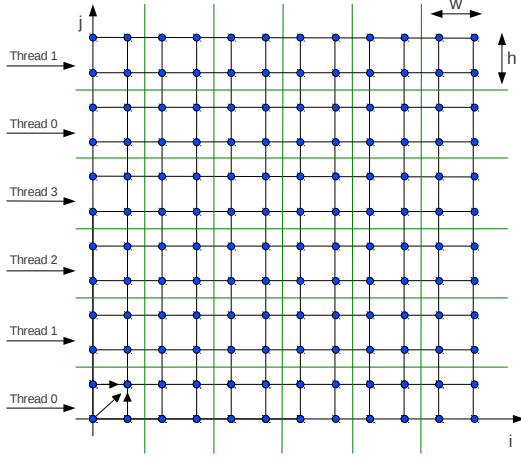


Fig. 11. Example for finding the execution time

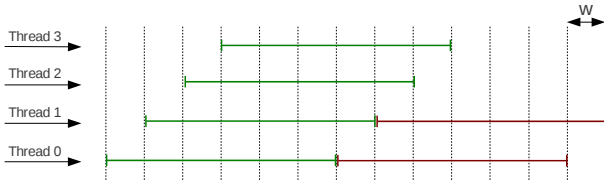


Fig. 12. Staggered execution of threads

VI. Experimental results

We benchmark our techniques on two out-of-order multi-core x86-64 machines, an Intel®Xeon®X5670 system with 12 cores and 24 threads ($p = 12, t = 2$ @ 2.93 GHz; a Nehalem architecture) and an Intel®Core™i5-2500 with four cores and four threads ($p = 4, t = 1$ @ 3.3 GHz; a Sandy Bridge architecture).

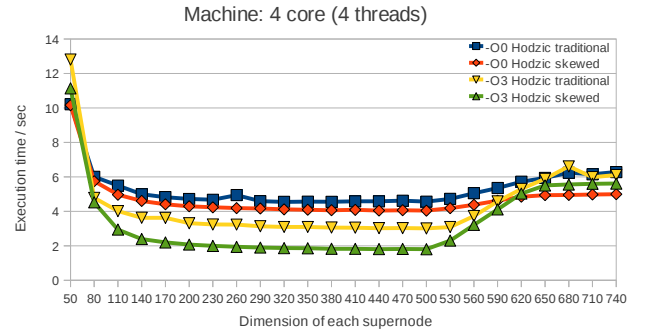
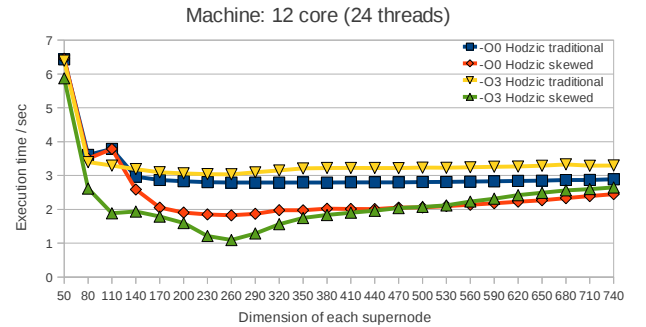
A. The benchmarks

We implemented each technique discussed and presented in this paper in ANSI C using Posix Threads. The benchmark performances are compared against each other and against the unmodified original loop nest according to the code in figure 2. The input size of all benchmarks is 43500 characters randomly generated stored on the heap and unmodified across all benchmarks. Four methods are tested: Hodzic's method (as shown in figure 4), Hodzic's method enhanced (as shown in figure 8) and our variation of the supernode transformation and threading strategy applied to the original loop nest as shown in figure 9 and applied to the skewed loop nest as shown in figure 10. Implementations based on Hodzic's method are benchmarked using squared supernodes as those are optimal for a squared index set according to [11]. For avoiding false sharing, mutexes which are stored globally are padded. Each benchmark is implemented with a maximum of four active threads on the four core machine ($p = 4, t = 1$) and

24 active threads on the 12 core machine with 24 hardware threads ($p = 12, t = 2$).

To measure the execution time the *gettimeofday* linux system call is used. This kernel routine takes use of the virtual dynamically-linked shared objects library (*VDSO*) of the linux kernel. Therefore, no switching from user space to kernel space is required, which minimizes the measurements overhead. The benchmarks were compiled using GCC 4.6.1 and tested on linux platforms with kernel version 2.6.35. The compilation process took advantage of all compiler optimizations by using flag `-O3`. If the unoptimized code (compiler flag `-O0`) performed better than the optimized code such as in figures 13 and 14 we show those results as well. Each value in figures 13, 14, 16, 17, 18 and 19 represents the arithmetic average of four repetitive runs. All values in tables 15, 16, 17, 18 and 19 represent the execution time in seconds.

B. The results

Fig. 13. Traditional supernode transformation and our enhancement at the i5core machine ($p = 4, t = 1$)Fig. 14. Traditional supernode transformation and our enhancement at the Xeon machine ($p = 12, t = 2$)

Figures 13 (four core machine) and 14 (12 core machine) show the benchmarks of Hodzic's method and our enhancement of it as described in subsection IV-A and subsection IV-B, respectively. The execution time is presented as a function of the supernode dimension, which is the side

length of the squared supernodes. In both cases the curves share the general shape with which the execution time decreases with the supernode dimension until it reaches an optimal grain size. The fastest execution time in each graph is scored by applying Hodzic's method to the compiler optimized skewed loop nest. The fastest execution time of our enhancement to Hodzic's method in relation to the fastest execution time of Hodzic's original supernode transformation scores a speedup of $3.01/1.80 = 1.67$ at the four core machine and $1.82/1.09 = 1.67$ at the 12 core machine. The optimal tile size for the compiler optimized transformed loop nest equals 500 at the four core machine and 260 at the 12 core machine. Interesting is also that execution times for tile sizes between 230 and 500 are consistently low at the compiler optimized transformed curve in figure 13 (4 core) while this benchmark in figure 14 (12 core) shows a more distinct optimal tile size.

Xeon	iScore
46.44	51.33

Fig. 15. Execution times of the original loop nest

Figure 15 shows the execution time of the original loop nest at each machine as presented in figure 2. Comparing the best performance of the traditional supernode transformation at the four core machine (iScore architecture) with the original loop nest, it shows a speedup of $51.33/3.01 = 17.1$ and a $51.33/1.80 = 28.5$ times speedup at the enhancement of Hodzic's method. The benchmarks for the 12 core machine (Xeon architecture) score a $46.66/2.79 = 16.7$ (traditional supernode transformation) and $46.44/1.09 = 42.6$ (enhanced supernode transformation) speedup. These results are better than expected. We experience the phenomenon of super-linear speedup, where the speedup is larger than the the increase of computation cores. This is mainly due to reduced data access times. As one thread is performing on specific data the whole cache line is loaded. Threads may be able to use data that is still present in the cache as it got loaded by other threads for reducing cache misses.

Figures 16 and 17 present the results of our variation of the supernode transformation and threading strategy on the four core machine applied to the original and the skewed loop nest. Figures 18 and 19 show the same techniques on the 12 core machine. We present the results in tabular form as we experiment with two parameters: The number of horizontal zones $\lceil N/w \rceil$ and the number of vertical zones $\lceil N/h \rceil$. The results show that the execution time decreases with an increasing number of vertical zones and increases after it reached an optimal number of vertical zones. Furthermore, the benchmarks tend to perform better with increasing number of horizontal zones. The fastest execution time for each number of vertical

		Number of vertical zones N/h							
		50	2000	4000	6000	8000	10000	12000	14000
Number of horizontal zones N/w	20	25.19	25.90	26.95	26.48	27.44	26.97	27.29	27.20
	50	7.34	8.39	8.41	8.09	9.98	9.48	8.35	8.63
	80	3.94	3.67	3.91	3.77	4.72	4.06	3.87	4.46
	110	3.61	3.15	3.39	3.24	4.03	3.48	3.21	3.66
	140	3.63	3.28	3.17	3.21	4.12	3.55	3.26	3.68
	170	3.66	3.18	3.21	3.22	4.09	3.50	3.22	3.72
	200	3.75	3.24	3.22	3.27	4.12	3.52	3.24	3.71
	230	3.69	3.43	3.19	3.23	4.12	3.55	3.24	3.70
	260	3.69	3.23	3.25	3.23	4.14	3.55	3.24	3.70
	290	3.68	3.23	3.23	3.24	4.16	3.55	3.25	3.71
	320	3.70	3.21	3.25	3.22	4.10	3.55	3.21	3.67
	350	3.75	3.30	3.29	3.29	4.24	3.58	3.27	3.74
	380	3.81	3.24	3.28	3.27	4.21	3.61	3.28	3.75
	410	3.81	3.27	3.25	3.31	4.19	3.60	3.32	3.76
	440	3.84	3.31	3.31	3.28	4.21	3.60	3.28	3.76
	470	3.83	3.26	3.26	3.28	4.20	3.61	3.28	3.76

Fig. 16. Our variation of the supernode transformation applied to the original loop nest at the iScore machine ($p = 4, t = 1$)

		Number of vertical zones N/h							
		50	4000	8000	12000	16000	20000	24000	28000
Number of horizontal zones N/w	20	8.56	7.70	8.18	9.97	12.18	14.41	16.26	17.27
	50	5.80	5.07	5.98	6.59	8.33	10.07	11.62	12.57
	80	3.33	2.54	2.89	3.11	3.97	4.80	5.59	6.08
	110	2.62	1.76	1.85	1.95	2.49	3.02	3.51	3.84
	140	2.66	1.83	1.89	1.93	2.49	3.05	3.51	3.83
	170	2.67	1.77	1.85	1.92	2.46	3.00	3.50	3.82
	200	2.67	1.80	1.85	1.91	2.45	3.00	3.50	3.84
	230	2.70	1.79	1.84	1.91	2.45	2.99	3.50	3.82
	260	2.71	1.79	1.98	1.90	2.44	2.99	3.50	3.82
	290	2.71	1.83	1.85	1.90	2.45	2.99	3.51	3.83
	320	2.74	1.84	1.91	1.90	2.44	2.98	3.50	3.82
	350	3.27	1.83	1.92	1.89	2.44	2.98	3.49	3.82
	380	2.81	1.96	1.85	1.92	2.44	2.99	3.50	3.83
	410	2.89	1.78	1.90	1.94	2.44	2.98	3.50	3.83
	440	2.98	1.77	1.90	1.92	2.45	3.03	3.51	3.84
	470	5.19	4.11	1.96	1.96	2.46	3.00	3.52	3.85

Fig. 17. Our variation of the supernode transformation applied to the skewed loop nest at the iScore machine ($p = 4, t = 1$)

		Number of vertical zones N/h									
		10	50	500	1000	2000	4000	6000	8000	10000	12000
Number of horizontal zones N/w	50	4.29	3.42	3.43	3.45	3.65	3.69	3.73	4.41	5.29	6.03
	250	3.46	3.03	2.97	2.98	2.98	3.04	3.12	3.30	3.46	4.02
	500	3.46	2.95	2.94	2.95	3.01	3.17	3.18	3.25	3.30	4.19
	750	2.11	1.07	1.55	1.75	1.68	1.01	1.48	1.95	2.54	2.89
	1000	1.96	1.06	1.54	1.66	1.41	1.00	1.46	1.92	2.43	2.81
	1500	1.88	1.09	0.89	0.92	0.89	1.07	1.44	1.90	2.52	2.97
	2000	1.99	1.14	0.88	0.88	0.85	0.99	1.45	1.91	2.40	2.88
	2500	2.07	1.17	0.89	0.86	0.85	0.99	1.44	2.00	2.34	2.79
	3000	2.12	1.19	0.87	0.86	0.84	1.13	1.46	2.01	2.46	2.78
	3500	2.12	1.17	0.89	0.85	0.83	0.93	1.38	1.83	2.27	2.80
	4000	2.06	1.17	0.85	0.85	0.82	0.87	1.71	1.85	2.20	2.61
	4500	2.07	1.22	0.85	0.81	0.79	1.00	1.20	1.65	2.11	2.32
	5000	1.99	1.18	0.80	0.79	0.84	0.81	1.13	1.48	1.97	2.27

Fig. 18. Our variation of the supernode transformation applied to the original loop nest at the Xeon machine ($p = 12, t = 2$)

zones is highlighted bold. While we experienced at the 12 core machine performance increases at larger numbers of horizontal zones we found that the four core machine has its performance optimum at lower numbers of horizontal zones. The data ranges in tables 16, 17, 18 and 19 present the critical subset out of all tested parameters. The new supernode transformation applied to the original loop nest scores a speedup of $51.33/3.17 = 16.2$ on the four core machine and $46.44/0.79 = 58.8$ on the 12 core machine

		Number of vertical zones N/h									
		10	50	250	500	1000	2000	6000	10000	14000	18000
Number of horizontal zones N/w	50	4.83	2.53	2.07	3.65	3.74	3.98	3.54	5.36	7.11	8.61
	250	1.89	1.05	0.81	1.05	0.83	0.92	1.49	2.37	3.31	4.13
	500	1.96	1.08	1.29	0.78	0.81	1.80	1.52	2.45	3.21	4.08
	750	2.41	1.50	0.82	0.80	0.83	0.92	1.98	2.27	3.09	3.93
	1000	2.41	1.12	1.87	0.80	0.84	0.89	1.84	2.63	4.46	4.03
	1500	2.35	1.21	0.84	1.13	0.84	0.89	1.41	2.30	3.19	4.09
	2000	2.46	1.81	0.86	0.82	0.85	0.94	1.47	2.53	3.27	4.19
	2500	2.37	1.32	0.89	0.84	0.83	0.97	1.48	2.35	3.31	4.29
	3000	2.43	1.30	0.88	0.95	0.85	0.93	1.45	2.33	3.29	4.18
	3500	2.47	1.33	0.88	0.83	1.22	0.92	1.80	2.24	3.09	4.02
	4000	2.56	1.31	0.87	0.82	0.85	0.90	1.39	2.17	3.18	3.82
	4500	2.51	1.33	0.91	0.82	0.83	0.88	1.65	2.31	2.84	3.69
	5000	2.56	1.34	0.90	0.82	0.81	0.87	1.63	2.24	2.76	3.58

Fig. 19. Our variation of the supernode transformation applied to the skewed loop nest at the Xeon machine ($p = 12, t = 2$)

over the original loop nest. The same method applied to the skewed loop nest achieves a $51.33/1.76 = 29.2$ speedup on the four core machine and a $46.44/0.78 = 59.5$ speedup on the 12 core machine. One observation is that skewing the loop nest with this method improved the performance only at the four core machine.

VII. Conclusion

This paper emphasized on supernode transformations and multithreading for the LCS problem by applying and enhancing previous work as well as presenting a variation of the supernode transformation and threading strategy. One major observation of this paper is that multithreaded implementations of the LCS score super-linear speedups. Applying Hodzic's method (traditional supernode transformation) to the LCS problem, it scores a 16.7 times speedup over the original loop nest on a 12 core MIMD machine. Enhancing this technique to eliminate loop carried dependencies along the inner loop within each supernode scored a 42.6 times speedup. To reduce the thread management overhead and improve the data reuse of threads we introduced a variation of the supernode transformation and threading strategy scoring a 59.5 times speedup. For each method we cited and presented the functions for the total execution time consider systems with infinite and systems with limited resources. The techniques presented in this paper are especially applicable to MIMD architectures. We benchmarked the techniques on two modern x86-64 multi-core machines with four cores and 12 cores. Techniques, ideas and formulas presented in this paper on the LCS problem are generally applicable to regular dependence algorithms.

We especially see potential for this technique at future MIC systems where a large number of processing cores are available. As these systems usually consist of in-order architectures, especially our new variation of the supernode transformation should be applied. The techniques and concepts presented in this paper may be improved

by applying software pipelining. Also, how to port the presented techniques to SIMD architectures such as most recent GPGPU's will be conducted in future work.

References

- [1] 1003.1 standard for information technology portable operating system interface (posix) rationale (informative). *IEEE Std 1003.1-2001. Rationale (Informative)*, pages i–310, 2001. 4
- [2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13*, pages 7–16, Juan-les-Pins, France, September 2004. 4
- [3] B. Ben Mabrouk, H. Hasni, and Z. Mahjoub. Parallelization of the dynamic programming algorithm for solving the longest common subsequence problem. In *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on*, pages 1–8, may 2010. 2, 5
- [4] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 39–48, 2000. 2
- [5] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 4
- [6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001. 3
- [7] T. Garcia, J.-F. Myoupo, and D. Seme. A coarse-grained multi-computer algorithm for the longest common subsequence problem. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 349–356, feb. 2003. 1, 2, 3
- [8] G. Goumas, N. Drosinos, and N. Koziris. Communication-aware supernode shape. *Parallel and Distributed Systems, IEEE Transactions on*, 20(4):498–511, april 2009. 2, 4
- [9] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24:664–675, October 1977. 2
- [10] E. Hodzic and W. Shang. On supernode transformation with minimized total running time. *Parallel and Distributed Systems, IEEE Transactions on*, 9(5):417–428, may 1998. 2, 3, 4, 5
- [11] E. Hodzic and W. Shang. On time optimal supernode shape. *Parallel and Distributed Systems, IEEE Transactions on*, 13(12):1220–1233, dec 2002. 2, 4, 8
- [12] S.-H. Hu, C.-W. Wang, and H.-L. Chen. An efficient and hardware-implementable systolic algorithm for the longest common subsequence problem. In *Machine Learning and Cybernetics, 2008 International Conference on*, volume 6, pages 3150–3155, july 2008. 2
- [13] D. Korkin, Q. Wang, and Y. Shang. An efficient parallel algorithm for the multiple longest common subsequence (mlcs) problem. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 354–363, sept. 2008. 2
- [14] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture, ISCA '10*, pages 451–460, New York, NY, USA, 2010. ACM. 4, 7
- [15] J. Liu and S. Wu. Research on longest common subsequence fast algorithm. In *Consumer Electronics, Communications and Networks (CECNet), 2011 International Conference on*, pages 4338–4341, april 2011. 1
- [16] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers, SIGGRAPH '08*, pages 18:1–18:15, New York, NY, USA, 2008. ACM. 4, 7
- [17] W. Shang and J. A. B. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Trans. Comput.*, 40(6):723–742, 1991. 3, 4

Hard-Sphere Collision Simulations with Multiple GPUs, PCIe Extension Buses and GPU-GPU Communications

K.A. Hawick

D.P. Playne

Computer Science, Institute of Information and Mathematical Sciences

Massey University – Albany

North Shore 102-904, Auckland, New Zealand

Email: {k.a.hawick,d.p.playne}@massey.ac.nz

Tel: +64 9 414 0800 Fax: +64 9 441 8181

Abstract

Simulating particle collisions is an important application for physics calculations as well as for various effects in computer games and movie animations. Increasing demand for physical correctness and hence visual realism demands higher order time-integration methods and more sophisticated collision management algorithms. We report on the use of single and multiple Graphical Processing Units (GPUs) to accelerate these calculations. We explore the performance of multiple GPUs (m-GPUs) housed on a single PCIe bus as well as the use of special purpose PCIe bus extender technology using GPU housing chassis systems such as Dell's C410x PowerEdge. We describe how a hard sphere collision system with gravitational interactions was developed as a benchmark. We compare the performance of various GPU models and show how algorithms that use GPU-GPU communications with NVidia's Compute Device Unified Architecture (CUDA 4) can considerably aid communications amongst multiple GPUs working on a single simulated particle system.

Keywords: hard-sphere collisions; m-GPU; GPU-GPU communication; CUDA 4; PCIe bus.

1 Introduction

Particle simulation is a technique used heavily in the computer games industry and also for constructing animation of sophisticated computer generated scene effects in the movie industry. Traditionally some rather poor approximations to the physics have been used in these applications to save on computational requirements and in many circumstances these are not noticed by the viewer or player.

We are interested in software for “physics engines” (Bourg 2002, Conger & LaMothe 2004) that make better approximations to the point of being able to explore the statistical mechanical behaviours or numerical experiments based on particles. We are therefore interested in high quality physics engines that might ultimately be used as games engines (Thorn 2011, Millington 2007, Gregory 2009, Menard 2011) as well – providing sufficient real time performance can be achieved.

Graphical Processing Units (GPUs) have found

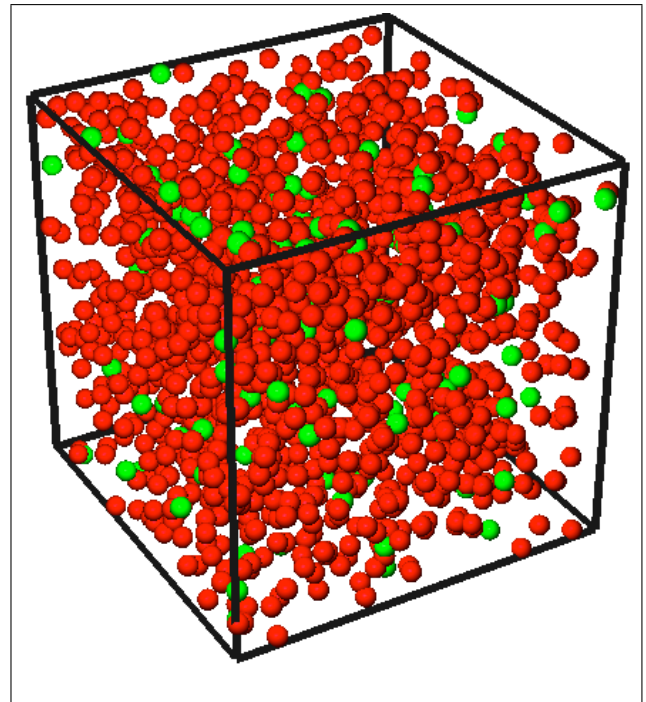


Figure 1: Simulated hard spheres, with density: 0.025(red); 0.0025(green).

many widespread recent uses in accelerating the performance of many scientific and simulation calculations. GPU commodity pricing and ubiquity means they are widely available in many games computers, but they are also finding sophisticated uses in supercomputers and indeed many of the world current top supercomputers (Meuer, Strohmaier, Simon & Dongarra 2010) employ GPU technology as general purpose “GPGPU” accelerators - to speed up calculations and not just graphical rendering (Wright, Haemel, Sellers & Lipchak 2011).

GPUs come in a number of different models with different numbers of cores; floating point capabilities and often very importantly different levels and amounts of memory (Leist, Playne & Hawick 2009, Playne & Hawick 2011). We have experimented with a number of individual GPU devices ranging from low priced game rendering models to top end gamer devices such as the GTX 580/590 series and to blade quality devices such as the C2050/C2070 series.

GPUs provide a powerful data-parallel architecture for classical N-body particle dynamics simulations which have in fact been used as a benchmark application for such devices (Hawick, Playne & Johnson 2011, Playne, Johnson & Hawick 2009, Nyland, Harris & Prins 2007, Stock & Gharakhani 2008, Kavanagh, Lewis & Massingill 2008). We extend this

idea to the simulation of hard spheres under the influence of a gravitational field. The use of hard-sphere collisions(Allen & Tildesley 1987) provides a communications challenge involving correct book keeping management of collisions as well as the computational challenge of performing accurate numerical integration(Hawick et al. 2011) of the classical mechanical equations of motion for an N-body system. We are interested in large and complex systems where there are many particles and interactions, but the system density gives us a parameter to vary along with the number of processing cores to use this application as a benchmark for exploring performance tradeoffs of modern GPUs.

An attractive and relatively recent development involves the use of multiple GPUs that all cooperate to support and accelerate the performance of a single CPU. Nvidia's Compute Unified Device Architecture(CUDA) – version 4 – offers software capabilities to manage direct communications between such cooperating GPUs and without passing data via the controlling CPU.

Graphical Processing Units (GPUs) are generally connected to their hosting processors via a Peripheral Component Interconnect Express (PCIe) bus(PCI-SIG 2010) and usually motherboards will support at most four PCIe devices. Although known as a bus, the PCIe standard is actually structured as point-to-point serial links with dynamic bandwidth negotiation. We construct a synthetic benchmark application to measure the bandwidth, latency and PCIe bus contention issues that arise as a multi-threaded CPU host program delegates work to m-GPU accelerator devices. We explore performance properties of different models of GPUs as well as that of PCIe extender cards and device chassis that support operation of more than four GPU devices from one CPU. In addition to benchmark data we discuss applications and appropriate software and threading architectures to make good use of GPUs and GPU-accelerated clusters configured in this manner.

We use this application as a benchmark for a m-GPU system built using Dell's C410x PowerEdge chassis(Dell Inc. 2010) for extending the PCIe bus and compare the performance of various GPU/cluster combinations. We discuss how the compute to communications ratio of an application that involves hard inelastic collisions differs from the simple N-body particle dynamics as a data-parallel benchmark.

Figure 1 shows a rendering of a simulation of several thousand colliding hard spherical particles with a density of around 0.025. This is in fact quite a high value that gives rise to many collisions in each simulated time unit. In this paper we explore how the number of particles in the system can be increased using multiple GPUs, but also how the performance changes as we increase the particle system density - and hence change the communications to computation ratio.

This paper is organised as follows. We describe the algorithms for approximating the mechanics of interacting hard cored spherical particles in Section 2. In Section 3 we describe the GPU configurations we employed and our GPU code implementations using CUDA. We present some performance results for various GPUs at different numbers and densities of simulated particles in Section 5 and discuss the scalability and implications for combining GPUs together in Section 6. We offer some conclusions and areas for further work in Section 7.

2 Hard Core Collisions & Interactions

In addition to game physics engine applications (Eberly 2006), a number of problems in chemistry and physics can be formulated in terms of interacting hard-core bodies. A hard-core body in this context simply means a rigid body that cannot be deformed beyond a certain point. Spheres are particularly useful for many models since they are very easily parameterised in terms of a position (of the centre) and a radius. Spheres can be rendered in a 3-D space with shading or false colour or texture maps and can be used to approximate planetary dynamical system or simple molecular models.

The packing density is essentially a measure of the wasted space when you pack a number of solids into a (large) box. A limiting fraction (between 0 and 1) gives a universal measure of this for different shapes. There are some important physical and chemical properties of various materials that relate to the packing density of their component molecules. The volume of a single sphere of radius a is just $\frac{4}{3}\pi a^3$ so if there are N non-overlapping spheres in a rectilinear box of Volume V , it is straightforward to relate the radius to the density ρ of the simulated system.

A number of authors have reported on well known approaches to parallelising the N-body particle dynamical problem. Approaches include use of space dividing oct-trees(Warren & Salmon 1993, Barnes & Hut 1986) to allow the $O(N^2)$ computation to be reduced to $O(N \log N)$ and data parallel computers have been applied successfully to this sort of problem for some years(Brunet, Mesirov & Edelman 1990, G.Fox, M.Johnson, G.Lyzenga, S.Otto, J.Salmon & D.Walker 1988, Playne 2008). A common approach is to divide the particle up amongst processing units and employ a one-dimensional communicating ring approach to allow all processors to access all (possibly reduced) information on the particles they do not have direct responsibility for updating.

Vector parallel techniques have also been successfully employed for updating collision lists of hard disks and hard spheres(Allen & Tildesley 1987, Donev, Torquato & Stillinger 2005). In this present paper we employ a hybrid approach since we use hard core particles that cannot intersect but also wish to apply high-order numerical integration methods to accurately track the changing trajectories of particles under the influence of gravity.

Our benchmark code is aimed at simulating the behaviour of interacting polydisperse hard-core spheres. The code computes collision dynamics for a system of N hard spheres, which are contained in a periodically repeating cell of unit edge length. The spheres, labelled by index i have diameters σ_i , and masses m_i . Their positions \mathbf{r}_i , velocities $\mathbf{v}_i = \frac{d\mathbf{r}_i}{dt}$ and accelerations $\frac{d^2\mathbf{r}_i}{dt^2}$ are tracked during the simulation. Hard spheres that only interact via inelastic collisions do not need accelerations to be recorded. Accelerations are however required and computed when a global gravitational field is applied. To make this physically meaningful we apply fixed boundaries for the roof and floor of the simulated box, but retain periodicity in the horizontal dimensions.

We consider a set of N particles, labelled $i = 0, 1, 2, \dots, N-1$ that interact via pair-wise interactions that are dependent on various properties of the particles. Central forces that depend solely upon the relative distance between particles i and j . For example the Newtonian gravitational potential arising on

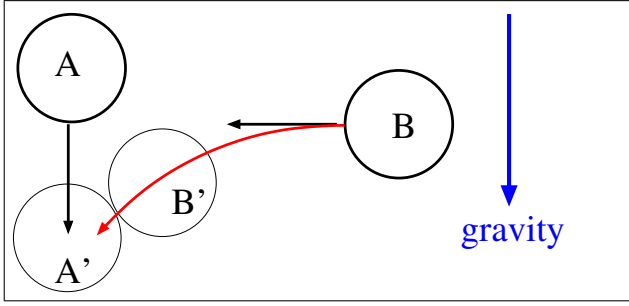


Figure 2: Hard Sphere Collision with gravitational forces applied.

the i 'th particle from the j 'th, $V(r_{i,j})$, can be written as:

$$V_{G_{i,j}}(r_{i,j}) = -\frac{Gm_i m_j}{r_{i,j}} \quad (1)$$

An alternative pair-wise force law that might model chemical van der Waal's force or some other form of long range attraction between particles (i, j) which can be approximated using a Lennard-Jones potential as:

$$V_{L-J}(r_{i,j}) = 4\epsilon \left[\left(\frac{\sigma}{r_{i,j}} \right)^{12} - \left(\frac{\sigma}{r_{i,j}} \right)^6 \right] \quad (2)$$

We can also add in a fixed extra term that depends solely on height $V_g(y) = m_i g |y|$ to model an overall gravitational field so that all particles tend to drift downwards within the simulated box – this can be incorporated into the total $V(\mathbf{r})$. The classical (Newtonian mechanical) force can then be written as the gradient of the potential:

$$\mathbf{F} = \nabla V(\mathbf{r}) \quad (3)$$

For centralised forces like gravitational systems, we can simply sum pair-wise forces along a vector connecting the particle centres, and for a single such axis the gradient is just a single-variable derivative and hence:

$$\mathbf{F}_i = \sum_j Gm_i m_j \mathbf{r}_{i,j} \quad (4)$$

Given Newton's third law: $\mathbf{F}_i = m_i \mathbf{a}_i \Rightarrow \mathbf{a}_i = \mathbf{F}_i / m_i$ and we can employ the separate x, y, z components of the acceleration in the Newtonian classical rigid body equations of motion so that we compute changes in particle i 's velocity and position.

We consider the possibility that particles interact over long ranges via such a force law, but that they also have some minimum distance of separation $2a$ so that they are hard-cored particles and cannot overlap.

Figure 2 shows the problem of two hard spheres that collide. The particles are hard spheres, and collide inelastically – they do not interact except for an impulse applied at the point of collision. However both do experience the externally applied gravitational field force and particle A which has a horizontal initial velocity follows a parabolic trajectory. The time and point of collision with particle B must be determined to correctly apply the collision behaviour.

Our model is summarised in Algorithm 1.

In effect then, the inelastic hard-core collisions are impulsive corrections to the time integration of the usual N-body Newtonian mechanics. We can use any time-integration algorithm we wish, but higher-order will yield better energy conservation. The inelastic

Algorithm 1

```

initialise  $N$  particles in 3D
for  $t \leftarrow 0$  to  $T_{\max}$  do
    compute gravitational and pair-wise force sums
    on each particle
    compute particle accelerations from forces
    time integrate all particles by  $h$ 
    check for penetrating core collisions
    while collision occurred do
        warp time back to earliest collision, undoing
        core penetrations
        compute impulses for collision, correcting ve-
        locities
        check for penetrating core collisions
    end while
    resume time integration
end for
    
```

collisions with box boundaries would change the overall energy and the simulated system would approach an equilibrium temperature given the particle core radii. In this present paper we focus on the GPU performance and benchmarking aspects and do not explore these statistical physics effects further.

3 GPU Implementation

Understanding the scaling and communications performance of multi-GPU systems (Spampinato 2009) is becoming very important as large scale supercomputers that use them become more prevalent – as evidenced by those on the present Top 500 world list of supercomputers (TOP500.org n.d., Meuer et al. 2010). A key aspect to understanding their behaviour is the scalability of the Peripheral Component Interconnect Express (PCIe) bus (PCI-SIG 2010) used to communicate between GPUs and CPU. PCIe is a sophisticated technology – it is a point-to-point serial structure with lanes, dynamic negotiation and has developed through a number of versions. It is implemented by a number of different vendors on different GPU models and boards.

Typically a PC motherboard is limited to having at most four PCIe slots into which GPU cards can be located. In some cases this is further limited by power requirements and physical geometry of the slots. Dell and other manufacturers are now making available a number of PCIe bus extenders in the form of an integrated chassis that allows various software controlled configurations of devices. The Dell C410x PowerEdge chassis (Dell Inc. 2010) we discuss in this present paper houses up to 16 GPUs each in its own bay, and provides power and cooling suitable for a machine room environment. This is a good deployment platform to experiment with blade-level GPUs, although much of our earlier work was successfully carried out using the very much cheaper gamer-quality GPUs that do not have error-corrected memory.

3.1 Single-GPU Implementation

This benchmark makes use of the well-known tiling algorithm to compute the all-pairs gravitational forces between particles (Nyland et al. 2007, Playne et al. 2009). This algorithm processing the particles in tiles stored in shared memory. Each block of threads will load one tile of particles at a time into shared memory and each thread computes the force of gravity exerted by each of these particles on the single particle that threads is responsible for.

A similar algorithm can be used to detect collisions between particles. Each pair of particles must be compared to determine if they have collided during the previous time step. If a collision has occurred, the time at which it happened is calculated and saved. If a thread's particle is involved in multiple collisions, it will record the time of the first collision and the index of the particle it collided with. The threads use atomic operations to compute the time at which first collision in the entire system occurred. The kernel used to detect these collisions is shown in Algorithm 2. This algorithm will determine the first collision particle i was involved in and will be computed for each particle by a separate thread. Each thread will compute $max_times[i]$ which is how long ago the first collision of that particle occurred.

Algorithm 2

```

collision detect kernel
 $p1 \leftarrow particles[i]$ 
for  $j \leftarrow 0$  to  $N$  do
   $p2 \leftarrow particles[j]$ 
   $d \leftarrow$  distance from  $p1$  to  $p2$ 
  if  $d < p1.R + p2.R$  then
     $t \leftarrow$  required step back
    if  $t > max\_times[i]$  then
       $max\_times[i] = t$ 
       $collision\_index[i] = j$ 
    end if
  end if
end for
AtomicMax( $system\_max\_time, max\_times[i]$ )

```

To ensure all collisions are processed correctly, they are resolved one at a time. Once the time of the first collision is determined, a kernel is launched which will test to see if the particle was involved in the first collision. If the thread's particle was not involved in the collision, the kernel will immediately return, this can be determined by comparing that particle's collision time with the time of the first collision. If the thread's particle was involved in the collision it will fetch the index of the other particle involved in the collision. The thread with the lowest index will step both particles back in time to the point of the collision, perform the collision and step both particles back to the current system time. The kernel to perform this collision process is shown in Algorithm 3.

Algorithm 3

```

collide particles kernel
if  $max\_times[i] == system\_max\_time$  then
   $j \leftarrow collision\_index[i]$ 
  if  $i < j$  then
     $p1 \leftarrow particles[i]$ 
     $p2 \leftarrow particles[j]$ 
    step particles  $p1$  and  $p2$  back by  $system\_max\_time$ 
    collide particles  $p1$  and  $p2$ 
    step particles  $p1$  and  $p2$  forward by  $system\_max\_time$ 
  end if
end if

```

This process of detecting the first collision and resolving it must be performed iteratively until no more collisions occur. The collision detection kernel will return 0 when no collisions have occurred. These kernels, along with the force calculation and integration kernels can be used to compute an N-body simulation with collisions on a single GPU. However, utilising multiple GPUs is somewhat more difficult.

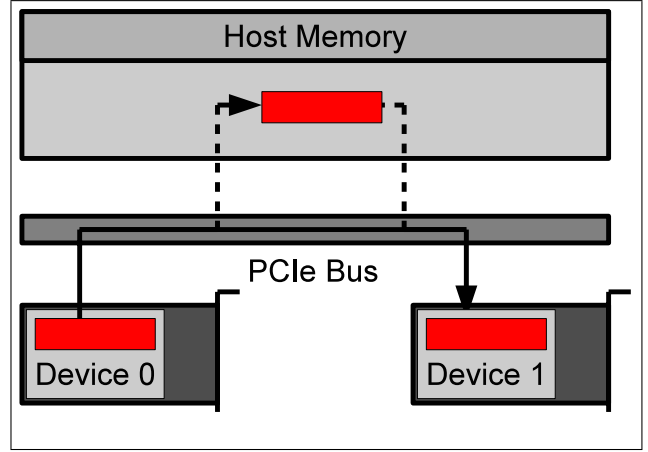


Figure 3: A diagram comparing GPUDirect communication and memory transfer through the host. The dotted lines show the transfer of the data through host memory.

3.2 M-GPU Implementation

The main challenge of computing an N-body simulation on an m-GPU system is managing the communication between them. This m-GPU implementation evenly distributes N particles between P devices such that each GPU device is responsible for $\frac{N}{P}$ particles. For each device to compute the total force on each of its particles, it must copy the particle data out of the other GPUs. Likewise to detect if any collision have occurred, each device must compare each of its particle with each other as well as the particles stored on other GPUs. Including m-GPU communication in the benchmark allows the test system to be evaluated in terms of computation as well as communication throughput.

This implementation makes use of the CUDA 4.0 functionality of peer-to-peer memory transfer. GPUDirect 2.0 allows data to be copied directly from one GPU device into another across the PCI-e bus. Without GPUDirect 2.0, data had to be first copied out of the device to host memory and then into the second device. This peer-to-peer communication can significantly improve the performance of m-GPU applications.

Algorithm 4 shows the basic algorithm for computing the total force on each particle. Initially a kernel is called to compute the forces the particles on the device exert on each other. Once this has been completed the device will loop through all the other device. For each iteration of the loop, the device will copy the particle data out of the other device into its memory. It then computes the total force those particles exert on its particles.

Algorithm 4

```

call compute force kernel
for  $d \leftarrow 0$  to  $num\_devices$  do
  if  $i \neq d$  then
    copy particles from  $device_d$  to  $device_i$ 
    call compute force kernel
  end if
end for
time integrate all particles by  $h$ 

```

A similar algorithm is used to detect particle collisions. Initially a kernel is called to detect collisions of particles on the same device. Once this is completed the particles on other devices are checked for colli-

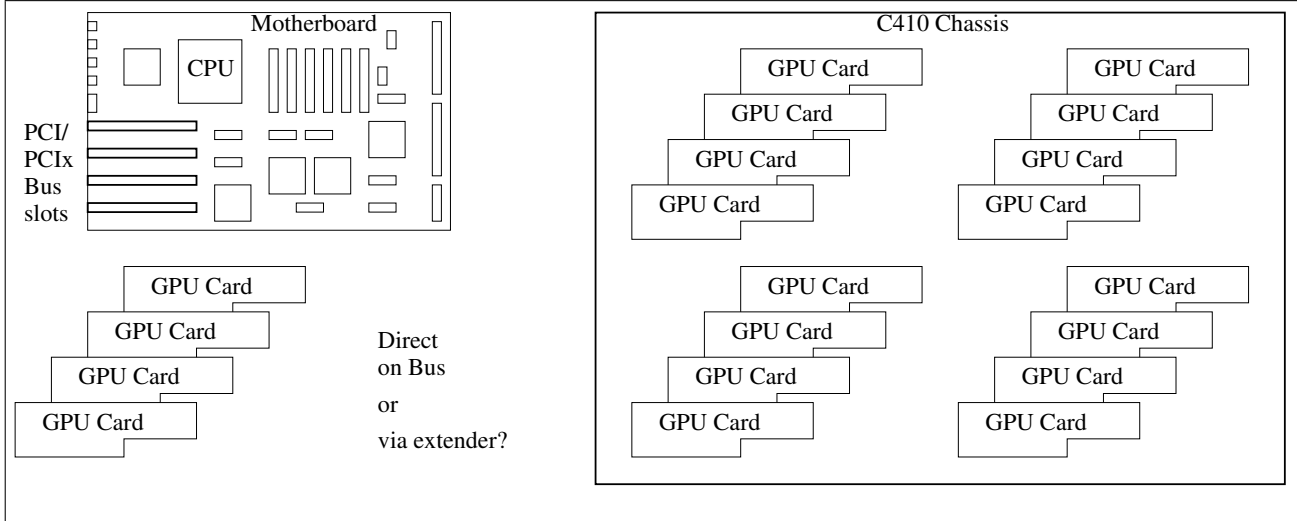


Figure 4: Use of Dell C410x chassis to extend PCI Bus to support more GPUs per CPU.

sions. Like the force calculation, the devices are iterated through and their particles copied to the current device. These particles are then checked for collisions with this device's particles. This process is presented in pseudo code in Algorithm 5

Algorithm 5

```

 $max\_time_i \leftarrow 0$ 
copy  $max\_time_i$  into  $device_i$ 
call collision detect kernel
for  $d \leftarrow 0$  to  $num\_devices$  do
    if  $i \neq d$  then
        copy particles from  $device_d$  to  $device_i$ 
        call collision detect kernel
    end if
end for
copy  $max\_time_i$  out of  $device_i$ 
    
```

To determine when the first overall collision occurred, the first collision times for each device must be compared. The time of the first collision is found and the index of the device is recorded. If the collision occurred between two particles on the same device, the same kernel as in Algorithm 6 will be launched. If the collision occurred between particles on different devices, the appropriate data will be exchanged and one kernel on each device will be launched. This kernel will update the particle on the device that was involved in the collision. The process for determining the time of the first collision and computing the collision is shown in Algorithm 6.

Algorithm 6

```

 $system\_max\_time \leftarrow 0$ 
for  $d \leftarrow 0$  to  $num\_devices$  do
    if  $max\_time_d > system\_max\_time$  then
         $system\_max\_time \leftarrow max\_time_d$ 
    end if
end for
if collision is on  $device_i$  then
    call collide particles kernel on  $device_i$ 
else if collision is on  $device_i$  and  $device_j$  then
    copy particles from  $device_i$  to  $device_j$ 
    copy particles from  $device_j$  to  $device_i$ 
    call collide particles kernel on  $device_i$ 
    call collide particles kernel on  $device_j$ 
end if
    
```

This basic algorithm can be used to implement an N-body simulation with hard sphere collision on

a m-GPU system. The main point of difference between the implementations is how the data transfer is performed. This includes both removing redundant communication as well as different methods of implementing communication between devices.

When collisions are detected and computed, the devices must exchange particle data. If multiple collisions occur within a single time step, the devices must exchange this data multiple times. However, most of the particle data will remain unchanged. Only the data about the particles that were involved in the collision needs to be propagated to the other devices. This is not a change in the fundamental algorithm, merely a reduction in the data that is communicated.

The more important difference in implementations is the method of CUDA communication. Fermi architecture GPUs have support for GPUDirect 2.0 which allows data to be directly communicated between devices. This is the preferred communication method used by the benchmark. However, Tesla architecture GPUs do not support this functionality and any data transfer must be communicated through the host memory.



Figure 5: The Dell C410x Chassis - shown with cooling fans exposed, in process of having GPUs installed and HCI connector cables linking the extended PCI bus to the internal PCI bus of a hosting PC..

4 Benchmarking

The benchmark has been used to evaluate a number of Fermi architecture GPUs in a variety of configurations. The testing has been focused on high-performance graphics cards and compares the gamer level GTX480 and GTX580 with the professional C2050 and C2070 GPUs. Both the single-GPU and m-GPU implementations have been executed on these cards to compare their performance. The C2050 and C2070 cards have been tested both hosted on a traditional motherboard and on the Dell C410x chassis.

To compare the performance of the different GPU configurations, the simulations have been run with three different system configurations which varies the computation between the highly parallel force computation and the more restricted collision detection. A particle configuration with very low density is very computationally similar to the benchmark without particle collisions as the computation is almost entirely parallel. Whereas particle configuration with a high density will have a great deal more collisions and requires more communication between devices to resolve these collisions.

To test the difference in performance based on particle density, three different initialisation configurations have been compared. The first initialises the particle positions and velocities randomly with a very low density. This configuration results in almost no collisions and purely tests the parallel processing performance. The second initialises the particle positions in a two-dimensional lattice with random velocities, this results in a medium density simulation with a collision rate of ≈ 0.03 collisions per particle per step. The final configuration initialises the system with the particles laid out in three-dimensional lattice with random velocities. In this configuration there are a higher number of collisions, the rate is ≈ 0.06 per particle per step. The number of collisions vs system size is shown in Figure 6.

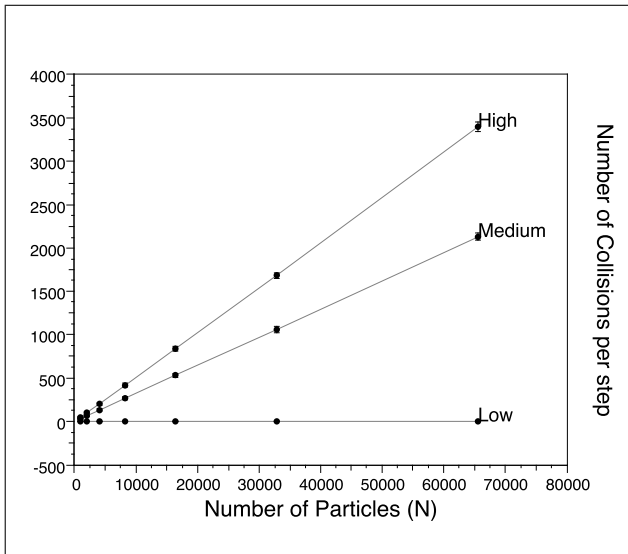


Figure 6: Number of collisions per step for Low (0.00), Medium (0.03) and High (0.06) density configurations.

5 Results

We present some selected performance timing data for the hard core gravitational particle simulation for a number of GPU devices in both single GPU and m-GPU configurations. These configurations are evaluated for a number of different system sizes but also with the three different particle densities discussed in the previous section - low, medium and high.

The first case is the dilute limit of density where no collisions occur during the period of the benchmark, effectively these systems have a collision per particle per step ratio of 0.00. This test will show the best possible performance as the computation can be executed entirely in parallel. Collisions are still detected but as they never occur then there is no serial process required to resolve them. The single-GPU implementation has been tested on the GTX480, GTX580, C2050 and C2070. The multi-GPU implementation has been tested on 2xGTX480, 2xGTX580, 2xC2050, 2xC2070, 4xC2050 and 4xC2070 configurations. The performance plot of this benchmark is shown in Figure 7.

From this plot it can be seen that the GTX580, C2050 and C2070 GPUs offer almost indistinguishable performance while the GTX480 performs significantly slower. However, for the m-GPU implementation the 2xC2050 and 2xC2070 configurations perform faster than the 2xGTX480 and 2xGTX580 systems.

The medium particle density systems (≈ 0.03 collisions per particle per step) shows an interesting change shift in single-GPU performance. For this test the gamer level GTX480 and GTX580 both performed faster than the C2050 and C2070 cards. However, once again the 2xC2050 and 2xC2070 both showed higher performance for the m-GPU implementations. The performance plots the medium density configurations (≈ 0.03) on the different devices can be seen in Figure 8.

The high particle density (≈ 0.06) configuration shows very similar results to the medium density (≈ 0.03) systems. The GTX480 and GTX580 cards both offer the best single-GPU performance but the Tesla compute cards provide the best m-GPU performance. These results suggest that the C2050 and C2070 Tesla compute cards have faster communication than the GTX480 and GTX580 graphics cards. NVidia datasheet specifies that Tesla computing GPUs support faster PCIe communication, evidenced by these findings (NVIDIA 2011). The performance results for the high-density configurations are shown in Figure 9.

The performance results for the C2050 and C2070 configuration shown in this section have been hosted on a Dell C410x chassis. This configuration has been compared to a configuration hosting the same GPUs on a traditional motherboard. The results show no measurable difference between the two configurations for single or m-GPU implementations and thus have not been presented separately. For this benchmark, hosting the devices on a C410x does not degrade performance.

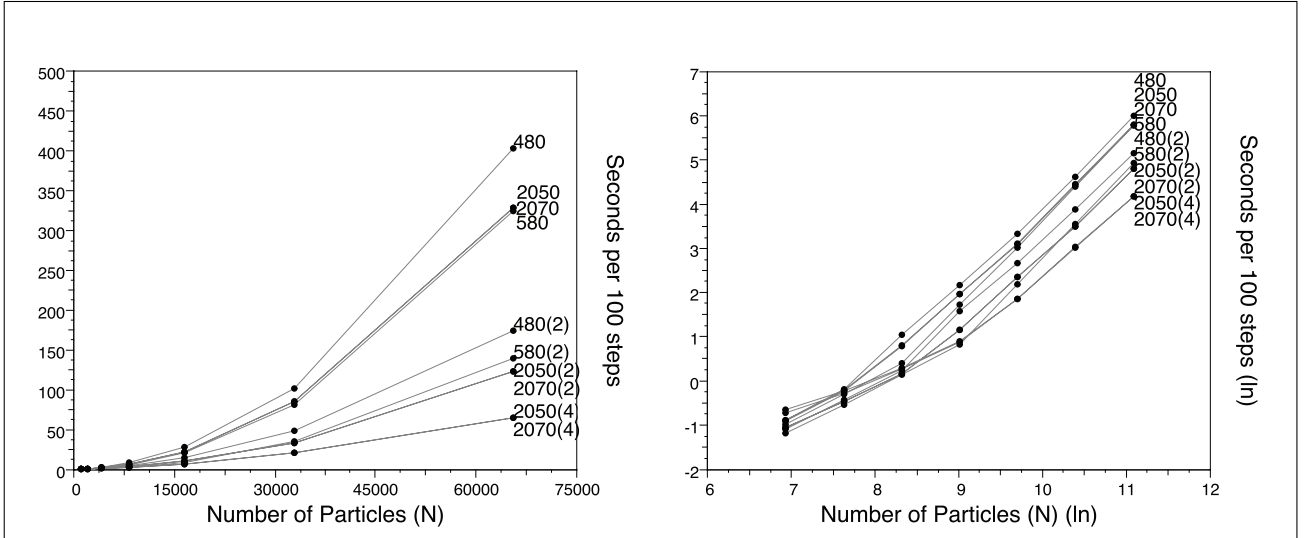


Figure 7: Comparison of GPU configurations computing N-body simulations initialised at the dilute limit of density. Results are shown for system sizes in the range $N = \{1024, 2048 \dots 65536\}$ in normal scale (left) and ln-ln scale (right).

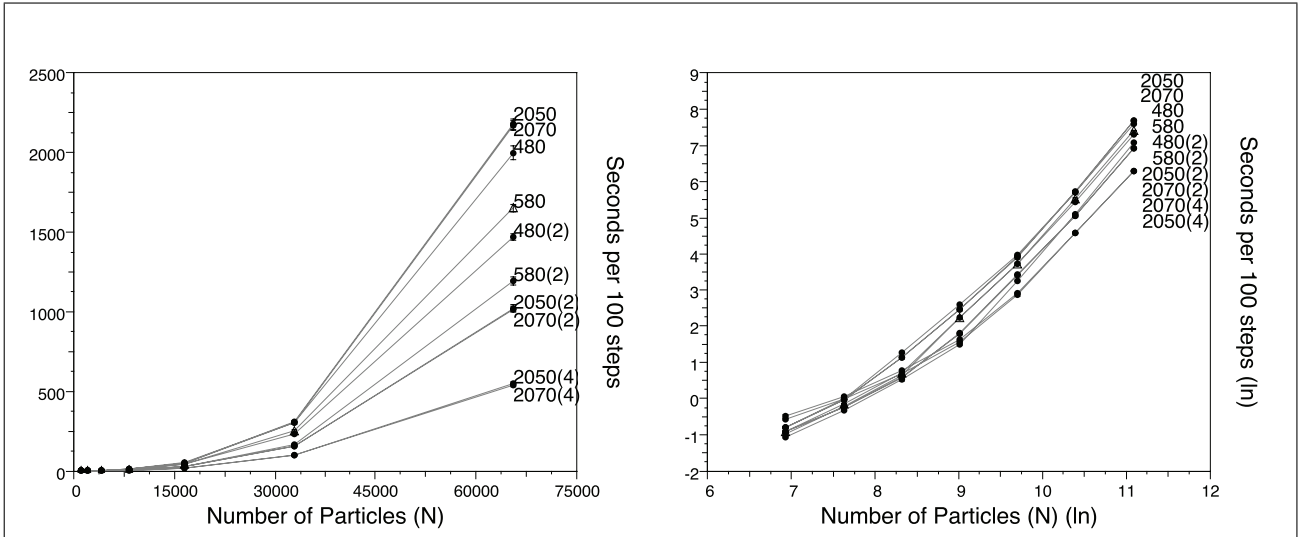


Figure 8: Timing data comparing GPU devices computing medium density configuration initialised in a two-dimensional lattice with random velocities. Results are shown for system sizes in the range $N = \{1024, 2048 \dots 65536\}$ in normal scale (left) and ln-ln scale (right).

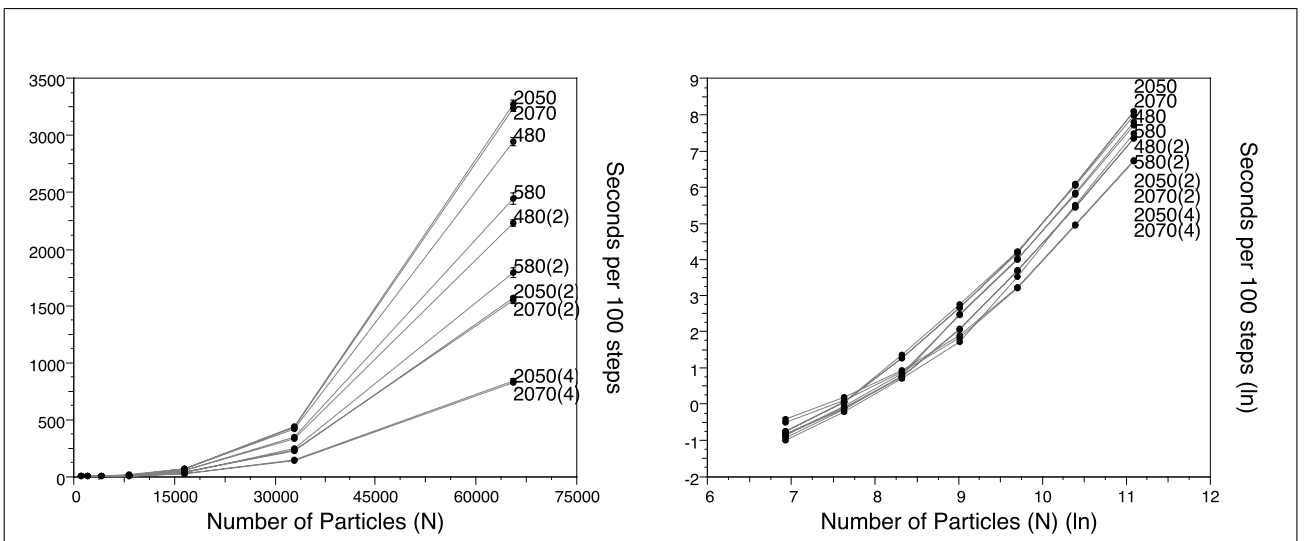


Figure 9: Timing data for single and m-GPU implementations on various devices with high density configurations initialised in a three-dimensional lattice with random velocities. Results are shown for system sizes in the range $N = \{1024, 2048 \dots 65536\}$ in normal scale (left) and ln-ln scale (right).

6 Discussion

Including hard-sphere particle collisions into the N-body benchmark allows greater insights into the performance of different GPU configurations. The gamer-level GeForce cards provided similar or slower performance compared the Tesla compute cards for low density configurations where collisions almost never occurred. However, for medium and higher density systems the GeForce cards both provided significant performance benefits over the Tesla compute cards.

However, for m-GPU implementations where device-device communication is required, the results showed the opposite. The Tesla compute cards provide a significant performance benefit as compared to the GeForce graphics cards, this performance difference is attributed to the Tesla compute cards' improved PCIe communication capabilities (NVIDIA 2011). This difference in performance was more pronounced for configuration medium and high particle densities as more communication is required to resolve collisions.

The Tesla compute cards are significantly more expensive than the consumer-level GeForce graphics cards yet for single-GPU applications where ECC is not required, the GeForce cards provide comparable or improved performance. However, the Tesla compute cards are the only GPUs that support ECC and for m-GPU implementations they provide significantly higher performance. We believe this is due to the better DMA transfer capabilities of the Tesla over GTX cards.

These Tesla compute cards can be hosted in PCIe extender chassis's such as Dell's C410x. For the single-GPU and m-GPU implementations tested in this research, devices hosted on this chassis provided performance that was indistinguishable from the same cards hosted on a traditional motherboard. This shows that the chassis does not degrade performance in any way, as was initially feared.

The m-GPU implementation was only tested with four devices on this chassis. Host machines are now available which can host multiple HCI cards to allow a single host to connect to up to eight devices hosted on the C410x. However, we do not currently own such a host machines and cannot currently test the performance of such a configuration.

7 Conclusions

We have described how a three dimensional N-body interacting particles model with hard-core collisions can be implemented on single GPU and m-GPU systems with several parallel algorithmic approaches all within a single application program. We have used this application as a parameterised benchmark, using the particle density and hence the average collisions per integration time-step as a benchmark parameter with which to explore computation to communications ratios.

We have discussed how features of NVidia's latest CUDA release aid the performance of this benchmark – in particular those that support direct GPU-to-GPU communications without passing through CPU code. We have also explored the performance capabilities of commodity priced gamer level GPU cards as well as significantly more expensive blade-quality production cards. We found that the gamer-level cards were better for an m-GPU approach, and that this

appears to be due to their enhanced ability to communicate rather than their floating point performance.

We have shown that the PCIe extender bus approach works well and without significant loss of performance for the regimes we have been able to explore. We expect performance to degrade with bus contention as more GPUs are added and we plan to explore this further as more hardware becomes available. We also anticipate availability of further improved GPU models that may have even better floating point performance and communications abilities than those models available to us.

There are open areas of computational physics such as the phase separation of polydisperse particles that can potentially be explored through fast simulations such as we describe. We have experimented with simple spherical particle collisions but there is scope for important work parallelising other and more general rigid body collision algorithms on data parallel architectures such as GPUs. Collision detection and particle dynamics continue to be important algorithms deployed in computer games and we anticipate applications such as we have described as becoming even more important as it becomes standard practice for "gamer computers" to have multiple GPUs available for performance acceleration.

8 Acknowledgments

It is with great pleasure the authors note their thanks to Arno Leist for technical assistance in configuring the C410x GPU chassis.

References

- Allen, M. & Tildesley, D. (1987), *Computer simulation of liquids*, Clarendon Press.
- Barnes, J. & Hut, P. (1986), 'A hierarchical $O(n \log n)$ force-calculation algorithm', *Nature* **324**(4), 446–449.
- Bourg, D. M. (2002), *Physics for Game Developers*, number ISBN 978-0596000066, O'Reilly.
- Brunet, J.-P., Mesirov, J. P. & Edelman, A. (1990), An optimal hypercube direct n-body solver on the connection machine, in 'Proc. Supercomputing 90', IEEE Computer Society, 10662 Los Vaqueros Circle, CA 90720-1264, pp. 748–752.
- Conger, D. & LaMothe, A. (2004), *Physics Modeling for Game Programmers*, Thompson.
- Dell Inc. (2010), *Dell PowerEdge C410x PCIe Expansion Chassis Hardware Owner's Manual*, Dell Inc. <http://www.dell.com/us/enterprise/p/poweredge-c410x/pd.aspx>.
- Donev, A., Torquato, S. & Stillinger, F. H. (2005), 'Neighbor list collision-driven molecular dynamics simulation for nonspherical hard particles: I. algorithmic details', *J. Comput. Phys.* **202**(2), 737–764.
- Eberly, D. H. (2006), *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, number ISBN: 978-0122290633, Morgan Kaufmann.
- G.Fox, M.Johnson, G.Lyzenga, S.Otto, J.Salmon & D.Walker (1988), *Solving problems on concurrent processors*, Vol. 1, Prentice Hall.
- Gregory, J. (2009), *Game Engine Architecture*, A K Peters.
- Hawick, K., Playne, D. & Johnson, M. (2011), Numerical precision and benchmarking very-high-order integration of particle dynamics on gpu accelerators, in 'Proc. International Conference on Computer Design (CDES'11)', number CDE4469, Las Vegas, USA.
- Kavanagh, G. D., Lewis, M. C. & Massingill, B. L. (2008), GPGPU planetary simulations with CUDA, in 'Proceedings of the 2008 International Conference on Scientific Computing'.

- Leist, A., Playne, D. & Hawick, K. (2009), 'Exploiting Graphical Processing Units for Data-Parallel Scientific Applications', *Concurrency and Computation: Practice and Experience* **21**, 2400–2437. CSTN-065.
- Menard, M. (2011), *Game Development with Unity*, Cengage.
- Meuer, H., Strohmaier, E., Simon, H. & Dongarra, J. (2010), '36th list of top 500 supercomputer sites', www.top500.org/lists/2010/11/press-release.
- Millington, I. (2007), *Game Physics Engine Development*, Morgan Kaufmann.
- NVIDIA (2011), 'NVIDIA Tesla Datasheet'.
URL: <http://www.nvidia.com/object/why-choose-tesla.html>
- Nyland, L., Harris, M. & Prins, J. (2007), Fast n-body simulation with cuda, in H. Nguyen, ed., 'GPU Gems 3', Addison Wesley Professional, chapter 31.
- PCI-SIG (2010), 'PCIe Express Base Specification 1.1', <http://www.pcisig.com/specifications/pciexpress/base>.
- Playne, D. & Hawick, K. (2011), Asynchronous communication for finite-difference simulations on gpu clusters using cuda and mpi, in 'Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)', number PDP2793, Las Vegas, USA.
- Playne, D. P. (2008), Notes on particle simulation and visualisation, Hons. thesis, Computer Science, Massey University.
- Playne, D. P., Johnson, M. G. B. & Hawick, K. A. (2009), Benchmarking GPU Devices with N-Body Simulations, in 'Proc. 2009 International Conference on Computer Design (CDES 09) July, Las Vegas, USA.', number CSTN-077.
- Spampinato, D. (2009), Modeling communication on multi-gpu systems, Master's thesis, Norwegian University of Science and Technology.
- Stock, M. J. & Gharakhani, A. (2008), Toward efficient GPU-accelerated N-body simulations, in 'in 46th AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2008-608'.
- Thorn, A. (2011), *Game Engine Design and Implementation*, Jones and Bartlett.
- TOP500.org (n.d.), 'TOP 500 Supercomputer Sites', <http://www.top500.org/>. Last accessed November 2010.
- Warren, M. S. & Salmon, J. K. (1993), A parallel hashed oct-tree n-body algorithm, in 'Supercomputing', pp. 12–21.
URL: citeseer.ist.psu.edu/warren93parallel.html
- Wright, R. S., Haemel, N., Sellers, G. & Lipchak, B. (2011), *OpenGL Superbible*, number ISBN 978-0-321-71261-5, fifth edn, Pearson.

A Comparative Study of Parallel Algorithms for the Girth Problem

Michael J. Dinneen

Masoud Khosravani

Kuai Wei

Department of Computer Science,
University of Auckland,
Private Bag 92019, Auckland, New Zealand
mjd,masoud,kuai@cs.auckland.ac.nz

Abstract

In this paper we introduce efficient parallel algorithms for finding the girth in a graph or digraph, where girth is the length of a shortest cycle. We empirically compare our algorithms by using two common APIs for parallel programming in C++, which are OpenMP for multiple CPUs and CUDA for multi-core GPUs. We conclude that both hardware platforms and programming models have their benefits.

1 Introduction

Graphs are models widely used in science and engineering, and graph algorithms are the basic blocks of many algorithmic solutions to real world problems. In this paper we study the problem of efficiently finding the girth in a graph or digraph on today's common workstations or servers, which often have several processing units (CPUs and GPUs). Modern graph applications require us to find fast algorithms capable of processing large volume of data. In such cases even a low-order polynomial time algorithm may not be able to accomplish a computational task in an acceptable time limit on a single CPU. As a common solution, one can deploy a large number of processors to do the task concurrently. We will discuss how to design and implement parallel girth algorithms and will present actual timing results for classes of hard test graphs.

1.1 Background on parallel programming

Designing parallel PRAM algorithms for graph problems has been the topic of a lot of research; see [3, 18, 19]. Several popular textbooks on parallel computing, such as [9], now address commonly-used shared memory parallel models like Pthreads (POSIX Thread API) and OpenMP (the standard directive-based parallel [17]). In addition to utilizing multiple CPU processors, recently there are more interests in the research community to explore the power of Graphics Processing Units (GPU) for solving graph problems. GPUs are high performance many-core processor devices that were originally designed to handle compu-

tation in image processing. General-Purpose computation on Graphics Processing Units (GPGPU) is the technique to use GPUs for solving a wider range of problems.

Among the first concrete results, Harish and Narayanan [10] introduced some parallel GPU algorithms for various graph problems. In [14], Katz and Kider presented an algorithm using GPUs for solving the all-pairs shortest path problem. Checking graph connectivity was the topic of the paper [20] by Soman, Kishore and Narayanan. As a final example, Leist and Playne [11] gave a GPU parallel algorithm for graph component labeling.

Despite the fact that designing and implementing parallel algorithms have been a major research topic, there are a few results on comparative studies of different APIs and architectures. Comparing CUDA and OpenMP for implementing various parallel girth algorithms is another focus of this paper. With respect to restrictions imposed by the architecture of GPUs, designing and implementing efficient parallel GPU algorithms for irregular data types is a challenging task. When one tries to implement a parallel algorithm for irregular data types on GPUs, there is a large gap between the theoretical and the actual results. Since GPGPU follows the Single Instruction Multiple Data (SIMD) paradigm, as an alternative benchmark, we use the OpenMP API standard, which supports multi-CPU shared-memory parallel programming. It supports C/C++, and Fortran programming languages on many architectures and operating systems. We note that CUDA is (currently) restricted to only NVIDIA graphic cards. However, OpenCL may also be easily used as an implementation choice for many other platforms (e.g. ATI Radeon GPUs) and usually with very little (if any) performance loss [6].

1.2 The girth problem

Girth is defined to be the length of a shortest cycle in a graph if one exists. Generating random graphs with large girth has applications in modelling and testing software systems and coding theory. Producing Tanner graphs with large girths is a main step in construction a Low-Density Parity-Check (LDPC) code; see [2, 12, 15]. A Tanner graph is a bipartite graph whose adjacency matrix is the parity-check matrix of a binary code.

Girth and diameter of a graph are related parameters. The diameter of a biconnected graph with girth

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the Tenth Australasian Symposium on Parallel and Distributed Computing (AusPDC2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 127, Jinjun Chen and Rajiv Ranjan, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

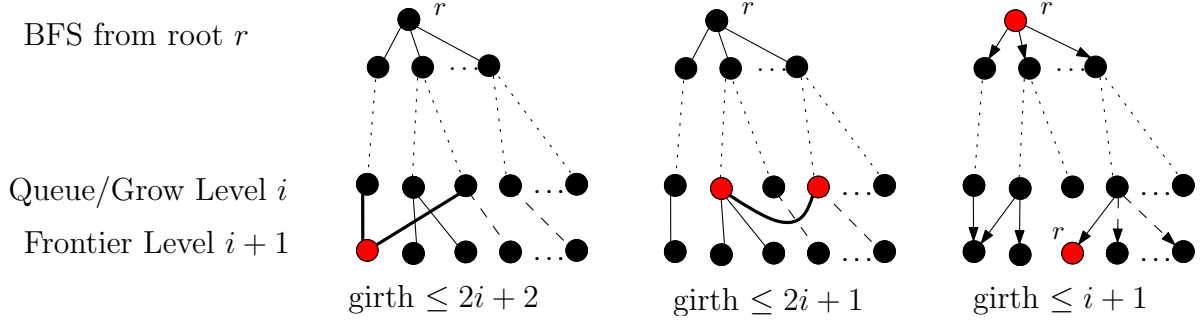


Figure 1: The process of detecting a short cycle via BFS.

$2d$ is at least d . The degree-diameter problem is the well-known problem of finding the largest possible graph with a given degree and diameter [5], and each best-known case usually has a large girth [8]. A large graph with bounded degree and diameter is a good model for an interconnection network topology that has some restrictions on the number of connections between hubs or routers and its maximum communication time between any two nodes.

There is an $O(mn)$ sequential algorithm for finding the girth of a graph G , where n is the order and m is the size of G (see [4]). One basically repeats a Breadth-First Search (BFS) algorithm from each node of a graph while tracking the cycles that are encountered. By imposing some restrictions on the input graph or relaxing the exactness of the solution, one can find a faster solution for the girth problem. For example, Itai and Rodeh [13] presented an $O(n^2)$ algorithm that finds a cycle which may have one edge more than the minimum. When a graph is restricted to be planar or has bounded genus, there is a linear time algorithm for the girth problem; see Djidjev [7]. Recently, Lingas and Lundell [16] presented a new approximation algorithm for the girth problem.

To our knowledge, this is the first study on parallel CPU and GPU implementations of the girth problem. One of our parallel algorithm uses parallel BFS, while the other algorithm is based on adjacency matrix multiplication. For each case we tailor our implementations to fit the hardware constraints (e.g. number and speed of processors; memory size and latency) of the selected platform.

1.3 Organization of the paper

The structure of this paper is as follows. In the next section, we introduce two parallel algorithms for computing the girth of a graph. Then in Section 3, we explain how those algorithms have been implemented by CUDA and OpenMP APIs. This section also describes a couple of potential optimizations. In Section 4, we describe the methods of generating four different sets of test graph data, specifically designed to strain our girth algorithms. A discussion on the results of testing our algorithms is the topic of Section 5. At the end of the paper we summarize our results and suggest topics for further study.

2 Two Parallel Algorithms

In this section we explain formally our parallel algorithms, including sample pseudo-code, for computing the girth.

Our first algorithm uses a slightly-modified parallel implementation of BFS, starting from each node. The algorithm (running in parallel from all roots) stops when the length of the first cycle is found. The approaches of detecting the girth in undirected or directed graphs are different.

1. For an undirected graph, a cycle is detected when a node in the frontier of the BFS has two parents already visited, or if it finds two nodes at the same distance (level) that are joined by an edge.
2. For a directed graph, a cycle containing the root is detected when the root node first appears in a frontier level of the BFS.

Figure 1 shows how an upper-bound of a smallest cycle is obtained via BFS. For undirected graphs we need to finish the current frontier for the two-parent case (left subfigure; even-length cycle); however, we can terminate immediately the search for the cross-edge case (middle subfigure; odd-length cycle). For directed graphs we can terminate the search when the root is first revisited (right subfigure; first cycle). The smallest upper-bound found over all BFSs is the actual girth of the graph and inter-process synchronization is needed to stop all parallel BFSs whenever the first cycle is found. See Algorithm 1.

Algorithm 1: Parallel girth algorithm via BFS.

Input: A Graph $G = (V, E)$

Output: The *girth* of G

$girth = |V| + 1;$

foreach node $v \in G$ *in parallel* **do**

 Run BFS algorithm rooted at v ;

 Let c be the length of first circuit detected;

$girth = \min(girth, c);$

Our second algorithm is based on doing repeated matrix multiplications of the adjacency matrix of a digraph. Let M be the adjacency matrix of a digraph G . It is well-known from graph theory, that the value of each entry $a_{i,j}$ of M^k represents the number of walks of length k from nodes i to j . Specially, the value on the diagonal entry $a_{i,i}$ shows the number of

directed circuits (closed walks) that start and end at i . This is easily adapted for our directed girth algorithm (Algorithm 2).

Algorithm 2: Girth via matrix multiplication.

Input: A Directed Graph $G = (V, E)$ as Adjacency Matrix M
Output: The *girth* of G
 $M^0 = I$;
 $M^1 = M$;
 $i = 1$;
while $\text{Trace}(M^i) = 0$ **do**
 Compute in parallel (binary matrix multiplication): $M^{i+1} = M^i \times M$;
 $i = i + 1$;
 $\text{girth} = i$;

To use this approach for undirected graphs, we need to adapt the aforementioned property of the powers of adjacency matrices to detect the smallest undirected cycle. First we need to ignore the circuits of length 2 (e.g. any edge (u, v) implies a circuit (u, v, u)). Secondly, note that any other smallest circuit of length at least 3 that we detected is, in fact, a cycle corresponding to the girth. Furthermore, we are only interested in knowing that the number of walks between i and j , $i \neq j$, is at least 2, so a possible optimization technique is to restrict to Boolean entries instead of integer entries.

Let the entry $b_{i,j}^k$ of matrix N^k denote the number of walks between i and j that do not traverse the same edge consecutively; clearly $b_{i,j}^k \leq a_{i,j}^k$ where $a_{i,j}^k$ is the entry of M^k . We can calculate N^k from matrices N^{k-1} , N^{k-2} , and $N^1 = M^1$, using a simple recurrence (modified vector products with respect to N^{k-2} where a row of N^{k-1} times a column of M yield an entry of N^k).

$$b_{i,j}^k = \bigvee_{\forall s : a_{s,j}^1 = 1} b_{i,s}^{k-1} \wedge \overline{b_{i,s}^{k-2}}$$

We parallelize by data partitioning the output rows of the matrix N^k ; rows assigned to the available processors. The undirected graph version of Algorithm 2 also uses the two-path idea as illustrated in Figure 1, where we stop computing when k is half of the actual girth. The process is to first test if the following odd-length cycle condition is met:

$$\exists \{r, u, v\} : a_{u,v}^1 \wedge b_{r,u}^k \wedge b_{r,v}^k$$

Then (if the previous condition is not met) test if the following even-length cycle condition is satisfied:

$$\exists \{r, u, v, w\} : a_{u,w}^1 \wedge a_{v,w}^1 \wedge b_{r,u}^k \wedge b_{r,v}^k$$

Note that all the values of the existential variables are distinct and both these conditions may be tested during the generation using the recurrence for N^{k+1} . Thus, the seemingly extra intra-level detection time of $O(n^3)$ is not required.

We end this section by mentioning the expected running times of our two algorithms for *sparse* graphs—those graphs with $m = O(n)$ edges. Sparse

```
int undirectedGirth(const Graph &G)
{
    int n = G.order();
    int level[n];
    int smallest = n+1; // value for infinity

    for (int r=0; r<n-2; r++) // minimum is 3-cycle
    {
        fill(level, level+n, -1); // unseen flags as -1
        level[r]=0;

        queue<int> toGrow; // sequential FIFO queue
        toGrow.push(r);

        while ( !toGrow.empty() )
        {
            int grow = toGrow.front(); toGrow.pop();

            // try next r if this BFS is too deep
            if ( level[grow]*2+1 >= smallest ) break;

            const vector<int> nbrs = G.neighbors(grow);
            for (int i=0; i<nbrs.size(); i++)
            {
                int u = nbrs[i];
                if ( u < r ) continue; // optimization

                if ( level[u] < 0 )
                {
                    level[u]=level[grow]+1; // now seen
                    toGrow.push(u);
                }
                else if ( level[u]==level[grow] )
                {
                    if ( level[u]*2+1 < smallest )
                        smallest=level[u]*2+1;
                    break; // try next r
                }
                else if ( level[u]==level[grow]+1 )
                {
                    if ( level[u]*2 < smallest )
                        smallest = level[u]*2;
                }
            }
        } // while BFS queue not empty
    }
    return smallest;
}
```

Figure 2: Sequential C++ girth function.

graphs are usually those graphs with large girth and will be prominent in the test cases for our algorithms. Also note that sparse $n \times n$ matrix multiplication can be done in time $O(n^2)$ if one uses the appropriate data representation [1, 6].

Theorem 1. *Given a sparse input graph G , Algorithm 1 runs on a machine with p processors in time $O(n^2/p)$.*

Theorem 2. *Given a sparse input graph G , Algorithm 2 runs on a machine with p processors in time $O(gn^2/p)$, where g is the girth of G .*

Note that the girth g is usually much smaller than the order n . Thus, both algorithms may be more practical than the other for different types of input cases.

3 Parallel Implementations

As noted in Section 2, BFS and adjacency matrix multiplication are the building blocks of our parallel

girth algorithms. Each program is a modified version of one of those basic algorithms. The program names that we introduce in this section correspond to the column headings of our final timing results of Tables 1 and 2 (at the end of the paper). Note the suffix `_p` on a program name denotes a “preprocessed” version, which is explained at the end of this section.

3.1 Cuda BFS program

In our programs `cuda_BFS` and `cudaBFS_p` we assign one CUDA *block* of threads (also known as *workgroup* in OpenCL) to each node. Each block is responsible to run one BFS in parallel using inter-block shared memory in addition to the GPU global memory. Furthermore, different blocks will run in parallel. Since our device memory size is limited, for large graphs we have to divide the nodes into separate groups.

The following proposition gives an optimization to save memory when searching for the girth in sparse undirected graphs.

Proposition 3. *When implementing the BFS-based algorithm for computing the undirected girth, it only needs to remember the nodes visited in the last three levels.*

Proof. Suppose we are exploring neighbors at level $i \geq 1$, where the root node is at level 0. By definition, the frontier level $i+1$ should only contain nodes not placed at any level $0 \leq j \leq i$, as depicted in Figure 1. Furthermore, a node x at level i can not have a neighbor y at distance $j < i-1$ from the root since that would imply x should be at level $j+1 < i$. Thus, we only need to remember nodes at levels $i-1$, i , and $i+1$ when doing the BFS search. \square

Our CUDA programs implement Algorithm 1 by using three arrays to represent the last three levels of the BFS search tree: one for parent (of the grow) level, one for the grow level, and one for the frontier level. The algorithm initializes the arrays with the root node as the only item in the parent level and the neighbors of the root as the grow level. The frontier level is processed by finding the neighbors of the elements of the grow level that are not already in either the parent or grow levels.

All threads are intra-block synchronized at the end of each level. At this time, we change the roles of the arrays by doing pointer exchanges to save time by not having to copy the grow level to the new parent level and the frontier level to the new grow level. The old parent level is emptied and becomes the target for the new frontier. As soon as the first cycle is detected by one thread, its length is compared (and atomic exchanged) with the length of the shortest-known cycle that is stored in shared memory.

We also save the value of the shortest cycle from shared memory to global memory (inter-block communication) for determining the minimization of all BFS searches. This allows for early termination of deep BFS trees and reduces the overall running time of the implementation.

3.2 Cuda matrix-based algorithms

As explained earlier in Section 2, the programs `cudaMAT` and `cudaMAT_p` use the powers of the adjacency matrix for finding a smallest cycle originating from any node. We have only one thread being assigned sole ownership in writing the values of the i -th row of the output matrix of paths of length k . Since, for large graphs, we have more rows than the total number of available threads. Thus, we assign a contiguous group of rows to a particular thread. In our implementation, we have to do block synchronization (unlike our BFS implementation). This is done by repeated kernel launches, as illustrated in the following CUDA snippet.

```
for (int dist=1; dist <= order/2; dist++)
{
    Girth_Kernel<<<NUM_BLOCKS, NUM_THREADS>>>
        (graph, girth_D, DistMAT, DistLens, dist);

    cudaMemcpy(&girth_H, girth_D, sizeof(int),
               cudaMemcpyDeviceToHost);
    if (girth_H <= order) break;
}
```

3.3 OpenMP implementations

Our OpenMP implementations (`ompBFS`, `openBFS_p`, `openMAT` and `openMAT_p`) of the algorithms follow the same logic as explained for CUDA implementations. These were developed by adding `#pragma omp` directives to our sequential C++ code (e.g. add one above the first C++ `for` loop of Figure 2).

3.4 Final implementation remarks

In some of the implementations (denoted with a suffix `_p`), we first apply a preprocessing procedure to eliminate all nodes that are not clearly involved in any cycle of the graph. In other words, we iteratively delete all nodes of degree at most one. For the digraph input cases, we iteratively delete all sinks and sources. Note that the preprocessing times are included in the reported computational elapsed times.

The speed-up of this procedure is the result of reducing the order of the input graph. If the graph is not reduced significantly, then the preprocessing procedure may increase the running time. To explicitly display the impact of this procedure, we use it for all our parallel implementations.

The algorithms for finding girth in directed graphs are implemented by applying proper changes to the algorithms for the undirected ones. In `d_cudaBFS` and `d_ompBFS` we discover the directed cycles as soon as a back edge to the root is detected (see Figure 1).

In `d_cudaMAT` and `d_ompMAT` we use the parallel version of adjacency matrix multiplication as explained in Algorithm 2.

While processing graphs, we ignore those nodes that have larger index than the current root. This helps shrink the search space with no change in the correctness of the algorithm. Consider a cycle C of shortest length. If we start a BFS at the node with smallest index of C we will detect this cycle since no nodes of C are ignored. This pruning method is applied for both BFS and adjacency matrix multiplication approaches.

4 Generating Girth Test Data

The standard random graph generators are not desirable because they either produce graphs with no cycles or very small girth (e.g. dense graphs). To test the performance and correctness of our algorithms, we produced several classes of graphs of large order and large girth.

We have four test suites for undirected graphs:

big cycles We construct random trees in which each node is replaced by a big cycle. Then we choose one node from each cycle that represent two neighboring nodes and connect them by an edge.

Cayley graphs Let S be a set of generator for a finite group (H, \cdot) . The nodes of a Cayley graph is the set H and $S \subseteq H$ is used to define edges. We connect a node h to a node h' if there is an element $s \in S$ such that $h' = h \cdot s$. The graph is undirected if S is closed under inverses (e.g. $s \in S$ implies $s^{-1} \in S$). We used the semi-direct product procedure given in [5] to generate large sparse graphs.

cycle graphs These graphs are generated by connecting a sequence of large cycles on a path and adding a few extra edges randomly. These extra edges may span the length of the connected cycles or may be a chord of one cycle.

sparse graphs These graphs are produced by taking random trees, generated by using Prüfer codes, and then randomly connecting pairs of nodes.

We also have three test suites for directed graphs:

directed big cycles These are generated using the same procedure as **big cycles** but with each cycle being a directed cycle.

cycle digraphs Each of these digraphs was created by generating a union of large random directed cycles.

sparse digraphs These digraphs are created by first generating a rooted random tree (all arcs directed from parent to children). Then several random directed edges are added from a descendant node to an ancestor to form directed cycles.

Each test suite¹ consists of eight subsets of [di]graphs, indexed from 0 to 7. Each subset, labeled by i , consists of 25 [di]graphs with the number of nodes ranging between $2^i \cdot 1000$ and $2^{i+1} \cdot 1000$. So the overall range of our test graphs varies from graphs with 1000 nodes up to graphs with 256000 nodes.

5 Comparative Study

We implemented our parallel algorithms using C++ (gcc 4.4) with the two APIs: CUDA 3.2 and OpenMP 3.0. To run our CUDA programs, we used an Nvidia Tesla C2050 series (Fermi class) graphics card. The C2050 has Nvidia compute capability 2.0 and consists of 14 multiprocessors (MPs). Each MP

has 32 cores and 3Gb cache (global memory). Each of the 448 cores operates at 1.15 GHz frequency. For our graphics card, each block (of threads) supports up to 1024 threads. For running our OpenMP programs, we used two hyper-threaded quad-core 2.5 GHz Intel CPUs which provides at least 8 and up to 16 independent Pthreads. Due to the hardware available to us, we are restricted to using a smaller number of OpenMP threads compared to what our GPU device has. However, one benefit of using OpenMP over CUDA is that the memory available is larger (48Gb vs 3Gb DRAM) and much faster (data transfer rate).

We provide in Table 1 and Table 2 a summary of our programs. These tables contain the average running times for each of the 25 graphs per subset of a test suite, then the average of all 200 graphs in each test suite, and finally the overall average running times. These times are wall clock times in seconds. For the CUDA implementation we do not include the I/O time for loading the graphs into device memory. We also do not include any disk I/O time for any program.

To have a better evaluation of our algorithms, we also use two sequential algorithms for computing the girth of a graph. One of them is the Sage’s (Mathematics Software²) algorithm for finding the girth of undirected graphs [21]. Our other sequential program, **girthseq**, for undirected graphs use the BFS algorithm that was presented earlier (and listed in Figure 2). We also have a similar C++ implementation, **d_girthseq**, for directed graphs.

In general, as expected, the overall average performance (the last row in the two tables) of the sequential algorithms (**girthsage** and **girthseq** in Table 1 and **d_girthseq** in Table 2) are much slower than our parallel implementations.

Our two parallel algorithms running on OpenMP (**ompBFS** and **ompMAT**) perform about the same, which is about eight times faster than **girthseq**. On the other hand, our two parallel implementations running on the GPU (**cudaBFS** and **cudaMAT**) have performances that vary for undirected graphs and directed graphs. Our **cudaBFS** has the best overall performance for undirected graphs (18.6 times speed-up), and **cudaMAT** has the best overall performance for directed graphs (31.5 times speed-up).

Even though CUDA programs have the best overall performance in both undirected and directed graphs, the OpenMP still have advantages for solving small graphs. More specifically, OpenMP programs always outperform on the smaller graphs (subset 0) in the four test suites of undirected graphs and (subsets 0–2) in the three test suites of directed graphs. When the graph orders increase, the CUDA programs show their advantages.

For both CUDA and OpenMP, we find that the pre-processed versions are faster for the graphs in **sparse_graphs**, but increases the computation time for all other test classes. This is expected because only the class of **sparse_graphs** contains many nodes that are not on any cycle. We note that for digraphs, we could not gain better performance with our chosen

¹These test suites are available by request.

²Note we selected this open-source platform for our base-line benchmark since it seems to out-perform our commercial software such as Mathematica 7.

preprocessing implementations.

6 Conclusions and Future Work

In conclusion, both OpenMP and CUDA based parallel programs improve the computation time of detecting the girth in undirected and directed graphs for our extensive test data. For small graphs/digraphs OpenMP seems to be faster (can't exploit multiple threads) than larger graphs. Both algorithm design approaches and both implementation APIs are valuable.

We note that the amount of human effort for CUDA is clearly expensive—we are waiting for higher-level programming tools (like OpenMP but for GPUs).

For the future we would like to try C# Parallel Task Library, CUDA 4.0 Thrust Library and new C++ Patterns Library (PPL). Also, we want to try other parallel hardware and possible different graph test cases for the girth problem. Also, we would like to consider performing performance evaluations on emerging, virtualized computing models (cloud resources) such as Amazon EC2 or Google AppEngine.

Acknowledgements

The authors would like to thank the University of Auckland for support in an FRDF grant 9843/3626216 for providing the necessary hardware and a Faculty of Science PhD research stipend for the second author.

References

- [1] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39:85–97, March 1996.
- [2] Shashi Kiram Chilappagari, Dung Viet Nguyen, Bane Vasić, and Michael W. Marcellin. Girth of the Tanner graph and error correction capability of LDPC codes. In *Communication, Control, and Computing, 2008 46th Annual Allerton Conference on*, pages 1238–1245, September 2008.
- [3] Francis Y. Chin, John Lam, and I-Ngo Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25:659–665, September 1982.
- [4] Michael J. Dinneen, Georgy Gimel'farb, and Mark C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages, 2nd Edition*. Pearson (Education New Zealand), 2009. ISBN 978-1-4425-1206-1 (pages 264).
- [5] Michael J. Dinneen and Paul R. Hafner. New results for the degree/diameter problem. *Networks*, 24:359–367, October 1994.
- [6] Michael J. Dinneen, Masoud Khosravani, and Andrew Probert. Using OpenCL for implementing simple parallel graph algorithms. In Hamid R. Arabnia, editor, *Proceedings of the 17th annual conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11), part of WORLDCOMP'11*, pages 1–6, Las Vegas, Nevada, July 18–21 2011. CSREA Press.
- [7] Hristo Djidjev. A faster algorithm for computing the girth of planar and bounded genus graphs. *ACM Transactions on Algorithms*, 7(1):3, 2010.
- [8] Geoffrey Exoo and Robert Jajcay. On the girth of voltage graph lifts. *European Journal of Combinatorics*, 32:554–562, May 2011.
- [9] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2 edition, January 2003.
- [10] Pawan Harish and P.J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor Prasanna, editors, *High Performance Computing HiPC 2007*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer Berlin / Heidelberg, 2007.
- [11] Kenneth A. Hawick, Arno Leist, and Daniel P. Playne. Parallel graph component labelling with GPUs and CUDA. *Parallel Computing*, 36(12):655 – 678, 2010.
- [12] Xiao-Yu Hu, Evangelos Eleftheriou, and Dieter-Michael Arnold. Regular and irregular progressive edge-growth Tanner graphs. *IEEE Transactions on Information Theory*, 51(1):386–398, 2005.
- [13] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7(4):413–423, 1978.
- [14] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH / EUROGRAPHICS Symposium on Graphics Hardware*, GH'08, pages 47–55. Eurographics Association, 2008.
- [15] Sunghwan Kim, Jong-Seon No, Habong Chung, and Dong-Joon Shin. Quasi-cyclic low-density parity-check codes with girth larger than 12. *IEEE Transactions on Information Theory*, 53(8):2885–2891, 2007.
- [16] Andrzej Lingas and Eva-Marta Lundell. Efficient approximation algorithms for shortest cycles in undirected graphs. *Information Processing Letters*, 109(10):493–498, 2009.
- [17] OpenMP. The OpenMP API specification for parallel programming, site visited 2011. <http://openmp.org>.
- [18] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Survey*, 16:319–348, September 1984.
- [19] V. Nageshwara Rao and Vipin Kumar. Parallel depth first search. Part I. Implementation. *International Journal of Parallel Programming*, 16:479–499, 1987.
- [20] Jyothish Soman, Kothapalli Kishore, and P.J. Narayanan. A fast GPU algorithm for graph connectivity. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.
- [21] William Stein. *Sage: Open Source Mathematical Software (Version 4.6)*. The Sage Group, October 2010. <http://www.sagemath.org>.

Table 1: Timings in seconds of girth algorithms on undirected graphs.

	Subset	girth_sage	girthseq	cudaBFS	cudaBFS_p	cudaMAT	cudaMAT_p	ompBFS	ompBFS_p	ompMAT	ompMAT_p
big cycles	0	0.0212	0.0018	0.0006	0.0007	0.0048	0.0043	0.0003	0.0003	0.0012	0.0023
	1	0.1312	0.0089	0.0017	0.0019	0.0193	0.0141	0.0010	0.0010	0.0057	0.0044
	2	0.5884	0.0373	0.0061	0.0065	0.0488	0.0387	0.0036	0.0036	0.0219	0.0196
	3	2.2220	0.1516	0.0218	0.0251	0.1202	0.0895	0.0147	0.0144	0.0770	0.0738
	4	11.6868	0.7531	0.1144	0.1203	0.3557	0.2623	0.0710	0.0710	0.3720	0.3727
	5	49.4324	3.2966	0.4731	0.4977	0.9750	0.7130	0.3387	0.3593	1.6402	1.6643
	6	174.0984	12.9000	1.6726	1.7588	3.5142	2.4929	1.2805	1.3231	5.4710	5.7810
	7	504.0968	38.1329	4.8217	5.0730	9.8500	6.9969	3.7729	3.8581	17.4768	18.8116
	Average	92.7847	6.9103	0.8890	0.9355	1.8610	1.3265	0.6853	0.7039	3.1332	3.3412
Cayley graphs	0	0.0596	0.0038	0.0036	0.0015	0.0352	0.0343	0.0006	0.0007	0.0031	0.0023
	1	0.1724	0.0133	0.0033	0.0036	0.1237	0.1040	0.0019	0.0020	0.0104	0.0084
	2	0.5064	0.0430	0.0109	0.0115	0.6081	0.5148	0.0060	0.0062	0.0529	0.0561
	3	1.1616	0.1668	0.0263	0.0292	1.1158	0.9550	0.0223	0.0218	0.1280	0.1220
	4	4.8736	0.6849	0.1209	0.1221	4.2231	3.6082	0.1106	0.1518	0.5886	0.6021
	5	9.2036	2.4132	0.2133	0.2172	5.0933	4.3397	0.3369	0.3764	0.9630	0.9246
	6	7.1400	4.9858	0.1338	0.1407	2.2464	1.9095	0.6583	0.7053	0.4896	0.4715
	7	12.7224	19.7854	0.1740	0.1836	1.4997	1.1844	2.6333	2.6962	0.6287	0.5478
	Average	4.4800	3.5120	0.0858	0.0887	1.8682	1.5812	0.4712	0.4951	0.3580	0.3418
cycle graphs	0	0.0212	0.0016	0.0007	0.0009	0.0027	0.0026	0.0003	0.0029	0.0012	0.0003
	1	0.0588	0.0066	0.0012	0.0035	0.0027	0.0026	0.0008	0.0046	0.0022	0.0049
	2	0.2236	0.0316	0.0034	0.0080	0.0039	0.0039	0.0031	0.0079	0.0056	0.0290
	3	0.5428	0.1294	0.0068	0.0095	0.0071	0.0050	0.0132	0.0186	0.0200	0.0183
	4	1.1220	0.4681	0.0133	0.0142	0.0056	0.0060	0.0494	0.0577	0.0366	0.0307
	5	2.0728	1.9861	0.0240	0.0255	0.0077	0.0084	0.2429	0.3891	0.1281	0.2356
	6	4.7428	7.9299	0.0533	0.0562	0.0161	0.0182	1.0435	1.2081	0.1983	0.2992
	7	7.7152	31.1896	0.0878	0.0905	0.0272	0.0314	4.2280	4.3701	0.3352	0.4226
	Average	2.0624	5.2179	0.0238	0.0260	0.0091	0.0098	0.6977	0.7574	0.0909	0.1301
sparse graphs	0	0.0096	0.0011	0.0015	0.0008	0.0175	0.0018	0.0003	0.0004	0.0009	0.0003
	1	0.0520	0.0064	0.0024	0.0010	0.0565	0.0041	0.0011	0.0011	0.0029	0.0010
	2	0.2084	0.0321	0.0027	0.0014	0.0571	0.0079	0.0048	0.0036	0.0102	0.0028
	3	0.6536	0.1291	0.0076	0.0025	0.2135	0.0132	0.0196	0.0124	0.0404	0.0066
	4	1.9240	0.5460	0.0207	0.0051	0.7572	0.0249	0.0848	0.0526	0.1384	0.0195
	5	5.6632	2.0746	0.0663	0.0102	2.3316	0.0435	0.2965	0.3819	0.5345	0.1590
	6	10.5944	6.4405	0.1040	0.0201	2.8499	0.0694	0.9124	0.9687	0.9201	0.2733
	7	34.1212	23.4932	0.4045	0.0382	17.3415	0.1912	3.4374	3.0170	3.4024	0.4003
	Average	6.6533	4.0904	0.0762	0.0099	2.9531	0.0445	0.5946	0.5547	0.6312	0.1078
AVERAGE		26.4951	4.9326	0.2687	0.2650	1.6728	0.7405	0.6122	0.6277	1.0533	0.9802

Table 2: Timings in seconds of girth algorithms on directed graphs.

	Subset	d_girthseq	d_cudaBFS	d_cudaMAT	d_ompBFS	d_ompBFS_p	d_ompMAT
directed big cycles	0	0.0016	0.0268	0.0068	0.0003	0.0005	0.0008
	1	0.0082	0.4085	0.0121	0.0008	0.0012	0.0051
	2	0.0372	0.5955	0.0274	0.0033	0.0042	0.0247
	3	0.1682	0.9255	0.0641	0.0143	0.0163	0.1168
	4	0.7007	0.2859	0.1419	0.0616	0.0662	0.4696
	5	2.7614	0.6990	0.2931	0.2471	0.2596	1.6903
	6	9.8746	1.5924	0.6623	0.9421	1.0185	5.5603
	7	41.9673	5.7967	2.1259	4.0266	4.2226	25.0513
	Average	6.9399	1.2913	0.4167	0.6620	0.6986	4.1149
cycle digraphs	0	0.0024	0.0086	0.0638	0.0004	0.0017	0.0061
	1	0.0076	0.0075	0.0934	0.0010	0.0030	0.0040
	2	0.0252	0.0025	0.0135	0.0029	0.0059	0.0015
	3	0.1074	0.0016	0.0030	0.0122	0.0158	0.0016
	4	0.4542	0.0030	0.0053	0.0505	0.0560	0.0033
	5	1.5451	0.0035	0.0025	0.1654	0.1769	0.0054
	6	7.4248	0.0083	0.0074	0.9728	1.0582	0.0465
	7	26.8069	0.0105	0.0064	3.5560	3.7332	0.1559
	Average	4.5467	0.0057	0.0244	0.5952	0.6313	0.0280
sparse digraphs	0	0.0013	0.5340	0.0025	0.0003	0.0008	0.0003
	1	0.0062	0.1696	0.0042	0.0009	0.0018	0.0007
	2	0.0287	0.0389	0.0084	0.0034	0.0054	0.0018
	3	0.1181	0.0193	0.0083	0.0136	0.0508	0.0038
	4	0.4792	0.0139	0.0164	0.0548	0.0630	0.0116
	5	1.7474	0.1223	0.0111	0.1933	0.2058	0.0859
	6	6.6283	0.0343	0.1097	0.8764	1.0621	0.2391
	7	19.8751	0.1582	0.1472	2.6206	2.8045	0.3118
	Average	3.6105	0.1363	0.0385	0.4704	0.5243	0.0819
AVERAGE		5.0324	0.4778	0.1599	0.5759	0.6181	1.4083

The Use of Fast Approximate Graph Coloring to Enhance Exact Parallel Algorithm Performance

John D. Eblen¹, Gary L. Rogers Jr.², Charles A. Phillips³ and Michael A. Langston³

¹ Center for Molecular Biophysics
Oak Ridge National Laboratory
Oak Ridge TN 37831 USA

² National Institute for Computational Sciences
University of Tennessee
Knoxville TN 37996 USA

³ Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville TN 37996 USA

(Extended Abstract)

Abstract

The significance of graph coloring is considered in the context of reducing the running time of a parallel branch and bound algorithm to solve the maximum clique problem. The greedy color preprocessing algorithm produces an upper bound u on the color degree c of a vertex v . The color degree of a vertex is defined to be the chromatic number, γ , of the neighborhood subgraph of vertex v . The graph instance is reduced by removing any vertex v , such that $u < k$, where k is the size of the largest known clique. The use of this graph coloring is extended and used in the interleaved preprocessing step during the branching phase of the algorithm. The basic techniques introduced can be extended to other problems such as minimum vertex cover and maximum independent set. Finally, results are presented from experiments using real biological data.

1 Introduction

Given a graph $G = \langle V, E \rangle$, the maximum clique problem asks what is the largest subset $C \subseteq V$ such that that every pair $\{u, v\} \in C$, then $\{u, v\} \in E$. Finding exact solutions to the maximum clique problem, *MCP*, has been extensively studied (Bomze et al. 1999, Pardalos et al. 1998). Exact solutions to *MCP* impacts many disciplines such as bioinformatics, image processing, and design of quantum

circuits (Tomita & Kameda 2007). Therefore, it is imperative to create efficient parallel maximum clique algorithms. Like many parallel algorithms, the basis for the parallel algorithm presented here is the serial algorithm Maximum Clique Finder, *MCF*. *MCF* was first introduced in (Eblen 2010) and is a derivative of research on the vertex cover problem, *VCP*, presented in (Abu-khzam et al. 2006). It is well known that *VCP* is fixed-parameter tractable, *FPT*, and that for any *FPT* problem, there exists a problem kernel, which is a reduced instance of the original problem. Using this knowledge, many of the kernelization methods that are applicable to *VCP* have been translated into comparable methods for *MCP*. These kernelization methods are referred to as the preprocessing rules and reduces the graph instance on which *MCF* is applied.

The basic preprocessing rules found in *MCF* are based strictly on vertex degrees. These rules include the (n-1)-degree rule, (n-2)-degree rule, and the low-degree rule. The (n-1)-degree rule automatically includes any vertex that has a degree of (n-1) to the clique, as it is connected to all other vertices in the graph. The (n-2)-degree rule includes any vertex v that is connected to all other vertices except vertex u . Vertex v is placed in the clique, while excluding vertex u from the clique. Finally, the low-degree rule removes any vertex that has degree less than $k - 1$, where k is the size of the largest known clique.

Graph coloring has been used to find an upper bound on the size of a maximal clique (Tomita & Kameda 2007, Bomze et al. 1999, Ostergard 2002). As exact solutions to the graph coloring problem can be time consuming, approximate colorings have been used. In (Tomita & Kameda 2007), an algorithm is presented that incorporates approximate graph coloring, however, the coloring is employed in the branching stage rather than preprocessing stage. Exploiting graph coloring in preprocessing, along with interleaving the coloring during the branching phase, decreases the overall runtime of the algorithm, as the search tree can be pruned to retain only the vertices that can generate a larger clique.

This research has been funded by the U.S. Department of Energy under the EPSCoR Laboratory Partnership Program. It has also been supported by an allocation of advanced computing resources provided by the U.S. National Science Foundation. Computations were performed on Kraken, a Cray XT5 housed at the National Institute for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA.

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 10th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 127, Jinjun Chen and Rajiv Ranjan, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

2 Parallel Maximum Clique Finder

The Parallel Maximum Clique Finder, *PMCF*, is a parallel implementation based on *MCF*. The parallelization of the algorithm is exploited in the branching phase. One common pitfall of parallel algorithms that are ported from serial algorithms is the lack of proper load balancing. This is typically due to the fact that static load balancing only works well when problems are embarrassingly parallel. Therefore, it is imperative to have a dynamic load balancing approach that continuously monitors the workload of the nodes and assigns jobs as needed (Weerapurage et al. 2011). *PMCF* uses a simple dynamic load balancing technique that only distributes jobs to worker nodes if a certain level of the search tree is reached and only distributes jobs that are sufficiently small. *MCF* handles the distribution of sufficiently small jobs by selecting candidate vertices in ascending order by degree.

3 Parallel Maximum Clique Finder With Coloring

Parallel Maximum Clique Finder with Coloring, *PMCFC*, incorporates approximate coloring of graphs with the *PMCF*. *PMCFC* uses a greedy approximate coloring heuristic in two stages. The first stage is the preprocessing stage of the initial graph. Along with other preprocessing rules, such as the low-degree rule, the color degree rule produces an upper bound on the size of the maximum clique and any vertex v that has a color degree less than $k - 1$ is removed from the graph. The second stage that exploits graph coloring is interleaved preprocessing. As the branching phase traverses the search tree, vertices are continually excluded from the search space. Interleaving the preprocessing step further reduces the number of vertices that can be excluded from the search space. In practice, the interleaved preprocessing stage has a large impact in the overall running time of the algorithm.

4 Experimental Results

In order to demonstrate the effect of graph coloring on *PMCF*, experiments were conducted on graphs derived from real biological data (Eblen 2010). The resulting graph is comprised of 17,338 vertices and 10,406,565 edges. The timings were completed on Kraken, the world's fastest academic supercomputer. Each node contains two 2.6 GHz six-core AMD Opteron processors (12 cores total) with 16 GB of memory. Figure 1 summarizes run times for a modest number of cores. The interleaved graph coloring steps have a significant impact in the overall runtime of the *PMCF* algorithm.

5 Conclusions and Direction for Future Research

The *PMCFC* algorithm uses a parallel framework that was derived from the serial algorithm, *MCF*. *PMCFC* exploits a number of strategies necessary for a parallel algorithm to be efficient, such as dynamic load balancing. This algorithm uses a simple dynamic load balancing algorithm that strives to keep the amount of overhead to a minimum. More complex and adaptive load balancing techniques, however, may increase the efficiency of the worker nodes. *PMCFC* also uses

a greedy graph coloring algorithm to generate an approximate coloring for the initial graph, as well as approximate colorings for neighborhood subgraphs for a vertex v . *PMCFC* prunes the search tree of vertices that would otherwise not be identified by other preprocessing methods such as the low-degree rule. While the current greedy coloring algorithm has shown positive results, it is possible that other approximate coloring algorithms could result in better overall algorithm performance. Improving these two portions of the *PMCFC* algorithm, however, is no trivial task. A balance must be struck between the time it takes to generate a load balancing scheme or a graph coloring and the overall time that is saved by having a better load balance or a more accurate coloring.

References

- Abu-khzam, F. N., Langston, M. A., Shanbhag, P. & Symons, C. T. (2006), 'Scalable parallel algorithms for FPT problems', *Algorithmica* **45**, 269–284. 1
- Bomze, I. M., Budinich, M., Pardalos, P. M. & Pelillo, M. (1999), The maximum clique problem, in 'Handbook of Combinatorial Optimization'. 1
- Eblen, J. D. (2010), The Maximum Clique Problem: Algorithms, Applications, and Implementations, PhD thesis, University of Tennessee. 1, 2
- Ostergard, P. R. (2002), 'A fast algorithm for the maximum clique problem', *Discrete Applied Mathematics* **120**(1-3), 197 – 207. Special Issue devoted to the 6th Twente Workshop on Graphs and Combinatorial Optimization. 1
- Pardalos, P. M., Rappe, J., Mauricio & Resende, M. G. (1998), An exact parallel algorithm for the maximum clique problem, in 'In High Performance and Software in Nonlinear Optimization', pp. 279–300. 1
- Tomita, E. & Kameda, T. (2007), 'An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments', *J. of Global Optimization* **37**, 95–111. 1
- Weerapurage, D. P., Eblen, J. D., Rogers, G. L. & Langston, M. A. (2011), Parallel vertex cover: A case study in dynamic load balancing, in 'Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011)', Vol. 118, pp. 25–32. 2

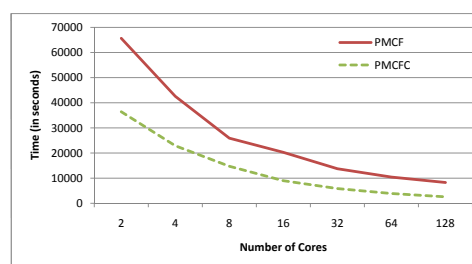


Figure 1: Both algorithms achieve speedup as the number of cores are increased within the range of this sample study. But clique finding is much faster with color preprocessing, and color preprocessing scales considerably better.

Managing Large Numbers of Business Processes with Cloud Workflow Systems

Xiao Liu¹, Yun Yang¹, Dahai Cao¹, Dong Yuan¹, Jinjun Chen^{2,1}

¹Faculty of Information and Communication Technologies
Swinburne University of Technology, Melbourne, Australia

²Faculty of Engineering and Information Technology
University of Technology Sydney, Sydney, Australia

{xliu, yyang, dcao, dyuan}@swin.edu.au, jijun.chen@uts.edu.au

Abstract

With the emergence of cloud computing which can deliver on-demand high-performance computing resources over the Internet, cloud workflow systems offer a competitive software solution for managing large numbers of business processes. In this paper, we first analyse the basic system requirements through a motivating example, and then, the general design of a cloud workflow system is proposed with the focus on its system architecture, functionalities and QoS (quality of service) management. Afterwards, the system implementation of a peer-to-peer based prototype cloud workflow system is demonstrated to verify our design. Finally, experimental results show that with the dynamic resource provisioning, conventional violation handling strategies such as workflow local rescheduling can ensure the on-time completion of large numbers of business processes in a more cost-effective way.

Keywords: Business Process Management, Workflow System, Cloud Computing, Cloud Workflow System

1 Introduction

With the rapid development of e-business and e-government in the global economy, both enterprises and government agencies are facing large numbers of concurrent business processes from the private and public sectors [1, 21]. For examples, a federal government taxation office receive millions of tax declaration requests at the beginning and end of the tax return period each year; a banking enterprise often needs to process millions of transactions including cheques everyday; and an insurance company may need to process over thousands of claims on a daily basis which may peak by a factor of tens or hundreds when some natural disasters happen, e.g. the Melbourne hailstorm in March 2010 results in 79,000 claims which worth A\$491 million¹. Failure of completing these process instances in time is not acceptable and will often results in significant loss. For example, the Australian federal government taxation office has to pay a large amount of interest to tax payers for the delay; the

time delays in stock exchange may result in significant loss to both sellers and buyers in the stock market.

For time constrained business processes, software performance (e.g. response time and throughput), as one of the basic dimensions of software quality, is very important [24]. To ensure satisfactory performance, enterprises and government agencies often need to invest a huge amount of money on their self-owned and self-maintained IT (Information Technology) infrastructures which are normally designed to have the capability to meet either the maximum or at least the average needs of computing resources. However, for the option to meet the average needs, the software performance during peak time can be significantly deteriorated. As for the option to meet the maximum needs, since the number of process instances during peak time can often be much larger than the average, such a design will often result in largely idle of computing resources, which means a huge waste of financial investment and energy consumption. In general, the running of larger numbers of business processes usually require powerful, on-demand and elastic computing resources. Specifically, the basic system requirements for business software can include: 1) scalable computing resource provision; 2) elastic computing resource delivery; 3) efficient process management and 4) effective QoS (quality of service) monitoring and control. Detailed analysis will be presented in Section 2.

Cloud computing, an exciting and promising new computing paradigm, can play an important role in this regard. In late 2007, the concept of cloud computing was proposed. Cloud computing, nowadays widely considered as the “next generation” of IT, is a new paradigm offering virtually unlimited, cheap, readily available, “utility type” scalable computing resources as services via the Internet [4, 6]. As very high network bandwidth becomes available, it is possible to envisage all the resources needed to accomplish IT functions as residing on the Internet rather than physically existing on the clients’ premises. With effective facilitation of cloud computing, many sophisticated software applications can be further advanced to stretch their limits and yet with reduced running costs and energy consumption. The advantages of cloud computing, especially its utility computing and SaaS (software as a service), enable entirely new innovations to the design and development of software applications [1, 18]. It is generally agreed among many researchers and practitioners that cloud applications are the future trend for business software applications since utility computing can provide unlimited on-demand and elastic computing

Copyright 2012, Australian Computer Society, Inc. This paper appeared at the 10th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 127. J. Chen and R. Ranjan, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

¹<http://www.theage.com.au/national/melbourne-storm-shaping-up-as-on-e-of-australias-costliest-20100320-qnah.html>

power while SaaS can provide massive software services with different capabilities [5]. Typical successful stories include NewYork Times which turns 11 million archived articles into pdf files in only one day costing \$240 by using Hadoop and computing power on Amazon's cloud²; Animoto employs Amazon's cloud to deal with nearly 750,000 new registered clients in three days and 25,000 people online at peak time³.

Workflow systems, with the benefits of efficient and flexible process modelling and process automation, have been widely used for managing business processes [2, 19]. Given the recent and rapid growth of cloud computing, we can envisage that cloud computing based workflow systems, or cloud workflow systems for short, can be a suitable solution for managing large numbers of business processes. Hence, the design of a cloud workflow system deserves systematic investigation. In this paper, we first employ a securities exchange business process as a motivating example to analyse the system requirements. Based on that, we propose the general design of a cloud workflow system with the focus on its system architecture, basic functionalities and QoS management. Afterwards, the system implementation of our SwinDeW-C prototype cloud workflow system is demonstrated. Finally, simulation experiments evaluate the effectiveness of SwinDeW-C in the running of large numbers of time-constrained business processes.

The remainder of this paper is organised as follows. Section 2 presents a motivating example and system requirements. Section 3 proposes the design of a novel cloud workflow system. Section 4 describes the prototype. Section 5 demonstrates the evaluation results. Section 6 introduces some related work. Finally, Section 7 addresses the conclusion and points out the future work.

2 Motivating Example and Basic System Requirements

2.1 Motivating Example

Securities exchange in the stock market is a typical instance intensive business process which involves a large number of transactions between different organisations and each of them is a relatively short process instance with only a few steps. Most steps of a process instance are executed in parallel. The example illustrated in Figure 1 is a securities exchange business process for the Chinese Shanghai A-Share Stock Market (<http://www.sse.com.cn/sseportal/en/>). There are more than one hundred securities corporations in this market and each corporation may have more than one hundred branches nation wide. It consists of six major stages (sub-process) in the securities exchange process. Due to the space limit, we only introduce the main facts here while leaving details in [15].

(1) The first stage is “lodge client entrustment” (Step 1). Every trading day, there are millions of clients online. The peak number of transactions can reach several millions per second and the average is around several thousands.

(2) The second stage is “fit and make deal” (Step 2 to Step 3). The raw entrustment data are first validated to check whether the clients have enough money to make the deal. After validation, the dealing results are recorded into the database in the securities corporation. This sub-process needs to complete in several minutes.

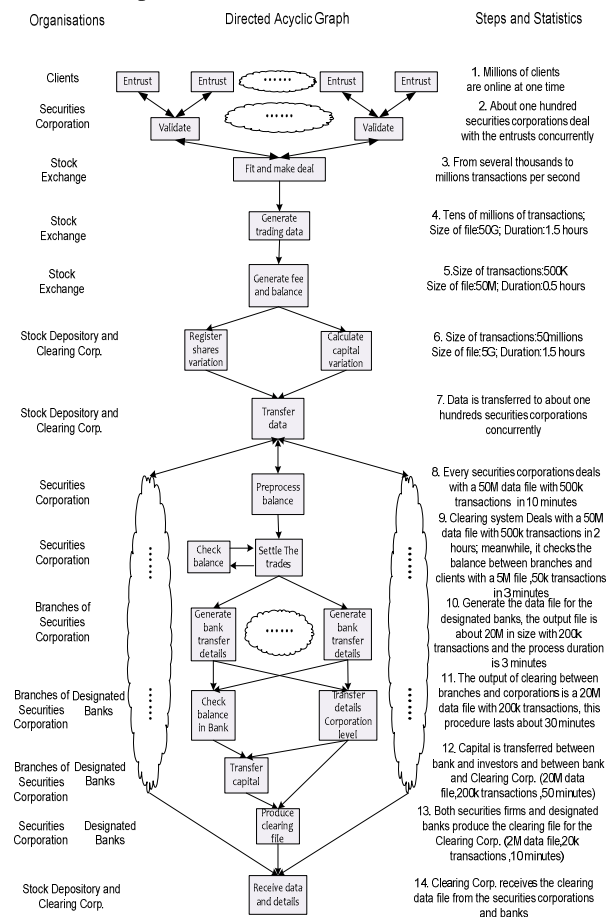


Fig. 1. A Typical Securities Exchange Business Process

(3) The third stage is “register shares variation and calculate capital variation” (Step 4 to Step 6). After 3:00 pm (closing time) of the trading day, all the completed deals need to be archived and summed up by securities corporations for clearing. The size of the output file is about 50G with tens of millions of transactions and the duration of the procedure is about 1.5 hours. All the trading data will be transferred to Shanghai Stock Depository and Clearing Corporation of China (<http://www.chinaclear.cn/>).

(4) The fourth stage is “settle the trades” (Step 7 to Step 9). The output files of the last step are divided by corporation ID and delivered to the securities corporations concurrently. There are three levels of clearings: the first level clearing is between Clearing Corporation and securities corporations, the second one is between securities corporations and their branches, and the third one is between branches and clients. For example, in the second level of clearing, the clearing system deals with a 50M size data file with about 500k transactions in roughly 2 hours. The clearing result of each level should match with each other.

(5) The fifth stage is “transfer capital” (Step 10 to Step 12). The output of the clearing process is the money

²<http://open.blogs.nytimes.com/2008/05/21/the-new-york-times-archive-s-amazon-web-services-timesmachine/>

³ <http://animoto.com/blog/company/amazon-com-ceo-jeff-bezos-on-animoto/>

transfer details for each client who made deals during the day. It is a 20M size data file with about 200k transactions and it should be sent to the designated banks. The designated banks check the bills in about 30 minutes at both the client level and the branch level to ensure each entity has enough money to pay for the shares. The money is then transferred between banks and clients, and between banks and the Clearing Corporation, which takes around 50 minutes.

(6) The last stage is “produce clearing files” (Step 13 to Step 14). Both securities corporations and designated banks should produce the clearing files for the Clearing Corporation. The balance of all the capital transferred should be zero at the Clearing Corporation level. Otherwise, exception handling should be conducted with manual intervention. The whole securities exchange workflow is ended afterwards.

To summarise, the securities exchange is a typical business process which involves many parallel process instances with strict performance requirements such as fast response time and high throughput. Failures of meeting these performance requirements could result in serious financial loss to both clients and securities corporations

2.2 System Requirements

Based on the above motivating example, we can identify the following four basic system requirements for managing large numbers of business processes.

1) Scalable computing resource provision. The running of large numbers of business processes requires powerful computing resources. To deal with millions of concurrent requests (e.g. the first and second stage) and processing these transactions after trading hours (e.g. the fifth and sixth stage), computing resources with high processing power and fast IO speed is required. Only in such a case, the satisfactory performance of the system such as short response time for each request and high throughput for processing massive transactions can be achieved.

2) Elastic computing resource delivery. The amount of computing resources required at peak-time (e.g. over millions of requests per second at the beginning and the end of the trading hours) is much higher than the average (e.g. thousands of requests per second in off-peak time). To ensure satisfactory system performance, huge capital investment is often spent on the IT infrastructure to meet the resource requirement during peak-time. However, this will result in large idle of computing resources and a huge waste of energy. Therefore, the elasticity in resource delivery, i.e. the resource pool can easily increase its size when necessary and decrease immediately after use, is very important for reducing the system running cost.

3) Efficient process management. Process automation is the key to improve the performance of running business processes. Besides, in the real world, the specific process structures may be subject to changes. For example, the introduction of new products and the practice of new market regulations may result in some process changes from the third stage to fifth stage. Therefore, the software system needs to have some flexibility for business process change, as well as some new functional and non-functional (quality) requirements coming with it [19]. To this end, efficient process management (e.g. process modelling,

process redesign, service selection, and task coordination) plays a significant role in process automation.

4) Effective QoS monitoring and control. Since a business software system needs to deal with massive processes with flexible business requirements, how to ensure that all the processes are running with satisfactory QoS requirements is a challenge. For example, if the response time for the second stage (fit and make deal) in the securities exchange process is over the time constraints, e.g. 5 minutes, it will probably result in the failure of the client’s requests, and thus will bring substantial loss to both the client and the securities corporation. Therefore, effective QoS monitoring and control is essential. Specifically, QoS monitoring is to constantly observe the system execution state and detect QoS violations while QoS control is to tackle detected QoS violations so as to ensure the specified QoS constraints can be satisfied.

3 The General Design of a Cloud Workflow System

Given the four basic system requirements discussed in Section 2.2, in this paper, we propose that a cloud workflow system is a competitive solution for managing large numbers of business processes. Naturally, a cloud workflow system is running in a scalable and elastic cloud computing environment (satisfying the first and second system requirements), and it is generally designed to have the basic system components for process modelling, resource management, runtime workflow monitoring and control (satisfying the third and fourth system requirements). Our strategy is to start with prototyping a core cloud workflow system, and then extend its structure and capabilities to meet the requirements for managing large numbers of business processes. In this section, we focus on the general system architecture, functionalities and QoS management, while leaving the details in the system implementation to be demonstrated in Section 4.

3.1 System Architecture

As depicted in Figure 2, the general cloud system architecture consists of four basic layers from the top to bottom: application layer, platform layer, unified resource layer, and fabric layer.

As shown in Figure 2, the general cloud workflow architecture can be a mapping of the general cloud system architecture [10]. Specifically, the application layer consists of cloud workflows (workflow applications for real-world business processes), the platform layer is the cloud workflow system which provides a development and deployment platform for cloud workflows. All the system functionalities of a cloud workflow system such as workflow management, cloud resource management and QoS management are included. The application layer and the platform layer are usually self-maintained by the business organisation⁴. The unified resource layer consists of both software services and hardware services that are required for the running of cloud workflows. Specifically, SaaS (software as a service) can provide massive number of software capabilities for processing different business

⁴A cloud workflow system can be encapsulated as a platform service, i.e. PaaS (platform as a service). In such a case, the platform layer is maintained by external cloud service providers.

tasks, while IaaS (infrastructure as a service) can provision on-demand and elastic computing power to meet the resource requirements for processing business activities. In practice, software and hardware services can also be integrated together and encapsulated to be delivered as VMs (virtual machines). The fabric layer is composed of low level hardware resources such as computing, storage and network resources. The unified layer and fabric layer are often maintained by external cloud service providers⁵.

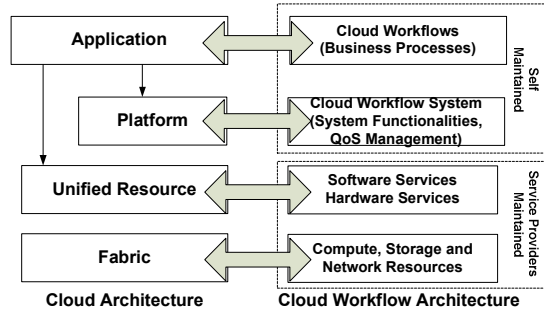


Fig. 2. Cloud Workflow Architecture

3.2 System Functionalities

A cloud workflow system is the combination of workflow system and cloud services. The workflow reference model [2] suggested by WfMC (workflow management coalition, <http://www.wfmc.org/>) defines the general components and interfaces of a workflow system. Therefore, instead of building from the scratch, we can design the basic system functionalities of a cloud workflow system by extending the workflow reference model with functionalities required for the integration of cloud services, such as cloud resource management and QoS management components. Given its critical importance in cloud workflow systems, the QoS management components will be introduced separately in Section 3.3.

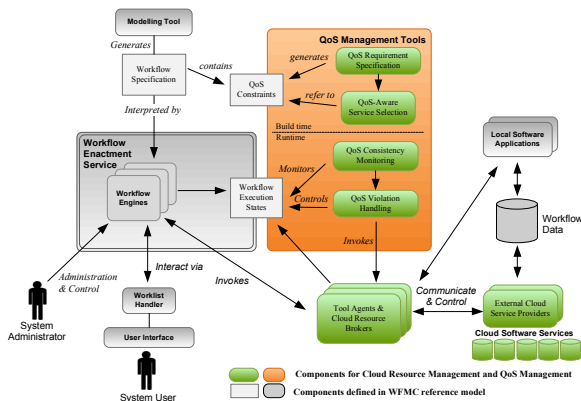


Fig. 3. System Functionalities and QoS Management in Cloud Workflow System

As depicted in Figure 3, the basic system functionalities of a cloud workflow system can be organised in the same way as the workflow reference model. Here, due to the space limit, we only focus on several key components. In a cloud workflow system, the workflow modelling tool provides the system clients an efficient way to create their business applications with the help of visual modelling components and/or scripting languages. Workflow

specifications created by the clients normally contain the information about the business process structures, the task definitions and the QoS requirements. The workflow enactment service is a collection of multiple parallel workflow engines which are in charge of interpreting workflow specifications and coordinating all the management tools and necessary resources for the workflow execution, such as the administration and control tools, work-list monitoring tools, workflow data and control flows, and software services. The workflow engines can invoke local software applications stored in the local repository (or private cloud) and external cloud software services. The workflow engines can search for cloud resources using the cloud resource brokers which perform the searching, reserving and auditing of cloud resources. After successful reservation of a cloud resource, a tool agent will be created which is in charge of the communications with external cloud service providers, and the control of cloud software services according to the instructions it received from the workflow engines.

3.3 QoS Management

Due to the dynamic nature of cloud computing, effective QoS management in a cloud workflow system is very important. Specifically, for managing large numbers of business processes, service performance (e.g. short response time for every client request and high throughput for processing massive concurrent client requests), service reliability (e.g. minimal failure rate for activity execution) and service security (e.g. stringent policies for the lifecycle protection of client data in its storage, transfer and destroy), are among the most important QoS dimensions which should be given higher priority in cloud workflow QoS management [17-19, 24]. Meanwhile, since a cloud workflow instance needs to undergo several stages before its completion, a lifecycle QoS management needs to be established.

In general, a lifecycle QoS management consists of four basic steps, viz. QoS requirement specification, QoS-aware service selection, QoS monitoring and QoS violation handling [16]. As depicted in Figure 3, as part of workflow built-time functionalities, QoS requirement specification and QoS-aware service selection are mainly interacted with the workflow modelling tool. The QoS requirement specification component would generate the QoS constraints, which are part of the workflow specification and the basic criteria for QoS-aware service selection. The QoS-aware service selection component will return the available (best and backup) software services satisfying the QoS constraints, through the cloud resource brokers. After the workflow specifications are submitted to the workflow enactment services, workflow instances can be executed by invoking software services which are managed by the tool agents. During workflow runtime, the workflow execution state will be constantly observed by the QoS monitoring component. The workflow execution state can be displayed to the client and system administrator by a watch list which contains runtime information such as time submitted, time finished, percentage of completion, service status and many other real-time and possible statistic data. When the QoS violations are detected, alert messages would be sent to invoke the QoS violation handling component. This

⁵The fabric layer can also be a virtual collection of local computing infrastructure (i.e. private cloud) and the commercial computing infrastructure (i.e. public cloud), i.e. hybrid cloud.

component will analyse the workflow execution state and the QoS requirement specification to decide further actions. Generally speaking, for QoS violation handling, firstly, we should try to minimise the existing loss through compensation, and secondly, we should prevent similar violations from happening in the subsequent workflow as much as possible [16, 19]. It is evident that due to the complexity of instance intensive business processes and the dynamic nature of cloud computing, satisfactory service quality of a cloud workflow system can only be achieved through such a lifecycle QoS management.

4 System Implementation: A Prototype P2P based Cloud Workflow System

Based on the general design of a cloud workflow system presented in Section 3, this section demonstrates the implementation of a prototype cloud workflow system. SwinDeW-C (Swinburne Decentralised Workflow for Cloud) [17] is running in SwinCloud which is built on the computing facilities in Swinburne University of Technology and takes advantage of the existing SwinGrid infrastructure, a grid computing test bed [17].

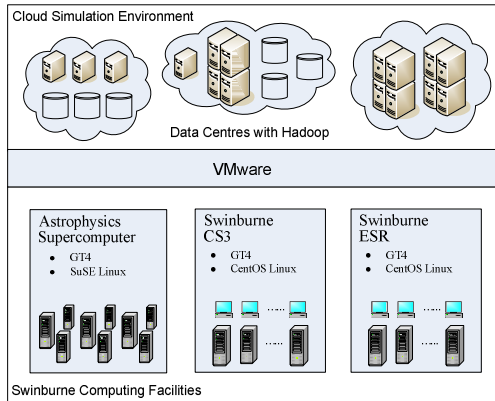


Fig. 4. SwinCloud Infrastructure

The migration of SwinGrid to SwinCloud is achieved in two steps. First, VMWare (<http://www.vmware.com/>) is installed in existing SwinGrid nodes so that they can offer unified computing and storage resources. Second, we set up data centres on the groups of SwinGrid nodes which can host different cloud services. In each data centre, Hadoop (<http://hadoop.apache.org/>) is installed to facilitate Map-Reduce computing paradigm and distributed data management. Different from SwinGrid, SwinCloud is a virtualised computing environment, where cloud services run on unified resources. By dynamically acquiring computing and storage units from VMWare, cloud services can flexibly scale up and down according to system requirements. SwinDeW-C inherits many features of its ancestor SwinDeW-G [23] but with significant modifications in its functionalities to accommodate the cloud computing paradigm and the system requirements for managing instance intensive business processes. Figure 4 depicts the SwinCloud infrastructure. More details about the system environment can be found in [17].

4.1 Architecture of SwinDeW-C

In order to overcome the problems of centralised management such as performance bottleneck, lack of scalability and single point of failure, SwinDeW-C is designed in a decentralised, or more specifically,

structured peer-to-peer fashion where all the workflow data and control flows are transferred among SwinDeW-C peers. Such a design can greatly enhance the performance and reliability of a workflow system in managing large

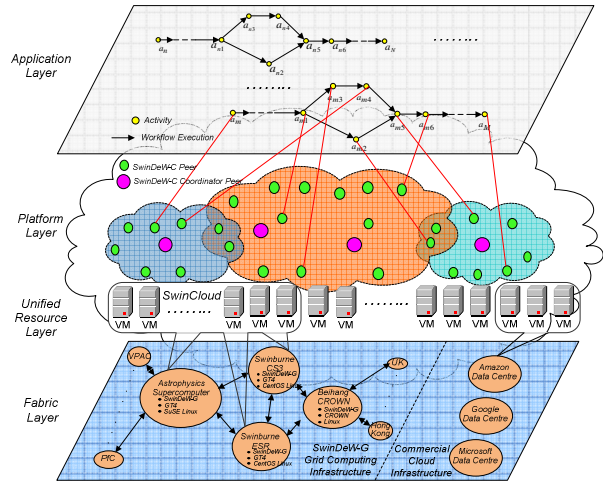


Fig. 5. Architecture of SwinDeW-C

numbers of workflows since all the system functionalities are implemented with distributed SwinDeW-C peers which will be introduced in Section 4.2.

The architecture of SwinDeW-C is depicted in Figure 5 [17]. Clients can access SwinDeW-C Web portal via any electronic devices such as PC, laptop and mobile phone as long as they are connected to the Internet. Compared with SwinDeW-G which can only be accessed through a SwinDeW-G peer with pre-installed client-side programs, SwinDeW-C Web portal can greatly improve the usability. Here, we describe the lifecycle of an abstract workflow application through its modelling stage, instantiation stage and execution stage to illustrate the system architecture.

At the modelling stage, given the cloud workflow modelling tool provided by the Web portal on the application layer, workflow applications are modelled by clients as cloud workflow specifications (consisting of such as task definitions, process structures and QoS constraints). After workflow specifications are created, they will be submitted to one of the coordinator peers on the platform layer. Here, an ordinary SwinDeW-C peer is a cloud service node which has been equipped with specific software services similar to a SwinDeW-G peer. However, while a SwinDeW-G peer is deployed on a standalone physical machine with fixed computing units and memory space, a SwinDeW-C peer is deployed on a virtual machine of which its computing power can scale dynamically. As for the SwinDeW-C coordinator peers, they are super nodes which are equipped with additional management functionalities.

At the instantiation stage, the cloud workflow specification is submitted to one of the SwinDeW-C coordinator peers. A coordinator peer conducts an evaluation process on the submitted cloud workflow instance to determine whether it can be accepted or not given the workflow specification, the available cloud services, and the resource prices. It is generally assumed that functional requirements can normally be satisfied given the unlimited scalable computing resources and software services in the cloud. In the case where clients need to run their own special programs, they can upload

them through the Web portal and these programs can be automatically deployed in the cloud data centre. However, the QoS requirements may not be always satisfied. Due to the natural limitations of cloud service quality and the unacceptable offers on budgets, a negotiation process between the client and the cloud workflow system may be conducted. The final negotiation result can be either the compromised QoS requirements or a failed submission of the cloud workflow instance. If it is successful, the workflow activities will be assigned to suitable SwinDeW-C peers through p2p based communication. The peer management such as peer join, peer leave and peer search, as well as the p2p based workflow execution mechanism, are the same as in SwinDeW-G system environment which are detailed in [23]. After all the workflow activities are successfully allocated (i.e. confirmation messages are sent back to the coordinator peer from all the allocated peers), a cloud workflow instance is successfully instantiated.

Finally, at the execution stage, each workflow activity is executed by a SwinDeW-C peer. Clients can get access to the final results as well as the running information of their submitted workflow instances through the SwinDeW-C Web portal. Each SwinDeW-C peer utilises the computing power provided by its virtual machine which can easily scale up and down according to the requests of workflow activities. As can be seen in Figure 4, SwinCloud is built on the previous SwinGrid infrastructure at the fabric layer. Meanwhile, some of the virtual machines can be created with external commercial IaaS (infrastructure as service) cloud service providers such as Amazon, Google and Microsoft.

4.2 Functionalities of SwinDeW-C Peers

The architecture and functionalities of SwinDeW-C peers are depicted in Figure 6. As mentioned above, the system functionalities of SwinDeW-C are distributed to its peers. SwinDeW-C is developed based on SwinDeW-G, where a SwinDeW-C peer has inherited most of the functionalities in a SwinDeW-G peer, including the components of task management, flow management, data management, and the group management [23]. Hence, a SwinDeW-G peer plays as the core of a SwinDeW-C peer. A SwinDeW-G peer is developed by Java with the Globus toolkit (<http://www.globus.org/toolkit/>) and JXTA (<http://www.sun.com/software/jxta/>).

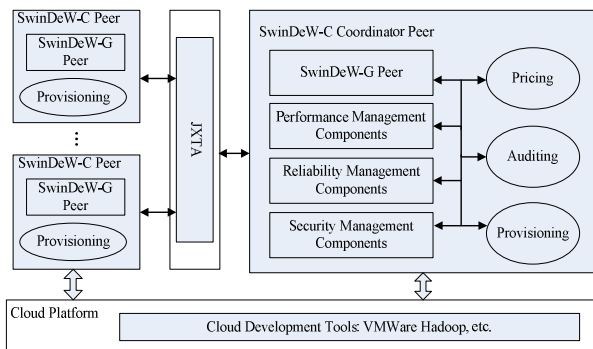


Fig. 6. Architecture and Functionalities of SwinDeW-C Peers

To accommodate cloud resources and the system requirements for instance intensive business processes, a coordinating peer is introduced to the SwinDeW-C system and significant modifications also have been made in other

normal peers. Besides the those functionalities inherited from SwinDeW-G peers, some new cloud resource management components are developed for SwinDeW-C peers based on the APIs offered by VMWare and Hadoop, and some existing components such as QoS management are further enhanced. Specifically:

First, a resource provisioning component is added to every SwinDeW-C peer. In SwinDeW-C, to meet the scalable and elastic resource requirement, a SwinDeW-C peer can scale up or down with more or fewer computing units. Meanwhile, through the SwinDeW-C coordinate peer, it can also scale out or in if necessary, i.e. to request the distribution of workflow activities to more or fewer SwinDeW-C peers in the same group. This is mainly realised through the APIs of VMWare management tools.

Second, the resource pricing and auditing components are equipped in SwinDeW-C coordinator peers. Since different cloud service providers may offer different prices, during the instantiation stage, a coordinator peer needs to have the pricing component to negotiate the prices with external service providers and set its own offered prices to its clients. Meanwhile, since the cloud workflow system needs to pay for the usage of external cloud resources, at the execution stage, an auditing component is required to record and audit the usage of cloud resources. These functionalities are mainly realised through the APIs of resource brokers and the external service provider's monitoring services such as the Amazon CloudWatch (<http://aws.amazon.com/cloudwatch/>).

Third, QoS management components in a SwinDeW-C coordinator peer have been extended to support for multiple QoS dimensions, viz. performance, reliability and security, which are regarded as three major QoS dimensions for running business processes. Specifically, performance management is mainly for the response time and throughput of business processes, reliability management is mainly for the reliability and cost of data storage services, and security management is mainly for transaction security and the protection of client privacy data. According to the lifecycle QoS management introduced in Section 3.3, these components need to interact with many other system build-time and runtime functional components which are implemented as parts of the SwinDeW-C coordinator peers.

4.3 QoS Management in SwinDeW-C

The major workflow QoS dimensions supported in SwinDeW-C are performance, reliability and security. Details can be found in [17]. In this section, we take the performance management component as an example. In our previous work, the performance management only focuses on the response time of a single workflow instance. In this paper, for the running of large numbers of time-constrained business processes, we also focus on the throughput of the cloud workflow system. Specifically, there are four basic tasks for delivering lifecycle performance management in SwinDeW-C:

Temporal Constraint Setting: In SwinDeW-C, temporal constraints consist of two types, viz. constraints for workflow response time, and constraints for system throughput. A probabilistic strategy is designed for setting constraints for workflow response time in SwinDeW-C.

Specifically, one overall deadline and several milestones are assigned based on the negotiation result between clients and cloud workflow service providers. Afterwards, fine-grained constraints for individual workflow activities can be derived automatically [13]. Besides the constraints for the workflow response time, we also need to setup some throughput constraints to monitor system throughputs along the workflow execution. Currently, we adopt a setting strategy where throughput constraints are defined as the percentage of completion and assigned at pre-defined time points with fixed equal time intervals. For example, given a set of 1,000 business processes (each with 10 activities) start at 10:00am and have an overall deadline by 12:00pm, the throughput constraints can be specified as at 10:30am, 25% of the total business processes should be finished, and 50% of them should be finished by 11:00pm, and so forth. Note that here 50% completion does not necessarily mean a total of 500 processes should be finished but rather mean a total of 5,000 activities are completed since significant delays often occur in the running of some business processes.

Temporal-Aware Service Selection: Given the fine-grained constraints for response time assigned in the first step, a set of candidate services which satisfy the constraints can be searched by the cloud resource broker from the cloud [6, 24]. Meanwhile, since different service providers may offer different prices, and there are often other QoS constraints such as reliability and security to be considered at the same time, a ranking strategy is designed to determine the best candidate for runtime execution. Furthermore, considering to the dynamic nature of cloud computing as well as the performance and reliability requirements for managing large numbers of business processes, a set of backup/redundant services should also be reserved during service selection. In fact, many cloud service providers such as Amazon provides special discount price for reserved instances⁶, which can be used as a source of reliable standby capacity.

Temporal Checkpoint Selection and Verification: During workflow runtime, the workflow execution state should be monitored against the violation of temporal constraints. Temporal verification is to check the temporal correctness of workflow execution, i.e. to detect temporal violations of workflow response time and system throughput. The verification of workflow response time constraints is conducted at the activity level and the verification of system throughput is conducted at the workflow level. In SwinDeW-C, a minimum time redundancy based checkpoint selection strategy [8] is employed which selects only necessary and sufficient checkpoints to detect the violations of workflow response time. Here, necessity means only the activity points with temporal violations are selected, and sufficiency means there are no omitted ones, hence the strategy is highly efficient for the monitoring of large numbers of workflow activities. As for throughput verification, it is conducted at the pre-defined time points which are specified at the constraint setting stage, hence in a static fashion.

Temporal Violation Handling: After a temporal violation is detected, violation handling strategies are

required to recover the error states such as the larger response time and lower system throughput. In SwinDeW-C, to decrease the overall violation handling cost on workflow response time, a three-level temporal violation handling strategy is designed. Specifically, for minor temporal violations, the TDA (time deficit allocation) strategy [7] is employed which can remove the current time deficits by borrowing the time redundancy of the subsequent activities. For moderate temporal violations, the ACOWR (ant colony optimisation based two stage workflow local rescheduling) strategy [12] is employed which can decrease the execution time of the subsequent workflow activities through the optimisation of resource allocation. As for major temporal violations, the combined strategy of TDA and ACOWR is employed which conducts TDA in the first step and followed by several iterations of ACOWR until the temporal violation is recovered. Based on such a design, the overall violation handling cost on workflow response time can be significantly reduced compared with a single expensive exception handling strategy [12]. However, since it has been well observed that short response time does not necessarily guarantee an overall high system throughput, we still need some violation handling strategies to recover throughput violations. Meanwhile, since most violation handling strategies such as TDA and ACOWR target the reduction of the response time of a single workflow instance, it may not be directly effective for the increase of system throughput. One of the options is to conduct these strategies repeatedly for many business processes so that the system throughput can be increase by the reduction of the average workflow response time. But this option is evidently very expensive. Currently, in SwinDeW-C, we adopt a simple elastic resource provision strategy which is to dynamically provision the reserved resources when throughput violations are detected, and release these resources when the system throughput is back to normal. In such a case, since many awaiting workflow activities will be processed immediately, the system throughput can be increased in a short period time. Details will be further illustrated in our experiments demonstrated in Section 5.

5 Evaluation

Based on the SwinDeW-C prototype system, the general design of a cloud workflow system proposed in Section 3 is successfully implemented to satisfy the basic system requirements discussed in Section 2.2. Specifically, the four-layer cloud workflow system architecture and the structured p2p based decentralised workflow management ensures efficient provision of scalable and elastic cloud computing resources for running instance intensive business processes (for the first and second system requirements); the visual modelling tool for workflow specification, the workflow enactment service and the application provision service, can effectively support the efficient process management (for the third system requirement); and the QoS management components can facilitate the effective QoS monitoring and control (for the fourth system requirement).

At the moment, to evaluate and improve its performance, a number of test cases with simulated large scale instance intensive workflows are designed and being

⁶ <http://aws.amazon.com/ec2/reserved-instances/>

tested in SwinDeW-C, including the securities exchange workflow and some large scale high performance applications with larger number of sub-processes such as a weather forecast workflow [13] and a pulsar searching workflow in Astrophysics [14].

On the setting of business processes and resources: In order to evaluate the performance of SwinDeW-C, we have simulated a large number of business processes running in parallel. The total number of business processes is 10K, which is similar to the total number of securities corporation branches nation wide. In our experiments, we focus on the offline processing part, i.e. from the step 4 to step 14, where all the daytime transaction data are to be batch-processed over night at the stock depository and clearing corporation. For the ease of simulation, we assume that each process has 20 activities to represent the basic batch processing steps, and correspondingly there are 20 types of cloud services in charge of running these activities. The total number of cloud service instances is set as 200, i.e. 10 instances for each type of service. Additionally, there is 1 reserved instance for each type of service to handle temporal violations. As for the resource price, we adopt the Amazon EC2 price model as a reference (<http://aws.amazon.com/ec2/pricing/>). The price for the primary services (similar to the EC2 Quadruple Extra Large Hi-Memory On-Demand Instances) is \$2.00 per hour, and the price for the reserved services (similar to the EC2 Large Standard Reserved Instances for 1 year fixed term) is about \$0.12 per hour.

The simulation will start from the parallel running of 100 business processes, i.e. the maximum workload for each service instance is set as 10. The activity durations are generated based on the statistics and deliberately extended by a mixture of representative distribution models such as normal, uniform and exponential to reflect the performance of different cloud services. The mean activity durations are randomly generated in a wide range of 30 milliseconds to 30 seconds. Meanwhile, some noises are also added to a random selected activity in each business process to simulate the effect of system uncertainties such as network congestion and performance down time. Different ratio of noises (the added time delays divided by the activity durations) from 5% to 30% are implemented. The process structures are specified as DAG graphs similar to the securities exchange business process.

On the setting of temporal constraints and monitoring strategies: For each business process, an overall temporal constraint is assigned. The strategy for setting temporal constraint is adopted from the work in [13] where a normal percentile is used to specify temporal constraints and denotes the expected probability for on-time completion. Here, we specify the normal percentiles as 1.28 which denotes the probability of 90.0% for on-time completion if without any temporal violation handling. This setting can be regarded as the norm, i.e. the satisfactory performance for most clients and service providers. We employ the state-of-the-art checkpoint selection strategy introduced in [8] as the strategy for detecting the violations on workflow response time. As for the monitoring of system throughput, we pre-define a set of time points with the equal fixed time interval as introduced in Section 4.3. Specifically, the fixed time interval in our experiments is set as 60 seconds,

i.e. around 20% of the average duration of a business process. Therefore, at the first 60 seconds, the system will verify whether 20% of the total activities (i.e. $20\% \times 100 \times 20 = 400$) have been finished, and at the next time point, i.e. at the time points for 120 seconds, the system will verify whether 40% (i.e. 800) of the total activities have been finished, and so on so forth until the completion of all the 10,000 business processes. The throughput verification will be conducted at every pre-defined time points.

On the setting of temporal violation handling strategies: For the comparison purpose, we record the global violation rates under natural situations, i.e. without any handling strategies (denoted as NIL). The violation handling strategies we implemented including the standalone Workflow Local Rescheduling strategy, the standalone Extra Resource Recruitment strategy, and the combined of the two strategies. The Workflow Local Rescheduling strategy is based on ACOWR and the Extra Resource Recruitment strategy is based on the simple elastic resource provision strategy (denoted as SERP), as introduced in Section 4.3. For the standalone ACOWR or SERP, the same strategy will be applied both to the violations of response time and system throughput. As for the combined strategy (denoted as ACOWR+SERP), ACOWR will handle the violations of response time and SERP will handle the violations of system throughput respectively. The parameter settings for ACOWR are same as in [14]. As for SERP, we employ one additional instance for each type of service when a throughput violation is detected, and immediately release them when the system throughput is back to normal at the next throughput constraint. Based on the resource settings mentioned above, the average cost for ACOWR is $\$3.08 \times 10^{-3}$ per time, which is mainly the computation cost for running the rescheduling strategy. Note that the cost for the re-allocation of workflow activities after rescheduling is not accounted here since the data transfer within a data centre is free in Amazon cloud. As for SERP, the cost is \$16.7 per round where 20 cloud services are reserved and dedicated for the entire running period, i.e. an average of 8 hours per day.

TABLE 1. Numbers of Temporal Violations

Rounds	Normal	Uniform	Exponential	Noise	Number of Response Time Violations	Number of Throughput Violations
R1	30%	50%	20%	0%	4892	63
R2	30%	50%	20%	5%	5108	252
R3	40%	50%	10%	10%	5322	473
R4	40%	50%	10%	15%	5601	690
R5	40%	40%	20%	15%	5446	671
R6	40%	50%	10%	20%	5715	890
R7	40%	40%	20%	20%	5687	886
R8	30%	50%	20%	25%	5719	1057
R9	40%	50%	10%	25%	5973	1104
R10	30%	50%	20%	30%	5895	1257

Based on the above experimental settings, 10 rounds of experiments are implemented and each runs for 100 times to get average values. Table 1 shows the number of temporal violations recorded in each round of experiment. Clearly, the number of response time violations for workflow instances and the number of throughput violations for the workflow system both increase rapidly with the increase of noise, i.e. the embedded time delays to represent various system uncertainties. For example, with the every 5% increase of noise, the average increase of response time violations is around 200. The distribution of

service performance seems to have less effect on the temporal violations. For example, given the same noise, the average difference of throughput violations (e.g. R4 and R5, R6 and R7, R8 and R9) is around 20.

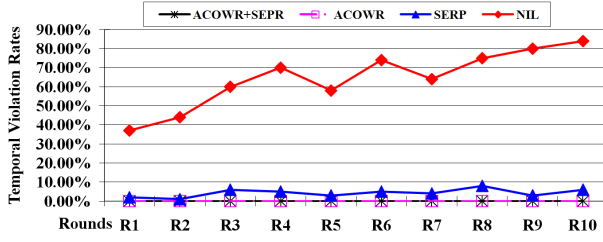


Fig. 7. Temporal Violation Rate with Different Violation Handling Strategies

Figure 7 depicts the temporal violation rates (the unsuccessful rate for on-time completion of the entire 10K business processes) with different violation handling strategies. For comparison purpose, the results of NIL represent the natural condition, i.e. without any handling strategies, where the violation rates increase from 37% to 84% with an average of 65%. Both the combined strategy of ACOWR+SEPR and the standalone ACOWR can ensure a very close to 0% violation rate. The standalone SERP strategy can also maintain a very low violation rate, i.e. with an average around 4%. The reason for such a difference is mainly because of the different granularity between ACOWR and SERP. Since ACOWR is triggered at every necessary and sufficient checkpoint while SERP only take place at pre-defined time points (a ratio around 13% according to the results shown in Table 1), there are some chances that significant time delays cannot be handled in time by SERP, and thus result in some unsuccessful on-time completion of business processes.

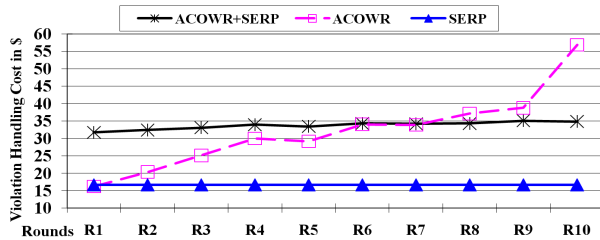


Fig. 8. Temporal Violation Cost with Different Violation Handling Strategies

Figure 8 demonstrate the total temporal violation handling cost for each type of handling strategies. The cost for the standalone SERP is static with \$16.7 in each round because only the reserved service instances are used. The cost for the combined strategy of ACOWR+SEPR is also very stable from \$31.7 to \$35.1 with an average of \$33.7. In contrast, the cost for the standalone ACOWR increases significantly from \$16.2 to \$56.8 with an average of \$32.1. This is mainly because the standalone ACOWR needs to run many times (from 2 to 10 times in our experiments) to handle throughput violations, and hence the cost increases rapidly with the number of throughput violations.

In summary, according to the experimental results presented above, we can see that the combined strategy of ACOWR+SEPR is the best one which can maintain the close to 0% violation rate while having the moderate cost among the three. Meanwhile, since our experimental settings actually allow for a probability of 10% violations (i.e. 90.0% for on-time completion), if a small violation

rate is tolerable by the clients, e.g. below 5%, the standalone SERP strategy is also applicable and can significantly reduce the violation handling cost. In general, we can see that thanks to the dynamic and elastic resource provision provided by cloud computing, conventional violation handling strategies such as ACOWR can ensure the on-time completion of large numbers of business processes in a more cost-effective way.

6 Related Work

Traditional workflow systems are normally designed to support the business processes in a specific domain such as bank, hospital and school. Therefore, they can only invoke existing software applications (or software components) which have already developed and stored in the local repository, which limits the flexibility in the support of general and agile business processes [2, 18]. In the last decade, with the rapid development of Web services, the employment of remote software services from external service providers becomes possible. Therefore, the design and application of Web service based workflow systems starts to attract most of the attention from both researchers and practitioners, for example, the Windows Workflow Foundation (<http://www.windowsworkflowfoundation.eu/>) and the Kepler project (<http://kepler-project.org/>). In recent years, with the fast growth of high performance computing (HPC) and high throughput computing (HTC) such as cluster and grid, workflow systems are also being used as a type of middleware service which often underlies many large-scale complex e-science applications such as climate modelling, astrophysics, and chemistry [9, 20]. The work in [24] proposes a taxonomy and summaries a number of grid workflow systems such as Condor (<http://www.cs.wisc.edu/condor/>), Gridbus (<http://www.gridbus.org/>), Pegasus (<http://pegasus.isi.edu/>), and Triana (<http://www.trianacode.org/>). However, they mainly target at processing data and computation intensive activities for a single scientific workflow instance, rather than massive concurrent workflow instances for business processes.

The design of a cloud workflow system, as the combination of workflow system and cloud computing, comes from the natural needs for efficient and effective management of large numbers of business processes. However, as a cutting-edge research issue, the investigation on cloud workflow systems is so far still in its infancy. Besides SwinDeW-C, there are currently a few existing grid workflow systems investigating the migration from grid to cloud such as Pegasus in the cloud (http://pegasus.isi.edu/pegasus_cloud.php) and GridBus to CloudBus (http://www.cloudbus.org/cloudbus_flyer.pdf), but most of them are for scientific applications. Response time and system throughput are the most important measurements for the performance analysis of cloud workflow systems [3]. The work in [11] proposes a throughput maximisation strategy for transaction intensive cloud workflows. The work in [22] investigates the dynamic resource allocation for efficient parallel data processing in the cloud.

To the best of our knowledge, this is the first paper that proposes the solution of a cloud workflow system to address the management of running large numbers of time-constrained business processes.

7 Conclusions and Future Work

The concurrency of large numbers of client request has been widely seen in today's e-business and e-government systems. Based on the analysis of a securities exchange business process, we have identified four basic system requirements for managing large numbers of business processes, viz. scalable computing resource provision, elastic computing resource delivery, efficient process management, and effective QoS monitoring and control. Based on that, the cloud workflow system is proposed as a competitive solution. We first present the general design of a cloud workflow system with the focus on its system architecture, basic functionalities and QoS management. Afterwards, based on such a general design, we have implemented a peer-to-peer based cloud workflow system prototype, SwinDeW-C. The architecture of SwinDeW-C (four-layered architecture), the system functionalities (realised in the functional components of SwinDeW-C coordinator and ordinary peers), and the QoS management (with the illustration of the performance management components) have been demonstrated to verify the effectiveness of our system design. The experimental results for the evaluation of system performance have shown that satisfactory on-time completion rate and better cost-effectiveness can be achieved with the dynamic and elastic provision of cloud resources.

In the future, the monitoring and violation handling strategies for the system throughput will be further enhanced. Meanwhile, the SwinCloud test bed will be extended in its size and capacity so that real world large scale business processes can be tested to further evaluate and improve our system design.

Acknowledgments. We are grateful for the discussions with Mr. Zhangjun Wu and his colleagues on the securities exchange business process. This work is partially supported by Australian Research Council under Linkage Project LP0990393.

References

1. Australian Academy Of Technology Science and Engineering, Cloud Computing: Opportunities and Challenges for Australia, <http://www.atse.org.au/component/remository/ATSE-Reports/Information-Technology/CLOUD-COMPUTING-Opportunities-and-Challenges-for-Australia-2010/>, accessed on 1st Aug. 2011
2. Aalst, W.M.P. van der, Hee, K.M.V.: Workflow Management: Models, Methods, and Systems. The MIT Press (2002)
3. Iosup, A., Ostermann, S., Yigitbasi, N., and et al.: Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems* 22(6), 931-945, (2011)
4. Armbrust, M., Fox, A., Griffith, R., Joseph, and et al: Above the Clouds: A Berkeley View of Cloud Computing. Technical Report, UCB/EECS-2009-28, University of California at Berkeley (2009)
5. Buyya, R., Bubendorfer, K.: Market Oriented Grid and Utility Computing. Wiley Press, New York, USA (2009)
6. Buyya, R., Yeo, C.S., Venugopal, and et al: Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems* 25, 599-616 (2009)
7. Chen, J., Yang, Y.: Multiple States based Temporal Consistency for Dynamic Verification of Fixed-time Constraints in Grid Workflow Systems. *Concurrency and Computation: Practice and Experience*, Wiley 19, 965-982 (2007)
8. Chen, J., Yang, Y.: Temporal Dependency based Checkpoint Selection for Dynamic Verification of Temporal Constraints in Scientific Workflow Systems. *ACM Transactions on Software Engineering and Methodology*, 20(3), article 9, (2011)
9. Deelman, E., Gannon, D., Shields, and et al.: Workflows and e-Science: An Overview of Workflow System Features and Capabilities. *Future Generation Computer Systems* 25, 528-540 (2008)
10. Foster, I., Yong, Z., Raicu, I., and et al: Cloud Computing and Grid Computing 360-Degree Compared. In: *Grid Computing Environments Workshop*, 1-10. (2008)
11. Liu, K., Chen, J., Yang, Y., and et al: A Throughput Maximization Strategy for Scheduling Transaction-Intensive Workflows on SwinDeW-G. *Concurrency and Computation-Practice & Experience* 20, 1807-1820 (2008)
12. Liu, X., Chen, J., Wu, Z., and et al: Handling Recoverable Temporal Violations in Scientific Workflow Systems: A Workflow Rescheduling Based Strategy. In: *10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 534-537. (2010)
13. Liu, X., Chen, J., Yang, Y.: A Probabilistic Strategy for Setting Temporal Constraints in Scientific Workflows. In: *6th International Conference on Business Process Management (BPM08)*, pp. 180-195. Springer-Verlag, (2008)
14. Liu, X., Ni, Z., Wu, Z., and et al: A Novel General Framework for Automatic and Cost-Effective Handling of Recoverable Temporal Violations in Scientific Workflow Systems. *Journal of Systems and Software* 84(3), 492-509, (2011)
15. Liu, X., Ni, Z., Yuan, D., and et al: A Novel Statistical Time-Series Pattern based Interval Forecasting Strategy for Activity Durations in Workflow Systems. *Journal of Systems and Software* 84(3), 354-376, (2011)
16. Liu, X., Yang, Y., Jiang, Y., and et al: Preventing Temporal Violations in Scientific Workflows: Where and How. *IEEE Transactions on Software Engineering*, published online <http://doi.ieeecomputersociety.org/10.1109/TSE.2010.99>, (2010)
17. Liu, X., Yuan, D., Zhang, G., and et al: SwinDeW-C: A Peer-to-Peer Based Cloud Workflow System. In: Furht, B., Escalante, A. (eds.), *Handbook of Cloud Computing*. Springer (2010)
18. Melvin B. Greer, J.: Software as a Service Inflection Point. *iUniverse* (2009)
19. Sadiq, S.W., Orłowska, M.E., Sadiq, W.: Specification and Validation of Process Constraints for Flexible Workflows. *Information Systems* 30, 349-378 (2005)
20. Taylor, I.J., Deelman, E., Gannon, D.B., and et al: Workflows for e-Science: Scientific Workflows for Grids. Springer (2007)
21. Wang, L., Jie, W., Chen, J. (eds.): *Grid Computing: Infrastructure, Service, and Applications*. CRC Press, Talyor & Francis Group (2009)
22. Warneke, D., Kao, O.: Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud. *IEEE Transactions on Parallel Distributed Systms* 22(6), 985-997 (2011)
23. Yang, Y., Liu, K., Chen, J., Lignier, J., Jin, H.: Peer-to-Peer Based Grid Workflow Runtime Environment of SwinDeW-G. In: *3rd International Conference on e-Science and Grid Computing*, 51-58. (2007)
24. Yu, J., Buyya, R.: A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing* 3, 171-200 (2005)

Use of Multiple GPUs on Shared Memory Multiprocessors for Ultrasound Propagation Simulations

Jiri Jaros¹, Bradley E. Treeby² and Alistair P. Rendell¹

¹Research School of Computer Science, College of Engineering and Computer Science
The Australian National University
Canberra, ACT 0200, Australia

jiri.jaros@anu.edu.au, alistair.rendell@anu.edu.au

²Research School of Engineering, College of Engineering and Computer Science
The Australian National University
Canberra, ACT 0200, Australia

bradley.treeby@anu.edu.au

Abstract

This paper outlines our effort to migrate a compute intensive application of ultrasound propagation being developed in Matlab to a cluster computer where each node has seven GPUs. Our goal is to perform realistic simulations in hours and minutes instead of weeks and days. In order to reach this goal we investigate architecture characteristics of the target system focusing on the PCI-Express subsystem and new features proposed in CUDA version 4.0, especially simultaneous host to device, device to host and peer-to-peer transfers that the application is going to highly benefit from. We also present the results from a CPU based implementation and discuss future directions to exploit multiple GPUs.

Keywords: Ultrasound simulation, 7-GPU system, CUDA, Matlab, FFT, PCI-Express, bandwidth, multi-core.

1 Introduction

In 1994 Becker and Sterling (1995) proposed the construction of supercomputer systems through the use of off-the-shelf commodity parts and open source software. Over the ensuing year, the so called Beowulf cluster computer systems came to dominate the top 500 list of most powerful systems in the world. The advantages of such systems are many, including ease of creation, administration and monitoring, and full support of many advanced programming techniques and high performance computing libraries such as OpenMPI. Interestingly, however, what was originally a major advantage of these systems, namely price and running costs, is now much less so. This is because for even a small to moderately sized cluster it is necessary to house the system in specially air-conditioned machine rooms.

Recently, developments in Graphics Processing Units (GPUs) have prompted another revolution in high-end computing, equivalent to that of the original Beowulf cluster concept. Although these chips were designed to

accelerate rasterisation of graphic primitives such as lines and polygons, their raw computing performance has attracted a lot of researchers to utilize them as acceleration units for special kind of mathematical operations in many scientific applications (Kirk and Hwu 2010). Compared to a CPU, the latest GPUs are about 15 times faster than six-core Intel Xeon processors in single-precision calculations. Stated another way, a cluster with a single GPU per node offers the equivalent performance of a 15 node CPU only cluster. Even more interestingly, the availability of multiple PCI-Express buses even on very low cost commodity computers means that it is possible to construct cluster nodes with multiple GPUs. Under this scenario, a single node with multiple GPUs offers the possibility of replacing fifty or more nodes of a CPU only cluster.

On the other hand, the development tools for debugging and profiling of GPU-based applications are in their infancy. Obtaining the peak performance is very difficult and sometimes impossible for a lot of real-world problems. Moreover, only a few basic GPU libraries such as LAPACK and BLAS have so far been developed, and these are only able to utilize one GPU in a node (CUDA Math Libraries 2011). GPU-based applications are also limited by the GPU architecture and memory model making general-purpose computing much more difficult to implement than a CPU-based application.

The purpose of this paper is to outline our efforts to migrate a compute intensive application for ultrasound simulation being developed in Matlab to a cluster computer where each node has seven GPUs. The utilised numerical methods are very memory efficient compared to conventional finite-difference approaches, and the Matlab implementation already outperforms many of the other codes in the literature (Treeby 2011). However, for large scale simulations, the computation times are still prohibitively long. Our overall goal is to perform realistic simulations in hours or minutes instead of weeks or days. This paper provides an overview of the ultrasound propagation application, the development of an optimised C++ version of the original Matlab code for the CPU that exploits streaming extensions, our attempts to characterise the multi-GPU target system, and a preliminary plan for the GPU code to run on that system.

Section 2 provides background on ultrasound simulation, the simulation method used here, and the time consuming operations. Section 3 introduces the architecture

of our 7-GPU Tian servers that will be used for testing and benchmarking our implementations written in C++ and CUDA. Section 4 gives preliminary results of the first C++ implementation using only CPUs and investigates the bottlenecks. Section 5 focuses on the GPU side of the Tian servers and measures the basic parameters of them in order to acquire necessary experience and investigate the potential architecture limitations. The last section summarizes open questions and issues that will be dealt with in the future.

2 Ultrasound Propagation Simulations

The simulation of ultrasound propagation through biological tissue has a wide range of practical applications. These include the design of ultrasound probes, the development of image processing techniques, studying how ultrasound beams interact with heterogeneous media, training ultrasonographers to use ultrasound equipment, and treatment planning and dosimetry for therapeutic ultrasound applications. Here, ultrasound simulation can mean either predicting the distribution of pressure and energy produced by an ultrasound probe, or the simulation of diagnostic ultrasound images. The general requirements are that the models correctly describe the different acoustic effects whilst remaining computationally tractable.

In our work, the k -space pseudospectral method is used to reduce the number of grid points required per wavelength for accurate simulations (Tabei 2002). The system of governing equations used is described in detail by Treeby (2011). These are derived from general conservation laws, discretised using the k -space pseudospectral method, and then implemented in Matlab (Treeby 2010). In order to be able to simulate real-world systems, both huge amounts of memory and computation power are required.

Let us calculate a hypothetical execution time requested for simulating a realistic ultrasound image using Matlab on a dual six-core Intel Xeon processor. The ultrasound image is created by steering the ultrasound beam through the tissue and recording the echoes received from that particular direction. The recorded signal from each direction is called an A-line, and a typical image is constructed from at least 128 of these. This means we need 128 independent simulations with slightly modified input parameters. Using a single computer, these must be computed sequentially. Every simulation is done over the 3D domain with grid sizes starting at $768 \times 768 \times 256$ grid points and 3000 time steps. From preliminary experiments performed using the Matlab code, each simulation takes about 27 hours of execution time and consumes about 17 GB of memory. Thus to compute one ultrasound image would require roughly 145 days. The objective of this work is to reduce this time to hours or even minutes by exploiting the parallelism inherent in the algorithm.

2.1 k -space Pseudospectral Simulation Method Implemented in Matlab

The Matlab code simulating non-linear ultrasound propagation using the k -space pseudospectral method is based on the forward and inverse 3-dimensional fast Fourier transformation (FFT) supported by a few 3D matrix operations such as element-wise multiplication, addition, sub-

traction, division, and a special `bsxfun` operation. This function replicates a vector in particular dimensions to create a 3D matrix on the fly and then performs a defined element-wise operation with another 3D matrix (such as multiplication denoted by `@times`). Most operations work over the real domain, however, some of them are done over the complex one.

The time step loop in a simplified form is shown in Figure 1. This listing identifies all the necessary mathematical operations and presents all matrices, vectors, and scalar values necessary for computation. For the computation, it is necessary to maintain the complete dataset in main memory. This data set is composed of 14 real matrices, 3 complex matrices, 6 real and 6 complex vectors.

An iteration of the loop represents one time step in the simulation of ultrasound propagation over time. The computation can be divided into a few phases corresponding to the particular code statements:

(1) A 3D FFT is computed on a 3D real matrix representing the acoustic pressure at each point within the computational domain. Despite the fact the matrix p is purely real, a 3D complex-to-complex FFT is executed in Matlab.

(2) - (4) New values for the local particle velocities in each Cartesian dimension x , y , z are computed. These velocities describe the local vibrations due to the acoustic waves. The result of `fft2n(p)` is element-wise multiplied by a complex matrix κ and then multiplied by a vector expanded into a 3D matrix in the given directions using `bsxfun`. After that, the 3D inverse FFT is computed. As we are only interested in real signals, the complex part of the inverse FFT is neglected. Other element-wise multiplications and subtractions are further applied. Note that the old values of the particle velocities are necessary for determining the new ones.

(5) The particle velocities in the x -direction at particular positions are modified due to the output of the ultrasound probe. (Note, additional source conditions are also possible, only one is shown here for brevity). The matrix `ux_sgx` is transformed to a vector and mask-based element-wise addition is executed.

(6) - (8) The gradient of the local particle velocities in each Cartesian direction is computed. First, the 3D FFT of the particle velocity is computed, then, the result is multiplied by κ and a vector in the complex domain. After that, the inverse 3D FFT is calculated. Only the real part of the FFT is used in the difference matrix.

(9) - (11) The mass conservation equations are used to calculate the ρ_{ox} , ρ_{oy} and ρ_{oz} matrices (acoustic density at each point within the computational domain). All operations are done over the real domain on 3D matrices. If an operand is a scalar or a vector, it is expanded to a 3D matrix on the fly.

(12) The new value of pressure matrix is computed here using data from all three dimensions. Two forward and inverse 3D FFTs are necessary for intermediate results. All other operations are done over the real domain.

(13) The pressure matrix is sampled and the samples are stored as the final result.

In summary, at a high level we need to calculate 6 forward and 8 inverse 3D FFTs, and about 50 other element-wise operations, mainly multiplications.


```

% start time step loop
for t_index = 2:Nt

    % compute 3D fft of the acoustic pressure
    1 p_k = fftn(p);

    % calculate the local particle velocities in
    % each Cartesian direction
    2 ux_sgx = bsxfun(@times, pml_x_sgx,
        bsxfun(@times, pml_x_sgx, ux_sgx)
        - dt./rho0_sgx .* real(ifftn(
            bsxfun(@times, ddx_k_shift_pos,
                kappa .* p_k) ))
        );
    3 uy_sgy = bsxfun(@times, pml_y_sgy,
        bsxfun(@times, pml_y_sgy, uy_sgy)
        - dt./rho0_sgy .* real(ifftn(
            bsxfun(@times, ddy_k_shift_pos,
                kappa .* p_k) ))
        );
    4 uz_sgz = bsxfun(@times, pml_z_sgz,
        bsxfun(@times, pml_z_sgz, uz_sgz)
        - dt./rho0_sgz .* real(ifftn(
            bsxfun(@times, ddz_k_shift_pos,
                kappa .* p_k) ))
        );

    % add in the transducer source term
    5 if transducer_source >= t_index
        ux_sgx(us_index) = ux_sgx(us_index) +
            transducer_input_signal(delay_mask);
        delay_mask = delay_mask + 1;
    end

    % calculate spatial gradient of the particle
    % velocities
    6 duxdx = real(ifftn( bsxfun(@times,
        ddx_k_shift_neg, kappa .* fftn(ux_sgx) )));
    7 duydy = real(ifftn( bsxfun(@times,
        ddy_k_shift_neg, kappa .* fftn(uy_sgy) )));
    8 duzdz = real(ifftn( bsxfun(@times,
        ddz_k_shift_neg, kappa .* fftn(uz_sgz) )));

    % calculate acoustic density rhox, rhoy and
    % rhoz at the next time step using a
    % nonlinear mass conservation equation
    9 rhox = bsxfun(@times, pml_x, (rhox -
        dt.*rho0 .* duxdx) ./ (1 + 2*dt.*duxdx));
    10 rhoy = bsxfun(@times, pml_y, (rhoy -
        dt.*rho0 .* duydy) ./ (1 + 2*dt.*duydy));
    11 rhoz = bsxfun(@times, pml_z, (rhoz -
        dt.*rho0 .* duzdz) ./ (1 + 2*dt.*duzdz));

    % calculate the new pressure field using a
    % nonlinear absorbing equation of state
    12 p = c.^2.*( ...
        (rhox + rhoy + rhoz)
        + absorbt_tau.*real(ifftn(
            absorbt_nabla1 .*
            fftn(rho0.*(duxdx+duydy+duzdz) ))
        - absorbt_eta.*real(ifftn(
            absorbt_nabla2 .*
            fftn(rhox + rhoy + rhoz) ))
        + BonA.*(rhox + rhoy + rhoz).^2
        ./ (2*rho0)
    );

    % extract and save the required storage data
    13 sensor_data(:, t_index) = p(sensor_mask_ind);

end
    
```

Figure 1: Matlab code for the k -space pseudospectral method showing the necessary operations.

3 Architecture of Tyan 7-GPU Servers

This section describes the architecture of the Tyan servers targeted for use in the ultrasound propagation simulations. The Tyan servers are 7-GPU servers based on the Tyan barebones TYAN FT72B7015 (Tyan 2011). The barebones consist of a standard 4U rack case and three independent hot-swap 1kW power supplies.

A schematic of the Tyan 7-GPU server configuration can be seen in Figure 2. The motherboard of the servers offers two LGA 1366 sockets for processors based on the Core i7 architecture in a NUMA configuration. The server is populated with two six-core Intel Xeon X5650 processors offering 12 physical cores in total (24 with HyperThreading technology). As each processor contains three DDR3 memory channels, the server is equipped with six 4GB modules (24 GB RAM). The memory capacity can be expanded up to 144GB using 12 additional memory slots.

Communication among CPUs and attached memories is supported by the Intel QuickPath Interconnection (QPI) with a theoretical bandwidth of 12 GB/s. This interconnection also serves as a bridge between CPUs and two Intel IOH chips that offer various I/O connections including four PCI-Express links.

By themselves, the four PCI-Express x16 links are insufficient to connect 7 GPUs and an Infiniband card at full speed. (We would have needed 128 PCI-E links, but unfortunately, had only 64.) Therefore, intermediate PEX bridges were placed between the IOH chips and other devices to double the number of PCI-E links. One PEX bridge is shared between two GPUs (or a GPU and an Infiniband card). The PEX bridges allocate PCI-Express links to the GPUs based on their actual requirements. If one GPU is idle the other one can use all 16 links.

As the servers are designed as a cutting edge GPGPU platform, the most powerful NVIDIA GTX 580 cards with 512 CUDA cores and 1.5GB of main memory have been used. These cards, based on the Fermi architecture, support the latest NVIDIA CUDA 4.0 developer kit and represent the fastest cards that can currently be acquired.

The operating system and user data are stored on two 500GB hard disks, one of which serves as a system disk and the other one as temporary disk space for users. The servers are interconnected using the Infiniband links and a 48 port QLogic Infiniband switch, and to the internet using one of four Gb Ethernet cards.

The operating system the servers are running is Ubuntu 10.04 LTS server edition. For our implementation we have decided to use standard GNU C++ compiler and the latest CUDA version 4.0. This introduces a lot of new features mainly targeted to multi-GPU systems, such as peer-to-peer communication among GPUs, zero-copy main memory accesses from GPUs, etc. OpenMPI is used to communicate between servers and OpenIB layer to directly access the infiniband network card.

4 CPU-based C++ Implementation

In order to accelerate the execution of the Matlab code, the time critical simulation loop has been re-implemented in C++ while paying attention to the underlying architecture to exploit all available performance. A good CPU implementation will serve as a starting point for a GPU

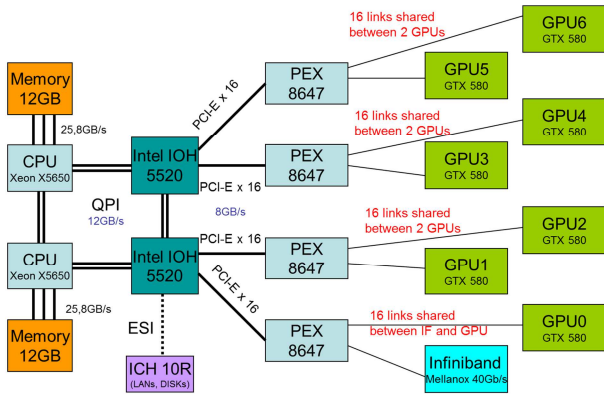


Figure 2: Architecture of 7-GPU server used for the acceleration of ultrasound simulations.

implementation, revealing all the hidden difficulties and ineffectiveness in the Matlab code while also providing ideas on how to improve the Matlab code.

First of all, the import and export of data structures from Matlab to C++ and back has to be designed. Fortunately, all Matlab matrices can be transformed into linear arrays (solving the problem with column-first ordering of multidimensional arrays in Matlab) and saved into separated files using an ASCII or binary format.

All imported matrices as well as six temporary matrices are maintained in main memory during the computation. In the C++ code, the matrices are treated as linear vectors and allocated using the `malloc` function. This organisation simplifies the computation because there is no need to use three indices in element-wise operations. The complex matrices are stored in an interleaved form (even indices correspond to real parts and odd indices the imaginary part of the elements). Another advantage of this data storage format is compatibility with FFTW and CUDA routines when implementing the GPU version.

The C++ code benefits from using an object oriented programming pattern. Each matrix is implemented as a class inheriting basic operations from base classes (real matrix class, complex matrix class) and introducing new methods reflecting the simulation method.

The C++ code does not follow the Matlab code in a verbatim way. Some intermediate results have been precomputed and several temporary matrices have been introduced and reused to save computational effort.

4.1 Complex-to-Complex FFT

Apart from easy to implement element-wise operations, the multidimensional FFT is computed many times in the code. Instead of creating a new implementation, the well-known FFTW library has been employed (FFTW 2011). This library is optimized for a huge number of CPU architectures including multi-core systems with shared memory and clusters with message passing and their streaming extensions such as MMX, SSE, AVX, etc.

A special class encapsulating the FFTW library has been designed in the C++ code. As Matlab uses complex-to-complex 3D FFTs even for real input matrices, the first version of the C++ code also employed the complex-to-complex in-place version of the 3D FFT. First, the input matrix is copied into the FFTW object and transformed

into a complex matrix. Then, the forward FFT is computed. As the FFTW class is compatible with other matrix classes, it serves as a temporary storage. Having computed the FFT, a few element-wise operations are performed on this complex matrix, and finally, the inverse FFT is computed. As FFTW does not use normalization, each element has to be divided by the product of the matrix dimension sizes.

4.2 Operation Fusion

The naïve C++ implementation, created at first, encodes each mathematical operation as a separate method parallelized using OpenMP directives. It allows us to understand the algorithm and validate the code. On the other hand, this implementation is extremely ineffective. It is caused by a very poor calculation to memory access ratio while processing very large matrices in the order of hundreds of MBs, and high thread management overhead.

The operation fusion reduces the memory accesses by performing multiple mathematical operations on corresponding elements at once and saving the temporary results in cache memories. As a result, memory bandwidth is saved enabling better scalability at the expense of more complicated code.

4.3 Real-to-Complex FFT

As all the forward FFTs take only real 3D matrices as an input, the results of the forward FFTs are symmetrical. Analogously, as we are only interested in real signals, the imaginary parts of the inverse FFTs are of no use.

Substituting complex-to-complex FFTs with real-to-complex ones saves nearly 50% of the memory and computation time related to FFTs. Moreover, as other operations and matrices are applied to the result of the FFT, we save additional computation effort and memory because of not having to store the symmetrical parts of auxiliary matrices such as `kappa`.

4.4 SSE Optimization and NUMA Support

The final version of the C++ code benefits from a careful optimization of all element-wise operations in order to utilize streaming extensions such as SSE and AVX. Some of the routines were revised so that the C++ compiler could utilize automatic vectorization to produce a highly optimized code. In the cases it was not possible to do so, the compiler intrinsic functions had to be used for rewriting the particular routines from scratch.

Finally, as the Tyan servers are based on the Non-Uniform Memory Access (NUMA) architecture, some policies preventing threads and memory blocks to migrate among cores and local memories have been incorporated into the code. First, all the threads are locked on CPU cores using an OS affinity property. Secondly, the shared memory blocks for all the matrices are allocated by the master thread and immediately initialized and distributed into local memories using a parallel first touch policy (Terboven, C., Mey, D., et.al. 2008). As the access pattern remains unchanged for element-wise routines, the static OpenMP scheduling guarantees all the matrices remain in the local memories. The only exception is the FFT computation, fortunately handed by FFTW library.

4.5 Execution Time Comparison

This section presents the first results of the C++ implementation and compares the execution time with the Matlab version on a dual Intel Xeon system with 12 physical cores and 24GB RAM memory.

Figure 3 shows the relative speed-ups of four different C++ implementations against Matlab and their dependency on the number of CPU threads. All the C++ versions utilize the FFTW library compiled with OpenMP and SSE extensions under single precision. Matlab could use all CPU cores (12) and worked also with single precision in all cases. It can also be noted the server is equipped with the Intel Turbo technology raising the core frequency up to 3.2GHz under one thread workload and decreasing the frequency to 2.66GHz under full 12 thread load.

The C2C, naïve implementation represents the simplest implementation of the problem. Although very simple, it is able to outperform Matlab by about 26%. Operation fusion brings an additional significant improvement. Utilizing all 12 cores, the results are produced in 2.7 times shorter execution time. Replacing Complex-to-Complex (C2C) FFTs with the Real-to-Complex (R2C) ones and reducing some matrices sizes led to an additional reduction in execution time. This version of C++ code is up to 5.2 times faster than Matlab. Finally, revising all element-wise operations to exploit vector extensions of the CPUs and implementing basic NUMA policy, we reached speed-ups of 8.4 times.

Analysing and profiling the C++ code, we learn that nearly 58% of execution time is consumed by FFTs (see Table 1). The other operations take only a fraction of the time. Unfortunately, they cannot be optimized as one, because of intermediate FFTs.

For larger problems, the memory requirements of the complex-to-complex C++ and Matlab codes are very close. The reduction of memory requirements in the real-to-complex version is about 20% considering that most of matrices remained unchanged.

A real-world example has also been examined. The domain size was set to 768x768x256 grid points and 3000 time steps simulated. Matlab needed 27 hours and 11 minutes to compute the result and consumed about 17GB of RAM memory. C2C version with operation fusion took 8 hours and 16 minutes to complete the task and 16.8GB of RAM memory. R2C version finished after 4 hours and 55 minutes using 13.3GB of RAM. The final version of the code reduced the execution time to 3 hours and 22 minutes. Recalling our hypothetical simulation example mentioned earlier, this would decrease the computational time from 145 days to 17 days.

Another important observation is the execution time necessary to perform an iteration of the loop. Assuming the real-world simulation space size of 768x768x256, and 3000 time steps, every iteration takes about 4.1s. As it is not possible to execute multiple iterations at a time, this is the granularity of parallelisation. Moreover, during this time the entire 13GBs of memory will be touched at least once.

Naturally, the outputs from the C++ version and Matlab version have been cross-validated with relative error lower than 10^{-6} for the domain sizes up to 256^3 , and 10^{-4} for domain sizes up to 768x768x256 grid points.

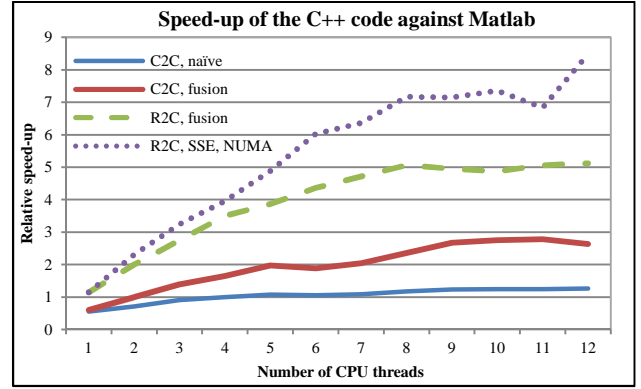


Figure 3: Relative speed-up of C++ against Matlab using a domain size of 256^3 and 1000 time steps.

% of time	Routine
30.84	Inverse FFT
26.73	Forward FFT
3.61	$BonA \cdot (\rho_{x0} + \rho_{y0} + \rho_{z0}) \cdot \Delta^2 / (2 \cdot \rho_{x0})$
3.46	Sum_subterms_on_line_12
3.10	$\rho_{x0} \cdot (du_{x0} + du_{y0} + du_{z0})$
2.81	$\rho_{x0} + \rho_{y0} + \rho_{z0}$
2.67	Compute_rhox
2.52	Compute_rhoy
2.42	Compute_rhoz
2.30	Compute_uy_sgy
2.16	Compute_uy_sgx
2.02	Compute_uy_sgz
15.36	Other operations

Table 1: Execution time composition of the C++ code.

5 Towards the Utilization of Multiple GPUs

In order to be able to solve real-world ultrasound propagation simulations in reasonable time, we need to reduce the execution time by an order of magnitude at least. For this reason we would like to utilize up to 7 GPUs placed in the Tian server to provide the necessary computational power as well as very high memory bandwidth.

First, we would like to start with one GPU and create a CUDA implementation of the simulation code. The most time consuming operations are the fast Fourier transformations. On the CUDA platform, the cuFFT library can be used. This library is provided directly by NVIDIA and runs on a single GPU (CUDA Math Libraries 2011). All other element-wise operations can be directly implemented as simple kernels, as the element-wise operations are embarrassingly parallel. On the other hand, these operations cannot benefit from the on-chip shared memory exploitation of which is often the key factor in reaching peak performance (Sanders and Kandrot 2010). This limitation can be partially alleviated by employing CUDA texture memory and its automatic caching.

There are a few strategies how to split the work among multiple GPUs. The obvious way is to calculate each dimension independently on a different GPU with the final pressure calculation performed on a single one. Looking at the listing shown in Figure 1, we can notice that nearly entire loop can be dimensionally decomposed. That is within each time step, the calculations for the x, y and z dimensions can be done independently. The only excep-

tion is line 12, where all three dimensions are necessary to compute the new pressure matrix. This could potentially utilize 3 GPUs for dimension independent calculations while only a single GPU for the final calculation.

Another strategy is to divide the computation of each operation among multiple GPUs. There is another reason to go this way. Utilizing only a single GPU or dimension partition scheme we are strictly limited by the GPU on-board main memory size, which is 1.5GB per GPU in our situation. This value is pretty small compared with 24GB of server main memory and does not allow us to treat larger simulation spaces. If we cut the loop into the smallest meaningful operations we would need two source 3D matrices and a destination one to reside in on-board GPU memory. This would allow us to solve problems with dimensions sizes up to 512^3 grid points in single precision. Our hypothetical example would be intractable because total memory required would be 1.7GB.

Dividing element-wise operations among multiple GPUs is straightforward. We can employ a farmer-workers strategy where a farmer (CPU) divides chunks of work to do. We can imagine a chunk as several rows of multiple 3D matrices that are necessary to compute several rows of a temporary result.

Currently, cuFFT does not run over multiple GPUs. Fortunately, the 3D FFT can be decomposed into a series of 1D FFTs calculated in the x, y and z dimensions and interleaved by matrix transpositions. Considering this, one possible scenario is that the CPU distributes batches of 1D FFTs over all 7 GPUs to compute the 1D FFT in the x dimension. Then a data migration is performed via CPU main memory or using the newly introduced CUDA peer-to-peer transfers followed by calculation of 1D FFTs in the y dimension etc. (An alternative strategy would be to use 2D FFTs on each GPU, with a transpose at the end of the 2D FFTs.)

As in many other distributed schemes, the overall performance will be highly limited by memory traffic, and in this case, also by the PCI-Express bandwidth. We must not forget that we will need to force tens of GBs through the PCI-Express which has a theoretical peak bandwidth of 8GB/s.

In order to gain necessary experience with our Tyan servers with 7-GPUs, we have designed several benchmarks to verify the key parameters of the servers such as PCI-Express bandwidth, zero-copy memory scheme, and peer-to-peer transfers among multiple GPUs. All these operations are going to be utilized in our future ultrasound code.

5.1 Peak PCI-Express Bandwidth with Respect to CPU Memory Allocation Type.

Having a good knowledge about PCI-Express characteristics, behaviour and performance is a key issue when designing and implementing GPGPU applications. As all data processed on the GPU (device) has to be transported from CPU (host) memory to device memory and the results back to the host memory to interpret on the CPU, PCI-Express can easily become a bottleneck debasing any acceleration gained using this massively parallel hardware. Considering the peak CPU-host memory bandwidth is 25GB/s and the peak GPU-device memory bandwidth

is 160GB/s, the theoretical throughput of PCI-Express x16 of 8GB/s is likely to be a place of congestion.

Any data structure (3D matrix or 1D vector in our case) designated for host-device data exchange has to be allocated on the host and device separately. Allocating memory on the device (GPU) side is easy as there is only one CUDA routine for this purpose. On the other hand, we need to distinguish between three different types of host memory allocation each intended for a different purpose:

- C/C++ memory allocation routines
- Pinned memory allocation with a CUDA routine
- Zero-copy memory allocation with a CUDA routine

C/C++ memory allocation routines such as `malloc` or `new` serve well for simple CUDA (GPGPU) applications. Their advantages are compatibility with non-CUDA applications and simple porting of C/C++ code onto the CUDA platform. However, using C/C++ memory allocation leads to PCI-Express throughput degradation caused by a temporary buffer for DMA introducing a redundant data movement in host memory. Moreover, only synchronous data transfers can be employed preventing communication-computation overlapping and sharing of host structures by multiple GPU and CPU cores.

A pinned memory allocation routine provided by CUDA marks an allocated region in host memory as non-pageable. This region is thus permanently presented in host memory and cannot be swapped onto disk. This enables Direct Memory Access (DMA) to this buffer, preventing any redundant data movement and allowing the buffer to be shared between multiple CPU cores and GPUs.

Zero-copy memory is a special kind of host memory that can be directly accessed by a GPU. No GPU memory allocations and explicit data transfers are needed any more. Data is streamed from host memory on demand. This is useful for GPU applications only reading input data or writing results once. However, this kind of memory allocation is extremely unsuitable for iteration-based kernels. It is important to note this has an impact on the ability of the CPU to cache this data and thus repeated accesses to the same data locations tend to be very slow. A possible scenario is that a CPU thread fills an input data structure for a GPU and never touches it again; the GPU reads it only once using zero-copy memory allowing a good level of computation and communication overlap.

Figure 4 shows the influence of host memory allocation type on the execution time needed to compute an element wise multiplication of 128M elements (512^3). First, three matrices are allocated on the host using a particular allocation type. After that, the matrices are uploaded into device memory (not in the case of zero-copy). Now, an element-wise multiplication kernel is run. The result is written into device memory and then transferred to host memory (not in case of zero-copy). The figure clearly shows the overhead of standard C memory allocation routines over the CUDA ones.

Zero-copy memory seems to be very suitable for our purposes. Although, the k -space method is iterative by nature, we are limited by the device memory size that does not allow us to store all global data (13GB) in

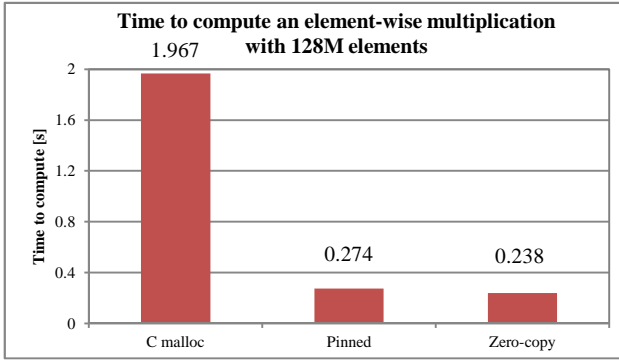


Figure 4: Time necessary to transfer two vectors of 128M elements to the GPU, perform element-wise multiplications, and transfer the resulting vector back to the CPU.

device memory even if we distribute the data over all 7 GPUs. Instead, we can leave some constant matrices in host memory and stream them to particular GPUs on demand. This perfectly suits line no. 6 in Figure 1, see below:

```
duxdx = real(ifftn( bsxfun(@times
    ddx_k_shift_neg, kappa .* fftn(ux_sgx))));
```

First, the 3D FFT of the matrix `ux_sgx` is calculated using a distributed version of `cuFFT`. The result is left in the device memory. Now we need to multiply the result of the forward FFT by the matrix `kappa`. As we need any element of `kappa` exactly once, there is no benefit in transferring the `kappa` matrix to the GPU. Instead, we could stream it from host memory using zero-copy memory. After that, we upload `ddx_k_shift_neg` vector into texture memory, because each element is read many times while expanding it to a 3D matrix on the fly and multiplying with the temporary result of the previous operation. Finally, the inverse 3D FFT is started using the data placed in the device memory.

5.2 Peak Single PCI-Express Transfer Bandwidth

Having chosen an appropriate memory allocation on the host side, we focused on measuring PCI-Express bandwidth between CPU and GPU taking into account different data block sizes starting at 1KB and finishing at 65MB. As the CPU has to serve multiple GPUs simultaneously, it is crucial to know the speed at which the CPU could feed the GPUs.

As the Tyan servers are special-purpose servers with a unique architecture using two IOH north bridges and PEX bridges, the peak bandwidth between the host and single devices was investigated in order to verify the throughput of different PCI-Express slots in both directions.

The experimental results are summarized in Figure 5. The measurements were repeated 100 times and averaged values were plotted. It can be seen that for small data blocks the PCI-Express bandwidth is degraded and reaches only a small fraction of the theoretical value of 8GB/s in one direction. In order to utilize the full potential of PCI-Express, data blocks with sizes of 500KB and larger

have to be transferred. The smallest chunk of data we can possibly upload to the GPU is one row of a 3D matrix, which for the size of interest represents 3KB. This is obviously too fine-grained a decomposition and we will have to send hundreds of lines in one PCI-Express transaction. This does not pose a problem, because a typical number of rows to process is in order of hundreds of thousands. The figure also reveals that device to host transfers are slightly faster than host to device ones.

A surprising variation in the peak bandwidth when communicating with different devices was observed. On one Tyan server, the first three GPUs are 2GB/s slower than the other four when transferring data from GPU to CPU memory. Although there are small oscillations from experimental run to experimental run, the results did not change significantly. We tried to physically shuffle the GPUs between slots but the results remained virtually unchanged. One explanation is that first three GPUs are connected to the Intel IOH chipset that is also responsible for HDDs, LANs, VGA, etc. On the other hand this does not explain the situation on the second Tyan server where GPUs 3, 4, 5 and 6 are significantly slower. Considering that both motherboards are the same, other peripherals should be connected to the same Intel IOH chipset. These results have also been cross validated with well-known SHOC benchmark proposed by Danalis et al. (2010).

5.3 Peak PCI-Express Bandwidth under Multiple Simultaneous Transfers

The second set of benchmarks investigates the PCI-Express bandwidth when communicating with multiple GPUs that is essential for work distribution over multiple GPUs. In all instances, pinned memory was used and four different transfer controlling (farmer) patterns considered:

- A single CPU thread distributes the data over multiple devices using synchronous transfer.
- Multiple CPU threads distribute the data over multiple devices using synchronous transfers. Each device is served by a private CPU thread.
- A single CPU thread distributes the data over multiple devices employing asynchronous transfers.
- Multiple CPU threads distribute the data over multiple devices by asynchronous transfers.

As each pair of GPUs share 16 PCI-Express links via a PEX bridge and different pairs are connected to different chipsets with NUMA architecture, we have investigated communication throughput in these configurations:

- (1) A pair of devices communicating with the host.
- (2) Two devices belonging to different pairs communicating with the host.
- (3) All even devices communicating with the host.
- (4) Two pairs of devices communicating with the host.
- (5) All seven devices communicating with the host.

The experimental measurements shown in Figure 6 demonstrate that a single CPU thread with synchronous transfers cannot saturate the PCI-Express subsystem of the Tyan servers; the peak bandwidth always freezes at the level of a single transfer. On the other hand, all remaining approaches are comparable, so there is no need to use multiple CPU threads to feed multiple GPU with data. We can employ the remaining CPU cores to work on tasks that are not worth processing on a GPU.

The second observation that can be made reveals the difference between the host to device and the device to host peak bandwidth. Whereas device to host transfers are limited by the 5.8GB/s, transfers managed by host scale up to 10.2GB/s (see Figure 6). We can conclude the device to host transfers are limited by the throughput of a single PCI-Express 16 channel while host to device by the QPI interconnection.

Table 2 presents the peak bandwidth in different configurations using one CPU thread and asynchronous transfers with respect to the numbering above. In all cases, device to host transfers cannot exploit the potential of the underlying architecture. In case (4), two different values were observed depending on the location of the pair. As we have mentioned before, the first three PCI-Express slots are slower than the other four. This leads to the fact that the first two pairs are slower than the other ones. The upper limit for host to device transfers lies around the 10GB/s level possibly limited by the QPI interconnection.

5.4 Peak Peer-to-Peer Transfers Bandwidth

One of the new features introduced in CUDA 4.0 is a peer-to-peer transfer. This feature enables Fermi based GPUs to directly access memory of another device via PCI-Express bypassing host memory. Data can be remotely read, written or copied. As peer-to-peer (p2p) transfers could serve the data exchange phase of distributed FFTs, we have investigated the performance of this technique and compared the results with user implemented device-host-device (d-h-d) transfers.

The Fermi GPU cards are only equipped with one copy engine, this device cannot act as source and destination of a peer-to-peer (p2p) transfer at the same time. Nevertheless, having seven GPUs we can create several scenarios where multiple devices are performing p2p transfers simultaneously. Also, we can use synchronous and asynchronous p2p transfers.

Figure 7 shows a comparison of p2p and d-h-d transfers running on two devices in different pairs, namely GPU 0 and GPU 1. We can see that the new p2p technique brings a significant improvement over the d-h-d transfer where the data has to be downloaded from the source device and, after that, uploaded on the destination device. The situation rapidly changes when performing multiple p2p transfers. The synchronous transfers become a bottleneck and asynchronous ones exploit more bandwidth. Figure 8 presents the performance of three simultaneous pairwise transfers (GPU 1 -> GPU 2, GPU 3 -> GPU 4, and GPU 5 -> GPU 6) where each device is either source or destination and all sources and destination are connected to different PEXs.

A d-h-d transfer in its asynchronous form consists of two phases. In the first phase, data packages are downloaded from all source devices and placed in host memory in asynchronous way. After synchronization the data packages are distributed over destination devices also in asynchronous way (more transfers at a time).

From the figure, CUDA 4.0 does not seem to be optimized for multiple simultaneous p2p transfers and user managed device-host-device transfers win. The difference is about 800MB/s. Taking into account this finding, it appears that it is better to implement highly optimized

Pattern	Host to Device	Device to Host
(1)	6GB/s	6.5GB/s
(2)	10GB/s	6.8GB/s
(3)	10GB/s	5.2GB/s
(4)	10GB/s	5.2 / 6.8GBs
(5)	10GB/s	5.4GB/s

Table 2: Peak bandwidth of multiple simultaneous transfers in different configurations.

device-host-device transfers that also involve CPU cores in data rearrangement and migration.

6 Discussion and Conclusion

This paper outlines our effort to migrate a compute intensive application of ultrasound propagation simulation to a cluster computer where each node has seven NVIDIA GPUs. The preliminary results from the CPU implementations have shown a speed-up of up to 8.4 compared to the original Matlab implementation. Given the computational benefits of using the k -space method compared to other approaches, this is a significant step towards creating an efficient model for large scale ultrasound simulation.

As the architecture of the Tyan 7-GPU server is not very common, we have examined a number of its specifications. We have designed several benchmarks that have revealed the behaviour of the PCI-Express subsystem.

In order to achieve the highest possible performance, we have to distribute the work over all seven GPUs. The CPU implementation of the code has revealed a low computation-memory access ratio. The asymptotic time complexity is only $O(n) = n \log n$. From the realistic experiments we found the CPU time for a single iteration is about 4.1s while global data of almost 13GBs has to be touched at least once.

Considering we could rework the code to access any element exactly once, and taking into account reachable CPU-GPU bandwidth, a naïve GPU based implementation would spend 2.1s or 1.3s distributing the data over one or multiple GPUs, respectively. Assuming all communication can be overlapped by computation using zero copy memory and the presence only one copy engine on a GPU, the realistic speed-up of a naïve implementation over a CPU one would be limited by 1.5 or 3.2 for one or multiple GPUs respectively.

On the other hand, if we accommodated all data in the on-board GPU memory we could reach much higher speed-up. Such an experiment has been carried out using a Matlab CUDA extension and a single NVIDIA Tesla GPU with 6 GB of memory and 448 CUDA cores. Using a domain size of 256^3 we have reached a speed-up of about 8.5 (compared to Matlab code), which is close to our CPU C++ implementation. Assuming we can optimize the GPU implementation in a similar way as in the CPU case, we may be able to improve on the Matlab CUDA code significantly.

The appropriate data distribution is going to play a key role in the application design. One way to reduce the data set is to calculate some matrices on the fly, exchanging spatial complexity for time complexity. Another possibility is to employ fast real-time compression and decompression of the data making the chunks smaller to transfer

through PCI-Express and between GPU on-board and on-chip memory. As many of the matrices are constant, the compression would have to be done only once. As long as we know that using asynchronous transfer one CPU core is sufficient to feed all seven GPUs, the remaining cores could execute other tasks that are not worth migrating to GPUs.

Data migration between GPUs will play another key role. Provided that we also need to perform data migration as a part of distributed FFT, we have revealed that the present CUDA 4.0 is not optimized for multiple simultaneous peer-to-peer transfers bypassing the host memory and thus, this communication pattern will have to be implemented as a composition of common device to host and host to device transfers.

7 Acknowledgments

This work was supported by the Australian Research Council/Microsoft Linkage Project LP100100588.

8 References

- Becker, D., Sterling, T., et al. (1995): Beowulf: A Parallel Workstation for Scientific Computation, *Proc. International Conference on Parallel Processing*, Oconomowoc, Wisconsin, 11-14.
- Kirk, D., and Hwu, W. (2010): *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann.
- Danalis, A., Marin, G., McCurdy, C., Meredith, J., Roth, P., Spafford, K., Tipparaju, V., Vetter, J (2010). The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. *Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processors (GPGPU 2010)*.
- Sanders, J. and Kandrot E. (2010): *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional.
- Treeby, B. E. and Cox, B. T. (2010): k-Wave: MATLAB toolbox for the simulation and reconstruction of photoacoustic wave fields. *Journal of Biomedical Optics*. **15**(2):021214.
- Treeby, B. E., Tumen, M. and Cox, B. T. (2011): Time Domain Simulation of Harmonic Ultrasound Images and Beam Patterns in 3D using the k-space Pseudospectral Method. *Medical Image Computing and Computer-Assisted Intervention*, **6891**(1): 369-376, Springer, Heidelberg.
- Tabeti M., Mast T. D. and Waag, R. C. (2002): A k-space method for coupled first-order acoustic propagation equations. *Journal of Acoustical Society of America*. **111**(1):53-63.
- Terboven, C., Mey, D., et.al. (2008): Data and Thread Affinity in OpenMP Programs. *Proceedings of the 2008 workshop on Memory access on future processors (MAW '08)*, New York, NY, ACM, 377–384.
- CUDA: Parallel computing architecture, NVIDIA http://www.nvidia.com/object/cuda_home_new.html, Accessed 15 Sep 2011.
- CUDA Math Libraries Performance 6.14, NVIDIA, <http://developer.nvidia.com/content/cuda-40-library-performance-overview>, Accessed 15 Sep 2011
- FFTW: Free FFT library, <http://www.fftw.org/>. Accessed 13 Sep 2011.
- Matlab: The Language of technical computing, MathWorks, <http://www.mathworks.com.au/products/matlab/index.html>, Accessed 15 Sep 2011.
- OpenMPI: Open Source High Performance Computing, The Open MPI project, <http://www.open-mpi.org/>, Accessed 15 Sep 2011.
- TYAN Computer: Tyan FT72B7015 server barebone, http://www.tyan.com/product_SKU_spec.aspx?ProductType=BB&pid=439&SKU=600000195, Accessed 15 Sep 2011.

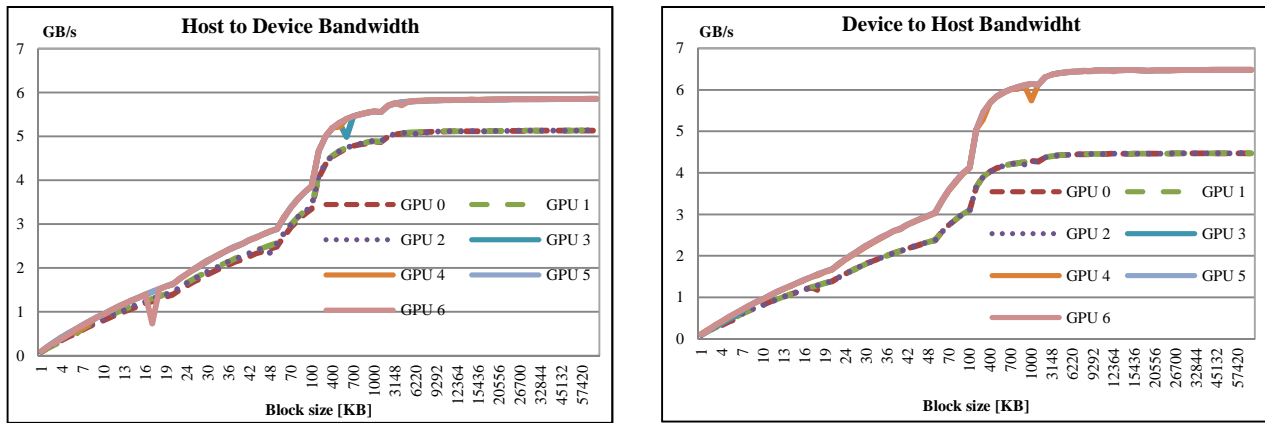


Figure 5: Peak bandwidth between host and a single device in both directions influenced by transported block size.

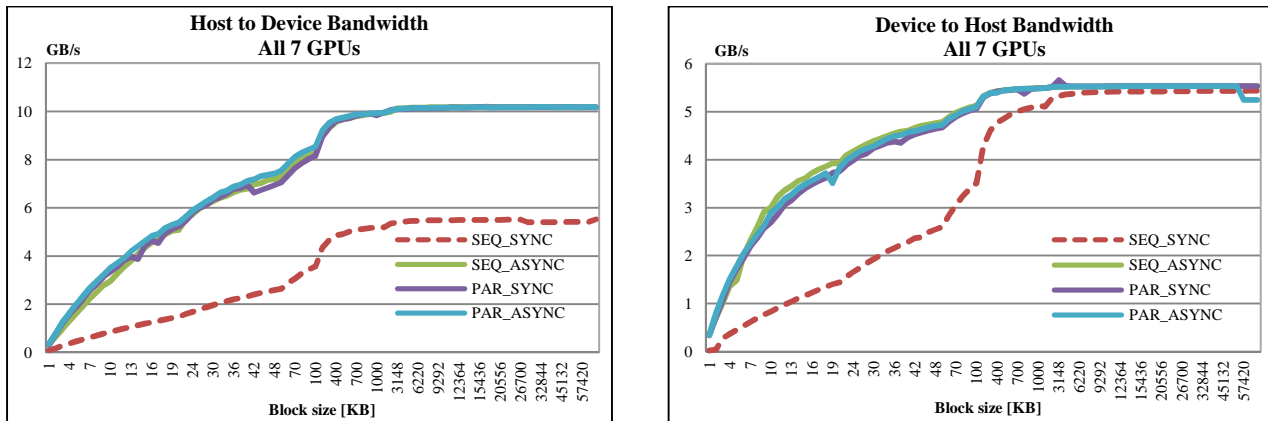


Figure 6: Peak bandwidth when host is communicating with all 7 GPUs in both directions.

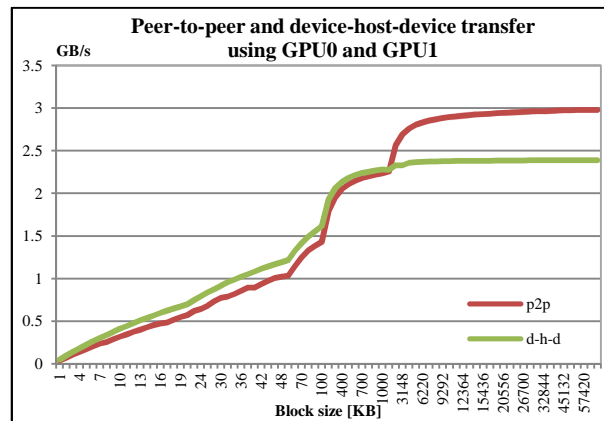


Figure 7: Peak bandwidth of a single peer-to-peer transfer and device-host-device transfer.

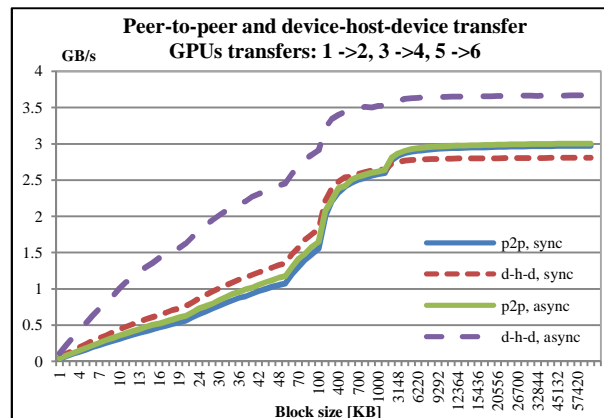


Figure 8: Peak bandwidth of multiple p2p and d-h-d transfers using three disjoint source-destination pairs.

Scaling Up Transit Priority Modelling Using High-Throughput Computing

Mahmoud Mesbah

School of Civil Engineering,
University of Queensland,
Australia

Mahmoud.Mesbah@uq.edu.au

Majid Sarvi

Dept. of Civil Engineering,
Monash University,
Australia

Majid.Sarvi@monash.edu

Jefferson Tan

Faculty of I.T.,
Monash University,
Australia

Jefferson.Tan@monash.edu

Fateme Karimirad

Dept. of Mechanical and
Aerospace Engineering,
Monash University,
Australia

Fatemeh.Karimirad@monash.edu

Abstract

The optimization of Road Space Allocation (RSA) from a network perspective is computationally challenging. An analogue to the Network Design Problem (NDP), RSA can be classified NP-hard. In large-scale networks when the number of alternatives increases exponentially, there is a need for an efficient method to reduce the number of alternatives while keeping computer execution time of the analysis at practical levels. A heuristic based on genetic algorithms (GAs) is proposed to efficiently select Transit Priority Alternatives (TPAs). The proposed framework allows for a TPA to be analysed by a commercial package that is a significant provision for large-scale networks in practice. We explore alternative parallel processing techniques to reduce execution time: multithreading and High-Throughput Computing (HTC). Speedup and efficiency are compared with that of traditional sequential GA, and we discuss both advantages and limitations. We find that multithreading is better when using the same number of processors, but HTC provides expandability.

Keywords: transport modelling, genetic algorithm, high-throughput computing, high-performance computing

1 Introduction

With ever-increasing travel demands, traffic congestion has become a challenge for many cities around the world. Construction of new roads or mass transit is not always possible, and reallocation of road space between transit vehicles and cars has emerged as a solution. Mesbah et al. (2011a, 2011b) proposed a bi-level optimization program for road space allocation (RSA). The objective was to identify the roads on which a bus lane should be introduced. The authors showed that this Mixed Integer Non-Linear (MINL) formulation is an NP-hard problem and is therefore computationally challenging. For large-scale networks, a heuristic approach is adapted to find reasonable solutions. This problem can be classified under the umbrella of Network Design Problems (NDP) that has a wide range of applications in Engineering. The network can be for roads, communication, power, water, or any network with a set of connected nodes and links.

The goal is to find the optimal combination of links to be added/modified to minimize a certain objective function.

The RSA problem is NP-hard, so the proposed optimization methods to large-scale problems requires extensive computational power, feasible with advanced techniques such as *High-Performance Computing* (HPC) (Strohmaier et al., 2005). While the term was applied broadly at first (Dongarra et al., 2005), HPC today typically applies to a tightly coupled system of many shared memory processors, particularly important when jobs must communicate among themselves. An alternative is *High-Throughput Computing* (HTC), aimed at providing large amounts of processing capacity taken together over a long period of time (Thain et al, 2005). *Many Task Computing* bridges the gap between HPC and HTC (Raicu et al., 2010), whether or not there are many long duration tasks, and regardless of the number of processors per computer. The common goal is to support simultaneous computations, where a long process is divided into small tasks, which are distributed across a set of interconnected processors to execute separately, simultaneously. Results are then gathered and combined. While HPC taken broadly may apply, the work described in this paper focuses on the HTC approach to distinguish the use of several independent computers on a network, as against our previous work using a single multiprocessor (Mesbah et al., 2011a). We demonstrate the application of HTC to solve a large-scale optimization problem in Transportation Engineering.

The proposed RSA is formulated as bi-level optimization. The upper level formulates an objective function and a set of constraints from the system managers' perspective. The lower level consists of user behavioural models, which requires a complex optimization program on its own. A number of commercial packages are available in order to analyse the user behaviour at the lower level, one of which is employed in this research. Many transport networks are already modelled in commercial packages, so there are benefits to sticking with them. Transport authorities have invested heavily in developing these models and already have confidence in their performance. Moreover, many transport planners are already trained to work with them. However, there are certain challenges in dealing with commercial applications such as we have had to do. We use a package called **Visum**. It requires Microsoft Windows, and uses a dongle for license management. The installer is more than 700MB, and requires interactive installation. While it can use multithreading on a machine with many processors (cores) and lots of memory, the

cost of such a machine can be prohibitive, and there are physical upper limits on cores and memory on any given machine. On the other hand, like other packages, it is not designed for HTC environments. Apart from a cluster, HTC can also be through a *computational grid*. This is an extensible aggregation of computational *resources*, such as clusters, belonging to independent organizations (Foster et al., 2001). Grids traditionally consist of Linux resources, while many engineering applications run on the Windows platform. Grids commonly support non-commercial applications with standard libraries provided almost out of the box, so a distributed execution of such applications is normally straightforward. RSA computations speed up if workload is distributed across such environments, but the nature of grids conflicts with the conditions for commercially licensed software. Licenses are typically limited to individual organizations while grids span across a *virtual organization* (VO) of several member organizations that remain autonomous. One cannot install or execute on just any resource, and such resources are normally not uniform anyway. We therefore have these three interesting challenges:

1. We use Visum, which requires Windows.
2. This is a commercial package and the source code is not accessible for reprogramming.
3. It must be pre-installed on each compute node with a large installer of over 700MB.

The proposed method can apply to many engineering applications where an iterative procedure is carried out using a commercial software package. A point we wish to make is that, despite the challenges, HTC can make many engineering applications scalable for large problems, even where the long runtime used to be a limiting factor.

The next section starts with a limited literature review on transit priority and continues with the bi-level optimization formulation. Then a solution algorithm is presented, based on a *genetic algorithm* (GA). It is implemented for (1) a single CPU on one machine, (2) multiple CPUs on one machine, and (3) multiple CPUs on multiple machines. Details are discussed subsequently, as is an example. In the last section, the results are discussed and the major findings are summarized.

2 Research Background

2.1 Road Space Allocation

The introduction of exclusive lanes to transit vehicles is one way to prioritize transit, an approach known as Road Space Allocation (RSA) (Black 1991, Currie et al., 2004). The literature on RSA can be classified into evaluation studies and optimization studies (see Figure 1).

Some evaluation studies focus on the local level, i.e. a link or corridor, e.g., Black (1991) presented a model on an urban corridor, evaluating several predefined scenarios based on total user travel time. Jepson and Ferreira (2000) assessed different road space priority treatments such as bus lane and setbacks based on delays in two consecutive links. Currie et al. (2007) considered a comprehensive list of impacts of RSA including travel time, travel time variability, initial and maintenance costs in a local priority project.

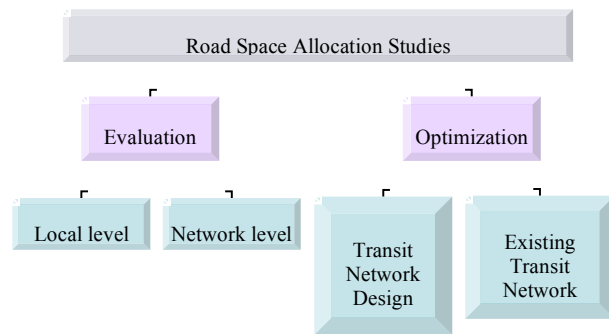


Figure 1. Classification of RSA studies.

Having compared performance measures in the literature, they proposed an approach to evaluate transit priority projects. Using the concept of intermittent bus lanes (Viegas 1996, Viegas and Lu 2004), Eichler and Daganzo (2006) suggested a new analysis method based on kinematic wave theory, which can be applied to a long arterial. At the network level, Bly et al. (1978) explored exclusive bus lanes to a link in different conditions, and the impact on the network was assessed using sensitivity analysis. Waterson et al. (2003) presented a macro-simulation approach which evaluates a given priority scenario at the network level. This approach considered rerouting, retiming, modal change, and trip suppression. Liu et al. (2006) proposed a similar approach with micro-simulation. Stirzaker and Dia (2007) applied micro-simulation to evaluate a major bus lane project in Brisbane. These studies evaluated a limited number of alternatives that do not necessarily include the best possible RSA over the network, and do not propose an optimization method to find the best set of bus lanes.

A number of studies have approached the problem using combined RSA optimization in Transit Network Design Problem (TNDP). Duff-Riddle and Bester (2005) applied a *trip focusing* process to design transit routes. The iterative method was able to put transit routes on the shortest travel time and shortest distance. The issue of express buses was also included with minute changes in the model. Chen et al. (2007) presented a design method in the form of a mathematical programming model. However, similar to Duff-Riddle and Bester (2005), the aim of their method was to design a new bus route.

Having first explored optimal TPAs in an existing transit network (Mesbah et al., 2008) with a general framework to find the optimal TPA at the network level, we have since then introduced a decomposition approach and a GA approach (Mesbah et al., 2011a, 2011b). This paper extends our work by employing HTC to reduce the runtime for large-scale transit networks.

2.2 High-Throughput Computing

HPC is a broad umbrella for a number of different environments (Strohmaier et al., 2005), but when performance is measured for many tasks across long periods of time, we may speak of high-throughput computing (HTC) (Thain et al., 2005). A neutral term bridging HPC and HTC is *many task computing* (MTC), with little distinction about the size of tasks (Raicu et al., 2010). Commodity computers can also be organized on high-speed networks. They are relatively low expenditure resources, compared to supercomputing facilities.

Beowulf-class clusters were probably the first (Sterling et al., 1998) of such environments, providing a queuing system for submitting and managing computational jobs. Another environment is the **Sun Grid Engine** (Gentzsch, 2001), and there are others, which uniformly share a preference for the UNIX or Linux environment.

Condor (Thain et al., 2005) uses computers that are normally used for other purposes, e.g., a desktop, and supports Windows nodes. Condor was originally dubbed “hunter for idle workstations” (Litzkow et al., 1988), i.e., when the user leaves the console for extended periods, e.g., after hours. This is the case for Monash University’s SPONGE resource, with up to 1000 cores running on computer laboratories across campuses during lean periods and after hours. While most nodes have two cores with modest memory, SPONGE collectively provides a considerable HTC resource.

3 Transit Priority Optimization

The RSA problem can be modelled as a ‘Stackelberg competition’ in which the system manager is the leader and transport users are followers (Simaan 1977, Bard and Falks 1982, Yang and Bell 1998, Liu et al., 2008). The system manager chooses a TPA, and in the subsequent system, users would choose their mode of travel and a path in order to maximize their own benefit.

The above design approach is formulated in this paper as a bi-level optimization program (Shimizu et al., 1997, Bard, 1998) (see Figure 2). At the upper level are the objective function and constraints from the system manager perspective. The upper level determines the TPA or the links on which priority would be provided for transit vehicles (decision variables). The aim of the upper level is to achieve System Optimal (SO) (Sheffi, 1984), thus the objective function includes a combination of network performance measures. The corresponding constraints are included in the upper level constraints.

The upper level can be formulated as follows:

$$\text{Min} Z = \alpha \sum_{a \in A} x_a^c(x) + \beta \left(\sum_{a \in B} x_a^b(x) + \sum_{l \in L} w_l^b \right) + \gamma \sum_{a \in A} \frac{x_a^c}{OCC^c} I_a Imp^c + \eta \sum_{a \in B} f_a s_a Imp^b \quad (1)$$

s.t.,

$$\sum_{a \in A_2} Exc_a \phi_a \leq Bdg \quad (2)$$

$$\phi_a = 0 \text{ or } 1 \quad \forall a \in A_2 \quad (3)$$

Variable definitions can be found in the annotation section. Note that $f_a = \sum_{p \in L} f_p \xi_{p,a}$, where $\xi_{p,a}$ is an element of the bus line-link incident matrix with $\xi_{p,a}=1$ if bus line p travels on link a and $\xi_{p,a}=0$ otherwise. The in-vehicle travel time is $t_a^b(x)$.

The first two terms in the objective function are the total travel time by car and bus. The next two terms represent the various other impacts of these two modes including emission, noise, accident, and reliability of travel time. The factors α , β , γ , and η not only convert the units, but also enable the formulation to attribute different relative weights to the components of the objective function (Mesbah et al., 2010). Equation (2) states that

the cost of the implementation should be less than or equal to the budget. The decision variable is ϕ_a by which the system managers try to minimize their objective function (Z). If $\phi_a=1$, then a bus lane is introduced on link a and buses can speed up to free flow speed, while the capacity of the link for cars is reduced from $Cpc_{0,a}^c$ to $Cpc_{1,a}^c$. If $\phi_a=0$, then buses will travel in the mixed traffic on a link with a capacity of $Cpc_{0,a}^c$. They are users who determine the link flows (x). Link flows are related to the decision variables by the lower level models.

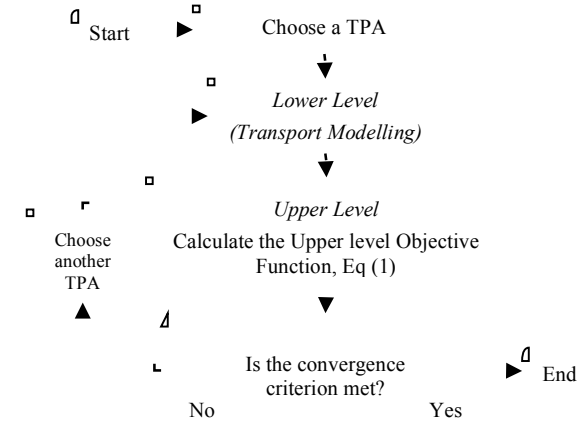


Figure 2. Outline of the proposed methodology.

At the lower level, it is the users’ turn to maximize their benefit. Based on the decision variables determined at the upper level, users make their trips. The traditional four-step method (Ortúzar and Willumsen, 2001) is adapted in this paper for transport modelling. It is assumed that the travel demand and the distribution of demand are not affected by the location of bus lanes (these conditions can be relaxed in future studies). Therefore, the origin-destination matrix remains constant. The lower level consists of three models: (1) modal split model, (2) traffic assignment model (car demand), and (3) transit assignment model (bus demand). Once the demand is determined, users choose their travel mode. Then, the car demand segment of the total demand is assigned to the network. The last step at the lower level formulation is the assignment of transit demand. Without loss of generality, in this study, a Logit model is used for the mode choice (Papacostas and Prevedourous, 1993), a User Equilibrium (UE) model is adapted for traffic assignment (Sheffi, 1984), and frequency-based assignment is applied to transit assignment (PTV AG, 2009). While these models are used for mode choice and assignment steps, the proposed HTC framework can be implemented by many other transport planning models. The lower level calculations are performed in Visum (PTV AG, 2009). As previously stated, many cities already use commercial packages. The proposed framework incorporates them instead of having to convert the models to other formats.

The bi-level structure, with a linear objective function and constraints, is NP-hard (Ben-Ayed and Blair, 1990). To complicate things further, the upper level objective function and the UE traffic assignment are non-linear. We employ a GA to find an approximate solution. The output

of the model is the combination of transit exclusive lanes which minimizes the proposed objective function.

4 The Genetic Algorithm Solution

A Genetic Algorithm (GA) is an iterative search method in which new answers are produced by combining two predecessor answers (Russell and Norvig 2003). Inspired from evolutionary theory in nature, the GA starts with a set of answers referred to as the *population*. Each individual answer in the population, a *chromosome*, is assigned a survival probability, based on the value of the objective function. The algorithm *selects* individual chromosomes based on this probability to breed the next generation of the population. GA uses *crossover* and *mutation* operators to breed the next generation, which replaces the predecessor generation. The algorithm is repeated with the new generation until a convergence criterion is satisfied. A number of studies applied GA to transit networks. Two recent examples are a transit network design problem considering variable demand (Fan and Machemehl, 2006) and minimization of transfer time by shifting time tables (Cevallos and Zhao, 2006).

In applying GA to the RSA problem, we define a gene to represent the binary variable ϕ_a , and a chromosome is the vector of genes (ϕ) which represents a TPA. A chromosome (TPA) contains a combination of links on which an exclusive lane may be introduced (set A_2). Therefore, the length of the chromosome is equal to the size of A_2 . The algorithm starts with an initial population with n chromosomes. The chromosomes of the initial population are produced randomly. When an initial chromosome population is produced, they are evaluated using the lower level models, i.e. the transport planning models of mode split, traffic assignment, and transit assignment. This evaluation is the time consuming component in the GA. Using the flow and travel time from the lower level, the values of the upper level objective function (Z) for all chromosomes are determined. Once the evaluated, the chromosomes are ranked from the lowest Z value to the highest. The fitness function, which determines the probability of a chromosome selection for breeding, is assumed to be an arithmetic series with the highest probability assigned to the top chromosome. The probability of the top ranked chromosome is assumed to be $P(1) = a_0 + 1/n$ where a_0 is a constant and n is the population size. Subsequently, other terms can be calculated using $P(i > 1) = P(1) - \gamma \times i$ where γ is the reduction factor so that $\sum_{i=1}^n P(i) = 1$.

$$\sum_{i=1}^n P(i) = P(1) + \sum_{i=2}^n (P(1) - \gamma \times i) = 1$$

$$\Rightarrow \gamma = \frac{n \times P(1) - 1}{n - 1} = \frac{n \times a_0}{n - 1}$$

A one point crossover is used in all experiments. The mutation involves flipping the value of a gene from 0 to 1 or vice versa. When a chromosome is selected for mutation, one gene from each set of 5 to 8 genes are flipped. That is about 12 to 20 flips for a chromosome 100 genes long. A common convergence criterion adapted here is to terminate if the number of iterations exceeds a predetermined value (*maxg*) or if the best objective function value found remains constant for a

number of generations (m). The process above is summarized in this algorithm:

0. Initialization: Set iteration number (n) to 1, best solution value or *upper bound* (UBD) to ∞ . Set max generations (*maxg*), and number of generations with same UBD, m .
1. Generate initial population.
2. Evaluation: Calculate the objective function value for all chromosomes (or TPAs) in the population, using the transport planning models at the lower level.
3. Fitness: Determine survival probabilities (fitness) and update UBD.
4. Convergence: If $n > \text{maxg}$ or UBD is constant for m generations, then stop.
5. Reproduction: Breed a new generation by performing selection, crossover, and mutation. Go to Step 2.

5 Implementation of the Genetic Algorithm

The most computationally intensive part of the GA is Step 2 where TPAs are evaluated. One evaluation involves running the four-step modelling for a network, which may take as long as a few hours on a typical desktop. Furthermore, the GA requires a large number of TPA evaluations, depending on the number of decision variables and attributes, e.g., probabilities of crossover and mutation. At this point, we decompose the processes in order to execute them in distributed fashion. This approach significantly reduces execution time.

The steps of Genetic algorithm in terms of dependency of processes are of two types. First is the evaluation step (Step 2). The evaluation of an individual chromosome (or TPA) is independent of other chromosomes (or TPAs) in a generation, which gives us a number of processes that can be executed independently. The second part of the GA involves fitness, convergence, and reproduction (Steps 3 to 5). These steps integrate the individual evaluations of Step 2 where the processes are interdependent. On the basis of the dependency attribute, two variants of the GA are proposed in the literature (Haupt et al., 2004, Goldberg, 2002, Cantú-Paz, 2000): *serial* (SGA) and *parallel* (PGA). Figure 3 illustrates these two variants. In SGA, all processes are carried out in a sequence, which means that, in Step 2, evaluation of a chromosome is completed before the evaluation of another chromosome is started. Then Steps 3, 4, and 5 are completed to produce another generation and then we cycle back to Step 2 (Figure 3 (a)). However, in PGA, evaluations are performed simultaneously. Therefore, Step 2 is executed in parallel, which is then followed by Steps 3, 4, and 5 in a sequence (see Figure 3(b)). SGA is simpler to implement, and details are explained in the next section. For PGA, we use two techniques of implementation: multithreading with multiple cores on one machine or HTC over several machines in a network.

5.1 Parallel GA - Multithreading (MT)

An operating system (OS) creates *threads* to run software. To run multiple applications simultaneously, multiple threads can be processed at a time, i.e.,

multithreading, if the machine supports multiple cores (Akhter and Roberts 2006, Evjen, 2004). To implement PGA by multithreading, the architecture of Figure 3(b) is used. The number of threads is selected equal to the number of processing cores on a machine (say p) plus a main thread. The main thread is reserved to control the flow of the GA from the start to the end. The main thread performs the fitness, convergence, and reproduction steps. The remaining p threads are used to execute TPA evaluations (objective function). When a generation is produced (see Figure 3(b)), n TPAs are queued for evaluation. The first p jobs in the queue are assigned p available threads. Once these p TPAs are evaluated, the next p TPAs are assigned. The next generation is produced when all TPAs are evaluated.

The speedup achieved depends on the number of cores on a machine and the efficiency of the OS in supporting multithreading. We implemented multithreading in Windows since the TPAs are evaluated by Visum, which requires Windows. The latter is commonly criticized for its performance, but there will always be cases where performance declines when the number of threads exceeds the number of cores (Akhter and Roberts, 2006), regardless of the OS. In that case, the OS must time-share the limited cores among so many executing threads, and we incur “time slicing” overhead. Moreover, the maximum number of cores that can go into one machine is subject to space and temperature constraints. There can also be a limit to gains due to memory latency and cache conflicts (Athanasaki et al., 2008). There is thus a cap on the speedup in multithreading, and the cost of purchasing many cores and supporting hardware can be high.

However, with TPA evaluations performed with commercial software, multithreading saves considerably on *license* costs for some packages. For example, one

Visum license is sufficient for one multithreading execution of the entire model on one machine, but performance will be constrained to what that machine can deliver. The next section discusses our HTC approach to avoid some of the limits of multithreading, although it requires multiple licenses. Our implementations are in Visual Basic .NET environment in this study.

A distributed computing approach such as HTC schedules TPA evaluations to several nodes on a network, each node having its own set of cores and local memory. Therefore, there is less of a limit on the number of tasks that can be executed simultaneously, as the number of computers in a network is not so tightly bounded. The trade-off is the complexity of distributing the task to available computers in the network, manage the queue, data transfers, provide an inter-process message-passing system in some cases, then collect and integrate the results.

5.2 Parallel GA –HTC with Condor

In Figure 3(b), p out of n evaluations in a population can be run in parallel. The ideal case is when p is equal to n , which means all n evaluations are done at the same time. However, as mentioned earlier, the number of threads supported on a given machine is limited. There are a number of existing systems for these, such as Condor. It was originally developed to use computers during idle periods (Thain et al., 2005), but is now one of the most flexible and powerful HTC platforms. Computers participate within a Condor *pool*. Owners can configure nodes to donate only some of their time. For example, as in the case of Monash University, the SPONGE pool consists of nodes that run from computer labs.

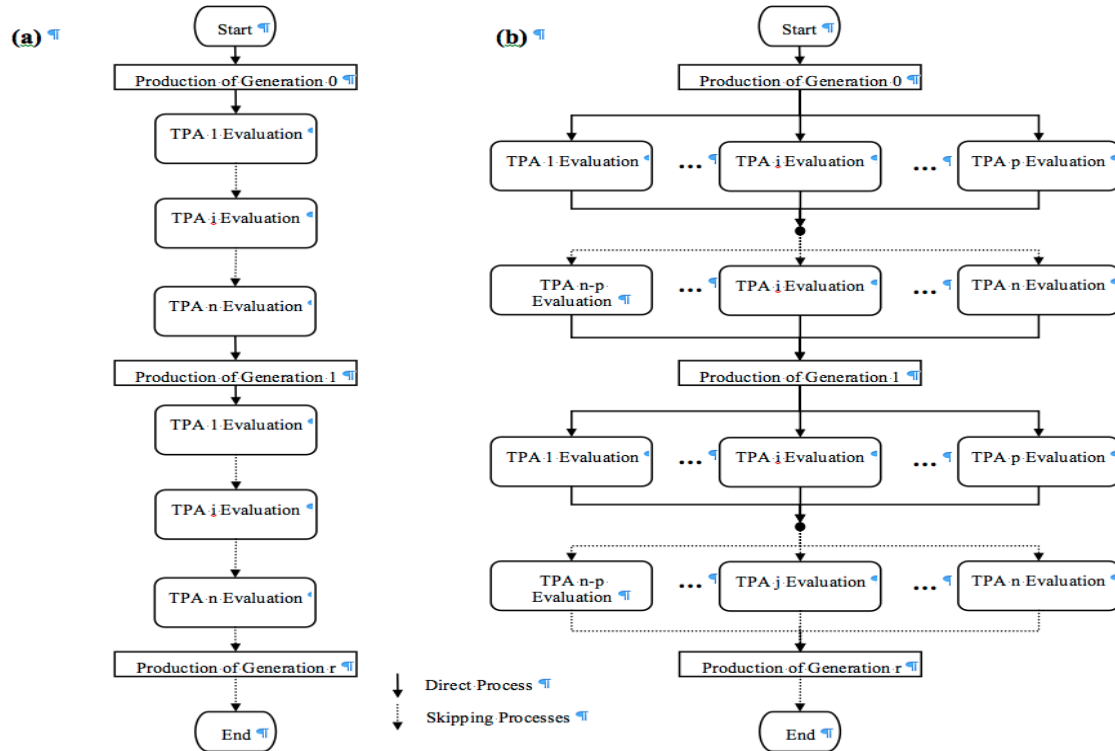


Figure 3. Sequence of components in serial genetic algorithm and parallel genetic algorithm.

They are only used when no one is currently using the desktop. These lab nodes are all running Windows XP or Windows 7, which works for us since our TPA evaluations are performed by Windows-based software. While issues emerging in adopting a general tool (HPC/HTC) tool to the RSA problem are tool-specific, important lessons can be learned. A license server restricts simultaneous runs of Visum with a hardware. The license server can run anywhere on the network, and need not be in the Condor pool. If x computers are in the network and y licenses are available, the maximum number of parallel TPA evaluations is $p = \min(x, y)$.

The parallel scheduling used in this HTC approach is to queue n TPA evaluations (the jobs) when a generation is produced (see Figure 3(b)). The jobs are assigned to the first set of available nodes. For instance, if $p < n$ nodes are available, p jobs are assigned and the remaining $n-p$ will wait in the queue. As soon as a job finishes on one node, the next queued job is assigned to that node. The next generation is produced when all TPAs are evaluated.

To evaluate the TPAs, a user submits jobs from the submission machine. For each job, Condor will copy input files and the objective evaluation program to the worker node and execute the program. Once completed, output data are copied back will be downloaded back to the submission machine. Some applications can be launched as a self-contained package, but Visum is not in that category. It requires interactive installation, with a 700-MB installer, which would require a considerable amount of time to copy to an execution node, even on a fast network. The solution was pre-installation of Visum on a subset of Sponge, where the owners were willing. Condor's scheduler must be told, upon submission, to send jobs only to nodes with Visum installed. This can be effected with Condor's *ClassAd* mechanism using custom *ClassAd* attributes, but in our implementation, we instead identified specific Visum-installed machines by name.

Windows differentiates between the local or remote launch of an application. Windows also consults the user permissions to run an application either locally or remotely. A *COM server* was configured to grant suitable permissions to launch Condor jobs from a remote user.

6 Numerical Example

Three GA implementations (SGA, PGA-MT, and PGA-HTC) are applied to an example transit network, the layout of which is in Figure 4. This grid network consists of 86 nodes and 306 links. All circumferential nodes together with Centroid 22, 26, 43, 45, 62, and 66 are origin and destination nodes. A 'flat' demand matrix of 30 persons/hr is traveling from all origins to all destinations. The total demand for all the 36 origins and destinations is 37,800 persons/hr. There are 10 bus lines covering transit demand in the network (see Figure 4). The frequency of service for the bus lines is 10 minutes. Parameters used are extracted from those calibrated for the Melbourne Integrated Transport Model (MITM), a four-step model used by the Victorian State Government for planning in Melbourne (Department of Infrastructure, 2004). Vertical and horizontal links are 400m long with two lanes in each direction and a speed limit of 36 km/hr. It is assumed that if an exclusive lane is introduced on a link on one direction, it may not necessarily be introduced

in the opposite direction. There are 120 links (uni-directional) in the network on which an exclusive lane can be introduced. These links are highlighted in black solid line. The following Akcelik cost functions (Ortúzar and Willumsen, 2001) are assumed for links with an exclusive lane (Equation (4)) and without (Equation (5)).

$$t_{1,a}^c = t_{0,a} + \frac{3600a}{4} \left[\left(\frac{x_a^c}{Cap_{1,a}^c} - 1 \right) + \sqrt{\left(\frac{x_a^c}{Cap_{1,a}^c} - 1 \right)^2 + \frac{8b}{ad} \left(\frac{x_a^c}{Cap_{1,a}^c} \right)} \right], t_{1,b}^c = t_{0,a} \quad (4)$$

$$t_{0,a}^c = t_{0,a}^b = t_{0,a} + \frac{3600a}{4} \left[\left(\frac{x_a^c + x_a^b}{Cap_{0,a}^c} - 1 \right) + \sqrt{\left(\frac{x_a^c + x_a^b}{Cap_{0,a}^c} - 1 \right)^2 + \frac{8b}{ad} \left(\frac{x_a^c + x_a^b}{Cap_{0,a}^c} \right)} \right] \quad (5)$$

where t_0 determines travel time with free flow speed, a is length of observation period, b is a constant, d is lane capacity, and other terms are as in the Section 8. Each link has 2 lanes, and:

$$a = 1hr, b = 1.4, d = 800veh/hr$$

$$Cap_{0,a}^c = 1800veh/hr$$

$$Cap_{1,a}^c = 900veh/hr$$

Mode share is determined using a Logit model. Traffic User Equilibrium (UE) and a frequency-based assignment is employed to model traffic and transit assignments, respectively. All these lower level transport models are implemented using Visum (PTV AG, 2009). The upper level objective function includes total travel time and total vehicle distance. The absolute value of the objective function can therefore be very large. A constant value is subtracted from the objective function value for all evaluations. Hence, the objective function value is relative. The weighting factors of the objective function are assumed to be 0.01. Regarding constraints the budget is assumed to allow for all candidate links for the provision of bus priority. The GA includes many parameters to tune. We suggest a particular set of values as a guideline in this example. It was assumed that population size, crossover probability (cp), and mutation probability (mp) are 40, 0.98, 0.01, respectively. The example demonstrates the HTC speedup compared to the serial approach. Although selection of the GA parameters may vary the absolute value of the execution time, the time differences on a relative basis are useful indicators to highlight the efficiency of the HTC approach. Table 1 describes seven computers we used in terms of the number of CPUs, versions of Windows, and of Visum. It demonstrates HTC incorporating diverse types of computers and software. Note that some processors can support two simultaneous threads per core. The first machine listed has four cores but can support eight threads, and perform up to eight TPA evaluations at a time. If all computers were allocated, 32 evaluations can be carried out simultaneously, requiring 32 licenses. The last column in Table 1 is the time spent evaluating one TPA on each machine. Machine 1 took the least time at 65 seconds, and Machine 7 was the slowest at 226 seconds. SGA, PGA by multithreading (MT), and PGA by HTC are explored.

□

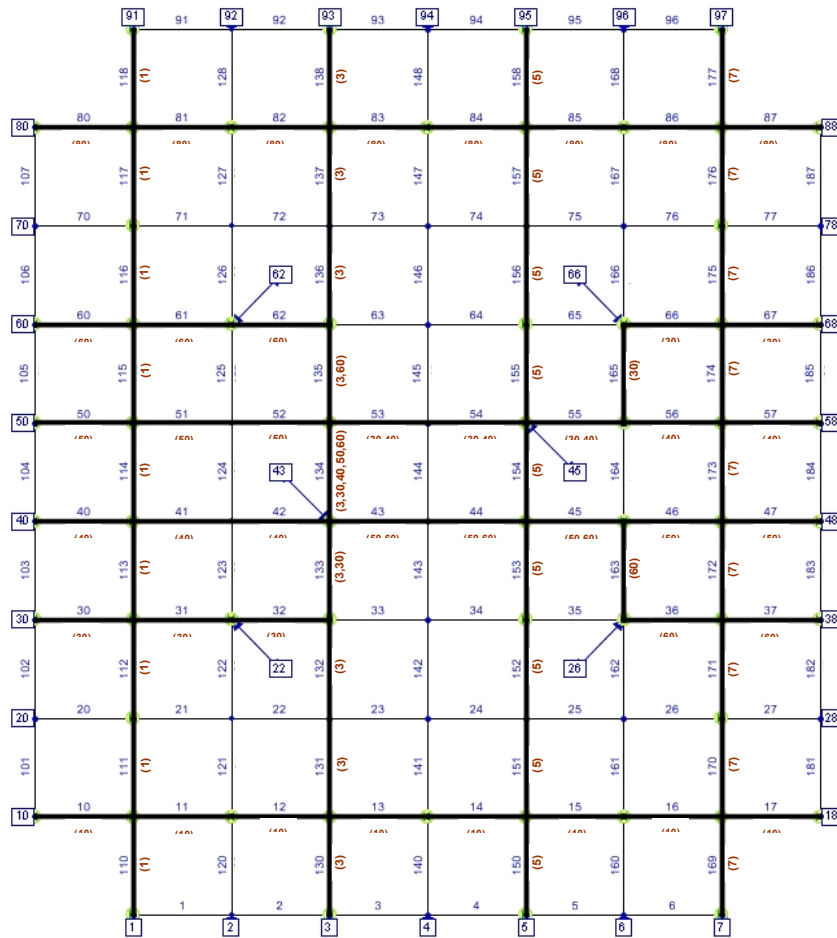


Figure 4. Example network with link numbers, origin destination nodes in boxes, and bus lines in parenthesis

Machine	CPU	Cores	Threads	Windows	Visum	Evaluation Time (s)
1	Intel Core i7 CPU 860 @ 2.8 GHz	4	8	7 64-bit	11.03 64-bit	65
2	Intel Core i7 CPU Q820 @ 1.73 GHz	4	8	7 64-bit	11.03 64-bit	147
3	Intel Core 2 Quad CPU Q6600 @ 2.4 GHz	4	4	7 64-bit	11.03 64-bit	101
4	Intel Core 2 Quad CPU Q6600 @ 2.4 GHz	4	4	XP 64-bit	11.01 32-bit	122
5	Intel Core 2 Quad CPU Q6600 @ 2.4 GHz	4	4	XP 64-bit	11.01 32-bit	121
6	Intel Core 2 Duo CPU E8500 @ 3.16 GHz	2	2	XP 64-bit	11.01 32-bit	88
7	Intel Pentium 4 CPU @ 3.2 GHz	2	2	XP 32-bit	11.01 32-bit	226

Table 1. Computers used in the experiments.

The base experiment (datum) for the MT approach is performed on Machine 4 with four threads, and for the HTC approach on Machines 1, 2, 3, and 6, with a total of 22 threads. The approach taken does not affect either the number of evaluations or the rate of improvement in the objective function. It does, however, affect the evaluation time. The minimum objective function value found in a run with 400 generations was -4.757.

The execution time of SGA is prohibitively long, being sequential. The number of generations was not carried past 300. All our four runs evaluated about 1700 TPAs each by the 50th generation. Although these runs do not follow exactly the same path in finding minimum, the trend shows that the value improves gradually at each successive evaluation. Figure 5 demonstrates the descent towards the minimum of the objective function value for two MT and two HTC runs. For comparison purposes, the SGA runs are also graphed. All approaches take the same downward trend to the minimum, but the implementation of the evaluation step results in different execution times. Three sets of experiments were organized with a population size of 40, crossover probability (*cp*) of 0.98, and mutation probabilities (*mp*) of 0.005, 0.01, and 0.02. The change in *mp* can change the number of evaluated TPAs. Figure 5 shows the quickest descent to the minimum of about 7.0 for HTC-1 and HTC-2 at about 100,000 seconds, with up to 32 simultaneous threads possible. MT-1 and MT-2 are not far behind at about 135,000 and 150,000 seconds, respectively, also to descend to a minimum of about 7.0. SGA runs went for much longer. For example, to reach a value of 30, SGA-3 takes about 170,000 seconds (two days) while HTC needs only 2,000 seconds. SGA-4 with 300 generations exceeded 5 days!

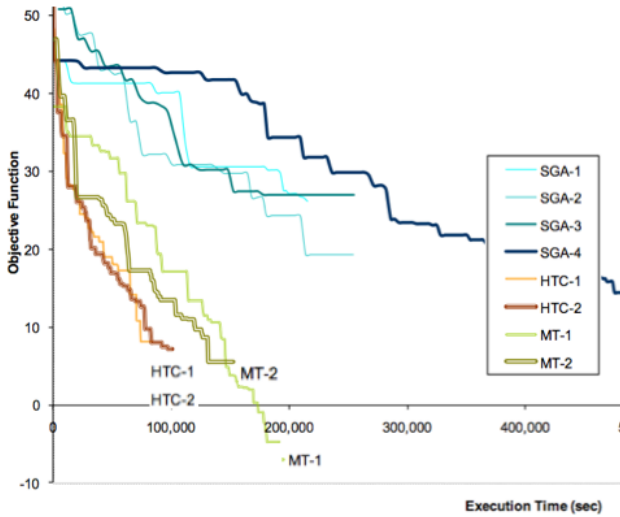


Figure 5. TPA evaluations in different modes.

Three measures were used in this study: (1) average time per evaluation (ATE), (2) speedup, which is the ratio of ATE in one run to the ATE of one SGA run, and (3) efficiency, which is the ratio of the speedup to the number of available threads. The speedup and efficiency of SGA runs are 1. Table 2 shows that ATE does not change significantly with mp . The number of cores are more significant, so the ATE for SGA, MT, and HTC runs are approximately 140, 40, and 14 seconds, respectively, where the number of cores are 1, 4 and 10, respectively. The efficiency measure demonstrates that in return for adding each thread in the MT approach, the execution time has improved by 80-90%. However, the efficiency in the HTC approach was just above 50% for the addition of each thread. There is considerable overhead incurred with distribution and queuing in HTC. Table 3 presents the effects of the number of available threads. Experiment E228 has the lowest ATE at 11.7 seconds. There are some important results in Table 3. The ATE did not improve when the number of threads went from 22 to 26. Experiment logs reveal that about 1650 TPA evaluations are performed in each run, to an average of 33 evaluations per generation. Nevertheless, this is not uniformly distributed. The TPA evaluations are recorded to prevent evaluating a TPA twice. Therefore, while the average number of evaluations per generation is 33, the first generations evaluate close to 40 (which is the population size) evaluations, while the last generations evaluate just over 20 TPAs. When close to 40 TPAs are being evaluated, both experiments E228 and E230 may allocate two or less evaluations to a thread. This means about two evaluations run in sequence. Similarly, when just above 20 TPAs are being evaluated, both E228 and E230 have enough threads to run all evaluations simultaneously. Therefore, an increase of four threads does not improve execution time. Accordingly, the ATE in experiment E231 should be similar to E228 and E230, but it increases instead. We added a very slow Machine 7 to the pool. In the time it takes for it to evaluate one TPA, other machines can evaluate between two to four. Machine 7 holds up the other available threads, extending the evaluation time of each generation.

7 Conclusions and Future Work

We presented a solution to Road Space Allocation using serial GA, parallel GA with multithreading, and parallel GA with HTC. The optimum was found regardless of the GA variant, but performance varied. PGA-MT with four threads reduced execution time by 3.2 to 3.7 times compared to SGA, and PGA-HTC with 18 threads by 9.3 to 9.8 times. MT is more efficient, but challenging to use for large-scale, realistic networks since the number of threads on a computer is generally constrained. In contrast, there is practically no limit in the HTC approach via incremental expansion.

A novel outcome is the successful implementation of HTC with commercial software on Windows. However, the overhead of pre-installed commercial software like Visum cannot be taken for granted. There is considerable benefit in grid computing, but it is not so accommodating to commercial packages. A logical follow-up is to explore *cloud computing* (Foster et al., 2008) with standard or custom settings and applications on the cloud resources. The framework is generic enough to apply to the entire family of Network Design Problems (NDPs). Applying the framework to NDP problems in large-scale networks can be a challenge. Moreover, substitution and comparison of other heuristic methods with the GA could be another area of future studies.

8 Notations

A : Set of all links in the network, $A = A_1 \cup A_2$

A_1 : Set of links in the network where provision of priority is impossible,

A_2 : Set of links where the provision of priority (introducing exclusive lane) is possible,

B : Set of links with a bus line on them, walking links, and transfer links,

L : Set of bus lines,

f_a : Sum of frequency of service for bus lines on link a ,

f_p : Frequency of service for bus line p ,

l_a : Length of link a ,

n : GA Population size

s_a : Bus service time on link a which is equal to running time plus dwell time at stops,

$t_{0-1,a}^{c,b}(x)$: Travel time on link a by mode car (c) or bus (b), which is a function of flow, with no exclusive lane (0), with exclusive lane (1)

$x_a^{c,b}$: Passenger flow on link a by car (c) or bus (b),

w_i^b : Waiting time and transfer time at stops.

Bdg : Available budget,

mp	Experiment Code	Approach	Number of Evaluations on Generation 50	Execution Time (sec)	Average time per evaluation	Number of Cores	Number of Threads	Speed up	Efficiency
0.005	E218	SGA	1649	240618	145.9	1	1	1	1
0.005	E220	MT	1513	59865	39.6	4	4	3.687	0.922
0.005	E219	HTC	1454	21607	14.9	10	18	9.821	0.546
0.01	E210	SGA	1680	241475	143.7	1	1	1	1
0.01	E223	MT	1543	66480	43.1	4	4	3.335	0.834
0.01	E227	HTC	1626	24918	15.3	10	18	9.378	0.521
0.02	E215	SGA	1721	231197	134.3	1	1	1	1
0.02	E224	MT	1714	72237	42.1	4	4	3.187	0.797
0.02	E214	HTC	1683	24204	14.4	10	18	9.343	0.519

Table 2. Comparison of the speedup using MT and HTC approaches.

Experiment Code	No. of Cores	No. of Threads	Machine ID	Evaluations on Gen. 50	Exec. Time (sec)	ATE (sec)	Speedup	Efficiency
E210	1	1	4	1680	241475	143.7	1	1
E225	6	10	2, 6	1724	37296	21.6	6.643	0.664
E226	10	14	2, 4, 6	1481	28013	18.9	7.597	0.543
E227	10	18	1, 2, 6	1626	24918	15.3	9.378	0.521
E228	16	22	1, 2, 4, 6	1659	19335	11.7	12.331	0.560
E230	20	26	1, 2, 4, 5, 6	1614	19053	11.8	12.173	0.468
E231	22	28	1, 2, 4, 5, 6, 7	1645	26235	15.9	9.011	0.322

Table 3. Comparison of HTC speedup, varying cores.

$Cpc_{0-1,a}^{c,b}$: Capacity of link a for mode car (c) or bus (b) with no exclusive lane (0), with exclusive lane (1)

Exc_a : Cost of implementing an exclusive lane on link a ,

$Imp^{c,b}$: Aggregate weight of operation costs of a car (c) or bus (b) to the community including: emissions, noise, accident, and reliability impacts.

Occ^c : Average occupancy rate for the car mode,

$\alpha, \beta, \gamma, \eta$: Weighting factors to convert the units and adjust the relative importance of each impact in the objective function, $\alpha, \beta, \gamma, \eta \geq 0$,

ϕ_a : Equals to 1 if there is an exclusive lane on link a , 0 otherwise

9 Acknowledgment

We received generous support from PTV AG, the Monash e-Research Centre (MeRC), and the Australian Research Council (ARC) for partial support.

10 References

- Akhter, S. and Roberts, J. (2006): *Multi-core Programming: Increasing Performance through Software Multi-threading*, Intel Press.
- Athanasaki, E., Anastopoulos, N., Kourtis, K. and Koziris, N. (2008): Exploring the performance limits of simultaneous multithreading for memory intensive applications. *Journal of Supercomputing*, **44**:64-97.
- Bard, J. F. (1998): *Practical Bilevel Optimization : Algorithms and Applications*, Kluwer, Dordrecht, The Netherlands.
- Bard, J. F. and Falks, J. E. (1982): Explicit solution to the multi-level programming problem. *Computers and Operations Research*, **9**:77-100.
- Ben-Ayed, O. and Blair, C. E. (1990): Computational difficulties of bilevel linear programming. *Operations Research*, **38**(3):556-560.
- Black, J. A. (1991): Urban arterial road demand management - environment and energy, with particular reference to public transport priority. *Road Demand Management Seminar 1991*, Melbourne, Australia. Haymarket, NSW, Australia, AUSTROADS.
- Bly, P. H., Webster, F. V. and Oldfield, R. H. (1978): Justification for bus lanes in urban areas. *Traffic Engineering and Control*, Feb. 1978, **19**(2):56-59.
- Cantú-Paz, E. (2000) *Efficient and Accurate Parallel Genetic Algorithms*, Boston, Mass., Kluwer.
- Cevallos, F. and Zhao, F. (2006): Minimizing transfer times in public transit network with genetic algorithm. *Transportation Research Record*, **1971**:74-79.
- Chen, Q., Shi, F., Yao, J.-L. and Long, K.-J. (2007): Bi-level programming model for urban bus lanes' layout. *Int. Conf. on Transportation Engineering, ICTE 2007*, Chengdu, China, 394-399.
- Currie, G., Sarvi, M. and Young, B. (2007): A new approach to evaluating on-road public transport priority projects: Balancing the demand for limited road-space. *Transportation*, **34**:413-428.
- Currie, G., Sarvi, M. and Young, W. (2004) A comprehensive approach to balanced road space allocation in relation to transit priority. *83rd TRB annual meeting*. Washington DC, Transportation Research Board.
- Department of Infrastructure (2004): *Melbourne Multi-modal Integrated Transport Model (MITM), User Guide*.
- Dongarra, J., Sterling, T., Simon, H. and Strohmaier, E. (2005): High-performance computing: clusters, constellations, MPPs and future directions. *Computing in Science and Engineering*, IEEE Computer Society, **7**:51-59.
- Duff-Riddell, W. R. and Bester, C. J. (2005): Network modeling approach to transit network design. *Journal of Urban Planning and Development*, **131**:87-97.
- Eichler, M. and Daganzo, C. F. (2006): Bus lanes with intermittent priority: Strategy formulae and an

- evaluation. *Transportation Research Part B: Methodological*, **40**:31-744.
- Evjen, B. (2004): *Professional VB.NET 2003*, Indianapolis, IN, J. Wiley.
- Fan, W. and Machemehl, R. B. (2006): Optimal transit route network design problem with variable transit demand: genetic algorithm approach. *Journal of Transportation Engineering*, **132**:pp 40-51.
- Foster, I. T., Kesselman, C. and Tuecke, S. (2001): The anatomy of the Grid: enabling scalable virtual organizations. *Int. Journal of Supercomputer Applications*, **15**(3):200-222.
- Foster, I. T., Zhao, Y., Raicu, I. and Lu, S. (2008): Cloud computing and grid computing 360-degree compared. *Grid Computing Environments Workshop (GCE '08)*, 1-10.
- Gentzsch, W. (2001): Sun Grid Engine -- Towards creating a compute power grid. *First IEEE/ACM Int. Symp. on Cluster Computing and the Grid*, 35-36.
- Goldberg, D. E. (2002): *The Design Of Innovation: Lessons From and For Competent Genetic Algorithms*, Boston, Kluwer Academic Publishers.
- Haupt, R. L., Haupt, R. L. and Haupt, S. E. (2004): *Practical Genetic Algorithms*, NY, Wiley Interscience.
- Monash University: SPONGE – *Harvesting Spare CPU cycles*.
<http://www.monash.edu.au/eresearch/activities/sponge.html> (last visited 2011).
- Jepson, D. and Ferreira, L. (2000): Assessing travel time impacts of measures to enhance bus operations. Part 2: Study methodology and main findings. *Road and Transport Research*, **9**:4-19.
- Litzkow, M. J., Livny, M. and Mutka, M. W. (1988): Condor – a hunter for idle workstations. *8th Int. Conf. on Distributed Computing Systems*, 104-111.
- Liu, R., Van Vliet, D. and Watling, D. (2006): Microsimulation models incorporating both demand and supply dynamics. *Transportation Research Part A: Policy and Practice*, **40**:125-150.
- Liu, W.-M., Jiang, S. and Fu, L.-F. (2008): Bi-level program model for multi-type freeway discrete equilibrium network design. *Zhongguo Gonglu Xuebao/China Journal of Highway and Transport*, **21**:94-99.
- Mesbah, M., Sarvi, M. and Currie, G. (2008): A new methodology for optimizing transit priority at the network level. *Transportation Research Record: Journal of the Transportation Research Board*, **2089**:93-100.
- Mesbah, M., Sarvi, M. and Currie, G. (2011): Optimization of transit priority in the transportation network using a genetic algorithm. *IEEE Transactions on Intelligent Transportation Systems*, **12**:908-919.
- Mesbah, M., Sarvi, M., Currie, G. and Saffarzadeh, M. (2010): A policy making tool for optimization of transit priority lanes in an urban network. *Transportation Research Record*, **2197**:54-62.
- Mesbah, M., Sarvi, M., Ouveysi, I. and Currie, G. (2011): Optimization of transit priority in the transportation network using a decomposition methodology. *Transportation Research Part C: Emerging Technologies*, **19**: 363-373.
- Ortúzar, J. D. D. and Willumsen, L. G. (2001): *Modelling Transport*, Chichester NY, J. Wiley.
- Papacostas, C. S. and Prevedouros, P. D. (1993): *Transportation Engineering and Planning*, Englewood Cliffs, NJ, Prentice-Hall.
- PTV AG (2009): *VISUM 11 User Manual*, 11th ed. Karlsruhe, Germany.
- Raicu, I., Foster, I. T. and Zhao, Y. (2010) Many-task computing for grids and supercomputers. *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*, 1-11.
- Russell, S. J. and Norvig, P. (2003): *Artificial Intelligence: a modern approach*, Upper Saddle River, N.J., Prentice-Hall.
- Sheffi, Y. (1984): *Urban Transportation Networks: Equilibrium Analysis With Mathematical Programming Methods*, Englewood Cliffs, N.J., Prentice-Hall.
- Shimizu, K., Ishizuka, Y. and Bard, J. F. (1997): *Nondifferentiable And Two-Level Mathematical Programming*, Boston, Kluwer Academic Publishers.
- Simaan, M. (1977): Stackelberg optimization of two-level systems. *IEEE Transactions on Systems, Man and Cybernetics*, **SMC-7**:554-559.
- Sterling, T., Becker, D., Warren, M., Cwik, T., Salmon, J. and Nitzberg, B. (1998): An assessment of Beowulf-class computing for NASA requirements: initial findings from the first NASA workshop on Beowulf-class clustered computing. *Proceedings of IEEE Aerospace Conference*, **4**:367-381.
- Stirzaker, C. and Dia, H. (2007): Evaluation of transportation infrastructure management strategies using microscopic traffic simulation. *Journal of Infrastructure Systems*, **13**:168-174.
- Strohmaier, E., Dongarra, J. J., Meuer, H. W. and Simon, H. D. (2005): Recent trends in the marketplace of high performance computing. *Parallel Computing*, **31**:261-273.
- Thain, D., Tannenbaum, T. and Livny, M. (2005): Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, **17**:323-356.
- Viegas, J. (1996): Turn of the century, survival of the compact city, revival of public transport. In *Meersman, H. and Van De Voorde, E. (Eds.) Transforming the Port and Transportation Business*, Antwerp, Belgium.
- Viegas, J. and Lu, B. (2004): The intermittent bus lane signals setting within an area. *Transportation Research Part C: Emerging Technologies*, **12**:453-469.
- Waterson, B. J., Rajbhandari, B. and Hounsell, N. B. (2003): Simulating the impacts of strong bus priority measures. *Journal of Transportation Engineering*, **129**:642-647.
- Yang, H. and Bell, M. G. H. (1998) Models and algorithms for road network design: a review and some new developments. *Transport Reviews*, **18**:257-278.

Author Index

Cao, Dahai, 33	Phillips, Charles, 31
Chen, Jinjun, iii, 33	Playne, Daniel, 13
Dinneen, Michael J., 23	Ranjan, Rajiv, iii
Eblen, John, 31	Rendell, Alistair P., 43
Hawick, Ken, 13	Rogers, Gary, 31
Jaros, Jiri, 43	Sarvi, Majid, 53
Karimirad, Fatemeh, 53	Shang, Weijia, 3
Khosravani, Masoud: Wei, Kuai, 23	Steinbrecher, Johann, 3
Langston, Michael A., 31	Tan, Jefferson, 53
Liu, Xiao, 33	Treeby, Bradley E., 43
Mesbah, Mahmoud, 53	Yang, Yun, 33
	Yuan, Dong, 33

Recent Volumes in the CRPIT Series

ISSN 1445-1336

Listed below are some of the latest volumes published in the ACS Series *Conferences in Research and Practice in Information Technology*. The full text of most papers (in either PDF or Postscript format) is available at the series website <http://crpit.com>.

- Volume 113 - Computer Science 2011**
Edited by Mark Reynolds, The University of Western Australia, Australia. January 2011. 978-1-920682-93-4.
Contains the proceedings of the Thirty-Fourth Australasian Computer Science Conference (ACSC 2011), Perth, Australia, 17-20 January 2011.
- Volume 114 - Computing Education 2011**
Edited by John Hamer, University of Auckland, New Zealand and Michael de Raadt, University of Southern Queensland, Australia. January 2011. 978-1-920682-94-1.
Contains the proceedings of the Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia, 17-20 January 2011.
- Volume 115 - Database Technologies 2011**
Edited by Heng Tao Shen, The University of Queensland, Australia and Yanchun Zhang, Victoria University, Australia. January 2011. 978-1-920682-95-8.
Contains the proceedings of the Twenty-Second Australasian Database Conference (ADC 2011), Perth, Australia, 17-20 January 2011.
- Volume 116 - Information Security 2011**
Edited by Colin Boyd, Queensland University of Technology, Australia and Josef Pieprzyk, Macquarie University, Australia. January 2011. 978-1-920682-96-5.
Contains the proceedings of the Ninth Australasian Information Security Conference (AISC 2011), Perth, Australia, 17-20 January 2011.
- Volume 117 - User Interfaces 2011**
Edited by Christof Lutteroth, University of Auckland, New Zealand and Haifeng Shen, Flinders University, Australia. January 2011. 978-1-920682-97-2.
Contains the proceedings of the Twelfth Australasian User Interface Conference (AUIC2011), Perth, Australia, 17-20 January 2011.
- Volume 118 - Parallel and Distributed Computing 2011**
Edited by Jinjun Chen, Swinburne University of Technology, Australia and Rajiv Ranjan, University of New South Wales, Australia. January 2011. 978-1-920682-98-9.
Contains the proceedings of the Ninth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia, 17-20 January 2011.
- Volume 119 - Theory of Computing 2011**
Edited by Alex Potanin, Victoria University of Wellington, New Zealand and Taso Viglas, University of Sydney, Australia. January 2011. 978-1-920682-99-6.
Contains the proceedings of the Seventeenth Computing: The Australasian Theory Symposium (CATS 2011), Perth, Australia, 17-20 January 2011.
- Volume 120 - Health Informatics and Knowledge Management 2011**
Edited by Kerry Butler-Henderson, Curtin University, Australia and Tony Sahama, Queensland University of Technology, Australia. January 2011. 978-1-921770-00-5.
Contains the proceedings of the Fifth Australasian Workshop on Health Informatics and Knowledge Management (HIKM 2011), Perth, Australia, 17-20 January 2011.
- Volume 121 - Data Mining and Analytics 2011**
Edited by Peter Vamplew, University of Ballarat, Australia, Andrew Stranieri, University of Ballarat, Australia, Kok-Leong Ong, Deakin University, Australia, Peter Christen, Australian National University, Australia and Paul J. Kennedy, University of Technology, Sydney, Australia. December 2011. 978-1-921770-02-9.
Contains the proceedings of the Ninth Australasian Data Mining Conference (AusDM'11), Ballarat, Australia, 1-2 December 2011.
- Volume 122 - Computer Science 2012**
Edited by Mark Reynolds, The University of Western Australia, Australia and Bruce Thomas, University of South Australia. January 2012. 978-1-921770-03-6.
Contains the proceedings of the Thirty-Fifth Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 123 - Computing Education 2012**
Edited by Michael de Raadt, Moodle Pty Ltd and Angela Carbone, Monash University, Australia. January 2012. 978-1-921770-04-3.
Contains the proceedings of the Fourteenth Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 124 - Database Technologies 2012**
Edited by Rui Zhang, The University of Melbourne, Australia and Yanchun Zhang, Victoria University, Australia. January 2012. 978-1-920682-95-8.
Contains the proceedings of the Twenty-Third Australasian Database Conference (ADC 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 125 - Information Security 2012**
Edited by Josef Pieprzyk, Macquarie University, Australia and Clark Thomborson, The University of Auckland, New Zealand. January 2012. 978-1-921770-06-7.
Contains the proceedings of the Tenth Australasian Information Security Conference (AISC 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 126 - User Interfaces 2012**
Edited by Haifeng Shen, Flinders University, Australia and Ross T. Smith, University of South Australia, Australia. January 2012. 978-1-921770-07-4.
Contains the proceedings of the Thirteenth Australasian User Interface Conference (AUIC2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 127 - Parallel and Distributed Computing 2012**
Edited by Jinjun Chen, University of Technology, Sydney, Australia and Rajiv Ranjan, CSIRO ICT Centre, Australia. January 2012. 978-1-921770-08-1.
Contains the proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 128 - Theory of Computing 2012**
Edited by Julián Mestre, University of Sydney, Australia. January 2012. 978-1-921770-09-8.
Contains the proceedings of the Eighteenth Computing: The Australasian Theory Symposium (CATS 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 129 - Health Informatics and Knowledge Management 2012**
Edited by Kerry Butler-Henderson, Curtin University, Australia and Kathleen Gray, University of Melbourne, Australia. January 2012. 978-1-921770-10-4.
Contains the proceedings of the Fifth Australasian Workshop on Health Informatics and Knowledge Management (HIKM 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 130 - Conceptual Modelling 2012**
Edited by Aditya Ghose, University of Wollongong, Australia and Flavio Ferrarotti, Victoria University of Wellington, New Zealand. January 2012. 978-1-921770-11-1.
Contains the proceedings of the Eighth Asia-Pacific Conference on Conceptual Modelling (APCCM 2012), Melbourne, Australia, 31 January – 3 February 2012.
- Volume 131 - Advances in Ontologies 2010**
Edited by Thomas Meyer, UKZN/CSIR Meraka Centre for Artificial Intelligence Research, South Africa, Mehmet Orgun, Macquarie University, Australia and Kerry Taylor, CSIRO ICT Centre, Australia. December 2010. 978-1-921770-00-5.
Contains the proceedings of the Sixth Australasian Ontology Workshop 2010 (AOW 2010), Adelaide, Australia, 7th December 2010.