

CONFERENCES IN RESEARCH AND PRACTICE IN  
INFORMATION TECHNOLOGY

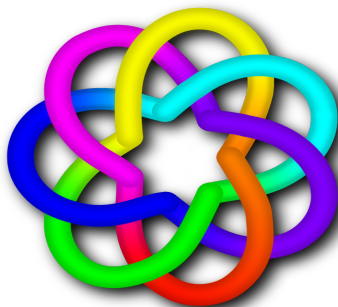
VOLUME 118

# PARALLEL AND DISTRIBUTED COMPUTING 2011

AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS, VOLUME 33, NUMBER 6



AUSTRALIAN  
COMPUTER  
SOCIETY



 **CORE**  
*Computing Research & Education*



# PARALLEL AND DISTRIBUTED COMPUTING 2011

Proceedings of the Ninth Australasian Symposium on  
Parallel and Distributed Computing  
(AusPDC 2011), Perth, Australia,  
17-20 January 2011

Jinjun Chen and Rajiv Ranjan, Eds.

Volume 118 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 2011.** Proceedings of the Ninth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia, 17-20 January 2011

**Conferences in Research and Practice in Information Technology, Volume 118.**

Copyright ©2011, 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 Information and Communication Technologies  
Swinburne University of Technology  
Melbourne, Victoria 3122  
Australia  
Email: [jchen@swin.edu.au](mailto:jchen@swin.edu.au)

**Rajiv Ranjan**

School of Computer Science and Engineering  
University of New South Wales  
Sydney, NSW 2052  
Australia  
Email: [rranjans@gmail.com](mailto:rranjans@gmail.com)

Series Editors:

Vladimir Estivill-Castro, Griffith University, Queensland  
Simeon J. Simoff, University of Western Sydney, NSW  
Email: [crpit@scm.uws.edu.au](mailto: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 118.  
ISSN 1445-1336.  
ISBN 978-1-920682-98-9.

Printed, January 2011 by University of Western Sydney, on-line proceedings  
Document engineering by CRPIT  
CD Cover Design by Dr Patrick Peursum, Curtin University of Technology  
CD Production by Snap St Georges Terrace, 181 St Georges Terrace, Perth WA 6000,  
<http://www.stgeorges.snap.com.au/>

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 Ninth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia, 17-20 January 2011

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 2011 Sponsors .....	xvi

## Contributed Papers

Speed and Portability Issues for Random Number Generation on Graphical Processing Units with CUDA and other Processing Accelerators .....	3
<i>Ken Hawick, Arno Leist, Daniel Playne and Martin Johnson</i>	
Data-Intensive Management and Analysis for Scientific Simulations .....	13
<i>Lynn Reid, John Norris, Randy Hudson, George Calhoun Jordan, Klaus Weide and Michael E. Papka</i>	
Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Grids .....	15
<i>Ghislain Charrier, Frédéric Desprez, Caniou yves</i>	
Parallel Vertex Cover: A Case Study in Dynamic Load Balancing .....	25
<i>Dinesh P Weerapurage, John D. Eblen, Gary Rogers, Michael A. Langston</i>	
A Parallel Approach to Social Network Generation and Agent-Based Epidemic Simulation .....	33
<i>Dimitri Perrin and Hiroyuki Ohsaki</i>	
Fast On-line Statistical Learning on a GPGPU .....	35
<i>Fangzhou Xiao, Eric McCreath, Christfried Webers</i>	
Author Index .....	43



## Preface

These proceedings contain the papers presented at the 9th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), held between 17-20 January 2011 in Perth, 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 this year, it is being referred to as AusPDC, has become the flagship symposium for Grid, Cloud, Cluster, and Distributed Computing research in Australia. In total, 14 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 three referees, and evaluated according to its originality, correctness, readability and relevance. A total of 6 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 ACSW organizing committee, those that submitted papers and those that attended the conference their work and contributions have made the symposium a great success.

**Jinjun Chen**

Swinburne University of Technology

**Rajiv Ranjan**

University of New South Wales

AusPDC 2011 Programme Chairs

January 2011

# Programme Committee

## Chairs

Jinjun Chen, Swinburne University of Technology, Australia  
Rajiv Ranjan, University of New South Wales, Australia

## Members

Jemal Abawajy, Deakin 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), Swinburne University of Technology, 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, University of New South Wales, 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

## Chair

Assoc. Prof. Mihai Lazarescu

## Co-Chair

Assoc. Prof. Ling Li

## Finance

Mary Simpson  
Mary Mulligan

## Catering and Booklet

Mary Mulligan  
Dr. Patrick Puersum  
Assoc. Prof. Mihai Lazarescu

## Sponsorship and Web

Dr. Patrick Puersum  
Dr. Aneesh Krishna

## Registration

Mary Mulligan  
Dr. Patrick Puersum

## DVD and Signage

Dr. Patrick Puersum  
Mary Mulligan

## Venue

Dr. Mike Robey

## Conference Bag

Dr. Sieteng Soh

# Welcome from the Organising Committee

On behalf of the Australasian Computer Science Week 2011 (ACSW2011) Organising Committee, we welcome you to this year's event hosted by Curtin University. Curtin University's vision is to be an international leader shaping the future through its graduates and world class research. As Western Australia's largest university, Curtin is leading the state in producing high quality ICT graduates. At Curtin Computing, we offer both world class courses and research. Our Computing courses cover three key areas in IT (Computer Science, Software Engineering and Information Technology), are based on the curricula recommendations of IEEE Computer Society and ACM, the largest IT professional associations in the world, and are accredited by the Australian Computer Society. Curtin Computing hosts a top level research institute (IMPCA) and offers world class facilities for large scale surveillance and pattern recognition.

We welcome delegates from over 18 countries, including Australia, New Zealand, USA, U.K., Italy, Japan, China, Canada, Germany, Spain, Pakistan, Austria, Ireland, South Africa, Taiwan and Thailand. We hope you will enjoy the experience of the ACSW 2011 event and get a chance to explore our wonderful city of Perth. Perth City Centre is located on the north bank of the Swan River and offers many fun activities and a wealth of shopping opportunities. For panoramic views of Perth and the river, one can visit Kings Park or enjoy a relaxing picnic in one of the many recreational areas of the park.

The Curtin University campus, the venue for ACSW2011, is located just under 10km from the Perth City Centre and is serviced by several Transperth bus routes that travel directly between Perth and Curtin University Bus Station, as well as several other routes connecting to nearby train services.

ACSW2011 consists of the following conferences:

- Australasian Computer Science Conference (ACSC) (Chaired by Mark Reynolds)
- Australasian Computing Education Conference (ACE) (Chaired by John Hamer and Michael de Raadt)
- Australasian Database Conference (ADC) (Chaired by Heng Tao Shen and Athman Bouguettaya)
- Australasian Information Security Conference (AISC) (Chaired by Colin Boyd and Josef Pieprzyk)
- Australasian User Interface Conference (AUIC) (Chaired by Christof Lutteroth)
- 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 Ker-ryn Butler-Henderson and Tony Sahama)
- Computing: The Australasian Theory Symposium (CATS) (Chaired by Taso Viglas and Alex Potanin)
- Australasian Computing Doctoral Consortium (ACDC) (Chaired by Rachel Cardell-Oliver and Falk Scholer).

The nature of ACSW requires the co-operation of numerous people. We would like to thank all those who have worked to ensure the success of ACSW2011 including the Organising Committee, the Conference Chairs and Programme Committees, our sponsors, the keynote speakers and the delegates. Many thanks go to Alex Potanin for his extensive advice and assistance and Wayne Kelly (ACSW2010 chair) who provided us with a wealth of information on the running of the conference. ACSW2010 was a wonderful event and we hope we will live up to the expectations this year.

**Assoc. Prof. Mihai Lazarescu and Assoc. Prof. Ling Li**

Department of Computing, Curtin University

ACSW2011 Co-Chairs

January, 2011



# CORE - Computing Research & Education

CORE welcomes all delegates to ACSW2011 in Perth. 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 ACSWin the mid 1990s now share this week with six other events - ACE, AISC, AUIC, AusPDC, HIKM, ACDC, which build on the diversity of the Australasian computing community.

In 2011, we have again chosen to feature a small number of plenary speakers from across the discipline: Heng To Shen, Gene Tsudik, and Dexter Kozen. I thank them for their contributions to ACSW2011. I also thank the keynote speakers invited to some of the individual conferences. The efforts of the conference chairs and their program committees have led to strong programs in all the conferences again, thanks. And thanks are particularly due to Mihai Lazarescu and his colleagues for organising what promises to be a strong event.

In Australia, 2009 saw, for the first time in some years, an increase in the number of students choosing to study IT, and a welcome if small number of new academic appointments. Also welcome is the news that university and research funding is set to rise from 2011-12. However, it continues to be the case that per-place funding for computer science students has fallen relative to that of other physical and mathematical sciences, and, while bodies such as the Australian Council of Deans of ICT seek ways to increase student interest in the area, more is needed to ensure the growth of our discipline.

During 2010, CORE continued to negotiate with the ARC on journal and conference rankings. A key aim is now to maintain the rankings, which are widely used overseas as well as in Australia. Management of the rankings is a challenging process that needs to balance competing special interests as well as addressing the interests of the community as a whole.

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 2010; in particular, I thank Alex Potanin, Jenny Edwards, Alan Fekete, Aditya Ghose, Leon Sterling, and the members of the executive and of the curriculum and ranking committees.

**Tom Gedeon**

President, CORE  
January, 2011

# 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.

**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

<b>ACDC</b>	Australasian Computing Doctoral Consortium
<b>ACE</b>	Australasian Computer Education Conference
<b>ACSC</b>	Australasian Computer Science Conference
<b>ACSW</b>	Australasian Computer Science Week
<b>ADC</b>	Australasian Database Conference
<b>AISC</b>	Australasian Information Security Conference
<b>AUIC</b>	Australasian User Interface Conference
<b>APCCM</b>	Asia-Pacific Conference on Conceptual Modelling
<b>AusPDC</b>	Australasian Symposium on Parallel and Distributed Computing (replaces AusGrid)
<b>CATS</b>	Computing: Australasian Theory Symposium
<b>HIKM</b>	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 2011 Sponsors

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



CORE - Computing Research and Education,  
[www.core.edu.au](http://www.core.edu.au)



Curtin University of Technology,  
[www.curtin.edu.au](http://www.curtin.edu.au)



Perth Convention Bureau,  
[www.pcb.com.au](http://www.pcb.com.au)



**AUSTRALIAN  
COMPUTER  
SOCIETY**

Australian Computer Society,  
[www.acs.org.au](http://www.acs.org.au)



# CONTRIBUTED PAPERS



# Speed and Portability issues for Random Number Generation on Graphical Processing Units with CUDA and other Processing Accelerators

K.A. Hawick

A. Leist

D.P. Playne

M.J. Johnson

Computer Science, Institute of Information and Mathematical Sciences,  
Massey University – Albany, North Shore 102-904, Auckland, New Zealand.  
Tel: +64 9 414 0800 Fax: +64 9 441 8181

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

## Abstract

Generating quality random numbers is a performance-critical application for many scientific simulations. Modern processing acceleration techniques such as: graphical co-processing units (GPUs), multi-core conventional CPUs; special purpose multi-core CPUs; and parallel computing approaches such as multi-threading on shared memory or message passing on clusters, all offer ways to speed up random number generation (RNG). Providing fast generators that are also portable across hardware and software platforms is non-trivial however, particularly since many of the powerful devices available at present do not yet support full 64-bit operations upon which many good RNG algorithms rely. We report performance data for a range of common RNG algorithms on devices including: GPUs; CellBE; multicore CPUs; and hybrids, and discuss algorithmic and implementation issues.

**Keywords:** Monte-Carlo simulation; random number generation; seed management; configuration management; portability.

## 1 Introduction

Quality Monte-Carlo simulation studies rely heavily on reliable and high-performance random number generators. Many application codes are still hand-crafted for specific scientific problems, especially in areas like computational physics. These are often necessary for studying problems that require many machine cycles to attain the required statistical accuracy. These sorts of problem are embodied by simulation problems like that of the Ising model (Hawick et al. 2009) of a magnet (as shown in figure 1) that is used to study critical phenomena and where a bias or correlation pattern in the random numbers employed leads to the wrong answer.

For such applications it is often important that the code be portable to support taking advantage of any and all computer cycles that are available on a wide variety of hardware and operating system platforms. There are a number of practical issues, not widely discussed in the literature, that are concerned with fast, reliable and portable random number generator algorithm implementations. This paper presents some of these issues, particularly with regard to different processing acceleration devices.

Copyright ©2011, Australian Computer Society, Inc. This paper appeared at the 9th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia.. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 118, J. Chen and R. Ranjan, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

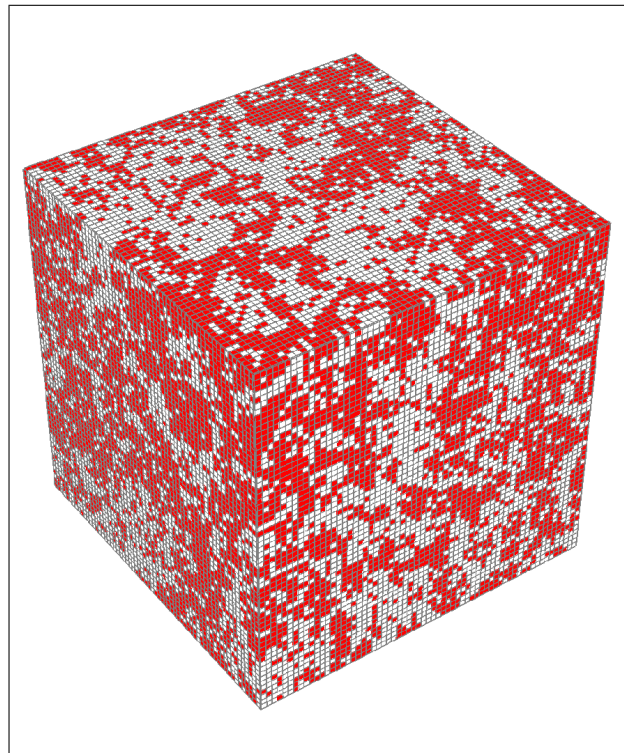


Figure 1: Monte Carlo generated Ising model on a  $64 \times 64 \times 64$  cubic mesh at its critical temperature.

Some relatively cheap accelerator devices such as Graphical Processing Units (GPUs), heterogeneous core processors such as the CellBE, or specialist processors such as Field Programmable Gate Arrays (FPGAs) or low power mobile devices like ARM, all offer potential price/performance advantages over conventional homogeneous core CPUs at the time of writing. Unfortunately many of these devices do not necessarily support full 64 bit operations, particularly for floating point algorithms. It is therefore interesting to consider what high-quality random number generator (RNG) algorithms can be deployed portably across a range of devices and what typical performance they can yield.

### 1.1 Historical Random Number Generation

Generating good quality fast random numbers (L'Ecuyer 2001, Brent 1997) on computers remains a long-standing challenge (Coddington 1994, Cuccaro et al. 1995). There is still an interesting algorithmic tradeoff space in which exist very high-quality generator algorithms such as the Mersenne-Twister (Matsumoto & Nishimura 1998)

that are significantly slower than those very-fast but lower-quality algorithms such as linear congruential generators. In between these extremes it is possible to improve low-quality generator algorithms by adding lag tables and shuffles tables to further randomise or decorrelate the sequences of random deviates and indeed to combine several independent algorithms. Random number generators are usually formulated in terms of mathematical recurrence relations (Johnsonbaugh 2001) whereby repeated application of a transformation will project a number to another in an apparently random or decorrelated sequence - at least to the extent that any patterns discernible in the resulting sequence are on a scale that is irrelevant to the application using them.

There are some philosophically deep questions concerning what it really means for a sequence of deviates to be truly random. For most scientific purposes it is sufficient to say that they need to be sufficiently uncorrelated that when used for a Monte Carlo simulation or other application the deviate quality does not lead to an observable bias (Knuth 1997). Or put more simply - that the random number generator does not lead the applications programmer to the wrong answer. Various statistical tests, both at a straightforward level (Coddington & Ko 1998) such as the spacing test, scatter-plots, that detect obvious patterns or simple statistics are possible, as well as very specific application related tests that are highly sensitive to correlations.

A related issue is the period length of the generator algorithm. A few deviates generated to make a game program behaviour "interesting" to a player does not require a generator with a challengingly long repeat length. However, Monte Carlo calculations that may take weeks or months of supercomputer resources must have generators with very long period lengths. In the last 20-30 years of steadily increasing supercomputer performance, there has been continued interest in ever longer period generator algorithms. This often ties in with the need for more bits used in the generator. The 16-bit integer based generators of the late 1970s, were superseded by 24-bit (floating-point) algorithms such as the Marsaglia lagged-Fibonacci algorithm (Marsaglia et al. 1987), by the 64-bit integer based Mersenne-Twister and in very recent times by 128-bit algorithms (Deng & Xu 2003) and even longer for cryptographically strong random number generation (Schneier 1996).

## 1.2 Generator Requirements

Randomness or lack of correlation amongst individual bits or patterns in the sequence of deviates is also very important. Some generators are known to have low correlation in some part of the generated bit patterns but not necessarily all, and therefore special operations can be used to only use those bit fields that are known to be decorrelated. Generally speaking if we have completely random bits we can generate random logical variables (obviously) but also integers and floating point to whatever precision we require. The reverse is not necessarily true and have a generator algorithm that produces a stream or sequence of integers or floating-point uniform deviates does not mean we can use them arbitrarily to reproduce random bits. A common target of many generator algorithms is to produce a sequence of random uniform floating point deviates - that is 32- or 64 bit floating point numbers on the range  $[0.0, 1.0)$ . A number of transformation algorithms (Hormann 1993) that can generate other statistically important distributions given a uniform stream of deviates are also well known. Some algorithms require particular low level data types to make them easily implementable. This can be an issue for

some programming languages (such as Java) that may not offer access to low level data field features such as unsigned integers (Coddington et al. 1999).

In many scientific programs that use random numbers, repeatability is important at least at the testing phase of a program. Deterministic testing using a completely repeatable and reproducible sequence of deviates is desirable for debugging a simulation program. Quantum physical devices are now available (ID Quantique White Paper 2010) that can inject a highly random (but irreproducible) stream of deviates into a calculation with some excellent non-correlation behaviour. However this is not always desirable for reproducibility purposes, and at the time of writing there is still a significant overhead in obtaining deviates from such devices as they are typically implemented as I/O or bus-based devices and are not yet integrated onto processing chips.

This gives rise to another important criteria for random number generators - ideally they should be well engineered in terms of having plug-compatible software programming interfaces. This means that a code can be tested and implemented using any number of different generator algorithms with little code change required. A further complication is that for many modern programs the random number generation must be part of a parallel computation (Coddington & Newall 2004, Newell 2003). This brings its own special problems concerned with ensuring independent processors have independent decorrelated streams of deviates, and it can make complete deterministic reproducibility impossible to guarantee without some sort of parallel synchronisation to avoid timing drifts between parts of a parallel computation. Some generator algorithms are more amenable to parallelisation than others depending upon the memory structure of the lag-table or whether the algorithm supports long sequence jumps that would allow separate processors to be initialised far apart in a shared (long) sequence.

In summary then, the field of computer generated random number algorithms is one of "horses for courses" - there is no single best algorithm that will satisfy all requirements. It is therefore of worth to review some algorithms in common use and their implementation on parallel computational systems and devices.

## 1.3 Paper Outline

In our present paper we give algorithmic details for implementing several different generator algorithms on different devices with various parallel programming models. In section 3 we illustrate some key algorithm implementations in CUDA and in other parallel frameworks including conventional multi core CPUs with both POSIX and Intel threading; single GPUs using a data-parallel strategy; multiple GPUs in a cluster; and CellBE processors. We also discuss how the algorithms were timed on the various platforms. In Sections 4.1 and 4.2 we present some detailed timings for various generators and discuss the implications in Section 5. Finally in Section 6 we offer some conclusions on good algorithmic choices for computational science applications and directions for future development of random number generators, given the current trends in parallel compute devices.

## 2 Accelerators Architectures

Rather than producing faster machines by simply making the Central Processing Unit faster and more powerful, architectures are being developed with more

specialised accelerators that can perform some specific computation much faster. We describe the architecture for two accelerators, Graphical Processing Units and Cell Broadband Engines.

Graphical Processing Units (GPUs) are proving to be very powerful processing accelerators for many scientific simulations and calculations and therefore in this paper we implement and test random number generators on GPUs using data-parallel techniques. We employ NVIDIA's compute unified device architecture (CUDA) programming language. CUDA supports very efficient code that fits the hardware, although the ideas and principles extend to the Open Compute language (OpenCL) specification (Khronos Group 2008) which is more widely supported by different vendors and devices. Some work has been done already on some generators for CUDA and GPUs (Langdon 2009, Giles 2009) and for other multicore parallel processors such as the STI Cell Broadband Engine (CellBE) (Bader et al. 2008) but with less emphasis on the topical issues of portability, performance tradeoffs and lack of 64-bit support.

### 2.1 Graphical Processing Units

Graphical Processing Units or GPUs have emerged in recent years as a very popular accelerator. This can be attributed to their reasonable prices, high computational throughput, common availability and relative ease of programming. Driven by the demands of modern 3D game graphics, both NVIDIA and ATI have developed highly parallel architectures to provide the processing required to supply these graphics in real-time. This architecture can be seen in Figure 2.

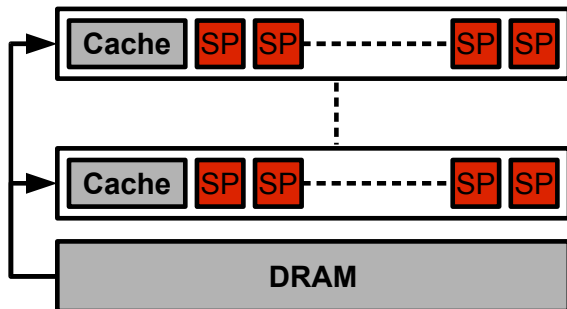


Figure 2: GPU architecture

GPUs contain many scalar processors (SPs) organised into multiprocessors (MPs). In the modern Fermi-based GeForce 400 series cards, each MP contains 32 SPs whereas in previous generations each MP contained only 8 SPs. GPU hardware can manage many thousands of threads as well as schedule and execute them on these multiprocessors. The multiprocessors can execute instructions independently from each other but the scalar processors within must execute the same instruction at the same time, this model is known as single instruction multiple thread (SIMT).

The main performance consideration for this architecture is how memory is accessed. All multiprocessors can access the main global memory (DRAM) of the GPU, however they also have some fast on-chip memory that the SPs within that multiprocessor can access. This allows SPs to reduce access to global memory and share information. Correct use of these on-board memory types has generally had the most impact on performance and has often been the main challenge of programming GPUs.

However, the release of Fermi-based GPUs has loosened the restrictions of global memory access.

These devices now have an automatic cache structure similar to that seen on most CPUs. This cache structure makes it easier to achieve high performance on such devices, while still giving the developer the option to fine-tune his code to explicitly use the fast on-chip cache where necessary.

It is often desirable to use multiple GPUs to achieve a higher computational throughput. Simply using multiple GPUs is relatively easy as each GPU connects to a host thread, however if they must communicate or share information it can become more of a programming challenge. GPUs cannot communicate directly and all information sent between them must go through the host CPU. This involves copying the information from the GPU to the host, exchanging it with the other host thread and then copying it to the other GPU.

### 2.2 Cell Broadband Engine

The Cell Broadband Engine (CellBE) has been less successful as an accelerator than the GPU but still represents an interesting target in terms of accelerator development. This architecture has one traditional processor (the PPE) with 8 less powerful cores (SPEs) that it can delegate tasks too. These cores are all connected to the main memory of the cell and exchange messages through the Element Interconnect Bus. This architecture is shown in Figure 3.

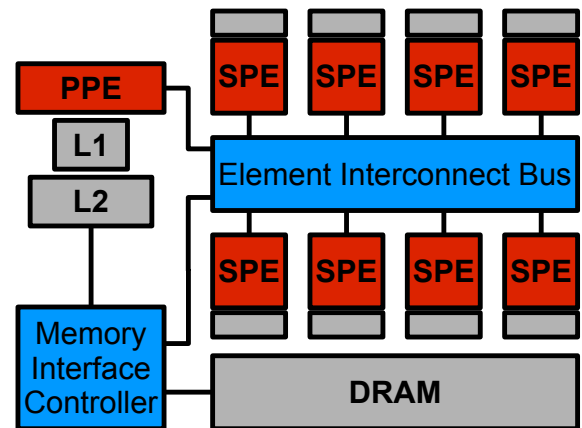


Figure 3: Cell Broadband Engine architecture

The major disadvantage encountered with the CellBE architecture was the programming API. CellBE applications require the programmer to manage the distribution of tasks and the exchange of data explicitly. Also to make full use of the CellBE processing power, the problem must be reworked to allow the SPEs to perform the computation in vector form. In many applications this can present a significant programming effort to rearrange the calculation to a vector form.

## 3 Random Number Generators

Random number generation is one of the most widely used facilities in computer simulations. A number of different algorithms are widely used (L'Ecuyer 2001, Marsaglia 1984), ranging from fast but low quality system supplied generators such as the `rand()`/`random()` generators available on Unix (BSD 1993) systems to slower but high quality 64-bit algorithms such as the Mersenne Twister generator (Matsumoto & Nishimura 1998). Marsaglia's lagged-Fibonacci generator (Marsaglia et al. 1987) is a 24-bit algorithm that produces good quality uniform deviates and which has been widely used in Monte

Carlo work(Binder & Heermann 1997). It is convenient for our purposes in this present paper as not all our target accelerator hardware platforms uniformly support 64-bit floating point calculations.

### 3.1 CPU - Sequential Algorithm

The Marsaglia lagged-Fibonacci random number generator (RNG) has been described in full elsewhere(Marsaglia et al. 1987), but in summary the details are given in Algorithm 1, which we provide for completeness.

---

**Algorithm 1** Marsaglia Uniform Random Number Generator, where an initialisation procedure sets the values as given below, and fills the lag table with deviates. The *id* and **signal** are not required for the sequential algorithm, but are used by the pThreads implementation described below.

---

```

function generate(id, seed)
  declare u[97]
  declare i ← 96
  declare j ← 32
  declare c ← 362436.0/16777216.0
  declare d ← 7654321.0/16777216.0
  declare m ← 16777213.0/16777216.0
  initialise(u, seed)
  for n ← 1 to N do
    uniform(i, j, c, d, m, u)
  end for
  signal complete(id)

```

---



---

**Algorithm 2** Marsaglia Uniform Random Number Generator, each call will generate a single random number.

---

```

function uniform(i, j, c, d, m, u)
  declare result ← u[i] − u[j]
  if result < 0 then
    result ← result + 1
  end if
  u[i] ← result
  i ← i − 1
  if i < 0 then
    i ← 96
  end if
  j ← j − 1
  if j < 0 then
    j ← 96
  end if
  c ← c − d
  if c < 0 then
    c ← c + m
  end if
  result ← result − c
  if result < 0 then
    result ← result + 1
  end if
  return result
end function

```

---

Where *i*, *j* index a lag table which is shown here of 97 deviates, but which can be any suitable prime, subject to available memory and where *c*, *d*, *m* are suitable values.

A number of optimisations for this sort of random number generation algorithm are possible on the various implementation platforms. One obvious one is to synchronise a separate thread that can produce an independent stream of random deviates that are consumed by the main application thread. Other algorithms, whose descriptions are beyond the space

limitations of our present paper, generate whole vectors or arrays of deviates together using a SIMD approach which can be used in applications that have similarly shaped work arrays or objects such as images or model data fields.

### 3.2 Multi-Core: POSIX Threads

The pThreads implementation of the lagged-Fibonacci generator launches multiple threads that each generate separate streams of random numbers. To do this each thread creates and initialises its own lag-table with a unique seed. The threads can then simply generate random numbers using this unique stream and the same **uniform** function as described in Algorithm 1.

Each thread that is created will generate *N* random numbers and then signal the main thread that it has completed its work. This code merely generates random numbers and does not make any use of them but it is assumed that any pThreads application that uses random numbers would make use of them within this thread loop.

### 3.3 Multi-Core: Threading Building Blocks

Like the pThreads implementation, the TBB implementation of the lagged-Fibonacci generator creates a number of independent RNG instances to generate streams of random numbers. However, the RNG instances are not associated with a particular hardware thread. Instead, they are each contained in a structure that can also store additional, application specific information related to the RNG instance. For example, it may also contain a pointer to an array that temporarily stores the generated deviates for later use, along with the array length. The structures are pushed into a vector after their RNG instances have been initialised. See Algorithm 3 for a description of this initialisation process.

---

**Algorithm 3** Initialising the TBB implementation of Marsaglia's random number generator. The parameters to the function are the seed *s*<sub>0</sub> and the desired number of RNG tasks *t*.

---

```

function initialise-tbb(s0, t)
  declare V //vector
  declare r0 ← new RngStruct
  initialise(r0, s0)
  for i ← 1 to t do
    declare ri ← new RngStruct
    declare si ← uniform(r0) * INT_MAX //seed
    initialise(ri, si)
    append ri at the end of vector V
  end for
  return V

```

---

The parallel random number generation using these RNGs is invoked by passing the begin and end iterators of the vector to TBB's **parallel\_for\_each** function, together with a pointer to a function that takes the structure type as its only argument. TBB applies the given function to the results of dereferencing every iterator in the range [begin,end). This is the parallel variant of **std::for\_each**.

The function called by **parallel\_for\_each** can then use the RNG instance passed to it to fill the array or array range specified in the same structure or to immediately use the random numbers in the application specific context. The process remains repeatable even though the thread that executes the function with a particular RNG structure instance as parameter can be different every time **parallel\_for\_each** is called.

TBB's task scheduler decides how many hardware threads are used and how they are mapped to the given tasks. While a larger number of RNG instances allows the code to scale to more processor cores, it also increases the overhead introduced by switching tasks. If there are no other processes or threads consuming a significant amount of processing resources, then setting the number of RNG instances equal to the number of hardware threads gives the highest and most consistent performance in our tests. If, however, other threads are using some processing power, too, then splitting the problem into a larger number of smaller tasks gives the task scheduler more flexibility to best utilise the remaining resources.

### 3.4 GPU - CUDA

The CUDA implementation of the lagged-Fibonacci random number generator is based on generating a separate stream of random numbers with every CUDA thread. This approach, referred to as CUDA 1, is repeatable and fast as race conditions are avoided and no communication between threads is required. Algorithms 4 and 5 illustrate the implementation of Marsaglia's algorithm in CUDA. A relatively small lag table should be used due to the memory requirements of this approach. The code example uses a table length of 97, which means 388-bytes for the table per thread. Other larger prime number sized tables can be used to improve the period at the expense of memory utilisation. The input seed value is used to initialise a random number generator (RNG) on the host, which is then used to generate the seeds for the CUDA threads. The CUDA implementations of the lag table initialisation and uniform random number generator functions are essentially the same as on the CPU, only that ternary expressions, which can be optimised by the compiler, are used to avoid branches and array indexing is adapted so that global memory accesses can be coalesced as long as the threads of a half-warp always request a new random number at the same time.

**Algorithm 4** CUDA implementation of Marsaglia's RNG that produces  $T$  independent streams of random numbers, where  $T$  is the number of threads. See Algorithm 5 for the CUDA kernel.

```

declare  $T = 30720$  //thread count
declare  $L = 97$  //lag table length
function RNG1( $s$ )
    Input parameters:  $s$  is the initialisation seed.
    declare  $S[T]$  //array of seeds
    initialise host RNG with  $s$ 
     $S \leftarrow$  generate  $T$  random deviates on the host
    declare  $S_d[T]$  in device memory
    copy  $S_d \leftarrow S$ 
    declare  $U_d[TL]$  in device mem. //lag tables
    declare  $C_d[T]$  in device mem. //array of  $c$  values
    declare  $I_d[T], J_d[T]$  in device mem. //indices
    do in parallel on the device using  $T$  threads:
        call KERNEL( $S_d, U_d, C_d, I_d, J_d$ )
    
```

A different approach has to be taken if a single sequence of random numbers is required. This approach, referred to as CUDA 2, only makes sense if most of the CUDA threads require the same number of random deviates and if giving the control back to the host before the next random number is needed does not come at a high cost or has to be done by the algorithm which consumes the random numbers anyway. The latter is necessary because this is the only way to synchronise across all CUDA threads. Algorithms 6 and 7 show how Marsaglia's algorithm can

**Algorithm 5** Algorithm 4 continued. The device kernel is the piece of code that executes on the GPU. The initialisation and uniform random number generator functions are essentially the same as on the CPU.

```

function KERNEL( $S, U, C, I, J$ )
    declare  $i \leftarrow$  thread ID queried from runtime
     $C[i] \leftarrow 362436.0/16777216.0$ 
     $I[i] \leftarrow L - 1$ 
     $J[i] \leftarrow L/3$ 
    declare  $s \leftarrow S[i]$  //load the thread's seed
    initialise the thread's RNG using  $s$ 
    generate random deviates when needed
    
```

be adapted to generate random numbers in parallel using a single, large lag table. This approach is based on the fact that the window between the table indices  $i$  and  $j$  is shifted by one every time a new random deviate is generated and that they start with an offset of  $\frac{2}{3}$  of the table size  $L$ . This means that  $\frac{1}{3}L + 1$  random numbers can be generated before index  $j$  reaches the starting index of  $i$ . It takes 3 iterations with either  $\frac{1}{3}L$  or  $\frac{1}{3}L + 1$  threads each to generate  $L$  random numbers, as the table length is a prime and therefore odd. The only value that changes every time a random number is generated is  $c$ , but this is not a problem as all future values can be calculated as shown in the code fragments. However, the values for  $c$  calculated in this way and thus the resulting random numbers are slightly different to those generated in the usual fashion due to floating point rounding errors. This means that in order to get the same results when running a simulation with the same seed multiple times, it is necessary to use the same RNG implementation every time and not use this CUDA implementation once and the CPU implementation the next time.

**Algorithm 6** CUDA implementation of Marsaglia's RNG that produces a single stream of random numbers using a large lag table. See Algorithm 7 for the CUDA kernel.

```

declare  $L = 92153$  //lag table length
declare  $T = L/3 + 1$  //thread count
declare  $D = 7654321.0/16777216.0$ 
declare  $M = 16777213.0/16777216.0$ 
function RNG1( $s$ )
    Input parameters:  $s$  is the initialisation seed.
    declare  $U[L]$  //the lag table
     $U \leftarrow$  initialise with  $s$ 
    declare  $U_d[L]$  in device mem. //the lag table
    copy  $U_d \leftarrow U$ 
    declare  $c \leftarrow 362436.0/16777216.0$ 
    while more random numbers required do
        //every iteration generates  $L$  required deviates
        declare  $o \leftarrow 0$  //offset into the lag table
        declare  $l \leftarrow L/3 + 1$  //update  $l$  table elements
        do in parallel on the device using  $T$  threads:
            call KERNEL( $l, o, U_d, c$ )
         $o \leftarrow o + l$ 
         $l \leftarrow \text{round}(L/3)$ 
        do in parallel on the device using  $T$  threads:
            call KERNEL( $l, o, U_d, c$ )
         $o \leftarrow o + l$ 
         $l \leftarrow L/3$ 
        do in parallel on the device using  $T$  threads:
            call KERNEL( $l, o, U_d, c$ )
         $c \leftarrow c - LD$  //update  $c$  for the next iteration
         $c \leftarrow c + \text{ceil}(\text{fabs}(c)/M)M$ 
    end while
    
```

The host code initialises the lag table before it is copied to the device. It then calls the CUDA kernel



**Algorithm 7** Algorithm 6 continued. The device kernel is the piece of code that executes on the GPU.

---

```

function KERNEL( $l, o, U, c$ )
  declare  $t \leftarrow$  thread ID queried from runtime
  if  $t < l$  then
    declare  $i \leftarrow L - 1 - t - o$  //index  $i$  into lag table
    declare  $j \leftarrow L/3 - t - o$  //index  $j$  into lag table
    if  $j < 0$  then
       $j \leftarrow j + L$ 
    end if
     $c \leftarrow c - (t + o + 1)D$  //calculate  $c$  for thread  $t$ 
    if  $c < 0.0$  then
       $c \leftarrow c + \text{ceil}(\text{fabs}(c)/M)M$  //until  $0 \leq c < 1$ 
    end if
    declare  $r \leftarrow U[i] - U[j]$  //new random deviate
    if  $r < 0.0$  then
       $r \leftarrow r + 1.0$ 
    end if
     $U[i] \leftarrow r$ 
     $r \leftarrow r - c$ 
    if  $r < 0.0$  then
       $r \leftarrow r + 1.0$ 
    end if
    do something with  $r$ 
  end if

```

---

3 times with different offsets into the lag table, generating  $L/3$  or  $L/3+1$  deviates in each call for a total of  $L$  new random numbers. With a lag table of length 92153 and a thread block size of 64, 30720 CUDA threads are executed in each call, 2–3 of which are unused.

Both CUDA implementations are mainly useful when the random numbers are consumed by other device functions, in which case they never have to be copied back to the host and often do not even have to be stored in global memory, but only exist in the local registers of the streaming multiprocessors. Lag table operations usually require global memory transactions, but if the conditions mentioned before are adhered, then all of these can be coalesced into 1 (approach 1) or 1–2 (approach 2) transactions per half-warp.

### 3.5 Multi-GPU - CUDA & POSIX Threads

The multi-GPU version of our approach to implementing Marsaglia's RNG in CUDA is basically the same as its single-GPU counterpart. One pThreads is created for every CUDA capable device in the system. These threads are used to control the CUDA kernel preparation and execution on the device associated to them. Instead of having to compute  $T$  random deviates as seeds for the thread RNGs, the host now has to generate  $T \times N$  seeds, where  $T$  is the number of threads per device and  $N$  is the number of devices. The multi-GPU implementation of Algorithm CUDA 2 does not distribute the lag-table across devices, but rather uses one lag-table per GPU.

### 3.6 Cell Processor - PS3

Implementing the lagged-Fibonacci generator on the Cell processor requires a certain deal of consideration. There are six separate SPEs each of which can process a vector for four elements synchronously. Vector types are used to make full use of the SPEs processing capabilities. Thus for each iteration, each SPE will generate four random numbers (one for each element in the vector).

To ensure that unique random numbers are generated, each element in the vector of each SPE must have a unique lag table. Six SPEs with four elements

per vector results in twenty-four lag tables. These lag tables are implemented as a single lag table of type `vector float` but each element of the vectors is initialised differently. Care should be taken when initialising these lag tables to make certain that the lag tables do not have correlated values and produce skewed results.

The lagged-Fibonacci generator algorithm has two conditional statements that affect variables of vector type. These conditional statements both take the form of `if( result < 0.0) result = result + 1.0;` (See Algorithm 1). As each element in the vector will have a different value depending on its unique lag table, different elements in the vector may need to take different branches.

---

**Algorithm 8** Pseudo-code for Marsaglia Lagged-Fibonacci algorithm implemented on the CellBE using vectors.

---

```

declare vector float  $u[97]$ 
initialise( $u$ )
declare  $i \leftarrow 96$ 
declare  $j \leftarrow 32$ 
declare  $c \leftarrow 362436.0/16777215.0$ 
declare  $d \leftarrow 7654321.0/16777215.0$ 
declare  $m \leftarrow 16777213.0/16777215.0$ 
function uniform()
  declare vector float  $zero \leftarrow \text{spu\_splats}(0.0)$ 
  declare vector float  $one \leftarrow \text{spu\_splats}(1.0)$ 
  declare vector float  $result \leftarrow u[i] - u[j]$ 
  declare vector float  $plus1 \leftarrow result + one$ 
  declare vector unsigned  $sel\_mask \leftarrow result > zero$ 
   $result \leftarrow \text{select}(result, plus1, sel\_mask)$ 
   $u[i] \leftarrow result$ 
   $i = i - 1$ 
  if  $i == 0$  then
     $i \leftarrow 96$ 
  end if
   $j = j - 1$ 
  if  $j == 0$  then
     $j \leftarrow 96$ 
  end if
   $c = c - d$ 
  if  $c < 0$  then
     $c \leftarrow c + m$ 
  end if
   $result \leftarrow result - \text{spu\_splats}(c)$ 
   $plus1 \leftarrow result + one$ 
   $sel\_mask \leftarrow result > zero$ 
   $result \leftarrow \text{select}(result, plus1, sel\_mask)$ 
  return  $result$ 
end function

```

---

There are two ways of overcoming this issue. The first method is to extract the elements from the vector and process them individually. This method is not ideal as it does not use the vector processing ability of the cell, instead the `spu_sel` and `spu_cmpgt` instructions can be used.

The `spu_cmpgt` instruction will compare two vectors (greater than condition) and return another vector with the bits set to 1 if the condition is true and 0 if the condition is false. The comparison is performed in an element-wise manner so the bits can be different for each element. The `spu_sel` can then select values from two different values depending on the bits in a mask vector (obtained from the `spu_cmpgt` instruction).

Using these two instructions the conditional statement `if( result < 0.0) result = result + 1.0;` can be processed as vectors with different branches for each element. The pseudo-code for this process can be seen in Algorithm 8.



## 4 Performance Results

We give some detailed performance results for the Lagged-Fibonacci generator running on multiple platforms (Section 4.1) as well as a study of different generator algorithms on GPU platforms (Section 4.2.)

### 4.1 Multi-Platform Lagged-Fibonacci Performance

The implementations of the lagged-Fibonacci generators on different architectures have been tested by generating 24 billion random numbers and measuring the time taken. Note that we do not store these deviates in memory as applications will typically consume them as they are generated. In the performance measures (See Table 1) the random numbers have not been used for any purpose as the only intention was to measure the generation time. This is obviously not useful in itself but it is assumed that any application generating random numbers such as these will make use of them on the same device as they were generated. Otherwise the random values can simply be written to memory and extracted from the device for use elsewhere.

Table 1: Comparison of the time taken to generate 24,000,000,000 random numbers using the lagged-Fibonacci generator on different hardware architectures. The CUDA measurements are done on a GeForce GTX295.

Device	Time (seconds)	Speed-up
CPU	256.45	1.0x
pThreads	66.72	3.8x
TBB	95.40	2.7x
Cell	23.60	10.9x
CUDA 1 (1 GPU)	8.56	30.0x
CUDA 1 (2 GPUs)	4.31	59.5x
CUDA 2 (1 GPU)	15.44	16.6x
CUDA 2 (2 GPUs)	8.33	30.8x

The platform we have used for all performance experiments except for the CellBE algorithms runs the Linux distribution Kubuntu 9.04 64-bit. It uses an Intel Core2 Quad CPU running at 2.66GHz with 8GB of DDR2-800 system memory and an NVIDIA GeForce GTX295 graphics card, which has 2 GPUs with 896MB of global memory each on board.

The platform used to run the CellBE implementations is a PlayStation 3 running Yellow Dog Linux 6.1. It uses a Cell processor running at 3.2GHz, which consists of 1 PowerPC Processor Element and 8 Synergistic Processor Elements, 6 of which are available to the developer. It has 256MB of system memory.

The results show that the concurrent implementations all perform well compared to the single-core CPU implementation. This comes as no surprise, as all threads and vector units execute independently from each another, using different lag tables and generating multiple streams of random numbers. The only exception to this is implementation CUDA 2, which generates a single stream of random numbers per GPU using a very large lag-table. The initial set-up time is insignificant compared to the time taken to generate 24 billion random numbers.

### 4.2 Multi-Algorithm GPU Performance

In this section we focus on the GPU and report on the relative performance of different algorithms and

the use of multiple GPUs. First we compare implementation CUDA 1 of the lagged-Fibonacci random number generator on different graphics devices, using 1, 2 or 3 GPUs and 3 different lag-table sizes to generate 24 billion random numbers in total. The results are given in Table 2. The algorithm scales almost linearly with the number of GPUs, which is not further surprising as the devices work independently and do not need to exchange any information. The GT200 series based devices show a significant performance drop when the lag-table size is increased, while the GTX480 is much less affected thanks to its improved memory hierarchy.

Table 2: Comparison of the time taken to generate 24,000,000,000 random numbers using implementation CUDA 1 of Marsaglia’s lagged-Fibonacci RNG with lag-tables of size 97, 1021 and 4093 on various CUDA devices. The timing results are reported in seconds.

Device	GPUs	Lag-table size		
		97	1021	4093
GTX260	1	7.67	7.98	8.91
GTX260	2	3.87	4.16	5.08
GTX260	3	2.69	2.97	3.85
GTX295	1	8.56	8.82	9.50
GTX295	2	4.31	4.55	5.27
GTX480	1	4.21	4.36	4.31
GTX480	2	2.12	2.17	2.19

The second performance comparison puts our implementation of Marsaglia’s RNG (CUDA 1) up against the algorithms Ran, Ranq1, Ranq2, Ranhash and Ranlim32 as described in Numerical Recipes 3rd edition (Press et al. 2007). The CUDA implementations of these RNG algorithms are straight forward and basically the same as the sequential CPU implementations. Each CUDA thread uses its own RNG instance and thus generates an independent stream of random numbers just like algorithm CUDA 1.

Two scenarios are used to compare the performance of these algorithms: **(a)** Generate 30.72 billion uniform deviates using 30720 threads and measure the execution time (lower is better); **(b)** Run an Ising simulation implemented in CUDA (Hawick et al. 2009) with 4096<sup>2</sup> cells for 16384 simulation steps and measure the hits per second (higher is better). The **Ranhash** algorithm is not well suited for the Ising simulation and has therefore not been used for those tests. Algorithm 9 describes how the different RNG implementations were tested for scenario **(a)**. The results are given in Table 3.

## 5 Discussion

As indicated in table 3 the generator algorithms we have employed can be implemented so that they provide broadly similar performance on a typical GPU. Other things being equal we therefore would be drawn to choose a quality algorithm that has been well tested, and employed and reported in the research literature. The Lagged-Fibonacci algorithm is our favourite for this purpose, but it can be configured with various different lag-table sizes to improve the deviate quality.

The lag-table size that we have employed for algorithms like the Lagged-Fibonacci generator has a relatively marginal effect in slowing down the GPUs. A larger table obviously requires greater processing, but the memory utilisation itself is more likely of greater

Table 3: The performance results for two test scenarios using different RNG implementations. Lower results are better in the first scenario and higher results are better in the second one. A GTX295 has been used for these measurement.

Performance Results	Ran	Ranq2	Ranq1	Ranhash	Ranlim32	Marsaglia
Generate $10^6$ uniform deviates per thread or 30.72 billion in total (seconds)	9.95	5.74	5.94	7.67	6.22	10.70
Ising simulation ( $10^9$ hits per second)	2.15	3.16	3.12	N/A	3.23	2.26

**Algorithm 9** This pseudo-code describes how the performance of the different RNG algorithms was measured. A RNG running on the host is initialised with a single seed and then used to generate the seeds for the CUDA RNGs, which are stored in  $s$ , before the CUDA kernel is called. `rng_params` is a place-holder for all algorithm specific parameters. Every thread sums the random numbers that it generates and finally stores the value to global memory to avoid code from being removed during the compiler’s optimisation phase.

---

```

tid ← thread ID queried from CUDA runtime
if tid < THREAD_COUNT then
  //init. the RNG stream with its individual seed
  initialise(tid, s[tid], rng_params)
  params ← load rng_params from global memory
  x ← 0.0
  for i ∈ {1, 2, 3, ..., 1000000} do
    x ← x + generate_uniform(params)
  end for
  rng_params ← save params to global memory
  results[tid] ← x //store x to global memory
end if

```

---

impact on simulations where GPU device memory is at a premium. This is particularly critical on the “gamer standard” GPUs we have employed in this work since current generation models typically have only around 1GByte of such device memory compared with “blade level” GPU products that have several times this amount.

The monotonic performance improvements we obtain on GPUs of increasing numbers of cores suggests an optimistic future for data parallelism on this architecture. AT the time of writing GPUs of approximately  $2^9$  cores are available and we believe the technology will readily support  $2^9$ - $2^{12}$  cores in the foreseeable future. We believe clusters using GPUs are already feasible, and it may even be beneficial to incorporate more than one GPU device per cluster node. This is certainly of use for applications where the work can be divided up into independent tasks. For other areas however, it may be more useful to allocate a specific accelerator device solely to producing random numbers. The CellBE architecture (if not this particular chip itself) shows some promise for that paradigm.

A particular application of interest for us is the Ising model(Onsager 1944) as described in (Hawick et al. 2009). It is notoriously difficult and computationally expensive to obtain high accuracy on critical parameters such as the critical temperature in the case of the three dimensional Ising system(Baillie et al. 1992). We are investigating how the critical temperature shifts when the underpinning lattice structure is changed according to a Watts-Strogatz re-wiring probability  $p$ (Hawick & James 2006). It is proving necessary to investigate  $p$  on logarithmic scales making the computational requirements even more severe. Using a portable generator such as the

Lagged-Fibonacci algorithm that works well on all platforms available to us is very important to the computational feasibility of such numerical simulation problems. The Ising model is so important, that we have in fact used “Ising model Monte-Carlo Hits per second” as a performance metric of the random number generator algorithms – as presented in table 3.

We have not reported on FPGA(Danese et al. 2007) performance data nor on low-power commodity mobile devices such as ARM processor(Sloss 2010). It is possible to devote programmable array die-space to 64-bit operations and some ARMs can indeed perform 64-bit floating point. At least at the time of writing and perhaps for some few years to come, the present transient generation of devices will not necessarily be able to perform 64-bit operations at a commodity price regime and therefore the issues we have discussed about portability on 32-bit devices will remain valid.

## 6 Conclusions and Future Work

We have explored the portability and performance of various random number generators on different accelerator devices using variety of parallel programming frameworks. The data-parallelism of the GPU architecture is particularly attractive for the Monte Carlo work we have focused on. While random number generation is surely still an area where the “horses for courses” argument applies, depending upon application context, we do believe the Marsaglia lagged-Fibonacci generator with suitable lag-table size is still a worthy portable candidate suited for use in quality Monte Carlo studies.

For future work we believe hybrid processor architectures such as the CellBE are interesting and will offer good specialist pipelining capabilities – such as generating random numbers. However at the time of writing we believe the software toolset available to help program such devices places quite high burdens on the applications programmer. While CUDA is not totally trivial it is certainly more application friendly, and we do expect its programming models to propagate further into systems like OpenCL. The number of cores available on GPUs continues to increase. There is scope for further work in tuning the thread block sizes to suit particular GPU devices with higher numbers of cores.

We have shown that some work is necessary to implement the various algorithms on different platforms but that CUDA’s similarities to C/C++ syntax does make this feasible. OpenCL holds some promise for portability although we have not reported on detailed performance data since at present the OpenCL platforms available to us are far out-stripped by CUDA. We do expect this situation will change as more vendors take OpenCL up. However, since RNGs tend to make use of low-level computational facilities such as bit operations, and would usually be written to take advantage of any vector or pipeline facilities available,

it is not clear whether OpenCL level portability will be enough.

In summary, we believe that at the time of writing, the Marsaglia Lagged-Fibonacci algorithm operating using single precision floating point is both portable across the devices we discuss and is readily implementable using the software technologies available - with suitable care and attention. The performance attainable multi-GPU data-parallel threads within the context of multiple conventional threads or processes of a multi-gpu cluster is particularly encouraging. This is our platform of choice for our current Monte-Carlo work, with the caveat of requiring suitable care and caution about seed initialisation and effective periodicity issues.

## References

- Bader, D. A., Chandramowlishwaran, A. & Agarwal, V. (2008), On the design of fast pseudo-random number generators for the cell broadband engine and an application to risk analysis, in 'Proc. 37th IEEE Int. Conf on Parallel Processing', pp. 520–527.
- Baillie, C., Gupta, R., Hawick, K. & Pawley, G. (1992), 'Monte-Carlo Renormalisation Group Study of the Three-Dimensional Ising Model', *Phys.Rev.B* **45**, 10438–10453.
- Binder, K. & Heermann, D. W. (1997), *Monte Carlo Simulation in Statistical Physics*, Springer-Verlag.
- Brent, R. P. (1997), A fast vectorized implementation of wallace's normal random number generator, Technical report, Australian National University.
- BSD (1993), *Random - Unix Manual Pages*.
- Coddington, P. D. (1994), 'Analysis of random number generators using monte carlo simulation', *J. Mod. Physics C* **5**(3), 547–560.
- Coddington, P. D. & Ko, S. H. (1998), Techniques for empirical testing of parallel random number generators, in 'Proc. International Conference on Supercomputing (ICS'98'. DHPC-025.
- Coddington, P., Mathew, J. & Hawick, K. (1999), Interfaces and implementations of random number generators for java grande applications, in 'Proc. High Performance Computing and Networks (HPCN), Europe 99, Amsterdam'.
- Coddington, P. & Newall, A. (2004), Japara - a java parallel random number generator library for high-performance computing, Technical Report DHPC-144, The University of Adelaide.
- Cuccaro, S., Mascagni, M. & Pryor, D. (1995), Techniques for testing the quality of parallel pseudo-random number generators, in 'Proc. of the 7th SIAM Conf. on Parallel Processing for Scientific Computing', SIAM, Philadelphia, USA, pp. 279–284.
- Danese, G., Loporati, F., Bera, M., Giachero, M., Nazzicari, N. & Spelgatti, A. (2007), 'An accelerator for physics simulations', *Computing in Science and Engineering* **9**(5), 16–25.
- Deng, L.-Y. & Xu, H. (2003), 'A system of high-dimensional, efficient, long-cycle and portable uniform random number generators', *ACM TOMACS* **14**(4), 299–309.
- Giles, M. (2009), Notes on CUDA implementation of random number generators. Oxford University.
- Hawick, K. A. & James, H. A. (2006), Ising model scaling behaviour on z-preserving small-world networks, Technical Report arXiv.org Condensed Matter: cond-mat/0611763, Information and Mathematical Sciences, Massey University.
- Hawick, K., Leist, A. & Playne, D. (2009), Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs, Technical Report CSTN-093, Computer Science, Massey University. To appear in Int. J. Parallel Programming (2010).
- Hormann, W. (1993), 'The transformed rejection method for generating poisson random variables', *Insurance: Mathematics and Economics* **12**(1), 39–45.
- ID Quantique White Paper (2010), Random Number Generation Using Quantum Physics, Technical Report Version 3.0, ID Quantique SA, Switzerland. QUANTIS.
- Johnsonbaugh, R. (2001), *Discrete Mathematics*, number ISBN 0-13-089008-1, 5th edn, Prentice Hall.
- Khronos Group (2008), 'OpenCL - Open Compute Language'.  
URL: <http://www.khronos.org/opencl/>
- Knuth, D. (1997), *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, 3rd edn, Addison-Wesley.
- Langdon, W. (2009), A Fast High Quality Pseudo Random Number Generator for nVidia CUDA, in 'Proc. ACM GECCO'09'.
- L'Ecuyer, P. (2001), Software for uniform random number generation: distinguishing the good and the bad, in 'Proc. 2001 Winter Simulation Conference', Vol. 2, pp. 95–105.
- Marsaglia, G. (1984), A Current view of random number generators, in 'Computer science and statistics: 16th symposium on the interface', Atlanta. Keynote address.
- Marsaglia, G., Zaman, A. & Tsang, W. W. (1987), 'Toward a universal random number generator', *Statistics and Probability Letters* **9**(1), 35–39. Florida State preprint.
- Matsumoto, M. & Nishimura, T. (1998), 'Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator', *ACM Transactions on Modeling and Computer Simulation* **8** No 1., 3–30.
- Newell, A. (2003), Parallel random number generators in java, Master's thesis, Computer Science, Adelaide University.
- Onsager, L. (1944), 'Crystal Statistics I. Two-Dimensional Model with an Order-Disorder Transition', *Phys.Rev.* **65**(3), 117–149.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (2007), *Numerical Recipes - The Art of Scientific Computing*, third edn, Cambridge. ISBN 978-0-521-88407-5.
- Schneier, B. (1996), *Applied Cryptography*, second edn, Wiley. ISBN 0-471-11709-9.
- Sloss, A. N. (2010), *ARM System Developer's Guide: Designing and Optimizing System Software*, Elsevier. ISBN 978-0080-490-496.



# Data-Intensive Management and Analysis for Scientific Simulations

Randy Hudson<sup>1</sup>

John Norris<sup>1</sup>

Lynn B. Reid<sup>2,3</sup>

G. Cal Jordan IV<sup>1</sup>

Klaus Weide<sup>1</sup>

Michael E. Papka<sup>1,4</sup>

<sup>1</sup> Flash Center, University of Chicago, Chicago, IL 60622, USA,

<sup>2</sup> Western Australian Geothermal Centre of Excellence, CSIRO, Kensington, WA 6151,

<sup>3</sup> School of Environmental Systems Engineering, University of Western Australia, Crawley, WA,

<sup>4</sup> Computation Institute, Argonne National Laboratory / University of Chicago, Chicago, IL 60622, USA  
Email: papka@anl.gov

## Abstract

Scientific simulations can produce enormous amounts of data, making the analysis of results and management of files a difficult task for scientists. The simulation management and analysis system (*Smaash*) described here is designed to allow scientists to easily capture, store, organize, monitor, and analyze simulation results. The system is automatic, standardized, and secure. *Smaash* was built using open-source tools and modularized to be independent of the scientific simulation. The web-based front-end allows the scientist to easily interact with the data, and has proved its usefulness in improving the efficiency of a scientific team's workflow.

**Keywords:** Data-intensive, scientific workflow management, FLASH astrophysical code

## 1 Introduction

High performance parallel computing allows scientists to solve complex physical problems through computer simulation. However, the massive amounts of data generated and the complex computing environment can create additional complications. A recent review by Ludäscher et al.(2009) describes how scientific workflows can assist scientists in extracting knowledge from these data-intensive operations by automating components within pipelines. Within the fusion community, Klasky et al.(2008) and colleagues have developed a system that handles the storage management, data movement, metadata generation and management, and a means to analyze the results. In response to scientists' needs, a simulation management and analysis system (*Smaash*) was developed at the University of Chicago and Argonne National Laboratory (USA). *Smaash* provides an integrated way to monitor simulations and analyze computational results; catalog, store, and retrieve simulations; and prepare output for publications. The system is independent of the particular simulation code, accessible from many HPC and browser-based platforms, and built around open-source software tools. Data security and provenance is considered throughout. The

analysis components are hidden behind a web-based front end, enabling scientists to focus on their results and not get bogged down by information overload.

## 2 Typical Data Requirements – FLASH code

The FLASH multiphysics adaptive mesh refinement code developed at the University of Chicago (Dubey et al., 2009) provided prototype data, and the astrophysicists of the Flash Center provided essential feedback to *Smaash* developers about analysis needs and scientific workflows. Typical scientific applications of the code include weakly-compressible turbulent flows (Fisher et al., 2008) and detonation of Type 1A supernovae (Jordan IV et al., 2008). The initial application is a set of parameter studies of Type 1A supernovae explosions which were calculated on the unclassified Purple HPC system at LLNL and the IBM BlueGene/P HPC system at Argonne National Lab.

A typical simulation of this three-dimensional physical system requires eight linked runs, or restarts, of 12 wall-clock hours each, which progressively cover only a few seconds of simulation time. Calculated on thousands of cores, the output might have: 90 large result files at 34GB each; 600 smaller analysis files at 9 GB each; log files recording integrated physical results and computational progress; and affiliated processing and visualization results. Total storage for one run may require eight terabytes of disk space; the parameter study required 16 of these complete simulations. The management of the data produced in this scientific study could be overwhelming for researchers who are primarily interested in abstracting physical insights from the computational results. Moreover, because of the limited availability of these HPC systems, the computed data must be carefully preserved and provenance understood.

## 3 Smaash Components

*Smaash* consists of three main parts: a back-end to capture, store, verify, and monitor simulation; a front-end designed for the scientists' needs; and a database. The front- and back-ends have modular components, allowing easy extensibility. The database is designed to be independent of the simulation code output formats and research genre.

The first step in the *Smaash* back-end is to automatically capture the data being generated and store it securely in the database. The archiving pipeline starts with a *Collector* which tracks the simulation progress through the simulation code's log file, and launches dynamically-loaded tools to record information into the database about each individual output type being generated. Next, the *Archiver* automates the transfer of files on the local filesystem into long-term mass storage, additionally storing details about

---

This work was part supported at the University of Chicago by the US Department of Energy under contract B523820 to the ASC Alliances Center for Astrophysical Nuclear Flashes.

Copyright ©2011, Australian Computer Society, Inc. This paper appeared at the Ninth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 118, J. Chen and R. Ranjan, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

the provenance in the database. The *Verifier* maintains the accuracy of the database, and the *Associator* keeps related analyses, such as post-processing results, connected to the original simulation results. The *Observer* monitors these back-end processes, and emails the user of significant changes. This program frees scientists from tending a lengthy simulation by being tied to a terminal, and allows efficient use of allocated computer processing time.

The back-end tools generally run on the same HPC system as the concurrent simulation, but on separate processors to avoid degrading simulation performance. They are robust and can recover from fatal conditions, and communicate securely.

### 3.1 User Interfaces

The control interface for Smaash is web-based and allows computational scientists to manage most post-processing and analysis from a web browser distant from the HPC system. Two primary front-end components are the *TreeView*, which hierarchically organises simulation sets, and the *GraphView* which shows visual details of simulation progress and provides easy access to data output. Figure 1 shows the TreeView in action, where multiple users can keep track of cascading simulations and their restarts. Figure 2 displays a user-definable concise window into the enormous quantity of data created by two simulations.

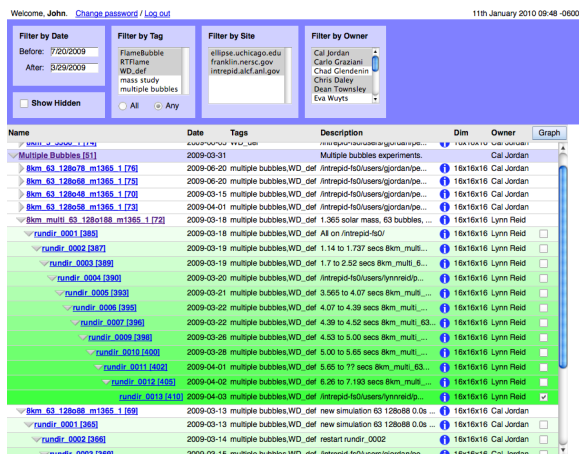


Figure 1: Web interface to the TreeView, showing multiple restarts in a single simulation.

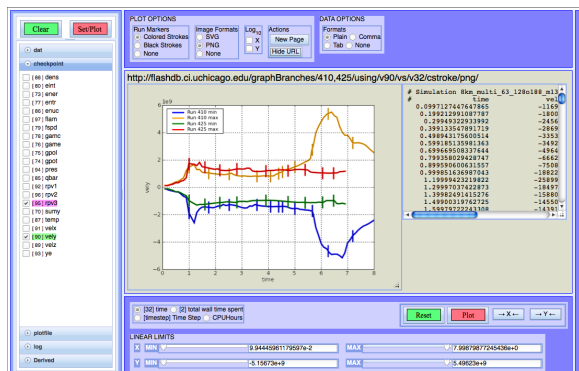


Figure 2: Web interface to the GraphView of two simulations, showing integral physical parameters plotted against simulation time. Curve coordinates are displayed on the right.

Other interfaces help the user keep track of simulation progress and do quick data analysis, such as the *Visualizer* pipeline, which provides graphical

snapshots of physical states over time. Summary web pages detail the accumulated information in the database and allow user annotations, while the robust URL feature of the graph pages allows a science group to share up-to-date notes through a wiki page.

### 3.2 Implementation

Smaash is designed to be easily adapted to a new scientific simulation framework by using modularity and standardization. Soft constraints encourage users to enter meaningful descriptions, and maintain data provenance. Off-the-shelf open-source tools such as MySQL, Django/Dojo/Dojango, and Matplotlib allow rapid development, extensibility, and provide well-considered security protocols. The Smaash development team is actively looking for new collaborations which would benefit from the integrated management and analysis tools described here.

### 4 Smaash in Action

The Smaash data management pipeline has greatly improved the efficiency of the scientific team's workflow. In one example, the front-end GraphView allowed easy amalgamation of multiple simulations into a clear picture showing differences in supernovae detonation times. Cross-referencing to the TreeView provided the means to pinpoint and extract crucial files for further analysis. In another computer science example, the FLASH programming team was able to spot a glaring inefficiency in CPU usage by plotting elapsed output file write times in the GraphView. Implementing a quick programming fix improved the use of precious allocated CPU time by forty percent. Smaash allows the scientist to shift focus from monitoring the simulation to analysing the results, while maintaining data integrity.

### 5 References

- Dubey, A, Antypas, K, Ganapathy, MK, Reid, LB, Riley, K, Sheeler, D, Siegel, A & Weide, K (2009), 'Extensible component based architecture for FLASH, a massively parallel, multiphysics simulation code', *Parallel Computing* **35**, 512–522.
- Fisher, RT, Kadanoff, LP, Lamb, DQ, Dubey, A, & 15 others 'Terascale turbulence computation using the FLASH3 application framework on the IBM Blue Gene/L system', *IBM Journal of Research and Development* **52** (1/2) 127–137, special issue on "Applications of Massively Parallel Systems".
- Jordan IV, G, Fisher, R, Townsley, D, Calder, A, Graziani, C, Asida, S, Lamb, D & Truran, J. (2008), 'Three-dimensional simulations of the deflagration phase of the gravitationally confined detonation model of Type Ia supernovae', *The Astrophysical Journal* **681**, 1448–1457.
- Klasky S, Barreto, R, Kahn, A, Parashar, M, Parker, S, Silver, D & Vouk, M (2008), 'Collaborative visualization spaces for petascale simulations', *International Symposium on Collaborative Technologies and Systems* 203–211.
- Ludäscher, B, Altintas, I, Bowers, S, Cummings, J, Critchlow, T, Roure, EDD, Freire, J, Goble, C, Jones, M, Klasky, S, McPhillips, T, Podhorszki, N, Silva, C, Taylor, I & Vouk, M (2009), Scientific process automation and workflow management, in A Shoshani & D Rotem, eds, 'Scientific Data Management: Challenges, Technology, and Deployment', Computational Science Series, Chapman and Hall/CRC, chapter 13.

# Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Grids

Yves Caniou<sup>1,2,3</sup>

Ghislain Charrier<sup>1,4</sup>

Frédéric Desprez<sup>1,4</sup>

<sup>1</sup>Université de Lyon, <sup>2</sup>UCBL, <sup>3</sup>CNRS (JFLI), <sup>4</sup>INRIA

Laboratoire de l'Informatique du Parallélisme (LIP), ÉNS Lyon

46 allée d'Italie, 69364 Lyon Cedex 07, FRANCE

Email: {yves.caniou, ghislain.charrier, frederic.desprez}@ens-lyon.fr

## Abstract

Grid services often consist of remote sequential or rigid parallel application executions. However, moldable parallel applications, linear algebra solvers for example, are of great interest but requires dynamic tuning which has mostly to be done interactively if performances are needed. Thus, their grid execution depends on a *remote and transparent* submission to a possibly different batch scheduler on each site, and means an *automatic tuning* of the job according to the local load.

In this paper we study the benefits of having a middleware able to automatically submit and reallocate requests from one site to another when it is also able to configure the services by tuning their number of processors and their walltime. In this context, we evaluate the benefits of such mechanisms on two multi-cluster Grid setups, where the platform is either composed of several heterogeneous dedicated clusters, or non dedicated ones. Different scenarios are explored using simulations of real cluster traces from different origins.

Results show that a simple method is good and often the best. Indeed, it is faster and thus can take more jobs into account while having a small execution time. Moreover, users can expect more jobs finishing sooner and a gain on the average job response time between 10% and 40% in most cases if this reallocation mechanism combined to auto-tuning capabilities is implemented in a Grid framework. The implementation and the maintenance of this heuristic coupled to the migration mechanism in a Grid middleware is also simpler because less transfers are involved.

**Keywords:** batch schedulers; computational grids; meta-schedulers; moldable tasks; reallocation

## 1 Introduction

In order to meet the evergrowing needs in computing capabilities of scientists of all horizons, new computing paradigms have been explored including the Grid. The Grid is the aggregation of heterogeneous computing resources connected through high speed wide area networks.

---

This work has been supported in part by the ANR project SPADES (08-ANR-SEGI-025).

---

Copyright ©2011, Australian Computer Society, Inc. This paper appeared at the 9th Australian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 118, J. Chen and R. Ranjan, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Computing resources are often parallel architectures managed by a local resource manager, called batch scheduler. In such a case, the local submission of a job requires at least a number of processors and a walltime. The walltime is the expected execution time for this job, given by the user or computed using data mining techniques. In most local resource management systems, when the walltime is reached, the job is killed, so users tend to over-evaluate the walltime to be sure that their job finishes its execution. Furthermore, giving an estimation of the execution time of a job is not an easy task and is influenced by the number of processors, which is generally chosen depending on external parameters such as the cluster load. Errors made at the local resource level may have a great impact on the global scheduling as shown by Beltrán & Guzmán (2009). Errors can come from mistakes on the walltime as well as a burst of submission as shown by Sonmez et al. (2009). Thus, having a mechanism to accommodate bad scheduling decisions is important.

The context of this work, described in detail by Caniou et al. (2010b), takes place in a heterogeneous multi-cluster Grid connected through a high bandwidth network: We propose a reallocation mechanism that takes into account scheduling errors by moving waiting jobs between clusters. The mechanism we propose can be used to connect different clusters together while each cluster keeps its local scheduling or resource allocation policies. Each job submitted onto the platform is executed automatically without any intervention from the user.

Two reallocation algorithms are studied with two heuristics each. We evaluate each couple (algorithm, heuristic) by comparing them on different metrics to an execution where reallocation is not performed. We extend the simulations realized by Caniou et al. (2010a) by focusing on moldable tasks instead of parallel rigid tasks. The middleware is able to determine the number of processors and the walltime automatically for each task. Furthermore, we study the algorithms on dedicated platforms as well as non dedicated platforms. *We aim at showing the expectations in terms of performance with regard to the increased complexity of the jobs management done by the middleware.* We analyze the results on different metrics, and we show that obtained gains are very good in the majority of the simulations we perform. Gains are larger on dedicated platforms than on non dedicated platforms. We show that in most cases reallocating jobs will let jobs to finish sooner and diminish their average response time between 10% and 40%. Furthermore, results definitely confirm the counter intuitive fact that even for moldable jobs, whose number of processors varies if migrated, the simplest heuristic, both algorithmically and in implementation complexity, is the best to use. Results presented in this work are only for heterogeneous platforms. A



more complete analysis, with results for both homogeneous and heterogeneous platforms, each with different batch scheduler policies, is available in research report (Caniou et al. 2010b).

The remainder of the paper is as follows. In Section 2, we present related work. In Section 3 we describe mechanisms and the scheduling algorithms used in this work. Then we explain the experimental framework in Section 4, giving information about the simulator we developed, on the platforms simulated with real-world traces, scenarios of experiments that were conducted as well as the metrics on which results are compared in Section 5. Finally we conclude in Section 6.

## 2 Background

Parallel applications are characterized by Feitelson et al. (1997) as rigid, moldable or malleable. A rigid application has a fixed number of processors. A moldable application can be executed with different number of processors, but once the execution started, this number can not change. Finally, the most permissive applications are malleable. The number of processors used can be modified “on the fly” during execution.

Cirne & Berman (2002) use moldable jobs to improve the performance in supercomputers. The user provides the scheduler *SA* with a set of possible requests that can be used to schedule a job. Such a request is represented by a number of processors and a walltime. *SA* chooses the request providing the earliest finish time. The evaluation of *SA* is done using real traces from the Parallel Workload Archive and their results show an average improvement on the response time of 44%, thus justifying the use of moldable jobs instead of rigid ones. In our work, we use the same kind of technique to choose the number of processors and the walltime of jobs. However, the user does not provide any information. The middleware is able to automatize everything thus facilitating the user’s actions and can choose to migrate jobs from on site to another one.

Guim & Corbalán (2008) present a study of different meta-scheduling policies where each task uses its own meta-scheduler to be mapped on a parallel resource. Once submitted, a task is managed by the local scheduler and is never reallocated. In order to take advantage of the multi-site environment considered in our work, we use a central meta-scheduler to select a cluster for each incoming task because we place ourselves in the GridRPC context where clients do not know the computing resources. Also, once a task is submitted to the local scheduler, our approach let us cancel it and resubmit it elsewhere.

Yue (2004) presents the Grid-Backfilling. Each cluster sends a snapshot of its state to a central scheduler at fixed intervals. Then the central scheduler tries to back-fill jobs in the queue of other clusters. The computation done by the central scheduler is enormous since it works with the Gantt chart of all sites. All clusters are homogeneous in power and size. In our work, the central scheduler is called upon arrival of each job in order to balance the load among clusters. During the reallocation phase, it gathers the list of all the waiting tasks and asks the local schedulers when a job would complete, but it does not perform complex computations. Furthermore, in our work, clusters are heterogeneous in size and power and we consider moldable jobs.

Huang et al. (2009) present a study of the benefits of using moldable jobs in an heterogeneous computational grid. In this paper, the authors show that using a Grid meta-scheduler to choose on which site to execute a job coupled with local resource management

schedulers able to cope with the moldability of jobs improves the average response time. In our work, instead of letting the local schedulers decide of the number of processors for a job, we keep existing infrastructure and software and we add a middleware layer that takes the moldability into account. Thus, our architecture can be deployed in existing Grids without modifications of the existing. Furthermore, this middleware layer renders reallocation between sites possible.

## 3 Task Reallocation

In this section, we describe the proposed tasks reallocation mechanism. First, we present the architecture of the Grid middleware (Section 3.1). Then we present the different algorithms used for the tasks reallocation (Section 3.2).

### 3.1 Architecture of the Middleware

Caniou et al. (2010b) describe the architecture that we use in this work. It is very close to the GridRPC (Seymour et al. 2004) standard from the Open Grid Forum<sup>1</sup>. Thus it can be implemented in GridRPC compliant middleware such as DIET (Caron & Desprez 2006) or Ninf (Sato et al. 1997). Because such a middleware is deployed on existing resources and has limited possibilities of action on the local resource managers, we developed a mechanism that only uses simple queries such as submission, cancellation, and estimation of the completion time.

The architecture relies on three main components: the **client** has computing requests to execute, and contacts an **agent** in order to obtain the reference of a **server** able to process the request. In our proposed architecture, one server is deployed on the front-end of each parallel resource, in which case it is in charge of interacting with the batch scheduler to perform the submission, cancellation or estimation of the completion date of a job. The server is also in charge of deciding how many processors should be used to execute the request, taking into account the load of the parallel resource. Benefiting from servers estimations, the *agent maps every incoming requests using a MCT strategy* (Minimum Completion Time (Maheswaran et al. 1999)), and *decides of the reallocation with a second scheduling heuristic*.

The process of submission of a job is depicted in Figure 1. 1) When a client wants to execute a request, it contacts the agent. 2) The agent then contacts each server where the service is available. 3) Each server able to execute the request computes an estimation of the completion time and 4) sends it back to the agent. 5) The agent sends the identity of the best server to the client which then 6) submits its request to the chosen server. 7) Finally, the server submits the task to the batch scheduler of the cluster. 8) When the agent orders a server to reallocate a task, the latter submits it to the other server provided by the agent.

### 3.2 Algorithms

This section presents the algorithm used to decide of the number of processors and walltime for each task (Section 3.2.1), the two versions of the reallocation mechanism (Section 3.2.2), and the scheduling heuristics used for reallocation (Section 3.2.3).

<sup>1</sup><http://www.ogf.org>



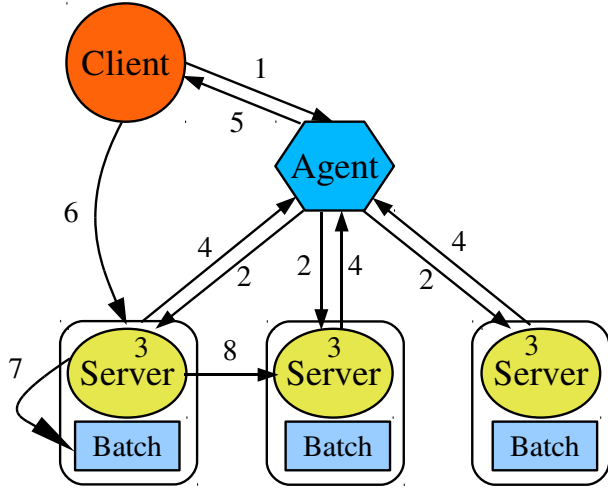


Figure 1: Architecture of the middleware layer for reallocation.

### 3.2.1 Tuning Parallel Jobs at Submission Time

The choice of the number of processors and walltime is done by the server each time a request arrives, either for the submission of a job or for an estimation of completion time. To determine the number of processors to allocate to the job, the server performs several estimations with different number of processors and returns the best size, *i.e.*, the one giving the earliest completion time. To estimate the completion time, the server can directly query the batch scheduler (but this capability is generally not present) or have it's own mechanism to compute the estimated completion time by simulating the batch algorithm for example.

The simplest idea to obtain the best size for the job is to perform an *exhaustive search*: For all possible number of processors (from one to the number of processors of the cluster), the estimation method provides a completion time as regard to the current load of the cluster. This method is simple and will choose the best size for jobs, however, it is time consuming. Indeed, each estimation is not instantaneous. Thus, for a large cluster, the estimation must be done a lot of times and the finding of the number of processors can require a long time.

Sudarsan & Ribbens (2010) benchmark different sizes of the LU application from the NAS parallel benchmarks<sup>2</sup>. Their study show a strictly increasing speedup up to 32 processors (adding processors always decreases execution time). But after this point, the execution time increases. It is due to the computation to communication ratio of the job becoming too small. This kind of job is not uncommon, thus we consider moldable jobs with *strictly increasing speedups until a known number of processors*.

Thus, in order to improve the speed in choosing the number of processors of a task, we can restrict the estimation from one processor to the limit of processors of the job. For jobs that don't scale very well, this will greatly reduce the number of calls to the estimation method thus reducing the time needed to find the most suitable number of processors.

Because of the hypothesis that speedup is strictly increasing until a maximum number of processors, we propose to perform a *binary search* on the number of processors to find how many of them to allocate to the job. Instead of estimating the completion time for each possible number of processors, we start by

estimating the time for 1 processor and for the maximum number of processors. Then, we perform a classical binary search on the number of processors. This reduces the number of estimations from  $n$  to  $\log_2 n$ .

In particular cases the binary search will not provide the optimal result because of the back-filling. Let us consider an example in order to illustrate this behavior. Consider a cluster of 5 processors and a job needing 7 minutes to be executed on a single processor. With a perfect parallelism, this jobs needs 3.5 minutes to run on 2 processors, 2.33 on 3, 1.75 on 4 and 1.4 on 5. Upon submission, the cluster has the load represented by hatched rectangles in Figure 2. First, the binary search evaluates the completion time for the job on 1 and 5 processors (top of the figure) and obtains completion times of 7 and 7.4 minutes respectively. Then, the number of processors is set to 3 (middle of 1 and 5). The evaluation returns a completion time of 7.33 (bottom left of the figure). The most promising completion time was obtained with 1 processor, thus the binary search looks between 1 and 3. Finally, the best completion for the tested values time is obtained for 2 processors: 6.5 minutes (bottom right). However, the best possible completion time the job could have is 1.75 minutes with 4 processors. Indeed, with 4 processors, the jobs can start as soon as submitted, but this value was disregarded by the binary search. During our tests to verify the behavior of the binary search on thousands jobs, the results were the same as the exhaustive search which means that the "bad" cases are rare.

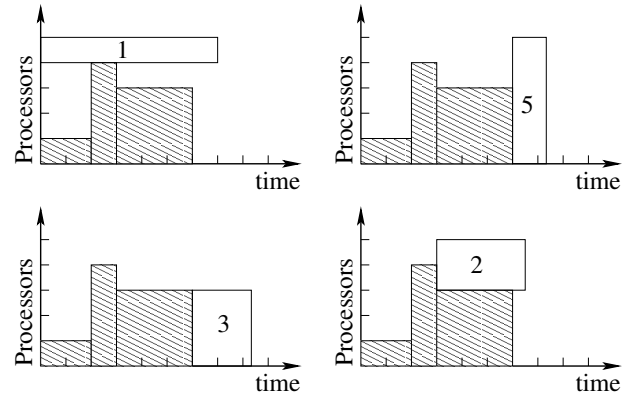


Figure 2: Estimations made by the binary search.

If the maximum number of processors of a job is large, using the binary search reduces enormously the number of estimations to do, potentially by orders of magnitude. For example, if a job can be executed on 650 processors the exhaustive search performs 650 estimations of completion time and the binary search performs only 10. The binary search in this case is thus 65 times faster.

### 3.2.2 Reallocation Algorithms

The first algorithm, **regular**, works as follows: It gathers the list of all jobs in the waiting queues of all clusters; it selects a job with a scheduling heuristic; if it is possible to submit the job somewhere else with a better estimated completion time (ECT) of at least a minute, it submits it on the other cluster and cancels the job at its current location; finally, it starts again with the remaining jobs. The one minute threshold is here to consider some small data transfer that can take place, and to diminish the number of reallocations bringing almost no improvement.

To have a better idea of what is done, consider an example of two batch systems with different loads (see

<sup>2</sup><http://www.nas.nasa.gov/Resources/Software/npb.html>

Figure 3). At time  $t$ , task  $f$  finishes before its wall-time, thus releasing resources. Task  $j$  is then scheduled earlier by the local batch scheduler. When a reallocation event is triggered by the meta-scheduler at  $t_1$ , it reallocates tasks  $h$  and  $i$  to the second batch system because their expected completion time is better there. To reallocate the tasks, each one is sequentially submitted to the second batch and canceled on the first one. In this example, the two clusters are identical so the tasks have the same execution time on both clusters, and the tuning of the parallel jobs (choice of number of processors to allocate to task  $h$  and  $i$ ) is the same due to the same load condition. In an heterogeneous context, the length and even the number of processors allocated to the tasks would change between the clusters. Note that a task starting earlier on a cluster does not imply that it will also finish earlier.

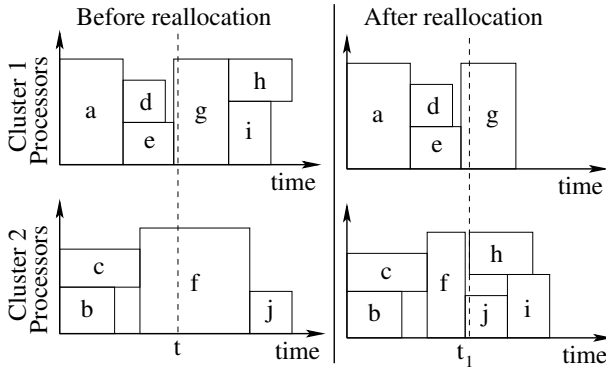


Figure 3: Example of reallocation between two clusters.

The second algorithm, **all-cancellation**, starts by canceling all waiting jobs of all clusters. The agent keeps a reference for all jobs. Then, it selects a job with a scheduling heuristic. Finally, it submits the job to the cluster giving the minimum estimated completion time and loops on each of the remaining jobs.

Note that it does not mean that all parallel jobs will be tuned in the maximum of their performance since platforms are not necessarily dedicated to the Grid middleware, each cluster has its own load. It may be better to use less resources, thus have a longer execution time, but start earlier.

The reallocation event in both versions of the algorithm is triggered periodically every hour, based on previous works conducted by Caniou et al. (2009) where a smaller period did not change the results but required more network transfers and potentially more reallocations.

Because both reallocation algorithms use an estimation of the completion time, it is mandatory that clusters use a batch scheduling algorithm able to give some guaranty on the completion time to guaranty the results. Feitelson et al. (2004) present the two main algorithms offering these guaranties are First-Come-First-Served (FCFS) and Conservative Back-Filling (CBF). Both algorithms make reservations for each job and jobs can never be delayed once the reservation done. However, jobs can be scheduled earlier if new resources become available. Batch schedulers using one of these algorithms are common. Other algorithms such as Easy Back-Filling (EBF) introduced by Lifka (1995) or the well-known Shortest Job First (SJF) presented by Feitelson et al. (1997) do not guaranty a completion time and thus should not be used without adding specialized prediction mechanisms to the servers.

### 3.2.3 Scheduling Heuristics for Reallocation

We focus on two heuristics to use to select a job at each iteration. With the first one, jobs are processed in their submission order. In the remainder of the paper, we refer to this policy as MCT because jobs are submitted in their original submission order and the jobs are submitted to the cluster with the Minimum Completion Time (MCT).

The second policy executes the MinMin heuristic on a subset of the jobs. MinMin asks the estimated completion time of all jobs and selects the job with the minimum of the returned values. In this paper, MinMin is executed on the 20 oldest jobs. We use this limit to avoid a too long reallocation time. Indeed, MinMin has to update the estimations of completion times of all the remaining jobs at each iteration to select the job with the minimum of the ECTs. Because the all-cancellation algorithm needs to resubmit all jobs, it executes MinMin on the 20 oldest jobs and then the remaining jobs are processed in their original submission order, leading to a MCT policy.

We have two scheduling heuristics, MCT and MinMin, as well as two reallocation algorithms, namely regular and all-cancellation. Thus, we have four couples of algorithm that we refer in the remainder of this paper as *MCT-reg*, *MCT-can*, *MinMin-reg*, and *MinMin-can*.

## 4 Experimental Framework

In this section we depict the experimental framework by presenting the simulator we implemented to run our experiments (Section 4.1), the description of the jobs (Section 4.2), the simulated platforms (Section 4.3), and the metrics used to compare the heuristics (Section 4.4). Finally, the experiments are described (Section 4.5).

### 4.1 Simulator

In order to simulate task reallocation in a distributed environment composed of several clusters, we use SimGrid (Casanova et al. 2008), a discrete events simulation toolkit designed to simulate distributed environments, and Simbatch (Caniou & Gay 2009), a batch systems simulator built on top of SimGrid. Simbatch, which has been tested against real life experiments, can simulate the main algorithms used in batch schedulers described by Feitelson et al. (2004). In this study, we use the *Conservative Back-Filling* (CBF) algorithm for the batch schedulers. Mu'alem & Feitelson (2001) introduces the CBF algorithm. It tries to find a slot in the queue (Back-filling) where the job can fit without delaying already scheduled jobs (Conservative). If it does not, the job is added at the end of the queue. CBF is available in batch systems such as Maui (Jackson et al. 2001), Loadleveler (Kannan et al. 2001), and OAR (Capit et al. 2005) among others.

The simulator is divided using the same components as the ones in the GridRPC standard introduced in Section 3.1:

The *client* requests the system for a service execution. It contacts the meta-scheduler that will answer with the reference of a server providing the desired service.

The *meta-scheduler* matches incoming requests to a server according to a scheduling heuristic (we use MCT in this paper) and periodically reallocates jobs in waiting queues on the platform using one of the reallocation scheduling heuristic described in Section 3.2.3.

The *server* is running on the front-end of a cluster and interacts with the batch system. It receives requests from the client and can submit jobs to the batch scheduler to execute the requests. It can also cancel a waiting job, return an estimation of the completion time of a request and return the list of jobs in the waiting state. For submission and estimation, the server uses an estimation function that automatically chooses the number of processors and the walltime of the request using the technique described in Section 3.2.1.

## 4.2 Jobs

We built seven scenarios of jobs submission, where for six of them, jobs come from traces of different clusters on GRID'5000 for the first six months of 2008. Table 1 gives the number of jobs per month on each cluster. The seventh scenario is a six month long simulation using two traces from the parallel workload archive (CTC and SDSC) and the trace of Bordeaux on GRID'5000. The trace from Bordeaux contains 74647 jobs, CTC has 42873 jobs and SDSC contains 15615 jobs. Thus, there is a total of *133135 jobs*. In the remainder of the paper, we refer to the different scenarios by the name of the month of the trace for the jobs from GRID'5000 and we refer to the jobs coming from CTC, SDSC, and GRID'5000 as "PWA-G5K".

Month/Cluster	Bordeaux	Lyon	Toulouse	Total
January	13084	583	488	14155
February	5822	2695	1123	9640
March	11673	8315	949	20937
April	33250	1330	1461	36041
May	6765	2179	1573	10517
June	4094	3540	1548	9182

Table 1: Number of jobs per month and in total for each site trace.

In our simulations, we do not consider advance reservations (present in GRID'5000 traces). They are considered as simple submissions so the batch scheduler can start them when it decides to. To evaluate the heuristics, we compare simulations together so this modification does not impact the results. However, we can not compare ourselves with what happened in reality. Furthermore, note that we add a meta-scheduler to map the jobs onto clusters at submission time, as if a grid middleware is used. On the real platform, users submit the cluster of their choice (usually they submit to the site closest to them) so the simulations already diverge from reality.

The traces taken from the Parallel Workload Archive were taken in their standard original format, *i.e.*, they also contain "bad" jobs described by Feitelson & Tsafir (2006). We want to reproduce the execution of jobs on clusters, so we need to keep all the "bad" jobs removed in the clean version of the logs because these jobs were submitted in reality.

### 4.2.1 Moldable Jobs

The jobs contained in the trace files are parallel rigid jobs. So, in order to simulate the moldable jobs, we defined 4 types of jobs using Amdahl's law ( $speedup = \frac{1}{(1-P) + \frac{P}{N}}$  with  $P$  the fraction of parallel code and  $N$  the number of processors). The law states that the expected speedup of an application is strictly increasing but the increase rate diminishes. The execution time of an application tends to the execution time of the sequential portion of the application when adding more processors.

To obtain the 4 types of moldable jobs, we vary the parallel portion of the jobs that is sequential as well as the limit of processors until the execution time decrease. The different values for the parallel portion of code are 0.8, 0.9, 0.99 and 0.999. Figure 4 plots the speedups for the different values of parallel code for different number of processors. Note that the y-axis is log-scaled. The figure shows that there is some point where the speedup increase becomes negligible. For the limits, we chose to use 32, 96, 256, and 650 processors. These values were chosen in accordance to the gain on the execution time of adding one processor. When the gain becomes really small, chances are that the internal communications of the job will take most of the time and slow down the task. Furthermore, the 650 limit is given by the size of the largest cluster of our simulations. So, the 4 types of jobs we consider are 4 couples (parallel portion, limit of processors):  $t1:(0.8, 32)$ ,  $t2:(0.9, 96)$ ,  $t3:(0.99, 256)$  and  $t4:(0.999, 650)$ .

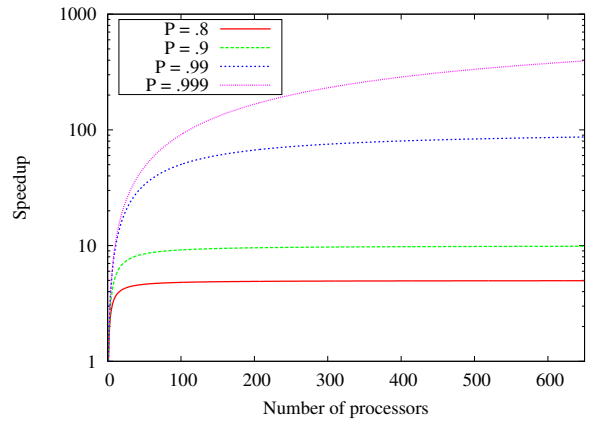


Figure 4: Speedups for the Amdahl's law for different parallelism portions.

In the traces, there are more tasks using a small number of processors than tasks using a lot of processors. Thus, each job from the trace files was given a moldable type. In each simulation we present, there are 50% of jobs of type  $t1$ , 30% of type  $t2$ , 15% of type  $t3$  and 5% of type  $t4$ . The type of a job is chosen randomly. In order to keep a more realistic set of jobs, we decided to keep the sequential jobs of the traces sequential.

### 4.2.2 Simulating Realistic Parallel Jobs

During the simulations, the server uses information from both the traces and the type of the job to choose a suitable number of processors and a walltime for the job. In order to do so, the server uses the binary search described in Section 3.2.1 to choose a number of processors and follows the following process to choose the walltime: First, it computes the speedup of the job in the trace file using Amdahl's law, the type of the job and the number of processors:  $spd = amdaahl(p, n_t)$  with  $p$  the parallel portion of the code and  $n_t$  the number of processors used in the trace file. Second, the server computes the walltime of the job on one processor:  $w_1 = w_{n_t} * spd$ . Third, the server computes the speedup of the job for the current number of processors chosen by the binary search:  $spd_b = amdaahl(p, n_b)$ . Then, the server computes the walltime for the job:  $w_b = \frac{w_1}{spd_b}$ . Finally, the runtime and walltime are modified *in accordance with the speed of the cluster* given in Section 4.3.1.

To obtain the actual execution time for the moldable jobs, we keep the same difference ratio as the

one in the trace file. Thus if the runtime of a job was twice smaller than walltime in the trace file, it will also be twice smaller than the walltime in the simulations, independently of the number of processors chosen for the job.

### 4.3 Platform Characteristics

#### 4.3.1 Computing Resources

We consider two sets of resources, composed of three sites, each with a different number of cores, and managed with a CBF policy.

The first set corresponds to the simulation of three clusters of GRID'5000 (Bolze et al. 2006). The three clusters are Bordeaux, Lyon, and Toulouse. Bordeaux is composed of 640 cores and is the slowest cluster. Lyon has 270 cores and is 20% faster than Bordeaux. Finally, Toulouse has 434 cores and is 40% faster than Bordeaux.

The second set corresponds to experiments mixing the trace of Bordeaux from GRID'5000 and two traces from the Parallel Workload Archive<sup>3</sup>. The three clusters are Bordeaux, CTC, and SDSC. Bordeaux has 640 cores and is the slowest cluster. CTC has 430 cores and is 20% faster than Bordeaux. Finally, SDSC has 128 cores and is 40% faster than Bordeaux.

#### 4.3.2 Dedicated Vs. Non Dedicated

On real life sites, tasks can be either submitted by a Grid middleware or by local users. Thus, we investigate the differences in behavior of our mechanism depending on heuristics: on **dedicated platforms**, where all tasks have been submitted through our middleware; on **non dedicated platforms** where two third of the jobs issued from the traces are directly submitted through batch schedulers by simulated local users. Both setups will be investigated in Sections 5.1 and 5.2 for the dedicated case and for the non dedicated platform respectively.

### 4.4 Evaluation Metrics

We choose the following metrics to compare the performance of reallocation depending on platforms, mechanisms and scheduling heuristics:

**The percentage of jobs impacted by reallocation** is the percentage of jobs whose completion time is changed compared to an execution without reallocation. In this study, we are only interested by these jobs.

We also study **the number of reallocations relative to the total number of jobs**. We give the percentage of reallocations in comparison of the number of jobs. A job can be counted several times if it migrated several times so it is theoretically possible to have more than 100% reallocations. A small value is better because it means less transfers.

On a user point of view, **the percentage of jobs finishing earlier with reallocation than without** is very important. This percentage is taken only from the jobs whose completion time changed with reallocation. A value higher than 50% means that there are more jobs early than late.

Feitelson & Rudolph (1998) presents the notion of response time. It corresponds to the duration between submission and completion. Complementary to the previous one, **the average job response time** of the jobs impacted by reallocation relatively to the scenario without reallocation defines the average ratio that the duration of a job can issue. A ratio of 0.8

means that on average, jobs spent 20% less time in the system, thus giving the results faster to the users.

Figure 5 illustrates why jobs can be delayed and others finishing earlier onto a platform composed of two clusters. At time 0 a reallocation event is triggered. A task is reallocated from cluster 2 to cluster 1 with a greater number of processors allocated to it according to our algorithm. Thus, some tasks of cluster 2 are advanced in the schedule. On cluster 1, as expected, the task is back-filled. However, assume the task finishing at time 6 finishes at time 2 because the walltime was wrongly defined (see the task with the dashed line). Thus, because of the newly inserted task, the large task on cluster 1 is delayed. Note that, even with FCFS, reallocation can also cause delay. If a job is sent to a cluster, all the jobs submitted after may be delayed. Inversely, the job that was reallocated to another cluster now leaves some free space and it may be used by other jobs to diminish their completion time.

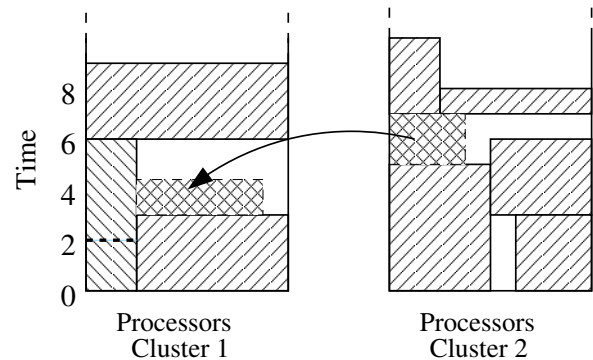


Figure 5: Side effects of a reallocation.

### 4.5 Experiment

An experiment is a tuple (reallocation algorithm, heuristic, platform-trace, dedicated, seed) where the seed is used to draw the type of a job in the trace, and concerning non dedicated platform, to draw if a job is submitted to the middleware or directly to the local scheduler. We used 10 different random seeds, hence, in addition to the reference experiment using MCT without reallocation, we conducted  $14+2*2*7*2*10$ , i.e., **574 experiments** in total.

## 5 Results

First, we present the results on dedicated platforms in Section 5.1. Then, Section 5.2 contains the results for non dedicated platforms. Finally, some concluding remarks on the results are given in Section 5.3.

Figures in this section show the minimum, the maximum, the median, the lower, and higher quartiles and the average of the 10 runs of each experiment. Concerning the figures in non dedicated platforms, results only take into account the jobs submitted to the Grid middleware. External jobs are not represented in the plots.

### 5.1 Dedicated Platforms

In this section, clusters are heterogeneous in number of processors and in speed (cf. Section 4.3). All requests are done to our Grid middleware, thus there are no local jobs submitted.

The percentage of jobs impacted is shown in Figure 6. In six experiments for the two traces March and June, extreme cases were almost 100% of the jobs that

<sup>3</sup><http://www.cs.huji.ac.il/labs/parallel/workload/>



were impacted by reallocation appear. This happens when the platform has a few phases with no job. If there are always jobs waiting, the reallocation is able to move jobs more often thus impacting a bigger portion of the jobs. Apart from these cases, the number of jobs impacted varies between the traces from 25 to 95%. All-cancellation algorithms usually impacts more jobs. MinMin-can impacts more jobs on average than the other heuristics. MCT-reg and MinMin-reg have close results.

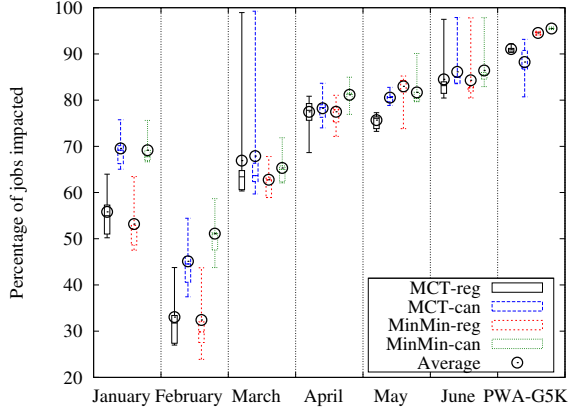


Figure 6: Jobs impacted on dedicated platforms.

The number of reallocations relative to the total number of jobs is plotted in Figure 7. All-cancellation algorithms always produce more reallocations. The regular algorithms give results inferior than 15% so the number of reallocations is quite small compared to the total number of jobs. However, with the all-cancellation algorithms, it is possible to go to a value as high as 50%. Because all-cancellation empties the waiting queues, more jobs have the opportunity to be reallocated. With the regular algorithms, jobs close to execution have a very small chance of being reallocated. The regular version of the reallocation algorithm is better on this metric.

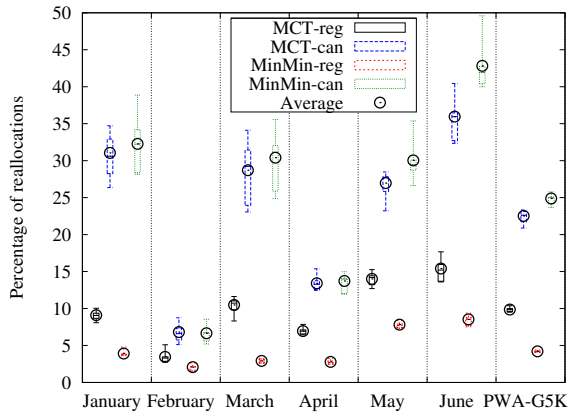


Figure 7: Reallocations on dedicated platforms.

Figure 8 plots the percentage of jobs early. In this case, 3 experiments produce more jobs late than early. In April without all-cancellation there are always more jobs late (less than 4%) when reallocation is performed. However in most cases, it is better to reallocate. MinMin-reg gives the worst results. It is followed by MCT-reg, then MinMin-can and finally MCT-can is the best with up to 64% of tasks early!

Concerning the average relative response time, the plot in Figure 9 shows a clear improvement in most

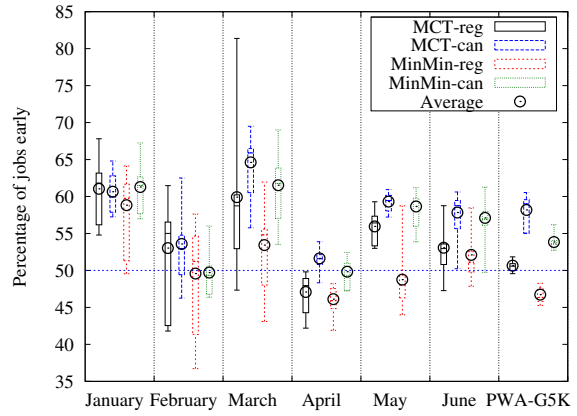


Figure 8: Percentage of jobs early on dedicated platforms.

cases. Excluding MinMin-reg, most gains are comprised between 10% and 40%. On average, MCT-can is the best heuristic. The reallocation without all-cancellation can worsen the average response time. It happened in 6 experiments (3 with MCT-reg and 3 with MinMin-reg). The loss is small for MCT-reg (less than 5%) thus it is not a problem. The all-cancellation versions are always better than their corresponding regular algorithm except in February for MCT-reg. Some experiments present a gain on the average response time while there were more jobs late than early (MCT-reg in April for example): The gains were high enough to compensate for the late jobs.

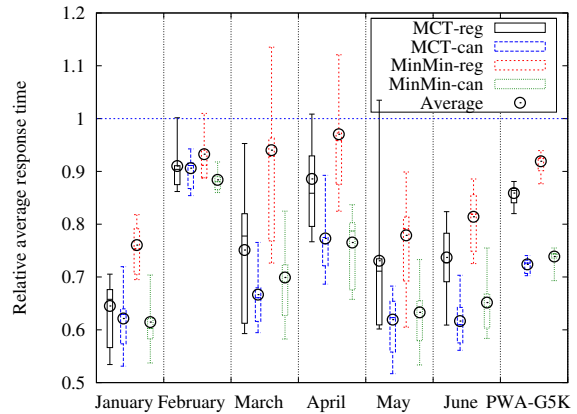


Figure 9: Relative average response time on dedicated platforms.

## 5.2 Non Dedicated Platforms

In this section, we present the results on non dedicated platforms where 33% of the jobs executed on the Grid platform are moldable and submitted to the Grid middleware.

The percentage of jobs impacted by reallocation is plotted in Figure 10. The two all-cancellation heuristics impact more jobs than the regular ones, but the difference is really small. There is one experiment in March where MinMin-reg impacts almost all jobs: a scheduling decision taken at the beginning of the experiment impacts all the following job completion dates. For a given trace, the number of impacted jobs usually does not vary a lot.

Figure 11 plots the number of reallocations relative to the total number of moldable jobs. The number of reallocations is very small. In most cases, there are

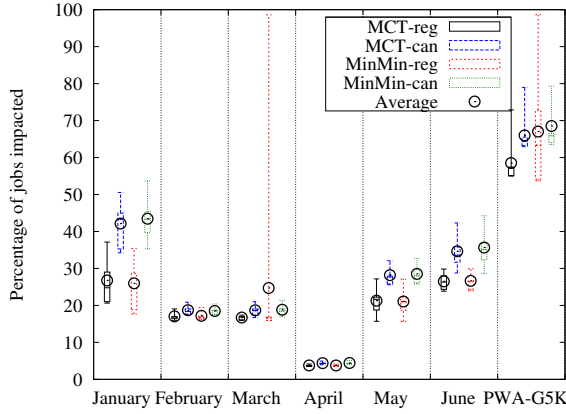


Figure 10: Jobs impacted on non dedicated platforms.

only a few dozens reallocations. The all-cancellation algorithms always reallocates more than the regular versions, *but not by far*. In a lot of cases, the number of reallocations corresponds to less than 1% of the number of jobs. Thus, on a non dedicated platform, the reallocation mechanism does not produce many transfers.

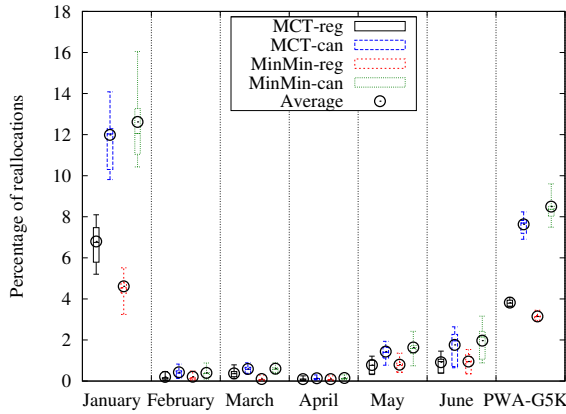


Figure 11: Reallocations on non dedicated platforms.

Most experiments except the worst case for PWA-G5K and March with MinMin-reg result *in more than half of the jobs early* as we can see in Figure 12. The 90% jobs late in March with MinMin-reg are from the same experiment where almost all jobs were impacted in Figure 10. Most experiments exhibit a percentage of jobs early close to 70%. All-cancellation again produces less jobs early than regular. MCT-reg and MinMin-reg are the two heuristics of choice, but MinMin-reg gives mitigate results for PWA-G5K so MCT-reg is a better choice.

Figure 13 shows that the different heuristics give results close to one another on the relative average response time. All-cancellation heuristics usually have a smaller difference between the minimum and the maximum gains. Depending on the experiment, results vary a lot. In some experiments, the average response time is divided by more than two, but in other it is augmented with a maximum of 40%. However on all experiments, the average gain is positive. Thus reallocation is expected to provide a gain.

### 5.3 Remarks on Results

MCT-reg and MinMin-reg usually give similar results on non dedicated platforms, often in favor of MCT-

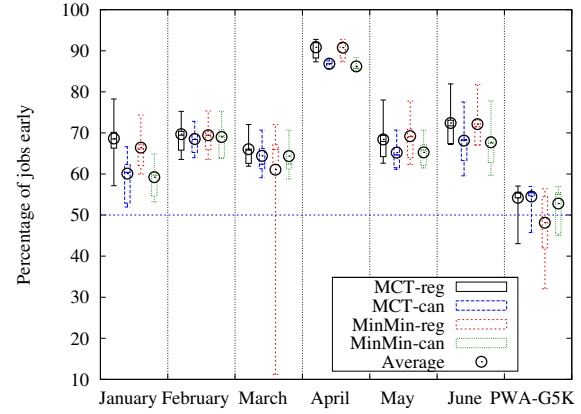


Figure 12: Percentage of jobs early on non dedicated platforms.

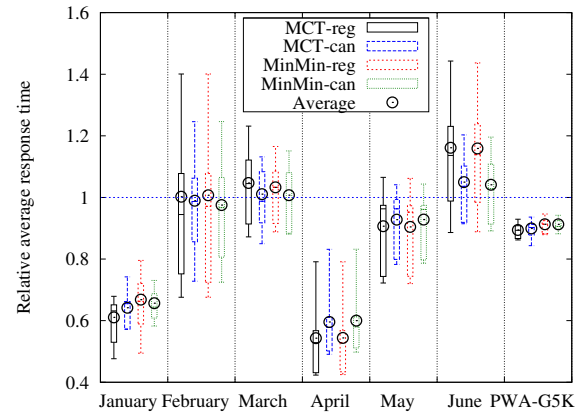


Figure 13: Relative average response time on non dedicated platforms.

reg. On dedicated platforms however, MCT-reg is clearly better than MinMin-reg. MinMin-reg may give better results if it is able to take more jobs into account during reallocation. But, if it takes more jobs into account, its execution time grows exponentially. Furthermore, the two algorithms with all-cancellation also give very similar results with a small advantage for the MCT-reg on dedicated platforms.

The all-cancellation algorithms can cause starvation. In a non dedicated platform, it is obvious that starvation can happen. Indeed, when canceling jobs, jobs from the external load (for the Grid middleware) will be rescheduled in front of the moldable jobs managed by the middleware system. This may explain the worst cases peaks in Figure 13. Even in a dedicated environment with MinMin-can, it is possible for a job to be delayed indefinitely. If the job is long, it will always be resubmitted after others and may never start execution. However, such cases did not happen in our simulations because there are always phases of low load where the queues can be emptied.

The results presented in this paper show that the heuristic of choice is MCT with or without all-cancellation whether the platform is dedicated or not. Indeed, MinMin is too complex in time to react in a decent time regarding the submission rate of jobs onto the platform. In a previous study Caniou et al. (2010a), we used several other selection heuristics such as MaxMin, Sufferage, MaxGain, and MaxRelGain but these heuristic did not prove better than MCT or MinMin. Because these algorithms have the same complexity than MinMin, we argue that they may also give poor results, especially because of worst

cases.

In this paper, due to space constraints, we present only results on heterogeneous platforms, since they are the most common in real life. But results on homogeneous platforms are presented in detail in the research report (Caniou et al. 2010b), where clusters have different sizes, but their speed is the same. Gains obtained by the reallocation are usually better by a few percents on homogeneous platforms than the one presented in this paper. The same patterns as the ones we see in this paper emerge: on dedicated platform, MCT-can and MinMin-can give the best results. MCT-reg produces less gains, and the worst is MinMin-can. On non-dedicated platform, all heuristics give similar results.

## 6 Conclusion and Perspectives

In this paper, we presented a reallocation mechanism that can be implemented in a GridRPC middleware and used on any multi-cluster Grid without modifying the underlying infrastructure. Parallel jobs are tuned by the Grid middleware each time they are submitted to the local resource manager (which implies also each time a job is migrated). We achieve this goal by only querying batch schedulers with simple submission or cancellation requests. Users ask the middleware to execute some service and the middleware manages the job automatically.

We have investigated two reallocations algorithms, the key difference between them being that one, regular, cancels a task once it is sure that the expected completion time is better on another cluster, and the other, all-cancellation, cancels all waiting jobs before testing reallocation. We also considered two scheduling heuristics to make the decision of migrating a job to another site. We conducted 564 experiments and analyzed them on 4 different metrics.

On dedicated clusters, the cancellation of all the waiting jobs proves to be very efficient to improve the average job response time. On the other hand in an non dedicated environment, the algorithm that does not cancel waiting jobs behaves better. On both platforms, surprisingly, there is not a great number of migrating tasks, but all tasks take benefit of those migrations since the percentage of impacted tasks is high. In term of performances, users can expect more jobs finishing sooner, and an improvement of the jobs response time from a few percents to more than 50%! Only a few cases give bad results leading to an increase of the average job response time.

The next step of this work is the implementation of the reallocation mechanism in the DIET GridRPC middleware. DIET already provides most of the needed features. The missing features are the cancellation of a job in batch schedulers (numerous are supported) which is easy to implement and the reallocation mechanism itself. This last point should be quite straightforward because all communications are already handled by the middleware. We intend to implement both reallocation mechanisms with MCT. Indeed, we need the regular algorithm to work on non dedicated platforms. We plan also to implement the all-cancellation mechanism because DIET can be used in a dedicated environment. Furthermore, we could use this in the SPADES<sup>4</sup> project where we plan to maintain a set of reserved resources on a site which are managed by our own embedded batch scheduler.

<sup>4</sup>ANRProject08-ANR-SEGI-025

## References

- Beltrán, M. & Guzmán, A. (2009), 'The Impact of Workload Variability on Load Balancing Algorithms', *Scalable Computing: Practice and Experience* **10**(2), 131–146.
- Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E. & Touché, I. (2006), 'Grid'5000: A Large Scale and Highly Reconfigurable Experimental Grid Testbed', *International Journal of High Performance Computing Applications* **20**(4), 481–494.
- Caniou, Y., Caron, E., Charrier, G. & Desprez, F. (2009), Meta-Scheduling and Task Reallocation in a Grid Environment, in 'The Third International Conference on Advanced Engineering Computing and Applications in Sciences (ADV-COMP'09)', Sliema, Malta, pp. 181–186.
- Caniou, Y., Charrier, G. & Desprez, F. (2010a), Analysis of Tasks Reallocation in a Dedicated Grid Environment, in 'IEEE International Conference on Cluster Computing 2010 (Cluster 2010)', Heraklion, Crete, Greece. To appear.
- Caniou, Y., Charrier, G. & Desprez, F. (2010b), Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Dedicated and non Dedicated Grids, Technical Report RR-7365, Institut National de Recherche en Informatique et en Automatique (INRIA).
- Caniou, Y. & Gay, J. (2009), Simbatch: An API for simulating and predicting the performance of parallel resources managed by batch systems, in 'Workshop on Secure, Trusted, Manageable and Controllable Grid Services (SGS), in conjunction with EuroPar'08', Vol. 5415 of *LNCS*, pp. 217–228.
- Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P. & Richard, O. (2005), 'A Batch Scheduler with High Level Components', *CoRR* p. 9.
- Caron, E. & Desprez, F. (2006), 'DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid', *International Journal of High Performance Computing Applications* **20**(3), 335–352.
- Casanova, H., Legrand, A. & Quinson, M. (2008), SimGrid: a Generic Framework for Large-Scale Distributed Experiments, in '10th IEEE International Conference on Computer Modeling and Simulation'.
- Cirne, W. & Berman, F. (2002), 'Using Moldability to Improve the Performance of Supercomputer Jobs', *Journal of Parallel and Distributed Computing* **62**(10), 1571–1601.
- Feitelson, D. & Rudolph, L. (1998), Metrics and Benchmarking for Parallel Job Scheduling, in 'Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS/JSSPP '98)', Springer-Verlag, pp. 1–24.
- Feitelson, D., Rudolph, L. & Schwiegelshohn, U. (2004), Parallel job scheduling - a status report, in 'Job Scheduling Strategies for Parallel Processing'.
- Feitelson, D., Rudolph, L., Schwiegelshohn, U., Sevcik, K. & Wong, P. (1997), Theory and Practice in Parallel Job Scheduling, in 'IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing', Springer-Verlag, pp. 1–34.

- Feitelson, D. & Tsafrir, D. (2006), Workload Sanitation for Performance Evaluation, in 'IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)', pp. 221–230.
- Guim, F. & Corbalán, J. (2008), A Job Self-scheduling Policy for HPC Infrastructures, in 'Job Scheduling Strategies for Parallel Processing (JSSPP)'.
- Huang, K., Shih, P. & Chung, Y. (2009), 'Adaptive Processor Allocation for Moldable Jobs in Computational Grid', *International Journal of Grid and High Performance Computing* **1**(1), 10–21.
- Jackson, D., Snell, Q. & Clement, M. (2001), Core Algorithms of the Maui Scheduler, in 'JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing', Springer-Verlag, pp. 87–102.
- Kannan, S., Roberts, M., Mayes, P., Brelsford, D. & Skovira, J. (2001), *Workload Management with LoadLeveler*, IBM Press.
- Lifka, D. (1995), The ANL/IBM SP scheduling system, in 'In Job Scheduling Strategies for Parallel Processing', Springer-Verlag, pp. 295–303.
- Maheswaran, M., Ali, S., Siegel, H., Hensgen, D. & Freund, R. (1999), 'Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems', *Journal of Parallel and Distributed Computing* **59**, 107–131.
- Mu'alem, A. & Feitelson, D. (2001), 'Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling', *IEEE Transactions on Parallel and Distributed Systems* **12**(6), 529–543.
- Sato, M., Nakada, H., Sekiguchi, S., Matsuoka, S., Nagashima, U. & Takagi, H. (1997), Ninf: A Network Based Information Library for Global World-Wide Computing Infrastructure, in 'HPCN Europe', pp. 491–502.
- Seymour, K., Lee, C., Desprez, F., Nakada, H. & Tanaka, Y. (2004), The End-User and Middleware APIs for GridRPC, in 'Workshop on Grid Application Programming Interfaces, In conjunction with GGF12'.
- Sonmez, O., Yigitbasi, N., Iosup, A. & Epema, D. (2009), Trace-Based Evaluation of Job Runtime and Queue Wait Time Predictions in Grids, in 'International Symposium on High Performance Distributed Computing (HPDC'09)'.
- Sudarsan, R. & Ribbens, C. (2010), 'Design and performance of a scheduling framework for resizable parallel applications', *Parallel Computing* **36**, 48–64.
- Yue, J. (2004), 'Global Backfilling Scheduling in Multiclusters', *Lecture notes in Computer Science* **3285**, 232–239.



# Parallel Vertex Cover: A Case Study in Dynamic Load Balancing

Dinesh P. Weerapurage, John D. Eblen, Gary L. Rogers Jr. and Michael A. Langston

Department of Electrical Engineering and Computer Science  
University of Tennessee  
Knoxville TN 37996-3450  
USA  
{dpemala, eblen, grogers, langston}@eecs.utk.edu

## Abstract

The significance of dynamic parallel load balancing is considered in the context of fixed parameter tractability. The well known vertex cover problem is used as a case study. Several algorithms are developed and tested on graphs derived from real biological data. Implementations are carried out on the Kraken supercomputer, currently the world's fastest computational platform for open science. We show that for certain difficult instances of biological data graphs our approach scales well up to 2400 processors.

**Keywords:** parallel computation, load balancing, data mining, genome scale analysis

## 1 Introduction

Given a graph  $G = \langle V, E \rangle$  and an integer  $k$  ( $k \leq n$ ,  $n = |V|$ ), the vertex cover problem asks whether  $V$  contains a subset  $C$  of size at most  $k$  so that every member of  $E$  has at least one endpoint in  $C$ . Applications are myriad, in large part because vertex cover is so easily transformed into clique and its dual (independent set). A huge number of domains are amenable (Bomze et al. 1999). Specific examples include bio-informatics (Samudrala 2006), chemistry (Rhodes et al. 2003), electrical engineering (Prihar 1956), and even social networks (Luce & Perry 1949). Vertex cover is fixed parameter tractable (FPT) for every fixed  $k$  (Downey & Fellows 1999). The asymptotically fastest currently known algorithm runs in  $O(1.2738^k + kn)$  time (Chen et al. 2006).

Despite the potential efficiencies offered by FPT, vertex cover remains  $\mathcal{NP}$ -complete (Garey & Johnson 1990). Accordingly, computational burdens are often onerous. Here we are interested chiefly in effective parallel load balancing schemes. We shall describe our work as follows. In Section 2, we present a simple sequential FPT vertex cover algorithm. Next, in Section 3, we devise a relatively straightforward parallel method employing a static form of load balancing to distribute the effort across processors. In Section 4, we develop a more complex but dynamic load balancing approach. A variety of experimental results are discussed in Section 5. In a final section

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 ©2011, Australian Computer Society, Inc. This paper appeared at the 9th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 118. J. Chen and R. Ranjan, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

we list a number of conclusions and directions for future research.

## 2 A Sequential Vertex Cover Algorithm

In the decision version of vertex cover, we require only a “yes” or “no” answer. In the case of “yes” we also solve the search version by producing a satisfying cover. To solve the optimization version, we employ the decision algorithm along with binary search to find the smallest  $k$  for which the answer is “yes.”

### 2.1 Kernelization

FPT's success as an algorithm design paradigm relies largely on kernelization, a form of problem reduction that ensures a compute core whose size depends only on  $k$  and not on  $n$  ( $n = |V|$ ). If  $(G, k)$  is an instance of vertex cover, an effective kernelization routine produces another instance  $(G', k')$ , in which  $k' \leq k$  and the number of vertices in  $G'$  is bounded above by some polynomial function of  $k'$ .  $G$  has a cover of size  $k$  if and only if  $G'$  has a cover of size  $k'$ . Numerous kernelization procedures have been proposed (Abu-Khzam et al. 2001, Chen et al. 2001). Those with the most theoretical appeal tend to be the most challenging to implement, while those with the simplest structure are often the most effective on real data. We have thus adopted only the following kernelization rules for this study.

#### Rule 1 (The degree one rule):

A vertex with degree one is excluded from the cover. This is because there is no gain in including a pendant vertex to cover its only neighbor.

#### Rule 2 (The high degree rule):

A vertex whose degree exceeds  $k$  must be in the cover. Otherwise, all of its neighbors must be in the cover, which is impossible.

In addition to the above two rules, there are kernelization algorithms to preprocess 2-degree vertices, 3-degree vertices and even higher degree vertices, but they become successively more difficult to implement. Other kernelization methods include linear programming algorithms (Abu-Khzam et al. 2006) and crown decomposition (Abu-Khzam et al. 2001). Analyzing the performance of different kernelization algorithms and implementations is a broader topic that requires a separate paper. Here, we focus our efforts on the hard computational core that kernelization produces, because that is where exponential run time occurs.

## 2.2 Decomposition and search

After reducing the graph using the two kernelization rules just described, the kernel should be searched for solutions. In FPT terminology this step is known as “branching.” The challenge is that the reduced problem core still might have an exponential number of solution candidates. We use a branching algorithm, which utilizes a search tree, to traverse through the search space. Each branch of the search tree represents a possible solution, and after searching the tree to a certain depth, we can make a decision on whether that particular branch contains a solution or not. This is an exhaustive search, and sometimes the entire solution space must be searched, such as when no solution exists.

In the branching tree, each branch is a possible choice for building the vertex cover. Each branch adds one or more vertices to the cover, and  $k$  is updated accordingly. If the value of  $k$  becomes zero or less than zero, a decision is made about the cover. If it is a valid cover, the search is ceased. Otherwise, searching is resumed on another branch.

Branching is illustrated in Figure 1. Letters in the box represent the current vertex cover while the graphs on either side show the modified graph. Black vertices represent vertices that have been deleted, whereas white vertices are vertices still in the graph. The neighbors of vertex  $v$  are shown as  $N(v)$  and number of neighbors as  $|N(v)|$ . First, vertex  $v$  is selected for the vertex cover in the left branch, which can be seen in the left box. Then the  $k$  value is reduced by one. The right branch contains a choice without vertex  $v$ . All edges connected to  $v$  should be covered. Therefore all  $N(v)$  should be in the vertex cover. Thus, the right box has vertex cover without  $v$  but including  $N(v)$ . Also,  $k$  is updated to  $k - |N(v)|$ .

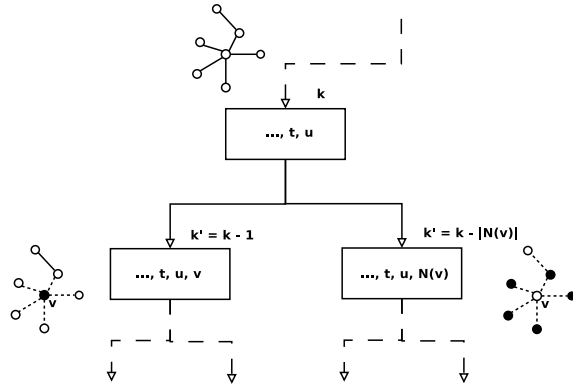


Figure 1: A branching algorithm

Figure 2 shows a simple graph with 7 vertices and 9 edges and Figure 3 shows the decomposition steps for  $k = 4$  on this graph. As in Figure 1, numbers in the boxes show the partial cover found at each branch. Graphs on either side illustrate the modified graph for each branch. Dotted lines are deleted edges, black vertices are deleted vertices, and white vertices are vertices in the graph. The  $k$  value in the upper right hand corner of each box is the updated  $k'$  value for the modified graph  $G'$ . When the cover size is  $k$  and all edges in the graph are covered, it is a valid vertex cover, depicted as a box within a box. Otherwise it is an invalid cover, depicted as a dashed-line box within a box. In this example, a current highest degree vertex is selected for the vertex cover. A vertex cover is found in the leftmost branch, and the search can be ceased. In the case where a graph might be structured in such a way that covers are in the rightmost branches, the algorithm has to search through all left branches before reaching the right side. It is important to note that kernelization can also

be applied during branching in subsequent recursive calls. This is termed “interleaving” (Niedermeier & Rossmanith 2000), which can help to reduce problem instances and improve algorithm performance.

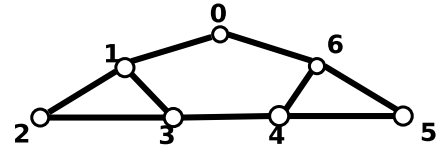


Figure 2: An example graph

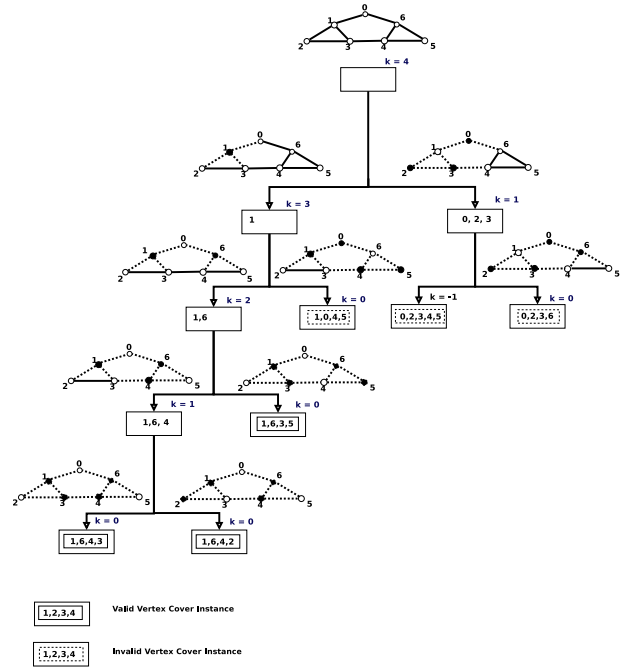


Figure 3: Branching tree for graph of Figure 2, with  $k = 4$

## 3 Parallel Vertex Cover

Decomposing a graph is a computationally demanding operation, and thus distributing work among several processors will help to decompose and search the graph more effectively. During the decomposition stage, branches are independent of each other, therefore branches can be searched separately and results are reported back when the computation is completed. The implementation has been done using MPI (Dongarra & Walker 1994), (Gropp et al. 1999) given that it is widely adopted and scales well. We employ a master-worker architecture, where each processor is assigned an independent branch to compute.

In Figure 4, experimental data on real biological data graphs are shown. The data sets, related to “Folic acid deficiency effect on colon cancer cells,” “Low concentrations of 17beta-estradiol effect on breast cancer cell line,” and “Interferon receptor deficient lymph node B cell response to influenza infection,” were obtained from NCBI (NCBI 2010). The graphs were created using genes as vertices and Pearson correlation p-values as the weights of edges between all pairs of genes. The graphs were then thresholded using p-values (0.45, 0.40, 0.35 and 0.30) and converted to undirected graphs. This is done by removing edges between vertex pairs that have a p-value greater than the threshold value. The minimum vertex cover of each graph was found using both the sequential algorithm

Graph	Sequential	Parallel
fo-0.45	2384.67	2400.00
fo-0.40	10023.07	9870.77
fo-0.35	29792.09	28502.76
fo-0.30	>1day	>1day
est-0.45	915.18	894.21
est0.40	14004.66	2630.92
est-0.35	12289.54	12043.73
est-0.30	56335.00	56150.65
inf-0.45	305.34	290.04
inf-0.40	696.76	671.87
inf-0.35	2089.13	2005.68
inf-0.30	7440.08	7243.81

Table 1: Sequential and parallel execution times (in seconds) for sample application graphs

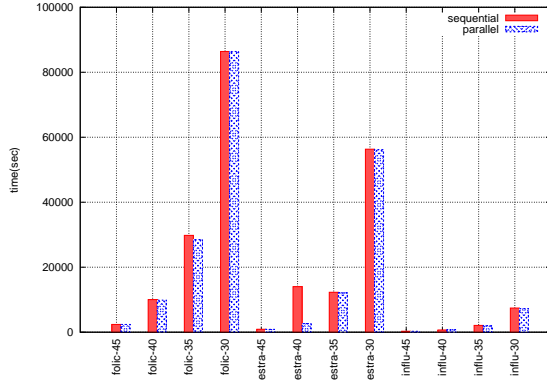


Figure 4: A comparison of sequential and parallel performance

and the parallel algorithm using a computing cluster that consisted of 32 Intel 3.2GHz processors, each with 4GB of main memory. Runs were limited to 24 hours (86400 seconds). Table 1 summarizes the results. (The folic-30 graph did not finish within 24 hours and the program was terminated.)

When using  $n$  processors for parallel algorithms,  $n$ -times speedup is expected, but these experiments fall far short of that goal. For instance, with the sequential algorithm folic-35 took 29792 seconds. With 32 processors, it took 28502.76 seconds. So even though 96% improvement in execution time was hoped, the actual execution time speedup was as low as 4.3%. The process utilization for the static load balancing algorithm for folic-35 is shown in figure 5. Low throughput happens as a result of uneven process utilization. (One processor does nearly all of the work.)

As we have seen in Figure 4, assigning independent branches to different processors is not always optimal, in terms of runtime. Given that not all branches are equal in terms of complexity, some processors work longer on difficult branches and other processors finish their job quickly. Even though searching is distributed to different processors, the workload will reduce to only a few number of busy processors quickly. If we can use idling processors to assist difficult branches it will help to increase overall efficiency. A load balancing algorithm will help to distribute the load across all processors evenly.

#### 4 Dynamic Load Balancing

We applied a scheduler-based load balancing algorithm to the parallel vertex cover algorithm. Several load balancing mechanisms for parallel algorithms have been in-

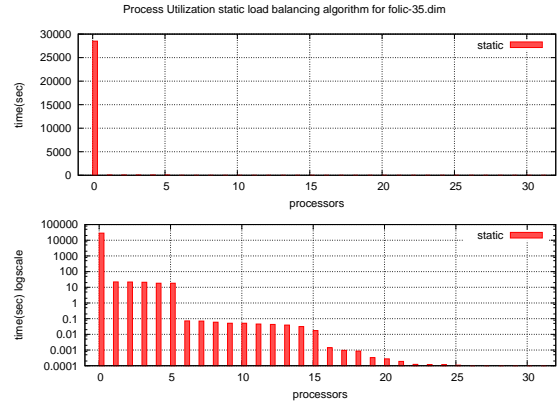


Figure 5: Processor utilization in static load balancing

vestigated in (Kumar et al. 1994). Six receiver initiated and three sender initiated algorithms have been discussed. Among them, a scheduler based algorithm (receiver initiated), which was also proposed in (Patil & Banerjee 1989), is more suitable for load balancing the parallel vertex cover algorithm. The sender initiated algorithm would be a push model that is hard to implement as the working processors finish time is unknown during execution. Also, since our algorithms select the current highest degree vertex during branching, we suspect that the leftmost branch is the hardest branch. So for balanced process utilization, idling processors should assist the leftmost branch.

Previously, at least two dynamic load balancing algorithms for parallel vertex cover have been suggested in (Abu-Khzam et al. 2006) and (Baldwin et al. 2004). Also, load balancing has been discussed in (Taillon 2007). Stack splitting and search frontier splitting methods, which were introduced in (Reinefeld & Schnecke 1994) and (Reinefeld 1994) have been discussed. The algorithm discussed in this paper is slightly different than the above two algorithms because this algorithm is tightly coupled with the branching tree whereas the aforementioned algorithms are more general load balancing algorithms. However, one could define it as a variation of the stack splitting method (Reinefeld 1994), because work is partitioned on demand to be delivered to the requester. The hardest instance is designated as the donor, and it is the job donating process. Therefore, the scheduler does not need to maintain a job queue. As we explained in the previous paragraph the hardest instance occurs in the leftmost branch. Processors that need more jobs contact the scheduler, who redirects job requests to the donor. Jobs are then delivered to the requesting processor directly. This process is shown in the diagram in Figure 6.

The diagram in Figure 7 shows the termination process. The scheduler is notified when the donor runs out of jobs. The scheduler then terminates the donor, and any processor requesting more work will be issued a termination signal.

Execution times in Figure 4 are compared with execution times for the dynamic load balancing algorithm in Figure 8. Some experiments with dynamic load balancing finished so quickly that DLB bars are not visible for some graphs. In order to have a clearer comparison, Figure 9 uses a log scale for execution times.

Although the parallel vertex cover algorithm with dynamic load balancing outruns both the parallel vertex cover algorithm without load balancing and the sequential algorithm, a load imbalance still occurs once the donor finishes to address this issue donor update algorithm is implemented.

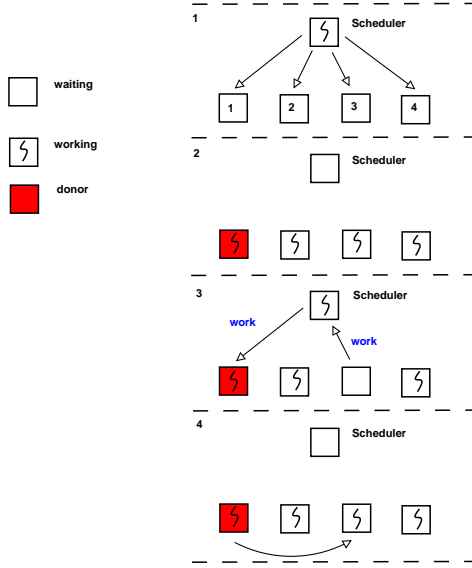


Figure 6: A dynamic load balancing algorithm

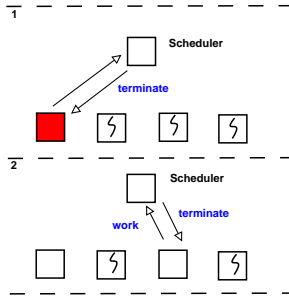


Figure 7: Load balancing termination

Graph	Sequential	Parallel	DLB
fo-0.45	2384.67	2400.00	108.28
fo-0.40	10023.07	9870.77	430.19
fo-0.35	29792.09	28502.76	1265.137
fo-0.30	>1day	>1day	6448.77
est-0.45	915.18	894.21	43.74
est-0.40	14004.66	2630.92	121.90
est-0.35	12289.54	12043.73	525.93
est-0.30	56335.00	56150.65	2485.52
inf-0.45	305.34	290.04	17.35
inf-0.40	696.76	671.87	33.70
inf-0.35	2089.13	2005.68	96.00
inf-0.30	7440.08	7243.81	345.50

Table 2: Sequential, parallel and DLB execution times (in seconds)

#### 4.1 Donor Update Algorithm

For the previous algorithm, after the initial job distribution, one processor will be designated as the donor, which is terminated when the work in the assigned branch is finished. All processors with subsequent calls for more jobs will be terminated too. At this moment any processor that has a difficult branch will work longer while others are being terminated. In order to achieve balanced process utilization, processors that finished should be used to support those still working.

To solve this problem, we create a donor update algorithm, which works as follows. The initial donor keeps track of the degree structures of the jobs that were dis-

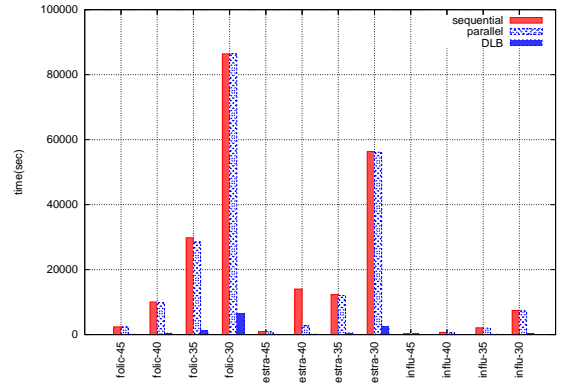


Figure 8: A comparison of sequential, parallel and DLB performance

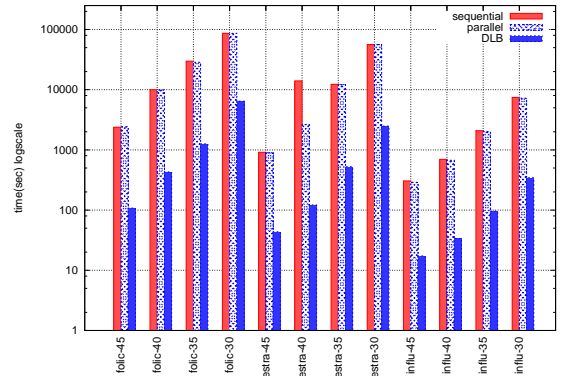


Figure 9: A log scale view of relative performance

tributed to workers. Upon termination the donor calculates the average degree for the last jobs distributed to the workers. Then these average degrees are sent to the scheduler, who selects the next donor. The scheduler selects the worker with the highest average degree job as the new donor.

It is important to note that only the first donor keeps track of the jobs sent to workers, because keeping track of jobs is expensive both in terms of memory and cycles. Moreover, initial donor is the hardest instance and all other processors have work that is donated from the initial donor. Therefore we assume that the jobs later donors have are not as hard as the jobs initial donor had. It is not effective to spend resources to obtain information from relatively easy jobs.

Initial donor sends degree information to the scheduler upon termination whereas all other donors terminate without sending additional information. The scheduler uses degree information to select next donors. After the initial donor exited, degree information for workers who ask for more jobs will be removed. As a result, degree information for the busy workers who later will be selected as donors will be remained. After all degree information are used, workers are issued the termination signal. The donor update process is depicted in Figure 10.

## 5 Experimental Results and Discussion

Twelve biological data graphs were tested using the decision version of the vertex cover algorithm, and a cluster of 32 Intel 3.2GHz processors with 4GB of main memory



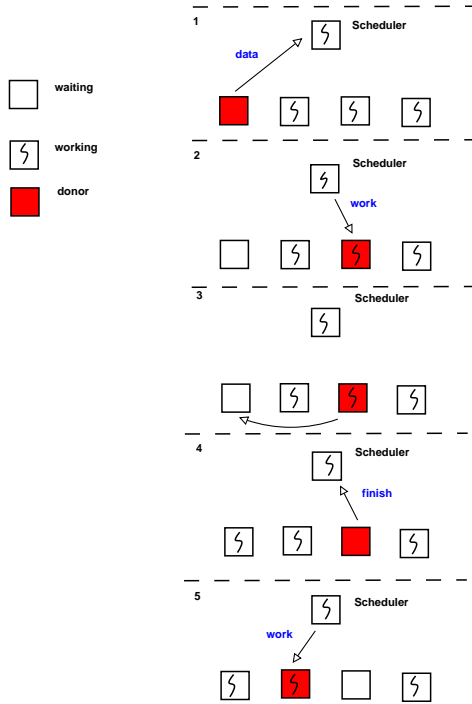


Figure 10: Donor update algorithm

Graph	Sequential	Parallel	DLB
fo-0.45	306.52	293.70	9.18
fo-0.40	147.13	140.30	35.18
fo-0.35	647.76	646.61	42.84
fo-0.30	6837.19	6831.89	339.28
est-0.45	23.46	23.43	2.5
est-0.40	3728.79	3637.11	33.00
est-0.35	836.74	837.04	55.13
est-0.30	3831.91	3812.89	216.68
inf-0.45	1.0	-	-
inf-0.40	0.6	-	-
inf-0.35	789.62	757.09	45.13
inf-0.30	3.0	3.3	0.59

Table 3: Times for hardest “yes” instances

was used for testing. Results in Table 3 are for the hardest “yes” instance while data in Table 2 are for the hardest “no” instance. When the parallel program is working on a “yes” instance, the search will be terminated as soon as one processor finds a solution, whereas in a “no” instance, the whole search space has to be searched. For example, in Table 2 for influenza-30, the graph vertex cover program spent 7440 seconds on the hardest “no” instance. On the other hand, in Table 3, the program spent only 3.0 seconds to find a “yes” instance. Thereby, the execution time of a “yes” instance cannot be used for meaningful comparisons. Hence execution times for “no” instances were used. A sequential program for a “yes” instance for influenza-0.45 and influenza-0.40 graphs finished very quickly (less than 1.0 sec). Thus a parallel program was not required.

Execution times for sequential algorithm, static load balancing (or parallel algorithm without explicit load balancing algorithm) and dynamic load balancing algorithm were compared in Table 2. Although the static load balancing algorithm uses 32 processors, it did not gain expected performance. As we saw in Figure 5, that happened as a result of uneven distribution of work load. The parallel algorithm with static load balancing will boil down to a small number of busy processors quickly, therefore

producing little improvement in execution time. In order to address this issue, a dynamic load balancing algorithm was introduced. The algorithm is explained in detail in Section 4. Dynamic load balancing gives 20–25 times speedup for the biological data graphs that have been selected. However, this speedup is still sub-linear. The main reason for this slowdown is communication between donor and workers, which is indeed inevitable. The intent of this dynamic load balancing algorithm is to have regular load distribution among the workers and to minimize idle time.

Table 2 lists execution time for the sequential, parallel and dynamic load balancing algorithms. Based on the results in Table 2, folic-30, estradio-30 and influenza-30 are the hardest instances in their category. Hardest instances were selected for further analysis of process utilization by the dynamic load balancing algorithm and the donor update algorithm. In this experiment, processor idling time is considered as the time the processor is idle. Time for communication is not counted as idling time.

Graph	Execution Time	Idling Time
fo-0.30	731.90	9.15
est-0.30	317.50	5.24
inf-0.30	32.36	0.92

Table 4: Execution and idling times for DLB on 120 processors

Graph	Execution Time	Idling Time
fo-0.30	694.16	62.87
est-0.30	276.00	25.82
inf-0.30	34.71	3.88

Table 5: Execution and idling times for DU on 120 processors

After completing all tests using the aforementioned cluster, load balancing codes were moved to the highly capable Kraken supercomputer at ORNL. Load balancing algorithms were tested for scalability ranging from 12 processors to 2400 processors. Each node on Kraken has two 2.6 GHz six-core AMD Opteron processors and contains 16 GB of main memory. Table 4 has information on execution time and idling time with the dynamic load balancing (DLB) algorithm. Also, Table 5 has execution time and idling time for the donor update (DU) algorithm. The DU algorithm has larger idling time compared to the DLB algorithm. Although DU algorithm has higher idling time it performs well compared to the DLB algorithm. Figure 11 has idling time as a percentage of execution time for folic-30 graph with 120 processors for both DLB and DU algorithms. DLB algorithm has less than 1% of idle time whereas DU algorithm has up to 10% of idle time. There is extra communication in DU compared to DLB as a result processors have to wait longer. However in DU collectively processors perform better than DLB because extra communication in DU make sure to occupy idling processors with work whereas in DLB after donor terminates workers who got harder job work harder while others terminating.

Both the DLB algorithm and the DU algorithm were tested for scalability using the hardest graph instances (folic-30, estradio-30 and influenza-30). The execution time plot for folic-30 is presented in Figures 12. Although execution times for folic-30, estradio-30 and influenza-30 were measured, only one plot of execution time is included because all three graphs are similar in shape. Normal measurement does not show any difference in execution time, so a log scale was applied. Note that the third plot from Figure 12 shows the difference in execution time between the DLB and DU algorithms more clearly than

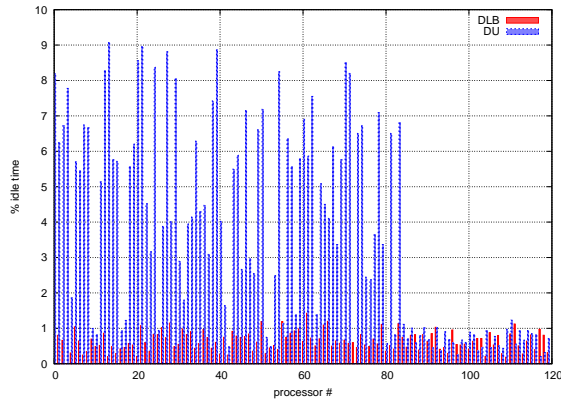


Figure 11: Percentage idle time for DLB and DU algorithms with folic-30 graph

the first plot. With smaller number of processors, both the DLB and the DU algorithm almost follow expected (linear) execution time, but for higher number of processors DU execution time is closer to the expected value than the DLB execution time. Note that speedup for each algorithm was computed by dividing sequential execution time by number of processors. Expected speedup is indicated as linear speedup on the plots. In addition an error bar representation is available on the speedup plots to show the consistency of the results.

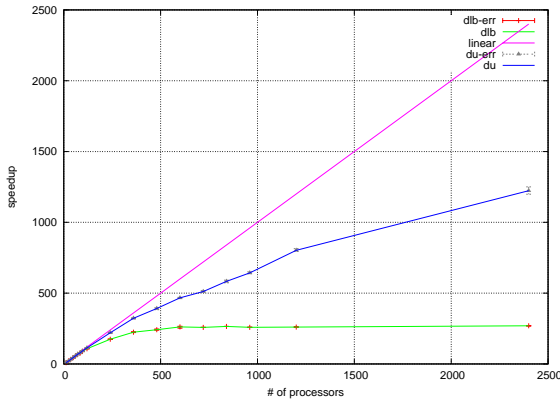


Figure 13: Speedups for folic-30 graph on Kraken

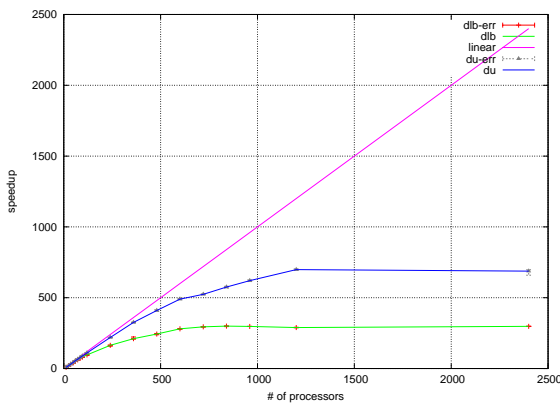


Figure 14: Speedups for estradio-30 graph on Kraken

Curves in Figures 13, 14 and 15 have noticeable

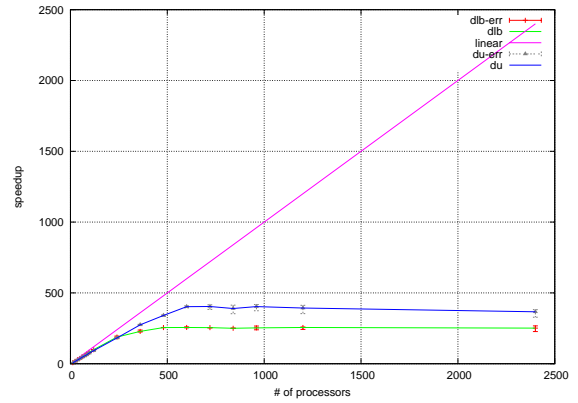


Figure 15: Speedups for influenza-30 graph on Kraken

bends because smaller samples were chosen for the experiment. The sample includes 12, 24, 36, 48, 72, 120, 240, 480, 600, 720, 840, 960, 1200 and 2400 processors. Plots were produced using data for the aforementioned sample. If experiments are done for a larger sample smoother curves can be produced.

The DU algorithm scales better for larger numbers of processors than the DLB algorithm. As seen in Figure 13, DLB does not scale up after 120 processors. Because after 120 processors donor finishes quickly and workers who get a harder job end up working longer, which leads to a smaller magnitude load imbalance again. On the other hand, the DU algorithm makes sure to assist harder jobs using idling processors until all harder jobs are finished. Therefore DU scales up well even up to 2400 processors.

Even though scalability of the DU algorithm can be clearly seen in Figure 13, in Figure 14 and in Figure 15 it is not as clear. Primarily this is because the estradio-30 and influenza-30 graphs are smaller compared to the folic-30 graph and thus not as hard. Since estradio-30 and influenza-30 graphs are not much harder there is not much work for larger number of processors hence communication overhead dominates the execution time.

After doing these experiments several times, maximum, average and minimum values for execution time are obtained. Average values use for plots, in addition minimum and maximum values are shown using an error bar representation. Error bars for all three graphs are small, which implies that execution time (and speedup) are consistent for both DLB and DU algorithms.

## 6 Conclusions and Direction for Future Research

Vertex cover remains an  $\mathcal{NP}$ -complete problem. As such, its computational requirements can be staggering. Yet it has many practical applications that demand exact solutions. Fixed parameter tractability is one approach for dealing with this conundrum. Kernelization and decomposition are key. Decomposition in particular can be based on independent branching actions, which are inherently parallel.

We initially developed a simple parallel decomposition algorithm. Different branches were statically assigned to different processors. Once a job finished, results were reported. Unfortunately, this scheme does not always scale well. Some branches are much harder than others, in which case one or only a few processors are left to do most of the work.

We therefore devised a dynamic load balancing algorithm. With it, the branch with what is perceived as the hardest task is selected as a donor, which then distributes

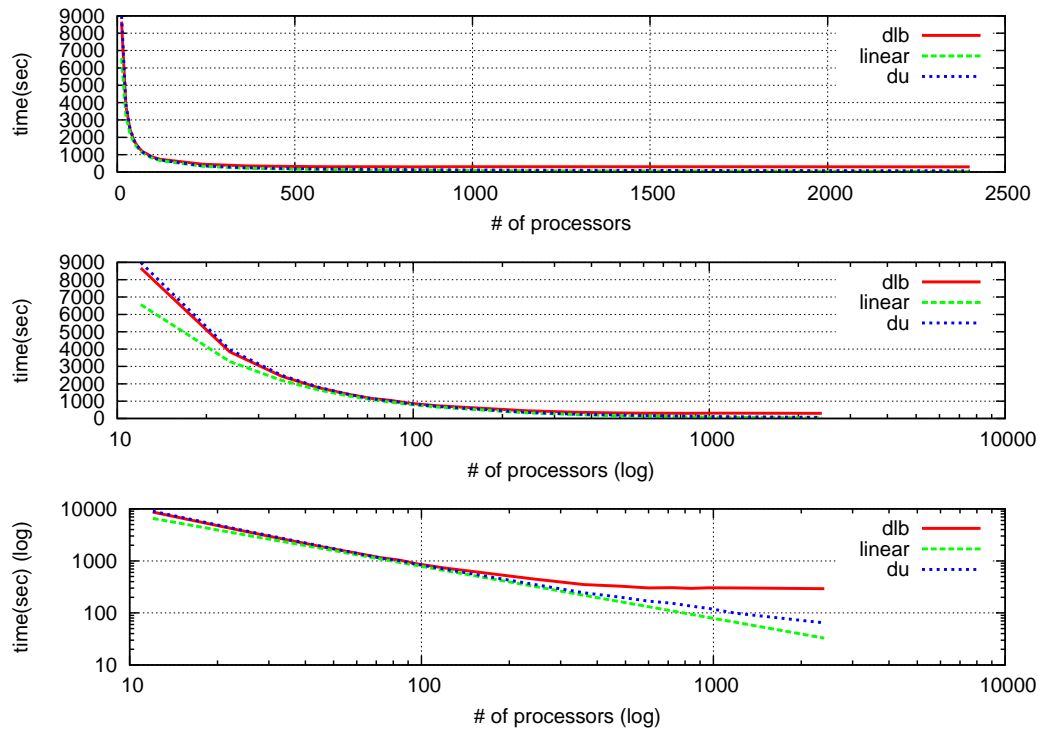


Figure 12: Execution times for folio-30 graph on Kraken

jobs to idling processors. Requests for more jobs will be made upon completion of a current job. There is a possibility of load imbalance, so a donor update algorithm is also implemented. When a current donor finishes, it will find a next donor and so on.

Primary testing was done using an in-house cluster containing 128 processors. More in-depth scalability analysis was performed on the Kraken supercomputer. With sufficiently difficult instances, dynamic load balancing shows linear speedup up to 2400 processors. On graphs without imposing execution times, we see only sub-linear speedup as communication overhead dominates.

We have previously observed numerous problems with static load balancing, curiously more so on real than on synthetic data. Thus this work was intended mainly as a case study. Nevertheless, we have found the switch to dynamic load balancing to be fairly straightforward and quite effective. We think it would be interesting to experiment with dynamic load balancing algorithms on other FPT problems. It may also be interesting to extend scalability testing to larger numbers of processors.

## References

- Abu-Khzam, F. N., Collins, R. L., Fellows, M. R., Langston, M. A., Suters, W. H. & Symons, C. T. (2001), 'Kernelization algorithms for the vertex cover problem: Theory and experiments (extended abstract)', *Combinatorial Optimization*, Kluwer Academic Publishers, pp. 1–74.
- Chen, J., Kanj, I. A. & Jia, W. (2001), 'Vertex cover: Further observations and further improvements', *Journal of Algorithms* **41**, 313–324.
- Chen, J., Kanj, I. & Xia, G. (2006), Improved parameterized upper bounds for vertex cover, in R. Krlovic & P. Urzyczyn, eds, 'Mathematical Foundations of Computer Science 2006', Vol. 4162 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 238–249.
- Dongarra, J. J. & Walker, D. W. (1994), 'MPI: A message-passing interface standard', *International Journal of Supercomputing Applications* **8**(3/4), 159–416.
- Downey, R. & Fellows, M. (1999), *Parameterized Complexity*, Springer, Berlin.
- Garey, M. R. & Johnson, D. S. (1990), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA.
- Gropp, W., Lusk, E. & Skjellum, A. (1999), *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*, MIT Press, Cambridge, MA, USA.
- Kumar, V., Grama, A. Y. & Vempaty, N. R. (1994), 'Scalable load balancing techniques for parallel computers', *J. Parallel Distrib. Comput.* **22**(1), 60–79.
- Luce, R. & Perry, A. (1949), 'A method of matrix analysis of group structure', *Psychometrika* **14**, 95–116. [10.1007/BF02289146](https://doi.org/10.1007/BF02289146).  
URL: <http://dx.doi.org/10.1007/BF02289146>
- NCBI (2010), 'The national center for biotechnology information advances science and health by providing access to biomedical and genomic information.' [Online; accessed 10/10/2010].
- Bomze, I. M., Budinich, M., Pardalos, P. M. & Pelillo, M. (1999), The maximum clique problem, in 'Handbook of

accessed 02-May-2010].

**URL:** <http://www.ncbi.nlm.nih.gov/>

Niedermeier, R. & Rossmanith, P. (2000), 'A general method to speed up fixed-parameter-tractable algorithms', *Inf. Process. Lett.* **73**(3-4), 125–129.

Patil, S. & Banerjee, P. (1989), A parallel branch and bound algorithm for test generation, in 'DAC '89: Proceedings of the 26th ACM/IEEE Design Automation Conference', ACM, New York, NY, USA, pp. 339–343.

Prihar, Z. (1956), 'Topological properties of telecommunication networks', *Proceedings of the IRE* **44**(7), 927–933.

Reinefeld, A. (1994), Scalability of massively parallel depth-first search, in 'In DIMACS Workshop', American Mathematical Society, pp. 305–322.

Reinefeld, A. & Schneck, V. (1994), Work load balancing in highly parallel depth first search, in 'In Scalable High Performance Computing Conference', pp. 773–780.

Rhodes, N., Willett, P., Calvet, A., Dunbar, J. B. & Humblet, C. (2003), 'Clip: similarity searching of 3d databases using clique detection', *Journal of Chemical Information and Computer Sciences* **43**(2), 443–448. PMID: 12653507.

**URL:** <http://pubs.acs.org/doi/abs/10.1021/ci025605o>

Samudrala, Ram; Moult, J. (2006), 'A graph-theoretic algorithm for comparative modeling of protein structure', *Journal of Molecular Biology* **279**(1), 287–302.

Taillon, P. J. (2007), On improving fpt k-vertex cover, with applications to some combinatorial problems, PhD thesis, Carleton University, Ottawa, Canada.



# A parallel approach to social network generation and agent-based epidemic simulation

Dimitri Perrin<sup>1,2</sup>

Hiroyuki Ohsaki<sup>2</sup>

<sup>1</sup> Centre for Scientific Computing & Complex Systems Modelling  
Dublin City University  
Dublin, Ireland

Email: dperrin@computing.dcu.ie

<sup>2</sup> Information Sharing Platform Laboratory  
Department of Information Networking  
Graduate School of Information Science and Technology, Osaka University  
1-5, Yamadaoka, Suita, Osaka 565-0871, Japan  
Email: oosaki@ist.osaka-u.ac.jp

## Abstract

Understanding the dynamics of disease spread is essential in contexts such as estimating load on medical services, as well as risk assessment and intervention policies against large-scale epidemic outbreaks. However, most of the information is available after the outbreak itself, and preemptive assessment is far from trivial. Here, we report on an agent-based model developed to investigate such epidemic events in a stylised urban environment. For most diseases, infection of a new individual may occur from casual contact in crowds as well as from repeated interactions with social partners such as work colleagues or family members. Our model therefore accounts for these two phenomena. Given the scale of the system, efficient parallel computing is required. In this presentation, we focus on aspects related to parallelisation for large networks generation and massively multi-agent simulations.

**Keywords:** Agent-based computing; Complex networks; Epidemics; Large-scale simulations; MPI; Parallelisation.

## Extended abstract

Dynamics of disease spread within a population are of crucial importance in terms of public health, (e.g. monitoring of existing outbreaks, evaluation of intervention policies). In order to avoid being limited to observation and subsequent intervention, computational models offer tools for *preemptive* analysis and decision-making. In this context, models have to deal with millions of people living in modern urban environments, each with a refined social behaviour. To date, most approaches have focused on tackling either one aspect or the other.

Funding from Dublin City University for the seminal work, and subsequently through an IRCSET - Marie-Curie International Mobility Fellowship in Science, Engineering and Technology, is warmly acknowledged. The authors would also like to thank the SFI/HEA Irish Centre for High-End Computing (ICHEC) for the provision of computational facilities and support.

Copyright ©2011, Australian Computer Society, Inc. This paper appeared at the 9th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 118. J. Chen and R. Ranjan, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Network-based models have been used to investigate the impact of social structures on disease spread, (see e.g. Kretzschmar & Wiessing (1998)). This is motivated by the fact that, for most infectious diseases, contact is required for a new infection to occur, (sexually-transmitted infections being obvious examples). Social structures are represented by a network where nodes correspond to individuals (or groups), and edges correspond to social links between these. Infection spread is then implemented as a stochastic propagation over the network (Chen *et al.* 2008).

There are, however, two limitations to this approach. First, networks with million of nodes are difficult to obtain from real data, or to generate *ab initio*. More crucially, because they are based on social structures, these models can not account for casual contacts between strangers, (e.g. in crowded areas and public transports), which are prevalent in common infectious diseases such as influenza.

Conversely, agent-based approaches are suited to model such infections between strangers, which emerge from individual behaviour: an infection between travellers on a bus occurs due to individual choices from each, which lead to them boarding, and not because of any link between them, (as opposed to colleagues who have to be in the same office, because of this work relationship). Such agent-based models can be efficiently parallelised and used for large-scale complex systems, as described previously for the immune model (Perrin *et al.* 2009). The main limitation, however, is the lack of a formal framework to include social complex structures.

Given the limitations of both paradigms, it becomes apparent that a hybrid model is the most efficient solution, combining elements of both approaches. In particular, network-based concepts can be used to generate socially-realistic populations, while the agent basis can simulate of an epidemic outbreak within these. This hybrid approach was successfully implemented, providing a realistic framework on which to investigate disease outbreaks and related policies (Claude *et al.* 2009).

In this presentation, we will detail the network generation aspects, and the approach taken to parallelise massively multi-agent simulations.

In our context, it is necessary for simulations to handle the social structures corresponding to a large population. To do so, we combine several types of social networks: household links between people living together, (whether they are part of the same fam-

ily or not), friendship links between people living in distinct households, (also covering extended family links), colleague relationships between co-workers, and considerations of sexual partnerships. These are linked through an overall network, which represents social participation.

The algorithm we will detail in the presentation, integrates these three layers. The required number of “social nodes” for the network is first created, where each network node corresponds to one agent in the subsequent simulations. These are distributed by age groups. Household and colleague relationships are generated from publicly available data, (e.g. Census for type and size distribution of households). Households are created by gathering selected nodes together.

Friendship relations are created using a network generation algorithm adapted from Keeling (2005). This algorithm is used here to generate a network of households. When two households are connected, some members are also cross-connected as “friends” (e.g. adult household members, children in similar age groups). These links are categorised as friendship type, but may also represent more distant familial relations.

This Keeling generation algorithm has been showed to be very useful for epidemic modelling, (see e.g. Badham (2008)). A standard implementation, using an adjacency matrix to store social links, would however be limited in terms of the size of networks it can handle. To address this, we re-implement and optimise the algorithm, taking into account that, if in theory any pair of individuals could be linked, in practice the number of links remain relatively low. A characteristic of social networks is, indeed, to have a relatively low average node degree.

The key idea is to store links directly within the nodes, as a list of neighbours. On a densely connected network, this would not be advisable. However, a 50,000-individual social network would only require storing 10 millions values if the average degree was 100, (which would be relatively large for such networks). Long integers occupy more memory space than booleans, but this still represents a 30-fold reduction in memory requirements, compared to a matrix-based storage that would involve 2.5 billion booleans, (i.e. over 2 Gb of memory).

This optimisation of memory usage enables handling networks with several tens of thousands nodes on desktop computers.

For larger networks, however, the algorithm is limited by the number of operations (and therefore the network generation time) increasing quadratically with network size. A single million-node network would take more than a day to generate, and a network ten times larger over four months, which is clearly not practical.

We therefore introduce a parallel version, which is based on the generation and linkage of smaller sub-networks. This is tested and evaluated on a large-scale cluster computer. This MPI-based parallelisation of the algorithm guarantees that large networks can be generated efficiently on recent clusters, and is a significant progress for large-scale network generation. Analysis of the influence of the number of such subnetworks on computing performances and on the impact of each algorithm parameter will be detailed during the presentation. While generating each sub-network is trivial, linking them in a manner complying with the specific structure of social networks is not, and deserves particular attention.

Once generated, the social network is used as an input to agent-based simulations. Again, this is a large-scale effort, with millions of agents, and parallelisation is necessary. As mentioned above, the method we use for this is similar to that developed for an immune model involving up to a couple of billions of agents. Here, the key concept is to take advantage of the city structure, and to handle separate neighbourhoods as mostly-independent units, on distinct nodes of the cluster. Communication between these nodes only occurs when an agent is travelling from a neighbourhood to another, and can therefore be kept to low levels.

Each node can handle regions of about 4 km<sup>2</sup>, so that recent clusters<sup>1</sup> are able to simulate weeks of epidemic outbreaks in large urban environments, with an area equivalent to that of the Dublin Region (920 km<sup>2</sup>) or Osaka Prefecture (1890 km<sup>2</sup>).

There currently is, to the best of our knowledge, no model of disease progression within very large human populations which offers the level of detail that we aim for in terms of both social and casual infections, nor which permits the inclusion of realistic individual mobility and social patterns through inclusion of both network-based and agent-based aspects.

Real-life *a priori* testing of future outbreaks is of course impossible, and such a model is therefore expected to complement and contribute significantly to existing evaluation processes for intervention policies.

Finally, an hybrid multi-approach framework with strong translational aspects will necessarily have an impact on the overall research field, both in refining the underpinning techniques and for other possible application areas.

This presentation should, therefore, be of interest to a large audience, from complex networks specialists to biomedical modellers, as well as the parallel computing community as a whole.

## References

- Badham, J.M. (2008), Role of social network properties on the impact of direct contact epidemics, Ph.D., University of New South Wales.
- Chen, Y., Paul, G., Havlin, S., Liljeros, F. & Stanley, H.E. (2008), ‘Finding a better immunization strategy’, *Physical Review Letters* **101**(5), 1–4.
- Claude, B., Perrin, D. & Ruskin, H.J. (2009), Considerations for a social and geographical framework for agent-based Epidemics, in ‘International Conference on Computational Aspects of Social Networks’, IEEE Computer Society, pp. 149–154.
- Keeling, M. (2005), ‘The implications of network structure for epidemic dynamics’, *Theoretical Population Biology* **67**(2005), 1–8.
- Kretzschmar, M. & Wiessing, L.G. (1998), ‘Modelling the spread of HIV in social networks of injecting drug users’, *AIDS* **12**(7), 801–811.
- Perrin, D., Ruskin, H.J. & Martin, C. (2009), *In silico* Biology: making the most of parallel computing, in ‘Handbook of Research on Biocomputation and Biomedical Informatics: Case Studies and Applications’, Medical Information Science Reference.

<sup>1</sup>DCU’s in-house cluster has 448 computing nodes, while ICHEC’s system and others have thousands of nodes.

# Fast On-line Statistical Learning on a GPGPU

FangZhou Xiao<sup>1</sup>

Eric McCreath<sup>1</sup>

Christfried Webers<sup>1,2</sup>

<sup>1</sup> School of Computer Science, College of Engineering & Computer Science  
The Australian National University,  
Canberra, ACT 0200, Australia,  
Email: Shaw.Xiao@anu.edu.au, Eric.McCreath@anu.edu.au

<sup>2</sup> NICTA,  
Canberra, ACT 0200, Australia  
Email: christfried.webers@nicta.com.au

## Abstract

On-line Machine Learning using Stochastic Gradient Descent is an inherently sequential computation. This makes it difficult to improve performance by simply employing parallel architectures. Langford et al. made a modification to the standard stochastic gradient descent approach which opens up the possibility of parallel computation. They also proved that there is no significant loss in accuracy in their approach. They did empirically demonstrate the performance gain in speed for the case of a pipelined architecture with a few processing units. In this paper we report on applying the Langford et al. approach on a General Purpose Graphics Processing Unit (GPGPU) with a large number of processing units. We accelerate the learning speed by approximately 4.5 times compared to a standard single threaded approach with comparable accuracy. We also evaluate the GPU performance for the sequential variant of the algorithm, which has not previously been reported. Finally, we investigate how changes in the number of threads, number of blocks, and amount of delay, effects the overall performance and accuracy.

*Keywords:* GPGPU, Asynchronous Optimisation, Statistical Machine Learning, On-line Learning

## 1 Introduction

Parallel architectures, such as the GPU or multi-core systems, are set to take over traditional serialised architectures given they facilitate a path to continued performance improvement. Given the GPU's outstanding floating point performance, it is a low cost solution for high-performance computing. Furthermore given the widespread deployment of graphics cards capable of CUDA or OpenCL, writing HPC applications on a GPU has become a very attractive option.

This work was supported in part by the Australian Research Council grant DP0987773.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

We would also like to thank Alistair Rendell and Alexander J. Smola for their guidance and encouragement with this project. Copyright 2011, Australian Computer Society, Inc. This paper appeared at the 9th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 118. J. Chen and R. Ranjan, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

GPUs have shown to be outstanding for some scientific modelling applications (Stone et al. 2007, Colange et al. 2007) by achieving surprising acceleration. Steinkraus et al. (2005) claimed that GPUs are pleasant substitutions to dedicated machine learning hardware, such as analog chips and coarse-grained parallel computers. Compared with CPUs or even multi-core CPUs, GPUs still exhibit advantages in many applications (Raina et al. 2009). Previous work on machine learning using GPUs mainly obtained acceleration through intrinsic parallel structures in applications. For example Raina et al. (2009) experimented with large-scale deep unsupervised learning on GPUs and Steinkraus et al. (2005) applied GPUs to a two-layer fully connected neural network. Generally, GPUs (or even other parallelised architectures) support applications which consist of highly symmetric and loosely coupled calculations. These applications are naturally parallelisable. Unfortunately, many applications also involve critical sequential blocks, which can not easily be parallelised. Such difficulty limits the performance of modern parallelised architectures. This paper tentatively applies the GPU's computational capability to one such sequential algorithm: on-line statistical machine learning using gradient descent. Although GPUs might not be the most suitable platform available to address sequential issues, other parallelised architectures more or less face similar issues of efficiency when utilising parallel computation capabilities.

Machine learning is concerned with the design and development of models and algorithms that allow computers to improve their performance over time based on data. Depending on whether or not all training data are used at each iteration step of the algorithm, one can distinguish between batch and on-line learning.

Batch machine learning approaches evaluate candidate hypotheses against the entire set of training instances. This can be very slow when the training set is very large. Furthermore, as all the data must fit into the memory, batch learning utilising large datasets is not well suited for GPUs which have a small local memory.

On-line learning takes one instance of data at a time, and improves its performance solely based on this data item. This process iterates through all the data (possibly a number of times) until either some convergence criteria are achieved or the model predicts unseen test data sufficiently accurately. As only the model parameter and one data item at a time have to be kept in memory, on-line learning better suits the architecture of GPUs. This is a great advantage especially when the training set is very large.

The serial nature of on-line approaches means that

they are difficult to parallelise. This is because the calculation of hypothesis  $x_{i+1}$  requires both the input of instance  $z_i$ , training label  $y_i$  as well as the previous hypothesis  $x_i$ . This entire process is depicted in Figure 1.

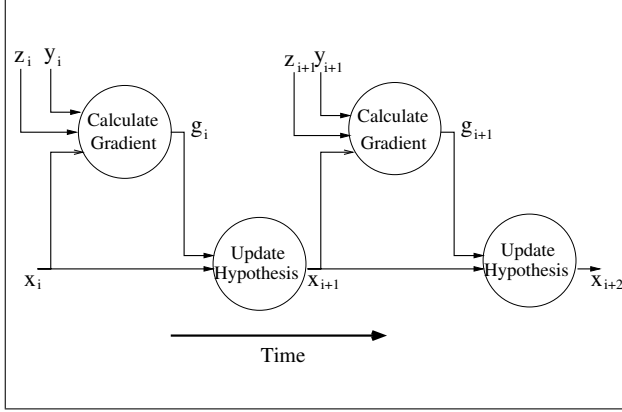


Figure 1: Data flow for the stochastic gradient descent algorithm. In order to calculate the hypothesis  $x_{i+1}$ , one data item  $z_i$ , training label  $y_i$ , as well as the previous hypothesis  $x_i$  are required. A stage can only start its calculations if the previous stage has finished all computations.

Langford’s (Langford et al. 2009) paper “Slow Learners Are Fast” modifies the standard gradient descent algorithm permitting the calculation of the gradient to use a delayed version of the hypothesis. This opens up the possibility of concurrently executing the calculation of the gradient. This is depicted in Figure 2 which shows the dependencies when the gradient calculation uses a delay of 1.

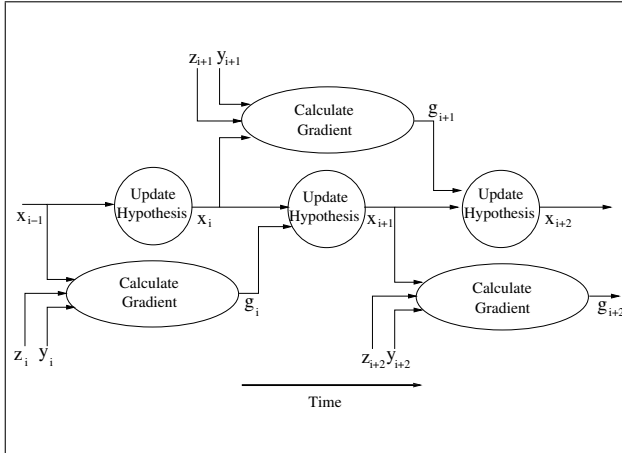


Figure 2: Data flow for delayed stochastic gradient descent algorithm. With a delay of 1, two threads can run concurrently. Thread A (the lower one) starts to calculate gradient  $g_i$  immediately after hypothesis  $x_{i-1}$  is available. Before it updates the hypothesis  $x_i$  with  $g_i$ , thread B (the upper one) uses the hypothesis  $x_i$  to calculate the gradient  $g_{i+1}$ . As can be seen from the data flow, gradient calculation in the threads and updating hypothesis can be run in parallel.

Clearly a new gradient calculation can be started after each hypothesis update. That means if the delay  $\tau$  is larger than one, potentially  $\tau$  threads can be run in parallel. On the other hand, if the delay is too large, the gradient updates become outdated because they are based on too old hypotheses. This limits the

number of the delay  $\tau$  from above.

Langford et al. provide the theoretical foundation for our series of experiments. They prove convergence properties of the convex minimisation problem. Yet they do not experimentally show the performance improvement on an actual parallel architecture. Our experiments implement the delayed mechanism on the TREC dataset (Cormack 2007), and verify the utility of this approach.

The stochastic gradient approach assumes that the training instances are independent and identical distributed (iid) data, this will generally require the training set to be randomly shuffled before being provided to the learner. This opens up the possibility of allowing the reordering of examples when they are used within the learner, which in turn enables us to arbitrarily allocate instances to threads without tightly enforcing an ordering via synchronisation.

The delayed approach trades accuracy with parallelism. Using this parallelism should make the program run faster, however, if you need to run more steps to gain the same accuracy then the speed gained via parallelism needs to outweigh the accuracy losses. Otherwise it is better to just run the standard serial code. We have explored and reported the effect on accuracy in this paper.

There are a number of challenges in implementing the delayed stochastic gradient descent algorithm using a GPU. They include:

1. synchronisation is difficult to implement across all the blocks,
2. synchronisation has the potential to be costly in terms of performance,
3. the GPUs memory size is relatively small, and
4. it is slow to transfer data between the host and the device memory.

## 2 Delayed Stochastic Gradient Descent

This problem is one of binary classification problems. Email  $t$  is denoted  $\mathbf{z}_t \in Z$  and given the label  $y_t \in \{\pm 1\}$ , so if  $y_t = -1$  then the message  $\mathbf{z}_t$  is labelled as spam, whereas, if  $y_t = 1$  then the message  $\mathbf{z}_t$  is labelled as not spam.  $Z$  belongs to an  $n$  dimensional space and has a corresponding  $n$  dimensional feature space  $X \subseteq R^n$ . This feature space contains the hypotheses we intend learning. To determine the classification of a new message  $\mathbf{z}$  we simply take the inner product between the message and that of our current hypothesis  $\langle \mathbf{z}, \mathbf{x} \rangle$  if this is negative then the message is predicted to be spam and if the inner product is positive then the message is predicted to be not spam.

The loss associated with email  $t$  using hypothesis  $\mathbf{x}$  is  $l(y_t \langle \mathbf{z}_t, \mathbf{x} \rangle)$ . The smoothed quadratic soft-margin loss function is used:

$$l(\chi) = \begin{cases} \frac{1}{2} - \chi & \text{if } \chi < 0 \\ \frac{1}{2}(\chi - 1)^2 & \text{if } \chi \in [0, 1] \\ 0 & \text{otherwise} \end{cases}$$

The aim is to find a hypothesis  $\mathbf{x}$  that minimises the sum of the losses over all the training instances. The basic idea of gradient descent, also known as steepest descent, is using a single example move the hypothesis  $\mathbf{x}$  in the direction of the negative gradient. This is repeated over all the instance of the training set over a number of repetitions. An annealing schedule

$\eta$  controls the convergence speed. The amount of delay used is denoted  $\tau \in N$ . Langford et al. (Langford et al. 2009) proved that if the delay is within a tolerable range, delayed stochastic gradient descent would converge to an acceptable value. The convex function used for calculating the gradient is:

$$f_t(\mathbf{x}_t) = l(y_t \langle \mathbf{z}_t, \mathbf{x}_t \rangle)$$

The algorithm for stochastic gradient descent with the delayed mechanism is given in the following steps:

1. Initialise weight vectors  $\mathbf{x}_1, \dots, \mathbf{x}_\tau = \mathbf{0}$
2. Compute the gradient:

$$\begin{aligned} \mathbf{g}_t &= \nabla f_t(\mathbf{x}_t) \\ &= \mathbf{z}_t \frac{\partial l(y_t \langle \mathbf{z}_t, \mathbf{x}_t \rangle)}{\partial \langle \mathbf{z}_t, \mathbf{x}_t \rangle} \\ &= \begin{cases} -y_t \mathbf{z}_t, & \text{if } \langle \mathbf{z}_t, \mathbf{x}_t \rangle \leq 0 \\ y_t (\langle \mathbf{z}_t, \mathbf{x}_t \rangle - 1) \mathbf{z}_t, & \text{if } \langle \mathbf{z}_t, \mathbf{x}_t \rangle \in [0, 1] \\ \mathbf{0}, & \text{otherwise.} \end{cases} \end{aligned}$$

3. Update  $\mathbf{x}_{t+1} = \mathbf{x}_t - \eta_t \mathbf{g}_{t-\tau}$ .
4. Repeat Steps 2 and 3.

Step 3 in the above algorithm is based on the assumption that the weight vector has not been changed greatly after  $\tau$  delays, so we can update the weight vector by using the delayed gradient. As the number of instances processed increases the annealing schedule, which is  $\eta_t = \frac{1}{t-\tau}$ , becomes small. This provides a guarantee that the weight vector will only change slowly.

Regarding to simplicity of implementation and consistency with Langford’s experimental settings, we used dot product in Euclidean spaces to calculate inner product mentioned.

### 3 Implementation Issues

This section generally covers concerns or possible issues if researchers try to replicate our experiments. Regarding to GPU hardware restrictions, we have to make changes to the codes running on the GPU. Where possible we kept similar experimental settings to that of Langford et al. (2009), if any other issues haven’t been mentioned in the paper.

#### 3.1 Machine Learning Issues

Our training set, the TREC dataset (Cormack 2007), consists of 75419 labeled e-mail messages. This dataset has three subsets, “full”, “partial” and “delay”. The “delay” part was used because it was large enough to properly evaluate the implementation, yet, small enough to still be able to run tests in a reasonable amount of time. In order to transform raw messages into manipulable data, we conducted a separate program from training codes to extract relevant information, called pre-processing. Note that this part of program had not been evaluated, only the performance of training codes was concerned. Information was extracted from the email header fields ‘From:’, ‘To:’, ‘Subject:’, and ‘Time:’, and also from the body of the email. Symbols other than alphabetic letters and numbers, such as ASCII code less than 48, were removed from the text. Then rest text was consisted of words that were separated by space. All the word were capitalised and recorded into a word dictionary. Langford’s experiment used both the bag of

words and the bag of words pairs representation. For simplicity reason we only adopted the bag of words representation. The sparse format of the message representation was transformed to a condensed format which listed just the features appearing within the email.

However techniques in text classification like excluding most frequently occurring words, such as “a”, “the” or words that only appear once from the word dictionary, had not been applied. Therefore less pre-knowledge or less artificial intervention was involved into the experiment. Without specific settings for text classification, our results might be meaningful for general machine learning problems. Furthermore our experiment results shown that weights of these features had very limited influence on classification results. Alternatives of these settings may affect accuracy. However, as our focus is on comparing the GPU implementation with that of a standard CPU approach, comparing both speed and accuracy. The relative performance, rather than absolute performance, of the two approaches is more informative.

To fulfil the assumption that input data are iid data, when messages were loaded, we randomised the data by repeatedly swapping randomly selected messages. This provided us with some confidence that the data provided to the learner was not correlated. For evaluation, we used the standard ten-fold cross validation to calculate average learning results.

#### 3.2 Data Structure on the GPU

In order to fit the experimental data into the GPU memory, we re-arranged some data structures. The message matrix that records feature indices was transformed into a long vector, called “datalist”. The start positions of each message was recorded in another vector called “positionlist”, with which we can easily extract the message from the long vector “datalist”. Target values of messages were put into a third vector called “targetlist”. The GPU has fast but small constant memory. We stored “positionlist” and “targetlist” into the constant memory. Because the information of “positionlist” and “targetlist” is frequently required, by storing it in the constant memory will significantly increase efficiency. The vector of feature weights was stored in global memory. Although this memory is very slow, global memory was the only space where all threads can read and update data in our current understanding. Furthermore for the same reason two vectors of “gradient” and “messageid” in global memory were used to store implicit gradient information. The size of “gradient” depended on the delay parameter.

Since the GPU used a different memory system with that of the host CPU system, thus we needed to copy data from the host memory to the device memory and copy results from the device memory back to the host memory after computing. We had to minimise this cost by avoiding frequently transferring data between the host and the device.

#### 3.3 Coding

In order to get benchmark results for evaluating the GPU’s performance, we wrote equivalent programs for both the CPU and the GPU. Both codes are written in C, and the CUDA API was used for calling the kernel executed on the GPU. We used an asynchronous approach to parallelise most parts of the program.

In our program, we can change the parameters of iteration number, delay amount, grid size and block

size (the total number of threads running is the product of the grid size and the block size) to measure the training time and the error rate. Repetitive learning was run over many iterations on the same dataset as indicated. Because every thread does the similar tasks, we could explain one thread as representative.

The thread loaded features of one message and the corresponding weights from the global memory. After calculating the gradient, this thread wrote the gradient and the message ID into “gradient” and “messageid” vectors. The index of the vectors was decided by the thread’s unique id. Then the thread updated the delayed gradient stored in “gradient” and “messageid” vectors (delay  $\tau$  times ago) to the global weight vector and cleared the outdated data.

Assuming every thread handles one parallel computation, then the thread size as Langford suggested is delay  $\tau$  plus 1. For example if the delay is zero, then at least one thread should be running and thus the delay mechanism is disabled. Note that total thread size did not necessarily equal to delay  $\tau$  plus 1. Total thread size is a number within the range of one and instances size. Delay size is between zero and the total thread size.

### 3.4 Scheduling

The delayed update mechanism required a well organised read and write schedule. However it was difficult for the GPU to keep such an organised schedule, because direct communications between threads is not easily available (although global memory could be used, it would be very slow). One thread that starts processing messages earlier cannot promise to finish earlier. The thread does not care about other threads’ status (especially threads in different blocks). In most cases, such a schedule was chaos and uncontrollable, especially when the disabled delay mechanism was used. Variances in the delay built up when the program had been run for some time. The delay mechanism would help to keep the schedule organised, because the more delay we assign, less threads would try to access the same shared data. Also each thread cleared its own “gradient” and “messageid” spaces after updates. For every thread, it required the results from  $t - \tau$  thread but it will not wait on previous threads. By clearing “gradient” and “messageid” spaces after every update, we can at least make sure that if a thread updated results earlier than previous threads finishing computing, it would at least make no changes to the final result.

## 4 Results and Evaluation

This section illustrates our experimental results regarding to the CPU and the GPU respectively. We evaluate the performance of the algorithm running on the CPU as benchmarks and investigate how changes in the number of threads, number of blocks, and amount of delay, effects the overall performance and accuracy. We also discuss the limitations of our implementation on the GPU.

The graphics card used is the NVIDIA Geforce GTX 295 and it consists of two identical GeForce GTX 200 GPUs. Each of these GPUs has 30 Streaming Multiprocessors and each of the Streaming Multiprocessors has 8 cores. We introduce SM as an abbreviation for Streaming Multiprocessors. Each of the GPUs has 895M global memory and 64K bytes constant memory. Execution blocks have 16K of shared memory and 16K of registers. The GPU runs at 1.24 GHz and has a CUDA capability of v1.3. The host

machine uses a AMD Phenom™ II X4 945 processor with 4GB of main memory. For simplicity we describe the NVIDIA GPU card as the ‘GPU’ and the AMD host as the ‘CPU’.

### 4.1 Experiments on the CPU

Figure 3 shows the results of Langford’s experiment based on “full” dataset. Our results based on both “full” and “delay” dataset were shown in Figures 4 and 5. In order to show curves clearly, we used different scales of x-axis from Langford’s figure.

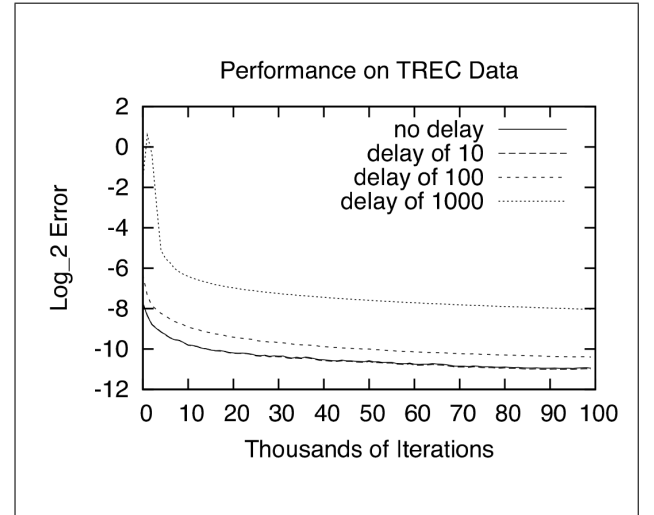


Figure 3: Results of Langford’s experiment on the full dataset. The curves of relatively small delay (compared with instance size and iteration times) are close to the curve of zero delay. The error rate increases when delay size increases.

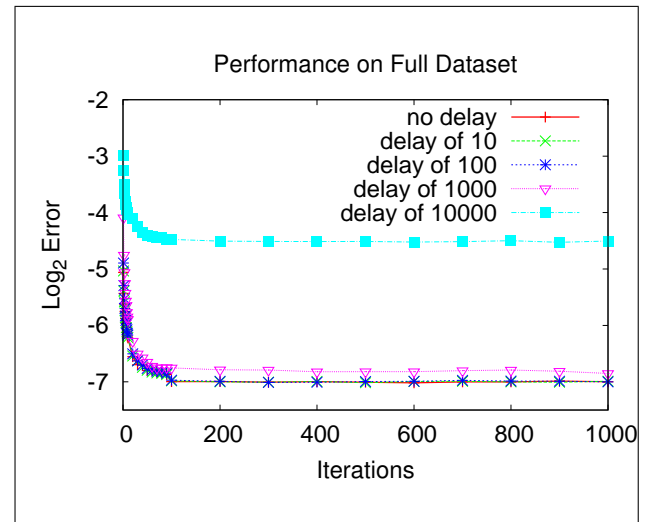


Figure 4: Performance of our experiment on the full dataset. Our results exhibit similar trends as Langford’s results. For a relatively large delay, the accuracy will be significantly affected.

In order to test that the learner was working properly, we created an artificial dataset. This artificial dataset includes some known positive and negative keywords. Through learning, the induced weight vector showed that these keywords had a much larger value than that of other less important words. Fur-

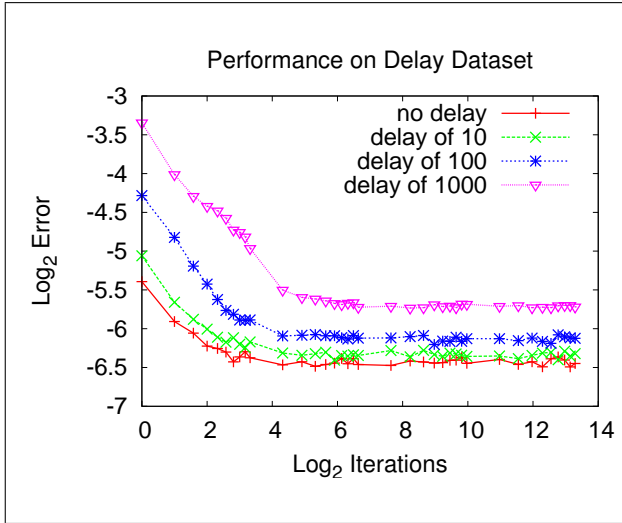


Figure 5: Performance of our experiment on delay dataset. This a similar experiment on a smaller dataset.

thermore, positive and negative keywords had opposite signs. Frequently occurring features ended up with small weights, which barely contributed to classification results.

Then we changed some target values of messages (adding some outliers) or exchanged position of messages pairs, the results showed that accuracy was affected in the first few iterations, but gradually converged to a similar accuracy level. Therefore we are confident that our algorithm, was not only able to distinguish important features from trivial ones, but also ignored outliers to some extent. Also we found that if the input data was large, the effect of delay could be minimised if we ran many repetitions in the learning.

Nevertheless our results did not exactly follow that of Langford’s results, since there were a few different settings used. Langford used the bag of word pairs as well as the bag of words to represent features, whereas, we only used bag of words for simplicity reason. We believe that if we adopted alternative settings, the error rate of classification could be much lower and more steady. Nevertheless all these results show the same trends: if the number of iterations increases, the error rate presents more steady and lower; if the delay amount increases, the less accuracy of converged results would achieve. Under our assumption that this experiment was a relative comparison, if we used our results of the same settings on the CPU as a benchmark, we could still explore the character of the GPU.

We tested both results that were calculated by single precision and double precision under our settings. The results showed that there were no obvious changes over the classification results within one hundred thousand iterations. The weight of each feature could be affected on the  $10^{-5}$  scale, but overall classification results remained the same. In this case, if the data clusters were distinctive, then such accuracy loss was tolerable. However in other applications, the precision of parameters may be more significant.

Figure 6 shows relationship between accuracy and minimum training time required. As we see, to gain higher accuracy will incur much more computations. It is beneficiary if GPUs can accelerate this procedure while they still be able to keep comparable accuracy.

We ran codes both on the CPU and the GPU using

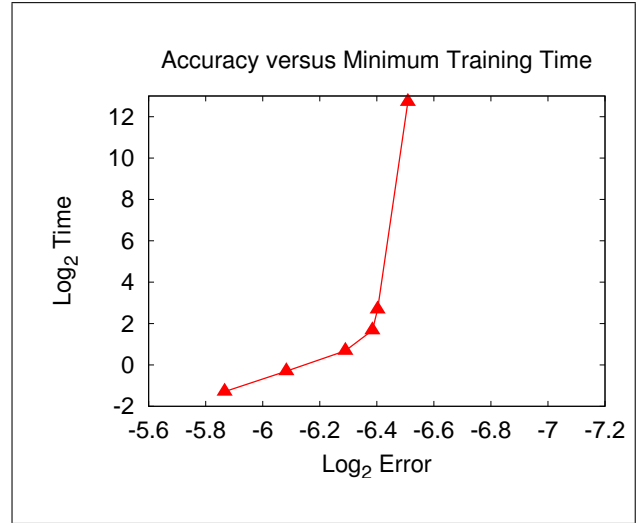


Figure 6: Example of minimum CPU sequential execution training time required to achieve different accuracy levels based on delay dataset. To achieve higher accuracy will require much more computations.

one thread on the “delay” subset without setting delay. The results showed that the CPU’s efficiency was roughly 35 times higher than that of the GPU (without considering a constant memory transfer time). Regardless of other facets, in order to gain acceleration from current settings, we ought to parallelise more than 35 threads (actually much more) on the GPU. If the CPU running time is less than the memory transfer time, then there is no point in using the GPU for acceleration. Only if the CPU’s running is much greater than the memory transfer time, could we possibly gain acceleration. Longer CPU running time is related to more repetitive learning or a larger dataset.

## 4.2 Experiments on the GPU

We outlined the performance of the GPU through changing parameters of delay, iteration, grid size and block size. Based on the knowledge of these parameters, we drew cures of acceleration that we could achieve in our understanding. More elaborate optimisation is still achievable.

### 4.2.1 Delay

According to Figure 4 and Figure 5, with iteration of eight hundred times, all curves of delay have converged to steady states. Therefore most of our experiments are tested under 800 iterations. Here we test results according to changing delay given total running threads as shown in Figure 7 and Figure 8. We set the grid size to 30 (equals to number of Streaming Multiprocessors, thus each SM execute one block) and the block size to 320 (10-fold of warp size). Note that memory transfer time is not considered. Figure 7 shows that accuracy increases when the delay goes up to 32. After that accuracy drops gradually when delay keeps increasing. Because when delay is smaller than 32, one warp of execution will definitely have two or more threads trying to access the same delay space. We cannot assure that former threads will finish computing and write back before later threads started to read. Yet if the delay is over the warp size, during one warp execution, in most cases, one thread



accessed one delay space and all threads will finish computing before next warp execution. However in terms of Langford's delay hypothesis, the proper delay should be much higher than 32 in this case. Using delay of 32 actually updated results earlier than expected. Figure 8 shows how execution time is affected by changing the delay. If the delay is smaller, then less processing time would be consumed. A big delay space will result in more cache misses, whereas small delay seemed to be more efficient. This indicates that with a delay of 32 we will help to achieve the fastest processing speed without affecting accuracy, yet this does not strictly follows Langford delay hypothesis.

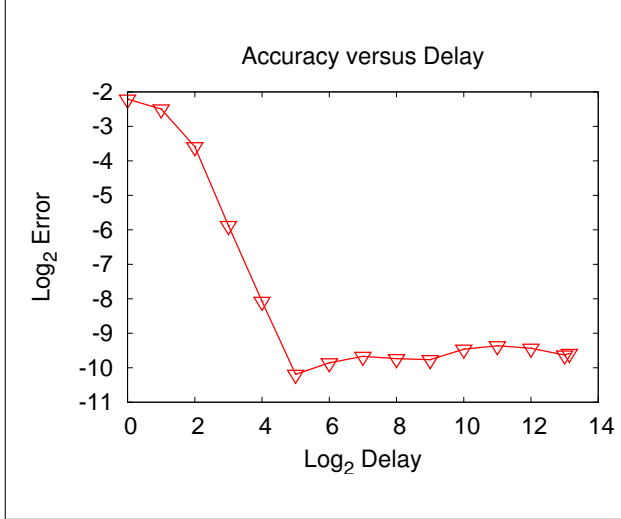


Figure 7: Example of accuracy versus delay curve tested given thread size and iterations. Accuracy increases when the delay goes up to 32. After that accuracy drops gradually when delay keeps increasing.

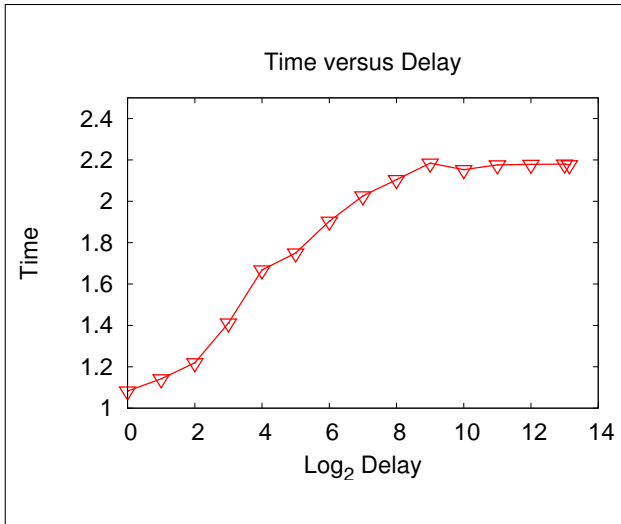


Figure 8: Example of time versus delay curve tested given thread size and iterations. Training time increases if the delay increases.

#### 4.2.2 Iteration

We tested the relationship between changing repetitive learning iterations and performance as shown in Figures 9 and 10. In these experiments: the grid size is 30; the block size is 320; and the delay is 32. Note

that memory transfer time is not considered. In Figure 10 we find that the processing time scales very well based on the number of iterations, which means computation and memory costs have a steady proportion. According to the results in Figure 9, it followed our simulation expectation that the error rate drops when the number of iteration increases. Even if we change the delay, it still shows a similar pattern. Surprisingly our classification results seems to be even better than the simulation code. A possibly reason might be that using the delay mechanism could initiate a better starting point for the gradient descent problem or it could avoid a local minimum trap. We would like to further experiment with this to understand what is happening.

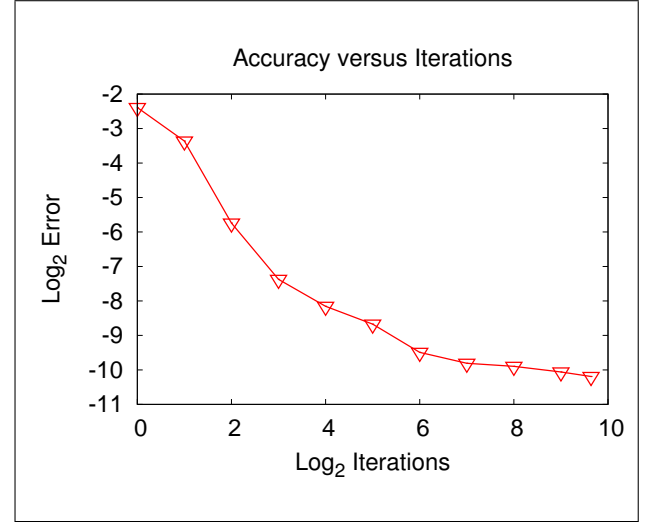


Figure 9: Example of accuracy versus iteration curve tested given threads and delay. The figure shows that error rate drops when iterations increase.

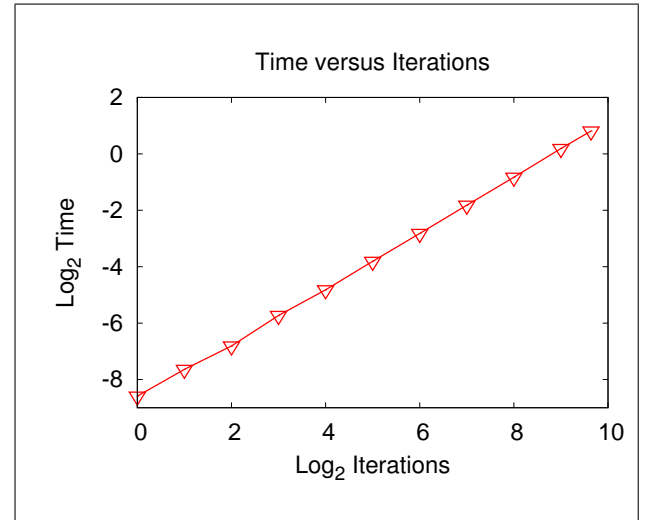


Figure 10: Example of time versus iteration curve tested given threads and delay. Training time is proportional to number of iterations.

#### 4.2.3 Grid size versus Block size

We tested codes with zero delay to find out how changes in grid size and block size will affect processing speed. In ideal case, if we double the number



of threads, the training time will be half the original. There are three ways to increase the total number of threads running on a GPU: increase block number per grid, increase thread number per block, and combined. The results through changing grid size and block size under 64 iterations is shown as Figure 11. If we only scale the grid size, we could see a repetitive pattern of 210 thread size (7-fold of SM number). SM can fit several blocks if the resources are available (Hong and Kim 2009). So in this case seven blocks could be executed at one time, therefore we have this repetitive cure. Therefore it was ideal to have the block size equal to SM's fold to achieve better acceleration. Note that we cannot fit as many as seven blocks into one SM as thread number increases. If we only increase the block size (biggest number of threads was 512 for our GPU), processing time drops steadily, although, it is not as steep as the ideal curve. Therefore we conclude that to efficiently use the GPU resources, it is advisable to have more threads running in the blocks. While at the same time we keep all SMs working. So the grid size of 30 seems to be the optimum setting.

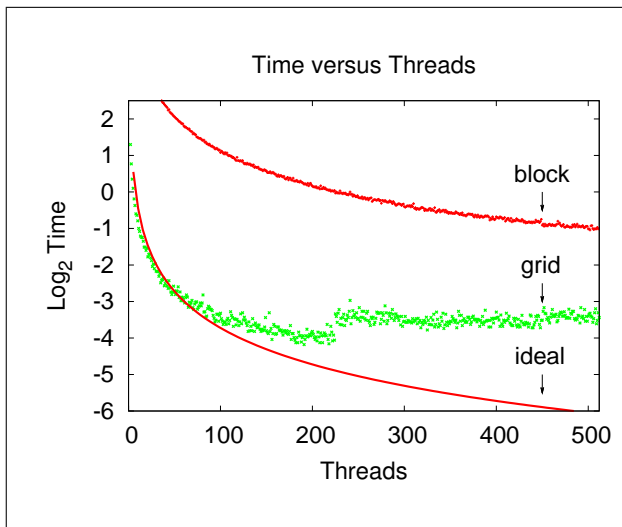


Figure 11: Example of time versus threads curve. The grid curve follows the ideal curve in the beginning. The block curve drops steadily.

#### 4.2.4 Acceleration

Figure 12 illustrates the acceleration achieved by comparing processing time between CPU and GPU through both running 800 iterations. There are three curves shown in the figure: “extreme”, “standard” and “tuned”. In order to increase total thread number, we first increased the grid size until it reached 30 (the number of SMs) and then increased the block size. “Extreme” speed-up was the ratio of CPU processing time to memory transfer time by assuming GPU processing time was zero. “Standard” curve was plotted that thread number equalled to delay plus one and “tuned” curve was tested under a fixed delay of thirty-two. Note that all curves take memory transfer time into account.

#### 4.3 Discussion

One big problem we found in the parallelism of asynchronous optimisation is that a direct and fast communication between threads is not available. Similar findings revealed by Xiao and Feng (2009) claimed

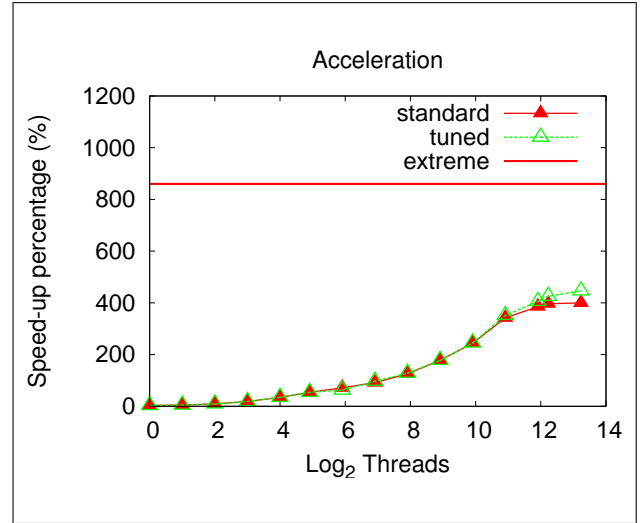


Figure 12: Acceleration curve that we could possibly achieve.

that inter-block GPU communication is a main contributor to total processing time. Furthermore, our control over thread scheduling is limited. There are a few possible approaches we could try to address these synchronisation issues:

1. Stop parallelism after some time, synchronise results and restart the parallelism. This approach would help to ensure correctness stage by stage. However this would introduce more overhead and also the efficiency of parallelism would be undermined.
2. Apply parallelism of pipelined optimisation to decompose loss function as Langford suggested (Langford et al. 2009). Assuming  $f_i(\mathbf{x}) = g(\phi(\mathbf{z}_i), \mathbf{x})$  The issue is to find out appropriate  $\phi(\mathbf{z}_i)$  and feed data  $\mathbf{z}_i$  to partial functions. The advantage of this approach is that it dramatically reduces synchronisation costs and updates partial values locally. Nevertheless when combining partial values we still need to make sure all threads are working on the same data.

Theoretical bandwidth of the NVIDIA GeForce GTX 295 is

$$1.24 \times 10^9 \times (512/8) \times 2 = 158.7 \text{ GB/sec}$$

We used *cudaprof* as profiler to test performance by setting the grid size to 30, the block size to 320 with delay 32. Overall global memory throughput was 29.9972 GB/sec. Occupancy for major function was 0.625. Only about 20% the GPU's theoretical capability had been achieved. This results were foreseeable if we implemented a sequential logical program onto parallel architectures. Bridges et al. (2007) experimented sequential codes of C benchmarks in SPEC CINT2000 on multi-core achieving speedup of 454% using 32 threads. Also some applications can hardly gain speedup. The bottleneck of the program was excessive access to global memory during synchronisation. The latency drag down the utilisation of computations. Xiao and Feng (2009) claimed that inter-block synchronisation is the main contributor to total processing time when computation is highly parallelised.

## 5 Conclusions and Future Work

Nowadays increasing parallel cores have become the stimulus for fast continuing growth in transistor count. However sequential applications have not taken the advantages of increasing computability. Therefore it is worthwhile to explore how to make use of tomorrow's processors. In this paper, we implemented Delayed Stochastic Gradient Descent Algorithm on a GPU platform. With the delay mechanism, we could parallelise an essentially sequential problem that has hardly been handled by general parallel architectures. Based on our experiment, we show that this alternative delay algorithm could achieve comparable accuracy to that of a sequential algorithm through parallel computations.

We also estimated GPU's performance on a strong dependency case, which had not previously been revealed. Our results showed that the GPU's high computability was not fully exerted compared to other GPU implementations. Because the application excessively attempted to access global memory, when parallelism became high, computation time contributed much less than memory access time. However global memory is the only all threads accessible memory, thus it was difficult to avoid such costs. Note that acceleration could still be increased if the dataset size or iterations increased, but synchronisation would still be the bottleneck. GPUs have very light weight threads which are not specially designed for complex operations. Because of its low scheduling design, GPUs gain benefits of fast growing computations but lose complex logic control over parallelism.

In order to solve the sequential problem proposed, there are two possible solutions: to find or to design an architecture that supports fast and complex logic over parallelism, or to revise current algorithms that minimise synchronisation cost however achieving applicable accuracy.

Experimenting on GPUs' simple cores will reveal its incapability of current design when solving some general problems. However further study of combining algorithms with architecture would possibly indicate what are the most important features to be necessarily included in future designs. For example, if synchronisation plays an important role in future parallelism program model, it assures that Fermi architecture (new generation of NVIDIA GPUs) with fast global cache is a smart choice. Furthermore it could be a good idea to have a central coordinate processor in parallel architecture such as PS3, which can sufficiently communicate with other parallel cores. This coordinator can gather information from other threads, but at the same time manage the scheduling and usage of memory. Besides we are also interested to experiment on other existing architectures. As Vuduc et al. (2010) suggested, a hybrid CPU/GPU architecture may perform better overall. This would also eliminate memory transfer time between the host system and the GPU's memory, which is currently one of limitations with current graphics card configurations.

To minimise the synchronisation cost, we can experiment with pipelined approaches of parallelism that require less synchronisation. Furthermore we could improve machine learning algorithms from frequently synchronisation required to only occasional synchronisation required. For example our application requires synchronisation after processing each instance. If an algorithm only requires synchronisation after processing one hundred instances, this would make a big difference.

In general our conclusions are limited in two main

ways. Firstly, our application was specifically on one machine learning approach, thus, limiting the generalisation of our claims. Secondly we have only tested the approach on the NVIDIA GTX295 using CUDA programming language. It would be interesting to explore the approach on other GPU models and other programming APIs.

In the next stage in this research we will experiment with other machine learning algorithms on various GPU models or other architectures. We would like to explore optimisation techniques, e.g. pipelined optimisation, orthogonal feature spaces. We could experiment on other stochastic algorithm to evaluate hypothesis of delayed update. Another direction is to explore other algorithms which are more suited to existing parallel architectures.

## References

- Bridges, M., Vachharajani, N., Zhang, Y., Jablin, T. & August, D. (2007), Revisiting the Sequential Programming Model for Multi-Core, in 'International Symposium on Microarchitecture - MICRO 2007'.
- Collange, S., Daumas, M. & Defour, D. (2007), Graphic processors to speed-up simulations for the design of high performance solar receptors, in 'IEEE 18th International Conference on Application-specific Systems'.
- Cormack, G. (2007), TREC 2007 spam track overview, in 'proceeding of the Sixteenth Text REtrieval Conference (TREC 2007)'.
- Hong, S. and Kim, H. (2009), An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, in 'Proceedings of the 36th annual international symposium on Computer architecture', Austin, TX, USA.
- Langford, J., Smola, A. & Zinkevich, M. (2009), Slow Learners are Fast, *Journal of Machine Learning Research*, Vol. 1, 1-23.
- Raina, R., Madhavan, A. & Ng, A. (2009), Large-scale Deep Unsupervised Learning using Graphics Processors, in 'Proceedings of the 26th Annual International Conference on Machine Learning'.
- Steinkraus, D., Buck, J. & Simard, P. (2005), Using GPUs for Machine Learning Algorithms, in 'Proceedings of the 2005 Eight International Conference on Document Analysis and Recognition (ICDAR'05)'.
- Stone, J., Phillips, J., Freddolino, P., Hardy, D., Trabuco, L. & Schulten, K. (2007), Accelerating molecular modeling applications with graphics processors, *Journal of Computational Chemistry*, Vol. 28, 2618-2640.
- Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M. & Shringarpure, A. (2010), On the Limits of GPU Acceleration, in '2nd USENIX Workshop on Hot Topics in Parallelism'.
- Xiao, S., and Feng, W. (2009), Inter-block GPU communication via fast barrier synchronization, *Technical Report TR-09-19*, Dept. of Computer Science, Virginia Tech.

## Author Index

Charrier, Ghislain, 15  
Chen, Jinjun, iii

Desprez, Frédéric, 15

Eblen, John D., 25

Hawick, Ken, 3  
Hudson, Randy, 13

Johnson, Martin, 3  
Jordan, George Calhoun, 13

Langston, Michael A., 25  
Leist, Arno, 3

McCreath, Eric, 35

Norris, John, 13

Ohsaki, Hiroyuki, 33

Papka, E. Michael, 13  
Perrin, Dimitri, 33  
Playne, Daniel, 3

Ranjan, Rajiv, iii  
Reid, Lynn, 13  
Rogers, Gary, 25

Webers, Christfried, 35  
Weerapurage, Dinesh P, 25  
Weide, Klaus, 13

Xiao, Fangzhou, 35

Yves, Caniou, 15

## 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>.

- |  |  |
|--|--|
| <p><b>Volume 91 - Computer Science 2009</b><br/>           Edited by Bernard Mans Macquarie University.<br/>           January, 2009. 978-1-920682-72-9.</p>   | <p>Contains the proceedings of the Thirty-Second Australasian Computer Science Conference (ACSC2009), Wellington, New Zealand, January 2009.</p>                               |
| <p><b>Volume 92 - Database Technologies 2009</b><br/>           Edited by Xuemin Lin, University of New South Wales and Athman Bouguettaya, CSIRO. January, 2009. 978-1-920682-73-6.</p>   | <p>Contains the proceedings of the Twentieth Australasian Database Conference (ADC2009), Wellington, New Zealand, January 2009.</p>  |
| <p><b>Volume 93 - User Interfaces 2009</b><br/>           Edited by Paul Calder Flinders University and Gerald Weber University of Auckland. January, 2009. 978-1-920682-74-3.</p>   | <p>Contains the proceedings of the Tenth Australasian User Interface Conference (AUI2009), Wellington, New Zealand, January 2009.</p>  |
| <p><b>Volume 94 - Theory of Computing 2009</b><br/>           Edited by Prabhhu Manyem, University of Ballarat and Rod Downey, Victoria University of Wellington. January, 2009. 978-1-920682-75-0.</p>  | <p>Contains the proceedings of the Fifteenth Computing: The Australasian Theory Symposium (CATS2009), Wellington, New Zealand, January 2009.</p>                               |
| <p><b>Volume 95 - Computing Education 2009</b><br/>           Edited by Margaret Hamilton, RMIT University and Tony Clear, Auckland University of Technology. January, 2009. 978-1-920682-76-7.</p>  | <p>Contains the proceedings of the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009.</p>                                  |
| <p><b>Volume 96 - Conceptual Modelling 2009</b><br/>           Edited by Markus Kirchberg, Institute for Infocomm Research, A*STAR, Singapore and Sebastian Link, Victoria University of Wellington, New Zealand. January, 2009. 978-1-920682-77-4.</p>                | <p>Contains the proceedings of the Fifth Asia-Pacific Conference on Conceptual Modelling (APCCM2008), Wollongong, NSW, Australia, January 2008.</p>                            |
| <p><b>Volume 97 - Health Data and Knowledge Management 2009</b><br/>           Edited by James R. Warren, University of Auckland. January, 2009. 978-1-920682-78-1.</p>  | <p>Contains the proceedings of the Third Australasian Workshop on Health Data and Knowledge Management (HDKM 2009), Wellington, New Zealand, January 2009.</p>                 |
| <p><b>Volume 98 - Information Security 2009</b><br/>           Edited by Ljiljana Brankovic, University of Newcastle and Willy Susilo, University of Wollongong. January, 2009. 978-1-920682-79-8.</p>   | <p>Contains the proceedings of the Australasian Information Security Conference (AISC 2009), Wellington, New Zealand, January 2009.</p>  |
| <p><b>Volume 99 - Grid Computing and e-Research 2009</b><br/>           Edited by Paul Roe and Wayne Kelly, QUT. January, 2009. 978-1-920682-80-4.</p>   | <p>Contains the proceedings of the Australasian Workshop on Grid Computing and e-Research (AusGrid 2009), Wellington, New Zealand, January 2009.</p>                           |
| <p><b>Volume 100 - Safety Critical Systems and Software 2007</b><br/>           Edited by Tony Cant, Defence Science and Technology Organisation, Australia. December, 2008. 978-1-920682-81-1.</p>  | <p>Contains the proceedings of the 13th Australian Conference on Safety Critical Systems and Software, Canberra, Australia, December, 2008.</p>                                |
| <p><b>Volume 101 - Data Mining and Analytics 2009</b><br/>           Edited by Paul J. Kennedy, University of Technology, Sydney, Kok-Leong Ong, Deakin University and Peter Christen, The Australian National University. November, 2009. 978-1-920682-82-8.</p>      | <p>Contains the proceedings of the 8th Australasian Data Mining Conference (AusDM 2009), Melbourne, Victoria, Australia, November, 2009.</p>                                   |
| <p><b>Volume 102 - Computer Science 2010</b><br/>           Edited by Bernard Mans, Macquarie University, Australia and Mark Reynolds, University of Western Australia, Australia. January, 2010. 978-1-920682-83-5.</p>   | <p>Contains the proceedings of the Thirty-Third Australasian Computer Science Conference (ACSC 2010), Brisbane, Queensland, Australia, January 2010.</p>                       |
| <p><b>Volume 103 - Computing Education 2010</b><br/>           Edited by Tony Clear, Auckland University of Technology, New Zealand and John Hamer, University of Auckland, New Zealand. January, 2010. 978-1-920682-84-2.</p>   | <p>Contains the proceedings of the Twelfth Australasian Computing Education Conference (ACE 2010), Brisbane, Queensland, Australia, January 2010.</p>                          |
| <p><b>Volume 104 - Database Technologies 2010</b><br/>           Edited by Heng Tao Shen, University of Queensland, Australia and Athman Bouguettaya, CSIRO ICT Centre, Australia. January, 2010. 978-1-920682-85-9.</p>   | <p>Contains the proceedings of the Twenty-First Australasian Database Conference (ADC 2010), Brisbane, Queensland, Australia, January 2010.</p>                                |
| <p><b>Volume 105 - Information Security 2010</b><br/>           Edited by Colin Boyd, Queensland University of Technology, Australia and Willy Susilo, University of Wollongong, Australia. January, 2010. 978-1-920682-86-6.</p>                                      | <p>Contains the proceedings of the Eight Australasian Information Security Conference (AISC 2010), Brisbane, Queensland, Australia, January 2010.</p>                          |
| <p><b>Volume 106 - User Interfaces 2010</b><br/>           Edited by Christof Lutteroth, University of Auckland, New Zealand and Paul Calder Flinders University, Australia. January, 2010. 978-1-920682-87-3.</p>   | <p>Contains the proceedings of the Eleventh Australasian User Interface Conference (AUI2010), Brisbane, Queensland, Australia, January 2010.</p>                               |
| <p><b>Volume 107 - Parallel and Distributed Computing 2010 2010</b><br/>           Edited by Jinjun Chen, Swinburne University of Technology, Australia and Rajiv Ranjan, University of New South Wales, Australia. January, 2010. 978-1-920682-88-0.</p>              | <p>Contains the proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), Brisbane, Queensland, Australia, January 2010.</p>       |
| <p><b>Volume 108 - Health Informatics and Knowledge Management 2010</b><br/>           Edited by Anthony Maeder, University of Western Sydney, Australia and David Hansen, CSIRO Australian e-Health Research Centre, Australia. January, 2010. 978-1-920682-89-7.</p> | <p>Contains the proceedings of the Fourth Australasian Workshop on Health Informatics and Knowledge Management (HIKM 2010), Brisbane, Queensland, Australia, January 2010.</p> |

**Volume 109 - Theory of Computing 2010**

Edited by Taso Viglas, University of Sydney, Australia and Alex Potanin, Victoria University of Wellington, New Zealand. January, 2010. 978-1-920682-90-3.

Contains the proceedings of the Sixteenth Computing: The Australasian Theory Symposium (CATS 2010), Brisbane, Queensland, Australia, January 2010.

**Volume 110 - Conceptual Modelling 2010**

Edited by Sebastian Link, Victoria University of Wellington, New Zealand and Aditya Ghose, University of Wollongong, Australia. January, 2010. 978-1-920682-92-7.

Contains the proceedings of the Seventh Asia-Pacific Conference on Conceptual Modelling (APCCM2010), Brisbane, Queensland, Australia, January 2010.

**Volume 112 - Advances in Ontologies 2009**

Edited by Edited by Thomas Meyer, Meraka Institute, South Africa and Kerry Taylor, CSIRO ICT Centre, Australia. December, 2009. 978-1-920682-91-0.

Contains the proceedings of the Australasian Ontology Workshop 2009 (AOW 2009), Melbourne, Australia, December, 2009.

