

CONFERENCES IN RESEARCH AND PRACTICE IN
INFORMATION TECHNOLOGY

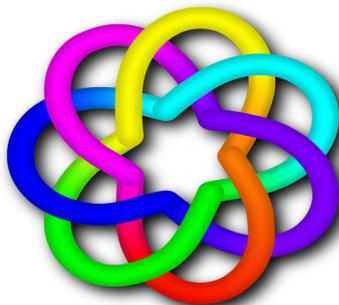
VOLUME 107

PARALLEL AND DISTRIBUTED
COMPUTING 2010

AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS, VOLUME 32, NUMBER 6



AUSTRALIAN
COMPUTER
SOCIETY



 **CORE**
Computing Research & Education

PARALLEL AND DISTRIBUTED COMPUTING 2010

Proceedings of the Eighth Australasian Symposium on
Parallel and Distributed Computing (AusPDC 2010),
Brisbane, Australia,
January 2010

Jinjun Chen and Rajiv Ranjan, Eds.

Volume 107 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 2010. Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), Brisbane, Australia, January 2010

Conferences in Research and Practice in Information Technology, Volume 107.

Copyright ©2010, 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
1, Alfred Street, Hawthorn,
Melbourne, Victoria 3122
Australia
Email: jchen@swin.edu.au

Rajiv Ranjan

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052
Australia
Email: rranjans@gmail.com

Series Editors:

Vladimir Estivill-Castro, Griffith University, Queensland
Simeon J. Simoff, University of Western Sydney, NSW

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 107.
ISSN 1445-1336.
ISBN 978-1-920682-88-0.

Printed, December 2009 by UWS Press, Locked Bag 1797, South Penrith DC, NSW 1797, Australia
Document engineering by Susan Henley, University of Western Sydney
Cover Design by Matthew Brecknell, Queensland University of Technology
CD Production by FATS Digital, 318 Montague Road, West End QLD 4101, <http://www.fats.com.au/>

The *Conferences in Research and Practice in Information Technology* series aims to disseminate 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 Eighth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), Brisbane, Australia, January 2010

| | |
|---|------|
| Preface | vii |
| Programme Committee | viii |
| Organising Committee | ix |
| Welcome from the Organising Committee | x |
| CORE - Computing Research & Education | xi |
| ACSW Conferences and the Australian Computer Science Communications | xii |
| ACSW and AusPDC 2010 Sponsors | xiv |

Contributed Papers

| | |
|---|----|
| A Technology to Expose a Cluster as a Service in a Cloud | 3 |
| <i>Michael Brock and Andrzej Goscinski</i> | |
| A New Integrated Unicast/Multicast Scheduler for Input-Queued Switches | 13 |
| <i>Kwan-Wu Chin</i> | |
| A Dynamic, Decentralised Search Algorithm for Efficient Data Retrieval in a Distributed Tuple Space | 21 |
| <i>Alistair Atkinson</i> | |
| A Distributed Heuristic Solution using Arbitration for the MMMKP | 31 |
| <i>Md. Mostofa Akbar, Eric. G. Manning, Gholamali C. Shoja, Steven Shelford and Tareque Hossain</i> | |
| Object Oriented Parallelisation of Graph Algorithms using Parallel Iterator | 41 |
| <i>Lama Akeila, Oliver Sinnen and Wafaa Humadi</i> | |
| Experience on the parallelization of the OASIS3 coupler | 51 |
| <i>Italo Epicoco, Silvia Mocavero and Giovanni Aloisio</i> | |
| Classification of Malware Using Structured Control Flow | 61 |
| <i>Silvio Cesare and Yang Xiang</i> | |
| Lazy Evaluation of PDE Coefficients in the EScript System | 71 |
| <i>Joel Fenwick and Lutz Gross</i> | |
| Author Index | 77 |

Preface

These proceedings contain the papers presented at the 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), held on 18 January in Brisbane 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, 16 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 8 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 of all paper submissions, without whom this conference would not be 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.

Rajiv Ranjan

University of New South Wales

Jinjun Chen

Swinburne University of Technology

AusPDC 2010 Programme Chairs

January 2010

Programme Committee

Chairs

Jinjun Chen, Swinburne University of Technology, Australia

Rajiv Ranjan, University of New South Wales, Australia

Members

Jemal Abawajy, Deakin of University, Australia

David Abramson, Monash University, Australia

Mark Baker, University of Reading, UK

David Bannon, Victoria Partnership for Advanced Computing, Australia

Boualem Bentallah, University of New South Wales, Australia

Rajkumar Buyya, University of Melbourne, Australia

Paul Coddington, University of Adelaide, Australia

Neil Gemell, University of Otago, New Zealand

Andrzej Goscinski, Deakin University, Australia

Kenneth Hawick, Massey University, New Zealand

John Hine, Victoria University of Wellington, New Zealand

Jane Hunter, University of Queensland, Australia

Martin Johnson, Massey University, New Zealand

Nick Jones, University of Auckland, New Zealand

Laurent Lefevre, University of Lyon, France

Andrew Lewis, Griffith University, Australia

Anna Liu, University of New South Wales, Australia

Piyush Maheshwari, Perot Systems, USA

Teo Yong Meng, National University of Singapore, Singapore

Manish Parashar, Rutgers University, USA

Srikumar Venugopal, University of Melbourne, Australia

Yun Yang, Swinburne University of Technology, 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 NSW, Australia

Dr. Wyne Kelly, Queensland University of Technology, Australia

Prof. Paul Roe, Queensland University of Technology, Australia

Dr. Andrew Wendelborn, University of Adelaide, Australia

Organising Committee

Co-Chairs

Dr. Wayne Kelly
Prof. Mark Looi

Budget and Facilities

Mr. Malcolm Corney

Catering and Booklet

Dr. Diane Corney

Sponsorship and Web

Dr. Tony Sahama

Senior Advisors

Prof. Colin Fidge
Prof. Kerry Raymond

Finance and Travel

Ms. Therese Currell
Ms. Carol Richter

Registration

Mr. Matt Williams

DVD and Signage

Mr. Matthew Brecknell

Satchels and T-shirts

Ms. Donna Teague

Welcome from the Organising Committee

On behalf of the Australasian Computer Science Week 2010 (ACSW2010) Organising Committee, we welcome you to this year's event hosted by the Queensland University of Technology (QUT). Striving to be a "University for the Real World" our research and teaching has an applied emphasis. QUT is one of the largest producers of IT graduates in Australia with strong linkages with industry. Our courses and research span an extremely wide range of information technology, everything from traditional computer science, software engineering and information systems, to games and interactive entertainment.

We welcome delegates from over 21 countries, including Australia, New Zealand, USA, Finland, Italy, Japan, China, Brazil, Canada, Germany, Pakistan, Sweden, Austria, Bangladesh, Ireland, Norway, South Africa, Taiwan and Thailand. We trust you will enjoy both the experience of the ACSW 2010 event and also get to explore some of our beautiful city of Brisbane. At Brisbane's heart, beautifully restored sandstone buildings provide a delightful backdrop to the city's glass towers. The inner city clusters around the loops of the Brisbane River, connected to leafy, open-skied suburban communities by riverside bikeways. QUT's Garden's Point campus, the venue for ACSW 2010, is on the fringe of the city's botanical gardens and connected by the Goodwill Bridge to the Southbank tourist precinct.

ACSW2009 consists of the following conferences:

- Australasian Computer Science Conference (ACSC) (Chaired by Bernard Mans and Mark Reynolds)
- Australasian Computing Education Conference (ACE) (Chaired by Tony Clear and John Hamer)
- Australasian Database Conference (ADC) (ADC) (Chaired by Heng Tao Shen and Athman Bouguet-taya)
- Australasian Information Security Conference (AISC) (Chaired by Colin Boyd and Willy Susilo)
- Australasian User Interface Conference (AUIC) (Chaired by Christof Lutteroth and Paul Calder)
- 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 Anthony Maeder and David Hansen)
- Computing: The Australasian Theory Symposium (CATS) (Chaired by Taso Viglas and Alex Potanin)
- Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Sebastian Link and Aditya Ghose)
- Australasian Computing Doctoral Consortium (ACDC) (Chaired by David Pearce and Rachel Cardell-Oliver).

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 ACSW2010 including the Organising Committee, the Conference Chairs and Programme Committees, our sponsors, the keynote speakers and the delegates. Special thanks to Justin Zobel from CORE and Alex Potanin (co-chair of ACSW2009) for his extensive advice and assistance. If ACSW2010 is run even half as well as ACSW2009 in Wellington then we will have done well.

Dr Wayne Kelly and Professor Mark Looi

Queensland University of Technology

ACSW2010 Co-Chairs

January, 2010

CORE - Computing Research & Education

CORE welcomes all delegates to ACSW2010 in Brisbane. 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 the week with seven other events, which build on the diversity of the Australasian computing community.

In 2010, we have again chosen to feature a small number of plenary speakers from across the discipline: Andy Cockburn, Alon Halevy, and Stephen Kisely. I thank them for their contributions to ACSW2010. 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 Wayne Kelly 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 2009, CORE continued to work 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. ACSW2010 includes a forum on rankings to discuss this process. Also in 2009 CORE proposed a standard for the undergraduate Computer Science curriculum, with the intention that it be used for accreditation of degrees in computer science.

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 2009; in particular, I thank Gill Dobbie, Jenny Edwards, Alan Fekete, Tom Gedeon, Leon Sterling, and the members of the executive and of the curriculum and ranking committees.

Justin Zobel

President, CORE
January, 2010

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.

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 AUC.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Conference Acronyms

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

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

ACSW and AusPDC 2010 Sponsors

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



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



CEED,
www.corptech.com.au



Queensland University of Technology,
www.qut.edu.au



CSIRO ICT Centre,
www.csiro.au/org/ict.html



SAP Research,
www.sap.com/about/company/research



AUSTRALIAN
COMPUTER
SOCIETY

Australian Computer Society,
www.acs.org.au



Manjrasoft Pty Ltd,
www.manjrasoft.com

CONTRIBUTED PAPERS

A Technology to Expose a Cluster as a Service in a Cloud

Michael Brock and Andrzej Goscinski

School of Information Technology, Deakin University
Pigdons Road, Waurn Ponds, Victoria 3217

{mrob, ang}@deakin.edu.au

Abstract

Clouds refer to computational resources (in particular, clusters) that are accessible as scalable, on demand, pay-as-you-go services provided in the Internet. However, clouds are in their infancy and lack a high level abstraction. Specifically, there is no effective discovery and selection service for clusters and offer little to no ease of use for clients. Here we show a technology that exposes clusters as Web services in the form of a Cluster as a Service for publishing via WSDL, discovering, selecting and using clusters.

Keywords: Cluster as a Service, WSDL Publishing and Selection, Dynamic Brokering, Clouds.

1 Introduction

Cloud computing is made possible through the combination of virtualization, Service Oriented Architecture (SOA), and Web and RESTful services. Virtualization allows any computer platform to be supported regardless of hardware and software. By abstracting cluster and server software, it improves the efficiency, availability, access and use of resources and applications. Virtualization enables the use of idle cycles of resources of datacenters, which are in 80% unused. SOA forms an architectural basis for the cooperation of clients, services, and registries (Papazoglou and van den Heuvel 2007). Web and RESTful services provide a high level abstraction and highly interoperable communication subsystem. Scalable data centers offer dynamic and huge hardware provisioning. The end result is an inexpensive, Internet accessible on demand environment where clients use computing resources on a pay-as-you-go basis as a utility and are freed from hardware and software provisioning issues.

For clients to access resources they must be discovered. Clouds and their computational resources are not easy to discover and it is difficult to select and use services. An analysis of the three main clouds (EC2 (Amazon 2007), Azure (Microsoft 2009), and AppEngine (Google 2009), has found that they and their services/resources are difficult to discover and offer little to no ease of use unless the user is a software developer.

Clusters are a basic component of clouds. However, it is difficult to discover clusters and select a cluster that satisfies client requirements. This implies that clients have to access every cluster from a list provided by a registry to learn about their state and characteristics. It is

also not easy to use cloud clusters as they are not exposed at a high level of abstraction (Jha et al. 2009).

This paper presents an outcome of our project that addresses some of these problems. It shows a technology that exposes a cluster as a service that offers a high level abstraction of clusters in the form of Cluster as a Service (CaaS). The proposed technology is based on the Resources Via Services (RVWS) framework (Brock and Goscinski 2008a; Brock and Goscinski 2008b). The technology proposed in this paper allows efficient exposing and publishing via WSDL documents of Web services exposing clusters, their discovery and selection of a requested cluster and makes its use easier. As the WSDL document is the most commonly requested and recorded object of a Web service, the inclusion of cluster's state and other information in the WSDL document makes the internal activity of the Web services exposing this cluster publishable.

It is important to mention that this paper does not address cloud SLAs (Service Level Agreements), business and provisioning models, security and reliability (network and computer system outages), although they are seen as critical aspects of clouds.

The rest of this paper is structured as follows. Section 2 discusses three well known clouds and concludes that they and their services/resources are difficult to discover and do not support service selection and ease of use well. Section 3 introduced a high level abstraction and architecture of the CaaS Technology. Section 4 discusses, following a brief characterization of the RVWS framework, the logical design of CaaS. All components responsible for the publication of clusters, their discovery and selection, and actual use are discussed extensively. Section 5 presents a proof of concept in the form of implementation and experiments carried out to demonstrate the way a cluster is published, found and used. Section 6 provides a conclusion.

2 Related Work

While the use of Web services has made cloud services interoperable, Web services are natively stateless, and can complicate the exposure of resources that depend heavily on state. The WSRF framework (Czajkowski et al. 2004) makes Web services stateful but the state itself is not publishable. The lack of published state forms a major obstacle: if the state of the Web service is not published, clients cannot learn if the Web service is ready for requests or not. Furthermore, while the standards behind the publication of Web services are extensive, their practice is limited greatly to static parameters such as the publication of Web service functionality, communication patterns, and provider contact details. Finally, the use the clouds are not easy and require clients to have good knowledge of them.

Virtualization lays the foundation for sharable on demand infrastructure, on which three basic cloud abstractions are offered on demand:

- Infrastructure as a Service (IaaS) – makes basic computational resources (e.g., storage, servers) available,
- Platform as a Service (PaaS) – makes offering that enable easy development and deployment of scalable applications, and
- Software as a Service (SaaS) – allows complete end user applications to be deployed, managed, and delivered over the Web.

The big four clouds, EC2 (Amazon 2007), Azure (Microsoft 2009), AppEngine (Google 2009) and Salesforce.com (Salesforce 2009) that represent these three basic cloud abstractions, only provide basic support.

Initially, EC2 was basically hardware as a service; it was for the user of EC2 to create the entire software stack starting with an operating system. Just recently, the Amazon announced their cluster services, i.e. Auto-Scale, Load-balance, and CloudWatch, moving the cloud to the PaaS category. However, features such as Auto-Scale, require involvement of the EC2 client. Before one can use Auto-Scale, one has to create multiple instances of Amazon images for Auto-Scale to utilize.

AppEngine is a PaaS cloud where clients are able to construct services and deploy them to AppEngine for execution without having to rebuild the software stack. However, AppEngine is very restricted in what language and technology can be used to build the services. At the time of writing, AppEngine only supports the Java and Python programming languages.

Both AppEngine and EC2 offer cluster like data processing in the form of MapReduce (Dean and Ghemawat 2004) and Hadoop (Apache 2009) respectively. In AppEngine, services are created to use Google's MapReduce framework while EC2 offers virtual servers with Hadoop installed. However, both MapReduce and Hadoop are restrictive because they are specialized for distributed data processing.

Salesforce is a SaaS cloud: specifically, CRM software as a service. Instead of maintaining hardware and software licenses, use of the software hosted on Salesforce servers for a minimal fee is offered. However, Salesforce is still primitive. The software cannot be customized and discovery of required software is only keyword based.

These approaches appear to have no means of discovery. At this time Windows Azure is seen as the only cloud that offers discovery. An underlying component to Azure is the .NET Services Bus. When a service is hosted in Azure, it is able to register a URI to the Bus so that clients can discover the service. As the Bus does the location resolution, clients are able to use the service no matter where the service is moved. While the Bus offers discovery, its solution is still not satisfactory. The only element that can be registered is a URI. All other elements such as attributes on activity and limitations are not published.

In summary, these four clouds provide some form of service management. However, they require, with the

exception of Azure, new and dedicated programming environments. Furthermore, clients must be heavily involved in configuration of virtual servers and execution of their applications in the same manner as programmers did years ago when they used a command driven Unix system (Chaganti 2008; VCL 2008). Clients face difficult problems of resource discovery and automatic services selection; dynamic sharing toward efficient management of resources; QoS and reputation of providers and clients; and fault tolerance. What is needed is an approach to simplify the publication of clusters, their discovery and actual use. Clients should be able to easily place required files and executables on the cluster, and get the results back without knowing any cluster specifics. A solution is in the proposed Cluster as a Service, a high level abstraction of clusters within clouds.

3 Cluster as a Service

Our Cluster as a Service (CaaS) Technology belongs to the category of PaaS clouds. The purpose of the CaaS Technology is to expose a cluster as a dynamically changing stateful service, manage the discovery and selection of clusters, the specification of cluster jobs¹, the upload of required files, the monitoring of execution and the download of result files. The CaaS Technology does not require any special development environment; it supports development, deployment and execution of applications that traditionally could be executed on a standalone cluster. The CaaS abstraction and technology is applicable to both public and private clouds.

This section discusses the CaaS technology in detail. In particular, as this the technology is immersed in the RVWS framework, the framework is briefly introduced. That is followed by the presentation of the CaaS high level abstraction, architecture, behavior, and the use of a stateful WSDL document.

3.1 RVWS Basics

While Web services have simplified resource access, it is not possible to know if the resource behind the Web service is ready for a request. In fact, it is out right impossible to easily find Web services that satisfy the client requirements. To do so requires clients to research extensively the services themselves before they are used.

To address these issues, we proposed our Resources Via Web Service (RVWS) framework. Diagram 1 shows an overall vision of RVWS in relation to clients and clouds. A key element of RVWS is the discovery of services and resources using state and characteristic attributes published to Web service WSDL documents.

The automatic service discovery allows for both a single service (e.g., a cluster) discovery and selection, and an orchestration of services to satisfy computation workflow requirements. The SLA (Service Level Agreement) reached by the client and cloud service provider specifies attributes of services, in particular clusters, that form the client's request or workflow. This

¹ It is good to recall the difference between processes and jobs. Jobs contain programs, data and even configuration and/or management scripts. A process is a program that is in execution. When clients use a cluster, they submit jobs and one or more processes are created to execute the job.

is followed by the process of services' selection using brokers. Thus, selection is carried out automatically and transparently.

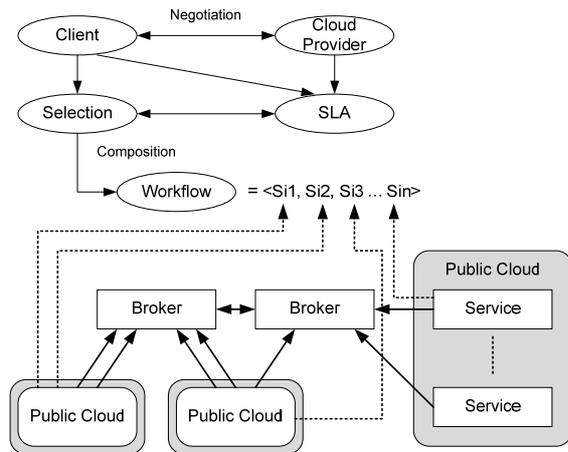


Diagram 1: Dynamic Discovery and Selection

There are two categories of dynamic attributes addressed in the RVWS framework (Brock and Goscinski 2008a), (Brock and Goscinski 2008b): state and characteristic. State attributes cover the current activity of the service and its resources thus indicating if a given service is ready for client requests. Characteristic attributes cover the operational and physical limitations of the service, the resources behind it, quality of service (QoS), price and even information about the providers of the services.

To keep stateful Web services current to their resources, a Connector (Brock and Goscinski 2008a) is used to detect changes in resources and then pass them on to the Web service. The Connector has three logical modules to keep the stateful Web service current: Detection, Decision and Notification. The Detection module routinely queries for attribute information from the resource. Any changes in the attributes are passed to the Decision module that decides if the attribute change is large enough to warrant a notification thus preventing excessive communication with the Web service. If notification is needed, the updated attributes are passed on to the Notification module. Once the attribute changes have been assessed, this module informs the stateful Web service that updates its internal state. When Clients requests the stateful WSDL document, the Web service returns the WSDL document with the values of all attributes at the request time.

Through the extensible nature of the WSDL document, it is possible to include additional information (state and characteristics) by encapsulating it in its own section. All information of service resources is kept in a new WSDL section called the Resources section. For each resource behind the Web service, a ResourceInfo element exists. Each ResourceInfo section has a resource-id attribute and two child sections, the state section and the characteristic section. When the Connector learns of the resource for the first time it publishes the resource to the Web service.

While the stateful WSDL document eliminates the overhead incurred from manually learning the attributes of the service and its resource(s), the issues behind

discovering needed services are still unresolved.

To help ease the discovery of services with stateful WSDL documents, a Broker was proposed (Brock and Goscinski 2009). The Broker is able to transparently contact other known Brokers if it cannot satisfy a Client request.

When publishing to the Broker, the Provider sends attributes of the Web service. The provider is even able to publish attributes about itself, e.g., name, price. After providing attribute information about the Web Service, the Broker gets the stateful WSDL document from the Web service. After getting the stateful WSDL document, the Broker has the complete attribute set that is stored across three stores: the Service, Resources and Provider stores. As the Web service changes, it is able to send a notification to the Broker (and also to the client if necessary) that then updates the relevant attribute in the relevant store.

When seeking desired services, the Client submits to the Broker three groups of attribute values for Service, Resource, and Provider. The Broker compares each attribute group on the related data store. Then, after getting matches, the Broker applies filtering. As the Client using the Broker could be anything from a human operator behind a web browser to another software unit, the resulting matches have to be filtered to suit its needs. Finally, the filtered results are returned to the Client.

3.2 CaaS – High Level of Abstraction

Clouds represent a high level of abstraction of a large distributed system achieved through layers of virtualization and abstraction. One of highest levels of the abstraction hierarchy is the concept of a service. This implies that cloud should offer a simple high level interface that allows clients to discover, select and access cloud resources (storage, CPU cycles) exposed as services easily and transparently.

An attempt of forming a hierarchical model of abstraction to address stateless Web service, stateful Web services based on the WSRF framework that do not expose state directly, and stateful Web services based on the RVWS framework that exposes dynamically changing state directly was presented in (Brock and Goscinski 2008a, Brock and Goscinski 2008b).

A proposal of creating a higher-level abstraction for grids to provide an explicit support for usage modes was introduced in (Jha et.al 2009). However, the authors have not moved beyond a general description of their vision.

This section shows a layer of abstraction that takes advantage of the hierarchy presented in (Brock and Goscinski 2008a, Brock and Goscinski 2008b) that sits within the cloud abstraction. The relationship among existing stateless Web service standards and frameworks, and the level of abstraction provided by the CaaS, an example of the PaaS cloud, that offers an interface for clients are shown in Diagram 2.

The CaaS provides the highest level of abstraction that hides all hardware and software features of cloud clusters. Clients only receive minimum operational amount of data (service location, invocation interface, current state and characteristics information) and are provided Web pages to deploy, run, and control execution of their jobs.

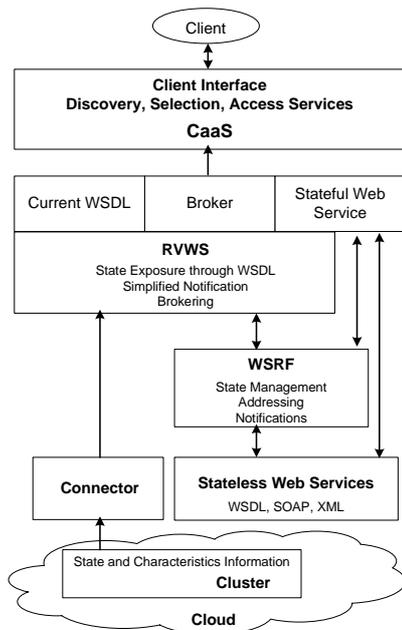


Diagram 2: CaaS Abstraction

3.3 CaaS Architecture and Behavior

The exposure of a cluster via a Service is intricate and comprises several services running with and on top of a physical cluster. Diagram 3 shows the complete solution with a cluster, RVWS and the proposed CaaS service.

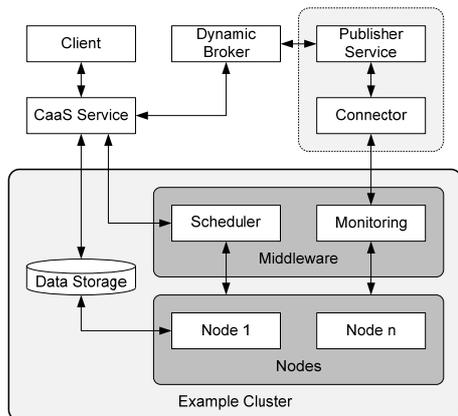


Diagram 3: Complete CaaS System

A typical cluster is comprised of four elements, nodes, fast networks, data storage and middleware. As the focus of this paper is abstraction, only cluster middleware is addressed here. Cluster middleware, a basic level of virtualization of clusters, is comprised of multiple components to manage the cluster and provide a single system image (Goscinski et al. 2002) thus preventing the process running on the cluster from needing to know the cluster organization.

With all the services in the middleware and the changes in node load, there is a lot of information to consider when finding a cluster. As time progresses, the amount of free memory, disk space and CPU usage of each cluster node changes dramatically. Furthermore, information about how quickly the scheduler can take a job and start it on the cluster is vital in choosing a cluster. Currently used Web services (stateless and even WSRF

stateful) do not take this changing information into account.

Thus while the Broker makes information about a cluster known (location and service invoking information), easing the use of a cluster was still left open. Clients could find required clusters but they still had to manually transfer their files, invoke the scheduler and get the results back. All three tasks require knowledge of the cluster and are conducted using work demanding tools.

To make information about the cluster publishable the RVWS framework was used to create a Cluster Connector and Publisher Web service. The role of the Publisher Web service was to show current cluster dynamic attribute information via a stateful WSDL document. To make the Publisher Web service (and the cluster behind it) discoverable, our Broker was used.

The role of the CaaS Service is to (i) find (using the Broker) matching clusters based on client requirements, (ii) provide easy and intuitive file transfer tools so clients can upload jobs and download results, and (iii) offer an easy to use interface for clients to use the cluster.

It may be required that data for cluster jobs be stored in a designated location (a directory) within the storage. As clients to the cluster cannot know any of this information, it is for the CaaS service to abstract the transfer of data files to the point where clients appear to operate the cluster storage as one of their own storage systems. Finally, the CaaS Service communicates with the cluster's scheduler, thus freeing the client from needing to know how the scheduler is invoked when submitting and monitoring jobs.

```
<definitions xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
  <resources>
    <resource-info resource-identifier="resourceId">
      <state element-identifier="elementId">
        <cluster-state element-identifier="cluster-state-root">
          <cluster-node-name free-disk="" free-memory="" native-os-name=""
            native-os-version="" processes-count=""
            processes-running="" cpu-usage-percent=""
            element-identifier="stateElementId" memory-free-percent="" />
          ...Other Cluster Node State Elements...
        </cluster-state>
      </state>
    </resource-info>
    <characteristics element-identifier="characteristicElementId">
      <cluster-characteristics node-count=""
        element-identifier="cluster-characteristics-root">
        <cluster-node-name core-count="" core-speed="" core-speed-unit=""
          hardware-architecture="" total-disk="" total-memory=""
          total-disk-unit="" total-memory-unit=""
          element-identifier="characteristicElementId" />
        ...Other Cluster Node Characteristic Elements...
      </cluster-characteristics>
    </characteristics>
  </resources>
  <types>...
  <message name="MethodSoapIn">...
  <message name="MethodSoapOut">...
  <portType name="CounterServiceSoap">...
  <binding name="CounterServiceSoap" ...>...
  <wSDL:service name="CounterService">...
</wSDL:definitions>
```

Figure 1: Publisher Web Service WSDL

3.4 Publisher Web Service Stateful WSDL

Through the extensible nature of the WSDL schema (Christensen 2001, Papazoglou 2008), it is possible to include additional information (specifically, state and characteristics) into existing WSDL documents. This is possible by encapsulating the additional information in its own WSDL section.

All information of service resources is kept in a new WSDL section called Resources. The core significance of RVWS is its combination of the WSDL of Web services with dynamic attributes. Figure 1 shows the resources section added to the WSDL of the cluster Web service.

Both the state and characteristics elements contain several description elements; each with a name attribute and (if the provider wishes) one or more attributes of the service. Attributes in RVWS use the $\{name: op\ value\}$ notations. An example attribute is $\{cost: \leq \$5\}$. As well as showing the attributes in the stateful WSDL document, the attribute information has to be organized so Clients viewing the stateful WSDL document immediately understand what each attribute means.

For the CaaS service to properly support the role of cluster discovery, extensive information about clusters and their individual nodes needs to be published to the WSDL document of the Cluster Web Service and subsequently to the Broker. Table 1 shows attributes of each node in a cluster.

4 CaaS Service Design

This section discusses the CaaS Service design. In particular, it shows the CaaS components, user interfaces, and their behavior. The CaaS service carries out three main tasks: Cluster Discovery and Selection, Job Management and File Management. Given the size and number of tasks, the CaaS service was modularized. Diagram 4 shows the structure of the service.

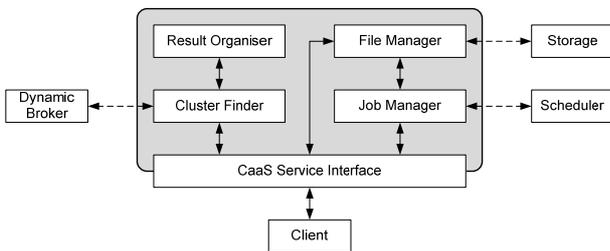


Diagram 4: CaaS Service Design

The modules inside the Web service are only accessed through an interface. The use of the interface means the Web service can be updated over time without requiring clients to be updated nor modified.

Invoking an operation on the CaaS Service Interface (discovery, selection, etc) invokes other operations on other modules. Thus, to describe the role each module plays in the CaaS service, the following sub-sections outline the various tasks the CaaS service carries out.

4.1 Cluster Discovery and Selection

The dynamic attribute information only relates to clients that are aware of them. Human clients know what the attributes are, owing to the section being clearly named. Software clients designed pre-RVWS ignore the additional information as they follow the WSDL schema that we have not changed.

When discovering services, the client must submit to the Broker three groups of attribute values (1 in Diagram 5); Service, Resource, and Provider. Thus to start discovery, clients provide cluster requirements in the form of attribute values, such as the number of nodes, to the CaaS Service Interface (1). Next, the CaaS Service Interface invokes the Cluster Finder module (2) that communicates with the Broker (3). The Broker returns (if any) an array of service matches which offer clusters that match the supplied requirements.

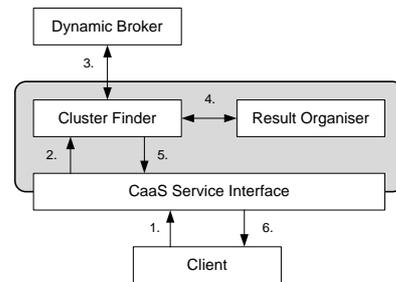


Diagram 5: Discovering suitable Clusters

To address the granularity of the Broker results, the Cluster Finder module invokes the Results Organizer module (4) that takes the Broker results and returns a summarized version. After getting the organized results, the results are returned to the client via the CaaS Service Interface (5-6). The organized results instruct the client what clusters exist and how each cluster matches up to the requirements. After reviewing the results, the client is

Table 1 Cluster Attributes

| Type | Attribute Name | Attribute Description | Source |
|-----------------|----------------------------|---|--------------|
| Characteristics | core-speed | Speed of each core | Cluster Node |
| | core-speed-unit | Unit for the core speed (e.g.: GigaHertz) | |
| | total-disk | Total amount of physical storage space | |
| | total-disk-unit | Storage amount unit (e.g.: Gigabytes) | |
| | total-memory | Total amount of physical memory | |
| | total-memory-unit | Memory amount measurement (e.g.: Gigabytes) | |
| | supported-software-name | Name of a single piece of software installed on the cluster | |
| | supported-software-type | Type of software installed on the cluster (eg: operating system) | |
| | supported-software-version | Version of a single piece of software installed on the cluster (eg: 6.1.0) | |
| | node-count | Total number of nodes in the cluster | Generated |
| State | free-disk | Amount of free disk space | Cluster Node |
| | free-memory | Amount of free memory | |
| | os-name | Name of the running operating system | |
| | os-version | Version of the running operating system | |
| | cpu-usage-percent | Overall percent of CPU used. As this metric is for the node itself, this value becomes averaged over cluster core | Generated |
| | memory-free-percent | Amount of free memory on the cluster node | |

informed enough to choose a cluster.

The automatic selection of services is performed to optimize a function reflecting client requirements. Time critical and high throughput tasks benefit by executing a computing intensive application on multiple clusters exposed as services of one or many clouds.

4.2 Job Submission

After selecting a required cluster, all executables and data files have to be transferred to the cluster and the job submitted to the scheduler for execution, as shown in the Diagram 6 workflow.

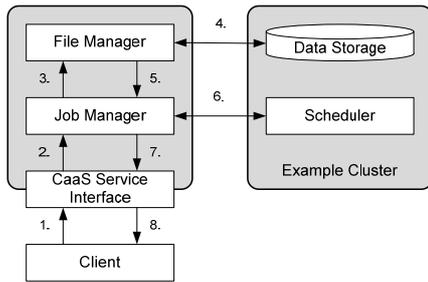


Diagram 6: Job Submission

All required job execution parameters, data files, executables, scripts, software libraries (if any) and are uploaded to the CaaS Service (1). Once the file upload is complete, the Job Manager is invoked (2). As the CaaS Service still has the required job files, the first task the Job Manager resolves is the transfer of all files to the cluster by invoking the File Manager (3). The File Manager makes a connection to the cluster data storage and commences the transfer of all files (4). Upon completion of the transfer (4), the outcome is reported back to the Job Manager (5). On failure, a report is sent and the client can decide on the appropriate response.

If the file transfer was successful, the Job Manager invokes the scheduler on the cluster (6) with the execution parameters given in (1). If the outcome of the scheduler (6) is successful, the client is then informed via the CaaS Service Interface (7-8). The information conveyed includes the response from the scheduler, the job identifier the scheduler gave to the job and any other information the scheduler provides.

4.3 Job Monitoring

Diagram 7 outlines the workflow the client takes when querying about his or her job after submitting it.

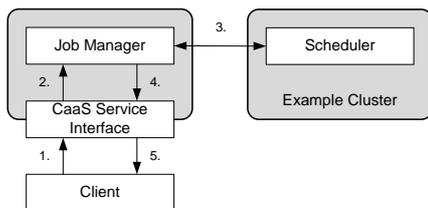


Diagram 7: Job Monitoring

The client first contacts the CaaS Service Interface (1) that then invokes the Job Manager module (2). No matter what the operation is (check, pause or terminate), the Job

Manager only has to communicate with the Scheduler (3) and reports back a successful outcome to the Client (4-5).

4.4 Result Collection

The final role of the CaaS Service is addressing jobs that have terminated or have completed their execution successfully. In either case, data/error files meant for the client need to be transferred to the client, Diagram 8.

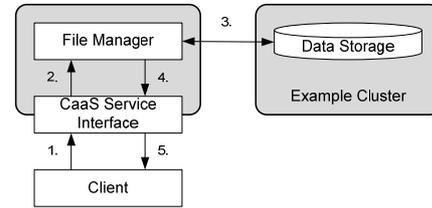


Diagram 8: Job Result Collection

Clients start the result/error file transfer by contacting the CaaS Service Interface (1) that then invokes the File Manager (2) to retrieve the files from the cluster's data storage (3). If there is a transfer error, the File Manager attempts to resolve the issue first before informing the client. If the transfer of files (3) is successful, the files are returned to the CaaS Service Interface (4) and then the client (5). When returning the files, URL link or a FTP address is provided so the client can retrieve the files.

4.5 User Interface

Users access systems based on their interfaces and how easy they are to use. Thus, to ease the use of CaaS service, a series of Web pages has been designed. Each page in the series covers a step in the process of discovering, selecting and using a cluster.

| | |
|---|--|
| Section A: Hardware | |
| Number of Nodes: | <input type="text" value="50"/> |
| Amount of Memory: | <input type="text" value="50"/> GB |
| Free Memory: | <input type="text" value="50"/> GB |
| Disk Free: | <input type="text" value="50"/> GB |
| CPU: | <input type="text" value="Pentium 4"/> <input type="text" value="64bit"/> <input type="text" value="3.2"/> GHz |
| Section B: Software | |
| Operating System: | <input type="text" value="Windows XP w/ Service Pack 2"/> |
| <input type="button" value="Discover ->"/> | |

Diagram 9: Cluster Discovery – Cluster Specification

Diagram 9 shows the Cluster Specification Web page to start cluster discovery. In Section A, the client is able to specify attributes about the required cluster. Section B allows the client to specify any required software the cluster job needs. Once all attributes have been specified, the attributes are then given to the CaaS service, which performs a search for possible clusters and the results are displayed in a Select Cluster Web page, Diagram 10. A match is indicated by showing a tick in the cell or a value of an attribute. The client can choose a cluster or go back to refine the discovery.

| | Cluster A select | Cluster B select |
|-------------------|-------------------------------------|-------------------------------------|
| Hardware | | |
| Number of Nodes: | <input checked="" type="checkbox"/> | |
| Amount of Memory: | <input checked="" type="checkbox"/> | |
| Free Memory: | <input checked="" type="checkbox"/> | |
| Disk Free: | | <input checked="" type="checkbox"/> |
| CPU: | <input checked="" type="checkbox"/> | |
| Architecture: | <input checked="" type="checkbox"/> | |
| Speed: | | <input checked="" type="checkbox"/> |
| Software | | |
| Operating System: | <input checked="" type="checkbox"/> | |
| Architecture: | <input checked="" type="checkbox"/> | |
| Version: | <input checked="" type="checkbox"/> | |

<- Refine Search

Diagram 10: Cluster Discovery – Cluster Selection

Next, the client goes to the Job Specification page, Diagram 11. Section A allows specifying information about the job. Section B allows the client to specify and upload all data files and job executables. If the job is complex, Section B also allows specifying a job script. Section C allows specifying an estimated time the job would take to complete.

Section A: Identification

Job Name: Travelling Sales Man

Job Owner: Joe Bloggs

Section B: Job File Specification

Executable: My_exec.exe

Script: my_script.pl

Data files: custom_set.dat

Proven.dat
Control.dat
Recent.dat

Output Filename: out.dat

Section C: Execution Specification

Estimated Time: 3d 14h

<- Change Clusters

Diagram 11: Job Execution – Job Specification

Once submitted, the Cluster Web service attempts to submit the job. The outcome of the submit attempt is shown in the Job Monitoring page, Diagram 12. Section A tells the client whether the job is submitted successfully. Section B offers commands to allow the client to refresh the displayed information, pause the job, and even halt it.

Section A: Submission Outcome

Outcome: Submitted Successfully

Job ID: cj404

Report: Delegating Submission request.... Request Accepted. Job has been started.

Section B: Job Control

Diagram 12: Job Execution – Job Monitoring

When the job is complete, the client is able to collect the results from the Collect Results page Diagram 13. Section A shows the outcome of the job. Section B allows the client to easily download the output file generated from the completed/aborted job via HTTP or using an FTP client.

Section A: Execution Outcome

Outcome: Completed Successfully

Time Finished: 16:59

Report: After a total of 2 days and 7 hours, your job has completed execution.

Section B: Results Download

HTTP: <http://download.clustera.org/cb404/out.dat>

Diagram 13: Job Execution – Result Collection

5 Proof of Concept

This paper proposes a new technology. Thus, it is important to demonstrate that it is feasible. This proof of concept is provided by showing the CaaS implementation and experiments carried out.

5.1 CaaS Implementation

The CaaS service was implemented using Windows Communication Foundations (WCF) of .NET 3.5 that uses Web services. The CaaS is shown in Diagram 14.

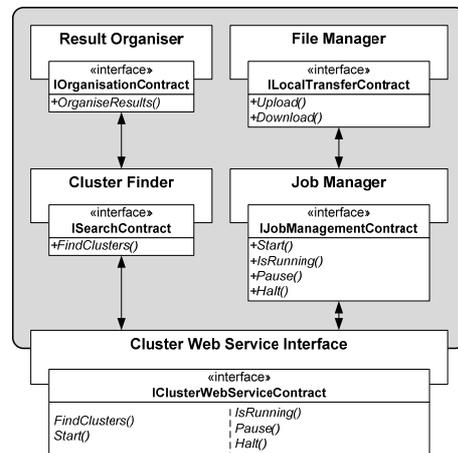


Diagram 14: Cluster Service Implementation

Each module presented in Section 4 is implemented as its own Web service. An open source library for building SSH clients in .NET called sharpSsh (Gal 2005) was used in the implementation of the Job and File Managers. As schedulers are mostly command driven, the commands and outputs were wrapped into a Service.

One final implementation change we made was adding automatic hostfile generation. This is to prevent over allocation of cluster nodes to the job. For example, if we only ask for 2 nodes, a host file containing two nodes will be generated to prevent the cluster from allocating more than two nodes.

5.2 Environment

The experiments were carried out on a single cluster exposed via CaaS; communication was carried out only through the CaaS service interface. To manage all the services and databases needed to expose and use the cluster via CaaS, VMware virtual machines were used extensively. Diagram 15 shows the complete test environment with the contents of each virtual machine.

All virtual machines have 512 MB of virtual memory and all (except the client VM) ran the Windows Server

2003. The client virtual machine ran Ubuntu 9.04. All Windows virtual machines run .NET 2.0; the CaaS virtual machine runs .NET 3.5.

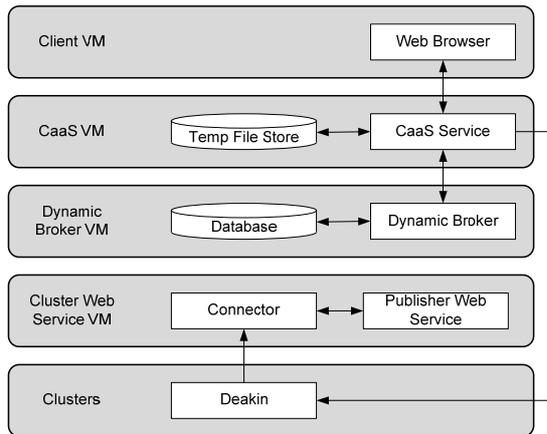


Diagram 15: Complete CaaS Environment

The first virtual machine is the Publisher Web service virtual machine. It contains the cluster Connector, the Publisher Web service and all required software libraries to make their execution possible. The Broker virtual machine contains the Broker and its database. The CaaS virtual machine houses the CaaS Server and a temporary data store. To improve reliability, all file transfers between the cluster and the client are cached.

5.3 Cluster and Job Specifications

The cluster used in the proof of concept consists of 20 nodes plus two head nodes (one running Linux and the other running Windows). Each node in the cluster has a two Intel Cloverton Quad Core CPUs running at 1.6 GHz, 8 Gigabytes of memory, 250 Gigabytes of data storage and all nodes are connected via Gigabit Ethernet and Infiniband. The head nodes are the same except they have 1.2 Terabytes of data storage.

For our tests, an mpiBLAST application was used. mpiBLAST is a distributed application used to find gene sequences in genome databases: a common testing bioinformatics. When running mpiBLAST, at least two files are needed: the database and a sequence file. To simplify testing, we placed the database on the cluster and only uploaded the sequence file.

5.4 Experiments and Results

5.4.1 Publication

Due to space limitations, the process of publishing a cluster and learning of its dynamic attributes via the Broker are omitted. However, this idea was well tested and the outcomes documented in (Brock and Goscinski 2009a).

5.4.2 Discovery and Selection

Diagram 16 shows the workflow behind the cluster's discovery. The required cluster information was submitted to the CaaS Service (1). We requested a cluster with at least 20 nodes, each with at least 6 Gigabytes of free memory and all nodes running a Linux system. The CaaS Service then contacted the Broker with the specified

requirements (2). The Broker queries its database for matches (3) and returns the results to the CaaS Service (4). The cluster matches are returned in (4). If 'information overload' is in place, the CaaS through its Results Organizer takes the matches, tabulates them and returns to the client (5).

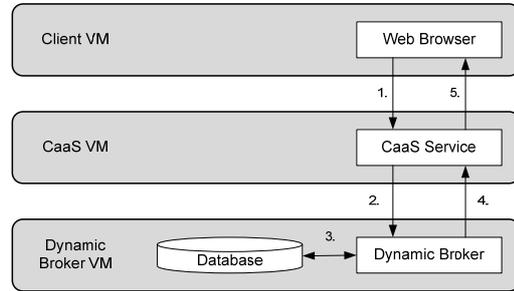


Diagram 16: Cluster Discovery

5.4.3 Job Submission

Diagram 17 shows the workflow behind submitting a job to the cluster. First, the cluster job is specified and submitted to the cluster (1). During submission, parameters such as the name of the job and its required execution time are specified. Along with the job parameters, our job script file and data files contained in a zip file are also submitted. To improve reliability, all files are kept in a temporary file store (2). Once all the files have been transferred, the CaaS service transfers the files to the chosen cluster (3). After all files are transferred, the scheduler is invoked and the outcome returned to the client (4).

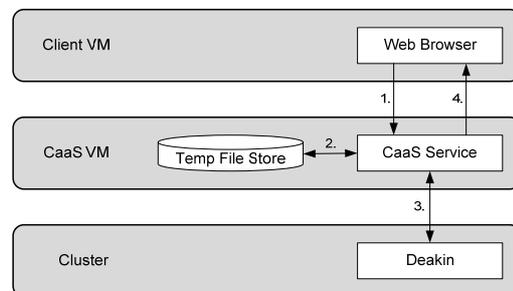


Diagram 17: Job Submission

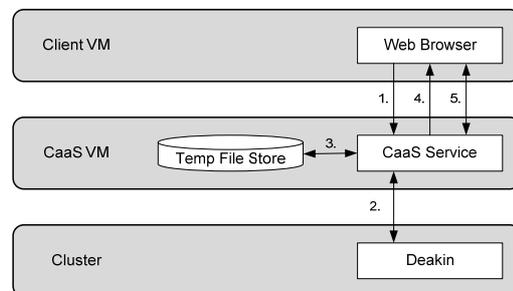


Diagram 18: Result Collection

5.4.4 Collection

The final experiment commences once the Job Monitoring Web page (Diagram 12) reports that the job has finished. Using the Web browser, a request to collect

the results is to be made (1 in Diagram 18) and the CaaS Service retrieves the result file to be stored in the File Store (2-3).

Once the file is transferred, it is expected that the CaaS Service show the Result Collection page (Diagram 13) and provide hyperlinks to download the result files over HTTP (4). After downloading the results files, confirmation is sent to the CaaS service (5) that removes the file from the File Store and instructs the cluster to remove its copy (6).

5.4.5 Results

Experiment 1: Discovery: As stated in Section II, cluster clients still have to discover clusters first. This is a problem as there is no discovery system for clusters. Hence, our first experiment was to see if a cluster was easily discovered through our CaaS Technology.

Diagram 19 shows the cluster discovery page populated with the requirements for our first mpiBLAST job. For this experiment, we only needed four cluster nodes, each with 8 Gigabytes of memory and did not have more than 10% utilization of their CPUs.

Section A: Hardware

Number of Nodes:

Amount of Memory:

Free Memory:

Disk Free:

CPU:

CPU Utilisation:

Diagram 19: Specifying Cluster Requirements

After specifying our requirements, the Web page passed the requirements to the CaaS Service for processing. As stated in Section III, the Dynamic Broker is first queried with the requirements and the results then processed by the Results Organiser. Diagram 20 shows a formatted version of the organized results.

| Cluster | Hardware | | | | | | Software | | | |
|---------|----------|-------------|-----------|-----------|-----------|-----------|------------|---------|---------|----------|
| | Nodes | Mem. Amount | Mem. Free | Disk Free | CPU Arch. | CPU Speed | Nodes Idle | OS Name | OS Ver. | OS Arch. |
| Deakin | 19 | 19 | n/a | - | 0 | - | 17 | 0 | - | - |

Diagram 20: Cluster Match Results

While the Web page was designed to have ticks, it was decided late in development to use numbers instead. The reason for this was the numbers gave a clearer view on how each requirement was satisfied by each cluster.

Overall, this experiment was a complete success. We had our cluster exposed via the Publisher Web service, and easily discovery the cluster through the CaaS Service. This is a significant contribution as a discovery service now exists to allow human operators to easily locate a required cluster.

Experiment 2: Job Submission: As stated in Section II, clusters can vary in how they are built. Specifically, not all clusters run the same middleware. Hence this experiment was carried out to see if the middleware specifics could be hidden.

Even though our cluster consisted of 20 nodes, only 19 were active at the time of testing. Furthermore, for this test, we only wanted to use two cluster nodes. Thus, a

successful outcome of this test had to show only two cluster nodes being used and not the whole cluster.

Diagram 21 shows the Job Specification Web page where our first mpiBLAST job was specified. For this test, we specified an mpiBLAST launch script, a sequence file to compare against a mouse database and the name of an output file was specified.

Section A: Job Identification

Job Name:

Job Owner:

Section B: Job File Submission

Executable:

Script:

Data Files:

Name of Output File:

Diagram 21: Specifying the Job

After completing the Job Specification Web page, our script and test file were uploaded to the CaaS VM and then transferred to the cluster. After all files were transferred, the scheduler (GridEngine) was invoked. Diagram 22 shows the resulting Job Monitoring Web page.

Section A: Submission Outcome

Outcome: Your job 76712 ("sge_script.sh") has been submitted
 Job ID: 76712
 Report: 30/10/2009 5:21:49 PM: Your job is still running.

Diagram 22: Monitoring the Progress of the Job

As the hostfile for our job was generated dynamically, we could not tell its contents during the submission process. After the completion of Experiment 4, the contents of the host file would be checked.

Experiment 4: Results Collection: Just as how clients need to be able to easily upload their jobs to clusters, they need to be able to download any result or error files.

To know when our job finished, the Job Monitoring Web page was refreshed a few times before indicating that the second job had completed (Diagram 23).

Section A: Submission Outcome

Outcome: Your job 76712 ("sge_script.sh") has been submitted
 Job ID: 76712
 Report: 30/10/2009 5:21:49 PM: Your job is still running.
 30/10/2009 5:23:12 PM: Your job is still running.
 30/10/2009 5:24:00 PM: Your job is still running.
 30/10/2009 5:24:36 PM: Your job is still running.
 30/10/2009 5:25:15 PM: Your job is still running.
 30/10/2009 5:26:17 PM: Your job is still running.
 30/10/2009 5:27:09 PM: Your job appears to have finished.
 30/10/2009 5:27:09 PM: Please collect your result files.

Section B: Job Control

Diagram 23: Completion Notification

Diagram 24 shows the Results Collection Web page

with a hyperlink to our result file. By clicking the link, we were able to download our results just like any other file on the Web.

With the completion of this experiment, our CaaS Technology with its Web pages, as a whole was proven successful. We were able to expose a cluster via Web services, discover it, run multiple jobs without any clashes, and easily get result data back. All of this was done with no computing expertise at all.

Section A: Execution Outcome

Outcome: Completed Successfully
Time Finished:
Report:

Section B: Result File Download

[HTTP: outcome_4.txt](http://outcome_4.txt)

Diagram 24: Collecting Job Results

With the completion of the job execution, we needed to examine the hostfile used to influence how the job was scheduled to the cluster. As Figure 2 shows, only two nodes were listed. This is a significant advancement as not only was the cluster made easy to use, but the client was also reserved the nodes available at the time of his or her request.

```
west-lin (mrab) 1005 $cat hostfile
west-07
west-16
west-19
west-12
```

Figure 2: Hostfile Contents

6 Conclusion

We have achieved the goal of this project by the development of a technology for building a Cluster as a Service (CaaS) using the RVWS framework. Through the combination of dynamic attributes, Web service's WSDL and Brokering, we successfully created a Service that quickly and easily published, discovered and selected a cluster, allowed to specify a job and execute it, and finally got the result file back.

The proposed technology forms a bridge between the dynamic attribute and Web service based Resources Via Services (RVWS) framework and a high level abstraction of clusters in clouds in the form of a CaaS was specified. This outcome could have significant impact on cloud computing.

7 References

- Amazon (2007) Amazon Elastic Compute Cloud. Accessed 1 August 2009, <http://aws.amazon.com/ec2/>.
- Amazon (2009) EC2StartersGuide, <https://help.ubuntu.com/community/EC2StartersGuide>
- Apache (2009) Hadoop, Accessed 1 August 2009, <http://hadoop.apache.org>
- M. Brock & A. Goscinski (2008a) Publishing Dynamic State Changes of Resources Through State Aware

WSDL. Int. Conf. on Web Services (ICWS) 2008. Beijing.

- M. Brock and A. Goscinski (2008b) State Aware WSDL. Sixth Australasian Symposium on Grid Computing and e-Research (AusGrid 2008). Wollongong, Australia, 82, 35-44, ACM.
- M. Brock and A. Goscinski (2009) Attributed Publication and Selection for Service-based Distributed Systems. Int. Workshop on Service Intelligence and Computing (SIC 2009). Los Angeles, IEEE.
- P. Chaganti 2008. Cloud Computing with Amazon Services, <http://ibm.com/developerswork/architecture/library/ar-cloudaws3/>
- E. Christensen, F. Curbera, G. Meredith and S. Weerawarana (2001) Web Services Description Language (WSDL) Version 1.1. Updated 15 March 2001, Accessed, <http://www.w3.org/TR/wsdl>.
- K. Czajkowski, et al. (2004) The WS-Resource Framework. 5 March 2004. <http://www.globus.org/wsrf/specs/ws-wsrf.pdf>.
- J. Dean and S. Ghemawat (2004) MapReduce: Simplified Data Processing on Large Clusters. Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
- T. Gal (2005) sharpSsh - A Secure Shell (SSH) library for .NET. Updated 30 October 2005, Accessed 1 March 2009, www.codeproject.com/KB/IP/sharppssh.aspx.
- Google (2009) App Engine. Accessed 17 February 2009, <http://code.google.com/appengine/>.
- A. Goscinski, M. Hobbs and J. Silcock (2002) GENESIS: An Efficient, Transparent and Easy to Use Cluster Operating System. Parallel Computing, Vol. 28 (2002), No. 4, April, 557-606.
- S. Jha, A. Merzky, G. Fox (2009) Using Clouds to Provide Grids Higher-Levels of Abstraction and Explicit Support for Usage Models, Version: 1.0, GFD-I.150.
- Microsoft (2009) Azure. Accessed 5 May 2009, <http://www.microsoft.com/azure/default.aspx>.
- M. Papazoglou and W-Jan van den Heuvel (2007) Service oriented architectures: approaches, technologies and research issues, The VLDB Journal (2007) 16:389-415.
- M. Papazoglou (2008) Web Services: Principles and Technology, Prentice Hall.
- Salesforce (2009) Accessed August 1 2009, www.salesforce.com
- VCL (2008) <http://vcl.ncsu.edu/>.

A New Integrated Unicast/Multicast Scheduler for Input-Queued Switches

Kwan-Wu Chin

School of Electrical, Computer and Telecommunications Engineering

University of Wollongong

kwanwu@uow.edu.au

Abstract

Researchers have thus far considered scheduling unicast and multicast traffic separately, and have paid little attention to integrated schedulers. To this end, we present a new integrated scheduler that considers both unicast and multicast traffic simultaneously and also addresses key shortcomings of existing approaches. Specifically, we outline a scheduler that achieves 100% throughput, and unlike existing schemes, do not require a tuning knob. Moreover, from our extensive simulation studies, we show that it works well in uniform, non-uniform and bursty traffic scenarios.

1 Introduction

The Internet is growing at a rapid pace, driven by the proliferation of high bandwidth applications capable of delivering voice and video traffic. This is particularly evident on Internet 2, where such applications are being used to deliver television programs, lectures, conduct video conferences, and to create interactive and collaborative research environments [1]. As a result, given their high bandwidth demands, Internet service providers are in need of switches/routers that are capable of switching unicast and multicast cells at high speeds.

To date, researchers have proposed a myriad of router designs capable of switching packets or cells at speeds ranging from gigabits to terabits per-second; see [5]. The most popular design is based on the input queued architecture, as it has good scalability with respect to switch size and link rate [5]. Figure 1 shows a block diagram of one such router with N inputs and N outputs connected by a crossbar fabric. It operates in cell mode where variable length packets are fragmented into fixed size cells before traversing the crossbar. They are then re-assembled at their respective output before leaving the router [8]. Each input has N virtual output queues (VoQs) for storing the corresponding unicast cells of N outputs. Without VoQs, a router will experience the head of line (HOL) blocking problem, which limits its throughput to only 58.6% [4]. Unlike previous router designs [3][15][6], which maintain $k < 2^N - 1$ multicast queues, our router has a single multicast queue and N staging buffers; their use will be explained in Section 3.

The scheduler is a key component of any high speed routers. It is responsible for arbitrating cells/packets from input ports across a switching fabric to output ports. Ideally, the scheduler must have 100% throughput and low complexity. In this respect, a significant amount of work has been devoted to unicast scheduling algorithms, the

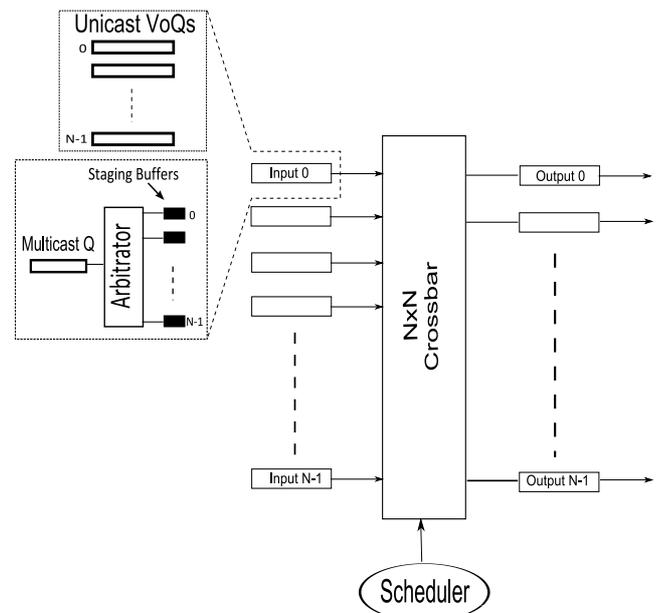


Figure 1: Input-queued switch architecture. Also shown is an *Arbitrator* managing N staging buffers.

most popular being iSLIP [10]. Similarly, a lot of efforts have been devoted to developing high speed multicast scheduling algorithms. Examples include Concentrate, TATRA and WBA [12], ESLIP [9] and Max-Scalar [6]. However, little attention has been paid to integrated schedulers. That is, a scheduler that considers both unicast and multicast cells simultaneously rather than separately.

This paper, therefore, adds to the existing state-of-the-art by proposing an integrated scheduler that overcomes limitations with existing approaches. Specifically, it works in conjunction with staging buffers to overcome the multicast cell HOL problem. Moreover, the proposed scheduler considers the weight of both unicast and multicast cells simultaneously, and hence works well in both uniform and non-uniform traffic scenarios. Our simulation studies involving uniform, non-uniform and bursty traffic sources show that our scheduler has 100% throughput, fair to both unicast and multicast traffic, and achieves superior performance over existing schedulers. Lastly, unlike Zhu et al. [16]'s scheduler, our scheme does not involve a tuning knob. This is a significant advantage because it frees the scheduler from continuously adjusting its behavior with changing traffic conditions.

This paper is organized as follows. We first review existing works and highlight their limitations in Section 2. After that, Section 3 outlines our integrated scheduler and the aforementioned staging buffers. Then, in Section 4, we discuss our simulation parameters. Section 5 presents our experimentation results on an $N \times N$ switch over varying traffic load and cell types. We then discuss our results in

Section 6, before concluding in Section 7. Note, in our discussions to follow, we use the term router and switch interchangeably.

2 Background

Each multicast cell has a fanout set that specifies its outgoing outputs. This is the key reason that complicates multicast cells scheduling, especially when cells have varying fanout sizes that can range from 1 to N , assuming a $N \times N$ switch. Hence, in each time slot, a router's load can increase by N^2 . Moreover, Andrews et al. [2] have shown that scheduling multicast cells is a NP-hard problem. Besides that, there is also the HOL multicast cell blocking problem. Assume cell C_1 and C_2 's fanout vector is $\{0,1,2,3\}$ and $\{1,2\}$ respectively. If C_2 is queued behind C_1 , then it will have to wait until all destination outputs of C_1 have a received a copy of C_1 before it receives service; i.e., at least four time slots. Note that each cell will have to contend with other multicast and unicast cells headed to the same output. Clearly, a switch's performance degrades when it persistently receives multicast cells with a large fanout. One naive approach to address this problem is to have $2^N - 1$ queues, where each queue stores cells headed to the same set of outputs. Unfortunately, this solution is not scalable, especially in large switches. Hence, researchers, such as [3], use k multicast queues instead, where $k < 2^N - 1$. As a rule of thumb, for a switch with N outputs, $2N$ multicast queues are needed to ensure good performance. We will show in Section 3 how staging buffers reduce this memory requirement further by storing only the address of a multicast cell.

As mentioned earlier, most researchers have developed schedulers that are optimized for either unicast or multicast traffic, and not many are designed for both unicast and multicast cells. In fact, only a handful of schedulers exist. Andrews et al. [2] propose that inputs transmit unicast traffic to outputs left unmatched by the multicast scheduler. Unfortunately, their approach leads to the starvation of unicast flows, and does not address the HOL blocking problem. Apart from that, their scheme is unfair to unicast traffic because it gives higher priority to multicast cells.

Schiattarella et al. [13] propose an approach that first uses a unicast and a multicast scheduler to independently derive the maximal matchings for unicast and multicast cells. A module then filters and integrates the matchings found from both schedulers in a fair manner. To avoid starvation, the module ensures edges that missed out in the current time slot will receive service in the next time slot. Their approach, however, is unnecessarily complex and do not consider the weight of unicast and multicast cells simultaneously.

Minkenberg [11] proposes to duplicate the address of a multicast cell into VOQs that correspond to its fanout. In effect, treating a multicast session with a fanout size of n as n unicast sessions. They showed that their scheme is better than the Concentrate scheme [12], but unfortunately its performance is worst than Concentrate for input queued switches. In particular, it does not take advantage of a crossbar switch innate multicast ability. Apart from that, it is not scalable, as input buffers need to have high write bandwidth.

In [8], McKeown presents ESLIP, a multicast extension of iSLIP [10]. Each input has a multicast queue, and a global multicast pointer a_M that points to the input receiving multicast service. The pointer a_M is updated in a round robin manner after the scheduler has sent a copy of a cell to all outputs in its fanout. In each round, inputs send a request to outputs corresponding to non empty queues. Outputs then consider these requests and send a grant to the input with the highest priority traffic. If that happens to be a multicast cell, the output sends its grant to the input a_M is pointing at. Inputs then send an accept to the output corresponding to its highest priority traffic.

ESLIP, however, suffers from the HOL blocking problem, and does not allow different multicast queues to receive service in the same round. Moreover, like iSLIP, it does not perform well when traffic are non-uniform.

Lastly, Zhu et al. [16] propose a scheduler, called slot-coupled integration algorithm (SCIA), that preferentially schedules unicast or multicast cells according to a probabilistic parameter called S_m . Specifically, if a time slot is marked as unicast, outputs will first consider unicast requests from inputs before considering multicast requests. Hence, a multicast request is only granted if there are no unicast requests. Similarly, input ports preferentially accept unicast grants. On the other hand, if a time slot is marked as multicast, then inputs and outputs will process multicast requests/grants first. The main limitation with Zhu et al.'s work is that their approach does not consider the weight of unicast and multicast cells simultaneously. For example, in a multicast time slot, some outputs in a multicast cell's fan-out vector may have higher weighted unicast cells awaiting transmission. Lastly, their scheme is designed for uniform traffic only, and is sensitive to the parameter S_m ; as we will show in Section 5.

3 Integrated Scheduler

To address the aforementioned limitations, we propose to have staging buffers at each input, and an integrated scheduler that makes use of them to schedule both unicast and multicast cells simultaneously. Note, we assume fanout splitting, as this ensures the switch is work conserving and have high throughput [6][2]. Also, the switch operates without any speedup.

3.1 Staging Buffers

Each input, see Figure 1, has N staging buffers corresponding to N outputs; each capable of holding the address of one cell. We refer to a buffer corresponding to output- j at input- i as S_{ij} , where i and j ranges from 0 to $N - 1$ for a $N \times N$ router. The aim of these buffers is to prevent the HOL blocking problem without having to maintain $2^N - 1$ multicast queues. All buffers are managed by the arbitrator, which is responsible for scanning the multicast queue and determining the next multicast cell destined for a given output. Specifically, when the arbitrator finds an empty buffer, say S_{ij} , it starts looking for the oldest cell in the multicast queue that is headed to output- j . This ensures cells destined for output- j are not transmitted out-of-order. Once a cell is found, the arbitrator stores the cell's address in S_{ij} .

Figure 2 shows an example staging buffers implementation using Ternary Content Addressable Memory (TCAM) and Random Access Memory (RAM) [14]. When a multicast cell arrives, a tag is created using the cell's fanout bitmap b and its timestamp ts ; the former is simply a bitstring of length N that identifies the set of outputs; e.g., 101 corresponds to outputs 1 and 2. The later is the modulo of the cell's arrival time and W ; i.e., ts is $\log_2(W)$ in size. The resulting tag is then associated with the cell's memory address in the cell buffer RAM. Lastly, ts is added to the corresponding per-output timestamp FIFO queues (POTQ).

The arbitrator is responsible for filling the staging buffers with the address of multicast cells. When a staging buffer S_j is empty, the arbitrator executes the following steps:

1. Set $ts = Dequeue(TS[j])$, where $TS[j]$ refers to the HOL ts value of output j 's POTQ.
2. Construct tag (j, ts) , and perform a TCAM lookup.
3. Copy the returned cell's address corresponding to (j, ts) into S_j .

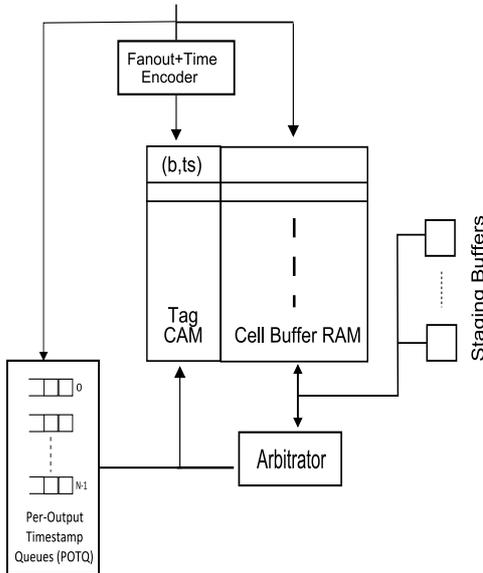


Figure 2: An example staging buffers architecture. b corresponds to the fanout bitmap, and ts is a cell's arrival timestamp (in slot).

After a cell has been transferred, the arbitrator decrements the cell's fanout counter. Here, we assume cells have an associated counter that stores their fanout size. Once the counter reaches zero, the arbitrator frees the memory occupied by the cell.

3.2 Scheduler

Given a $N \times N$ switch with unmatched inputs and outputs, the scheduler executes the following steps at each iteration until no more matches are found:

1. *Request.* Each unmatched input sends a request to every unmatched output corresponding to non-empty VoQs. Requests are also sent for each non-empty staging buffer corresponding to an unmatched output.
2. *Grant.* An unmatched output processes these requests and grants the request with the highest weighted cell. Moreover, the output informs the input whether the grant is due to a unicast or multicast cell.
3. *Accept.* An unmatched input first determines the highest weighted cell with a grant. If it is a unicast cell, the input sends an accept to the corresponding output. However, if the grant is for a multicast cell, the input sends an accept to all outputs that have sent a grant for that cell. In other words, for each staging buffer with a grant and holding the highest weighted cell, an accept is sent to the corresponding output port.

The time complexity at each output is $O(2N)$, since there are N unicast and N multicast requests. In the worst case scenario, the convergence time is $O(N)$ because in each round it is possible only one request is granted. However, in our experiments, convergence time is far smaller than N , especially when multicast cells have a large fanout.

3.3 Example

Figure 3 shows a 3×3 input queued switch. Input-0 has two cells for output 0 and 2, input-1 has a unicast cell for output-0 and also a multicast cell for outputs 1 and 0. Lastly, output-2 has a cell for output-1. In this example, assume that the multicast cells have a higher weight than

all unicast cells; this would be the case if they did not receive any service in previous time slots. Moreover, we only show staging buffers at input-1.

Starting at the Request stage, all inputs send a request message to outputs corresponding to non-empty VoQs and staging buffers. Notice that input-1 sends three requests to output-1, corresponding to its unicast and multicast cells. Each output then considers the cells' weight, as specified in each request message, and sends a grant to the input with the highest weight. In this example, input-1 receives two grants from input-0 and 1 respectively, and input-0 has a grant from output-2. Finally, input-0 sends an accept to output-0, and input-1 accepts both output 0 and 1's grant, thereby allowing the cell to be transferred using the crossbar's multicast capability.

4 Simulation Methodology

To study our integrated scheduler, we used SIM [7], and conducted experiments on a $N \times N$ switch; the value of N is specific to the experiment, and the crossbar connecting them has a speedup of one. All simulation runs are for 10 million slots time, and after each run we compute the average latency of unicast and multicast cells. We also record the switch's throughput – the number of matches over the number outputs. All inputs have infinite buffer size. In addition, we use cells' age as weight. Lastly, we set the maximum number of iterations in each time slot to be five.

For comparison purposes, we implemented the following scheduling algorithms:

- *iSLIP-Emulate* [11]. This algorithm creates copies of a multicast cell, and inserts them in VoQs corresponding to the cell's fan-out vector. The VoQs are then scheduled using iSLIP [8].
- *SCIA* [16]. At each input, we maintain $k = 4$ multicast queues, similar to [16]. Cells are always added to the shortest k queue. Apart from that, we set each queue's weight to be the queue length multiply by the HOL cell's age. To determine an appropriate S_m value, we iterate from 0 to 1 at an increment of 0.05 to determine the S_m that provides the best delay to both unicast and multicast cells for a given input load. Lastly, in unicast time slots, we use the oldest cell first (OCF) matching algorithm [8], and for multicast time slots, we use WBA [12]

5 Results

We now present results from our experiments on a $N \times N$ switch with uniform, non-uniform and bursty traffic. In addition, we also investigate the impact of different switch sizes.

5.1 Uniform Traffic

Our first experiment is on a 8×8 switch. We generate uniform Bernoulli traffic with a load of 0.1 to 0.55, and designate half of the traffic to be multicast. Lastly, each multicast cell has a random fanout ranging from 1 to 8.

Figures 4 and 5 show the delay incurred by unicast and multicast cells respectively. We see that iSLIP-Emulate, although low in complexity, has the highest unicast and multicast delay. SCIA and our integrated scheduler have comparable unicast and multicast delays. Note that, ours has the advantage of not requiring a tuning knob. In other words, SCIA's performance is achieved by tuning S_m iteratively. As the input load increases, SCIA's multicast delay becomes significantly higher, whereas unicast cells experience lower delays than those scheduled by our integrated scheduler. This is because SCIA has to probabilistically provide time slots to service unicast cells, which

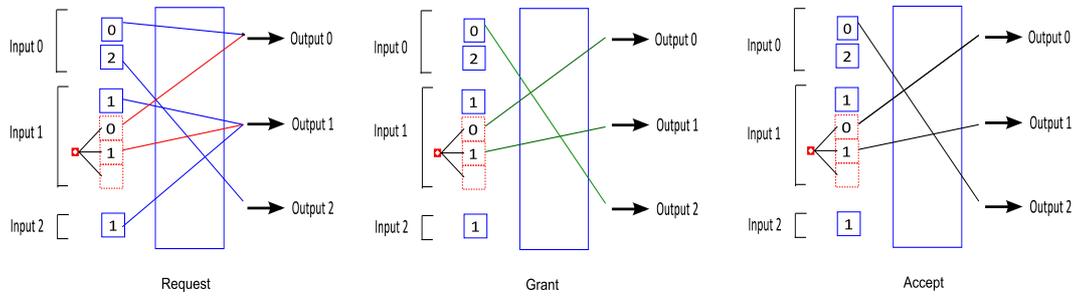


Figure 3: Unicast and multicast scheduling example.

reduces the throughput of multicast traffic, hence increasing delay significantly.

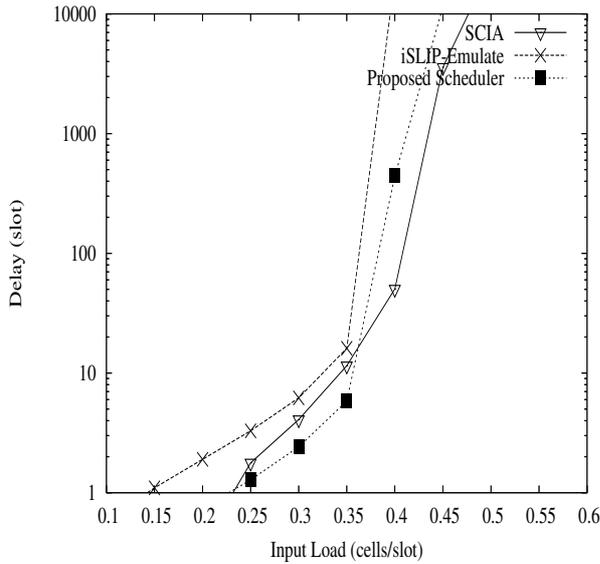


Figure 4: Average delay of unicast cells. Results are for an 8x8 switch with uniform i.i.d Bernoulli arrivals.

On the other hand, our integrated scheduler treats both unicast and multicast cells equally, which results in both traffic types experiencing similar delays. Apart from that, our scheduler utilizes the crossbar fabric's innate multicast ability when the opportunity arises, thereby increasing throughput. This is particularly critical during high loads, as it delays queue instability.

In the next experiment, we study what happens when inputs have increasing multicast cell arrivals. We fix the input load at 0.45, and increase the percentage of multicast traffic slowly from 10% to 55%. Figures 6 and 7 indicate that iSLIP-Emulate has the worst performance, and our scheduler results in both unicast and multicast cells having similar delays. When the percentage of multicast cells is at 35%, the queues in SCIA become unstable. In other words, SCIA is unable to provide sufficient scheduling opportunities to cells. This is exacerbated by the fact that SCIA probabilistically prefer unicast over multicast cells, and vice-versa. On the other hand, our scheduler ensures that the most urgent cells are transferred in a given time slot, hence it is able to delay queues instability.

5.1.1 Impact of Fanout

An important observation is the impact of multicast cells' fanout. To illustrate the detrimental effects of large fanout, we used a 3x3 switch. Input-0 has a single multicast flow that has a fixed fan-out of three, and an input load of 0.33, thereby yielding an effective load of 1.0. Other inputs have unicast flows that transmit cells uniformly across all outputs. In our experiments, we vary their load from 0.1 to 1

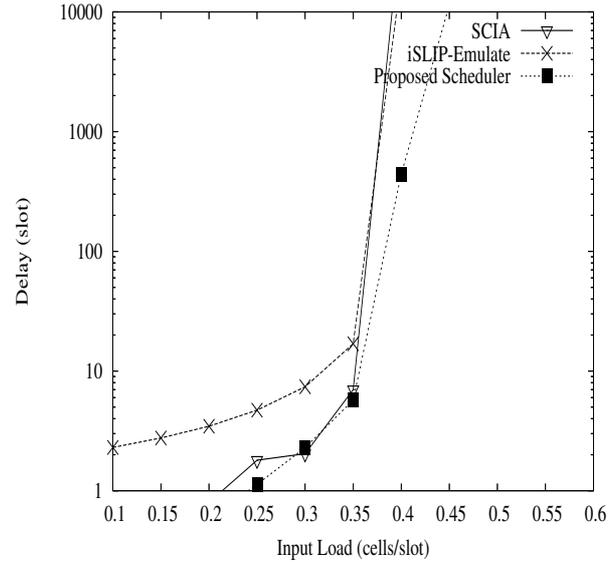


Figure 5: Average delay of multicast cells. Results are for an 8x8 switch with uniform i.i.d Bernoulli arrivals.

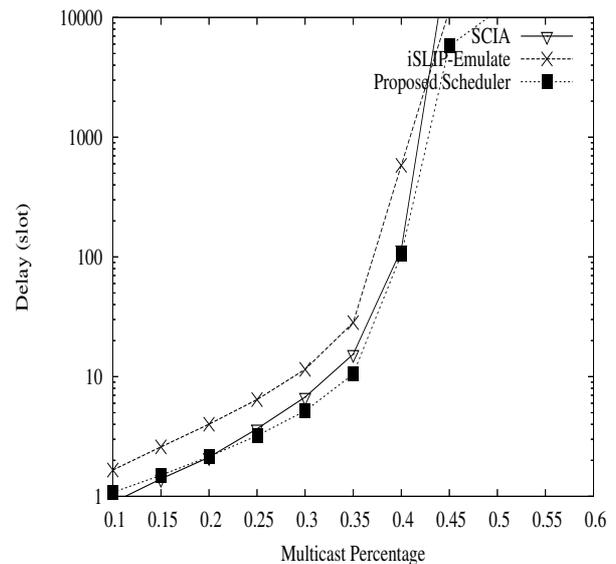


Figure 6: Average delay of unicast cells with uniform i.i.d Bernoulli arrivals.

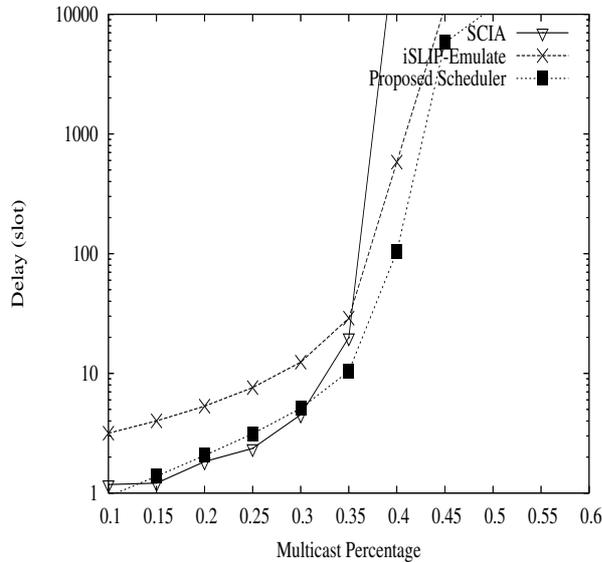


Figure 7: Average delay of multicast cells with uniform i.i.d Bernoulli arrivals.

to determine their impact on the multicast flow, and vice-versa.

Figures 8 and 9 show the delay incurred by unicast and multicast cells respectively. We see that iSLIP-Emulate has the lowest unicast delay, but has the highest multicast delay. This is due to iSLIPs inability to handle non-uniform traffic, since input-0 has a much higher load than other inputs. SCIA has the lowest multicast delay. This, however, is achieved at the expense of unicast cells. In particular, when the inputs have a load greater than 0.8, unicast cells experience high delays. We can reduce their delay by adjusting the parameter S_m , whereby we dedicate more time slots to unicast cells. Unfortunately, doing so increases the delay of multicast cells. The proposed scheduler, however, does not have the above limitations. The delay experienced by unicast cells is comparable to iSLIP-Emulate. On the other hand, even though multicast cells using our proposed scheduler have a slightly worse delay than SCIA, our scheduler does not cause severe performance degradation to unicast cells.

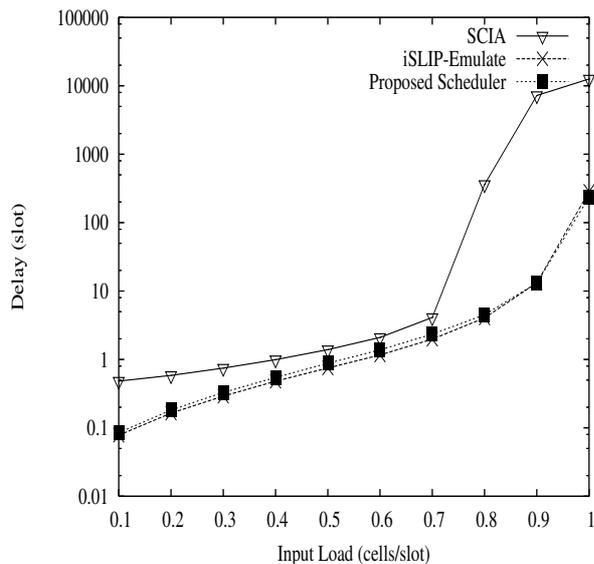


Figure 8: Average delay of unicast cells. Results are for a 3x3 switch with uniform i.i.d Bernoulli arrivals.

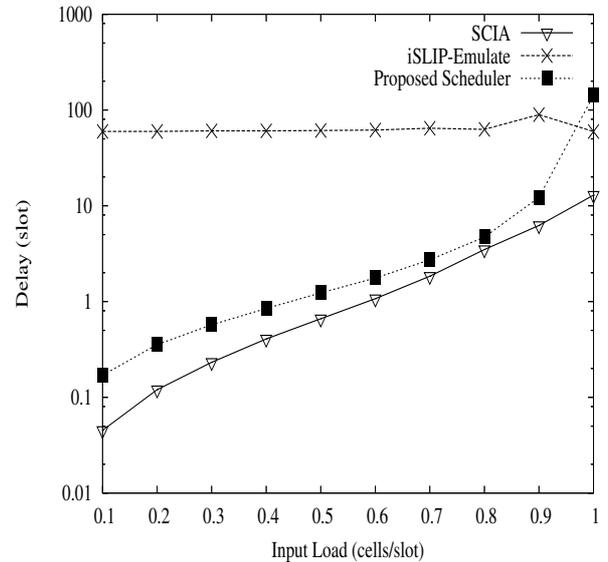


Figure 9: Average delay of multicast cells. Results are for a 3x3 switch with uniform i.i.d Bernoulli arrivals.

5.2 Non-Uniform Traffic

Using the same 8x8 switch, we change inputs' arrival to non-uniform Bernoulli traffic. Each input has a random load to each output that ranges from 0.0 to 0.1. As before, we designate half of the traffic to be multicast.

From Figures 10 and 11, we see that the proposed scheduler yields the best delay for both unicast and multicast cells. Comparatively, SCIA and iSLIP-Emulate have higher delays because both of them are known to have poor performance when traffic is non-uniform [16][8]. Intuitively, if a subset of inputs have a high unicast and multicast load, these schedulers will not consider these cells in the same round. For example, in SCIA, in a multicast time slot, it will try to maximize the number of multicast matchings without any regards to inputs with higher weighted unicast cells. In contrast, our scheduler considers both cell types in the same round, and schedules only the highest weighted cells. Moreover, it does not try to maximize the number of matchings unless multiple outputs deem a multicast cell to have the highest weight amongst all HOL cells that are destined for them.

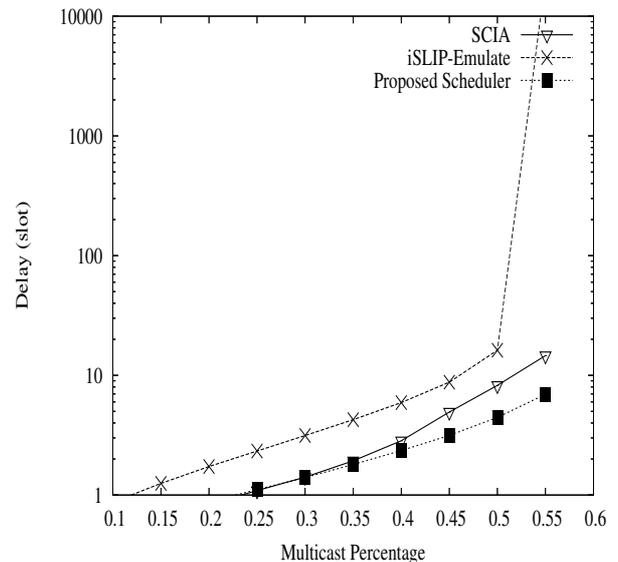


Figure 10: Average delay of unicast cells with non-uniform i.i.d Bernoulli arrivals.

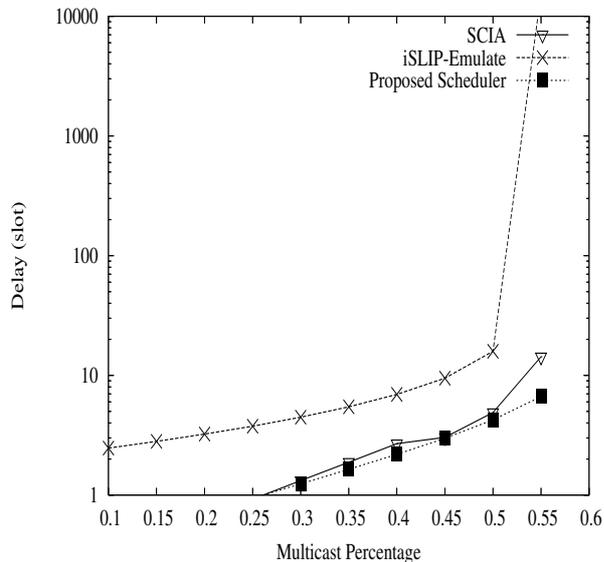


Figure 11: Average delay of multicast cells with non-uniform i.i.d Bernoulli arrivals..

5.3 Uniform Bursty Traffic

We now experiment with bursty traffic on a 8x8 switch. We start with uniform bursty traffic, where we increase the load of each input from 0.20 to 0.40 at an increment of 0.02. Moreover, we set the average burst size to 10 cells, and designate half the traffic to be multicast. Note, larger burst sizes simply cause a proportional increase in cell delay.

Figures 12 and 13 indicate that our scheduler has the lowest delay. SCIA has comparable delays, both for unicast and multicast cells, until the input load increases beyond 0.35. After such point, SCIA consistently prefers multicast over unicast cells, resulting in unicast queues becoming unstable. In comparison, the queues in our scheduler remain stable for a much higher input load, and treats both unicast and multicast cells fairly, as both cell types experience similar delays.

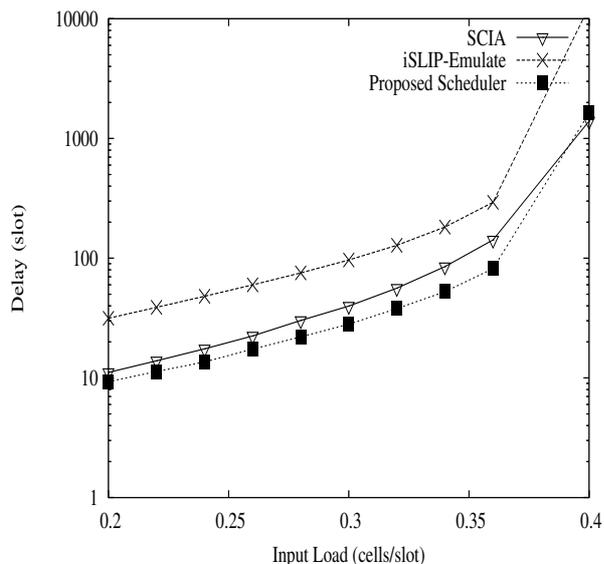


Figure 12: Average delay of unicast cells. Results are for an 8x8 switch with arrival burst length of 10 cells.

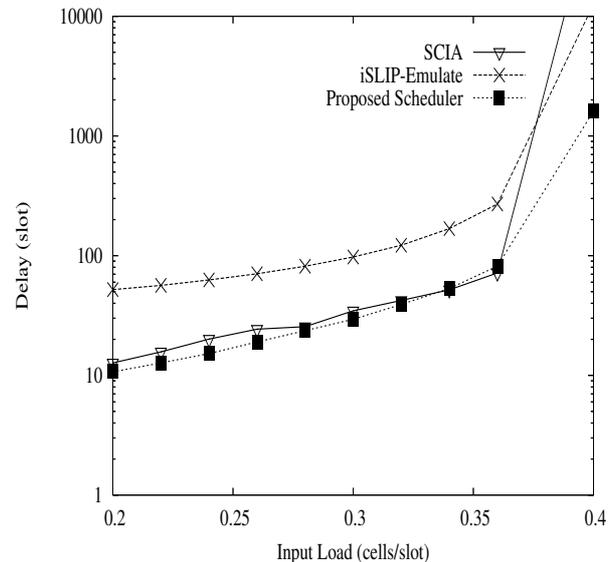


Figure 13: Average delay of multicast cells. Results are for an 8x8 switch with an arrival burst length of 10 cells.

5.4 Non-Uniform Bursty Traffic

We continue the previous experiment but with non-uniform bursty traffic. Figures 14 and 15 show the same trend as the previous experiment. As the percentage of multicast cells increases, SCIA spends more time scheduling multicast cells at the expense of unicast cells. We could easily adjust S_m to reduce the delays of unicast cells by providing more opportunities to schedule them first. In practice, however, determining the best tradeoff is difficult as different flows at different points in time will have varying delay requirements.

5.5 Throughput

A key performance metric is a scheduler's throughput. From Figure 16, we see that our scheduler achieves 100% throughput when traffic is uniform and non-uniform, a significant advantage over iSLIP-Emulate and SCIA, given that they achieve 100% throughput only when traffic is uniform.

5.6 Switch Size

Lastly, we investigate how a switch's size, i.e., the number of inputs and outputs, impact our scheduler's performance. Figure 17 shows the delays incurred by unicast cells on a 8x8, 16x16 and 32x32 switch. We omit the plot for multicast cells because the delays are similar given that our scheduler treats both traffic types fairly. We see that our scheduler's performance degrades with increasing switch sizes. In a 32x32 switch, when the input load reaches 0.12, there is a significant increase in delay. This is due to multicast cell's large fanout. In fact, a multicast cell can have up to 32 outputs! We have also experimented with smaller fanout sizes. Our results indicate delays of cells for all switch sizes increase proportionally to the load and number of outputs.

6 Discussion

Our integrated scheduler performs better than existing approaches because of the following reasons:

Firstly, it considers the weight of both unicast and multicast cells simultaneously. This is in contrast to existing schemes that have thus far considered both traffic types separately. From our experimentations, we found this to

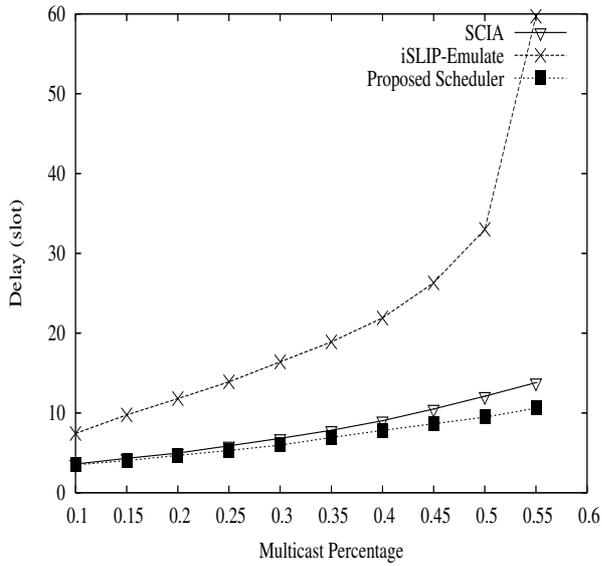


Figure 14: Average delay of unicast cells with non-uniform bursty traffic.

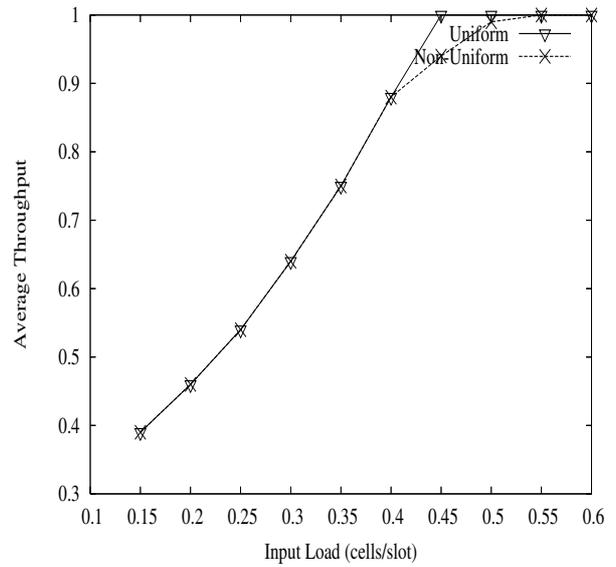


Figure 16: Average Throughput. Results are for an 8x8 switch with uniform or non-uniform traffic.

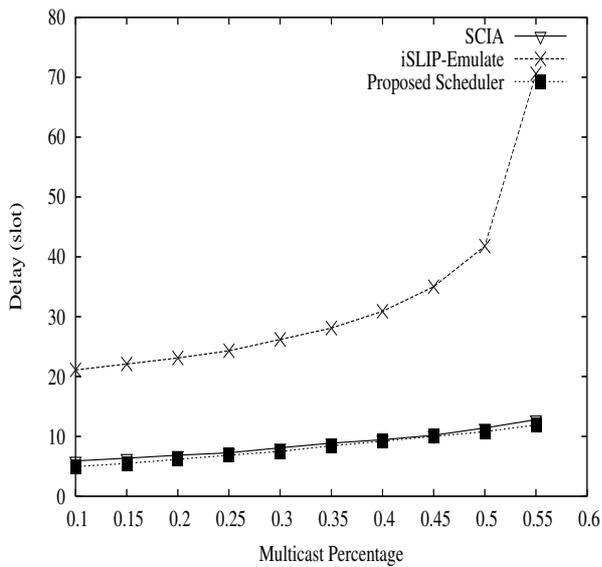


Figure 15: Average delay of multicast cells with non-uniform bursty traffic.

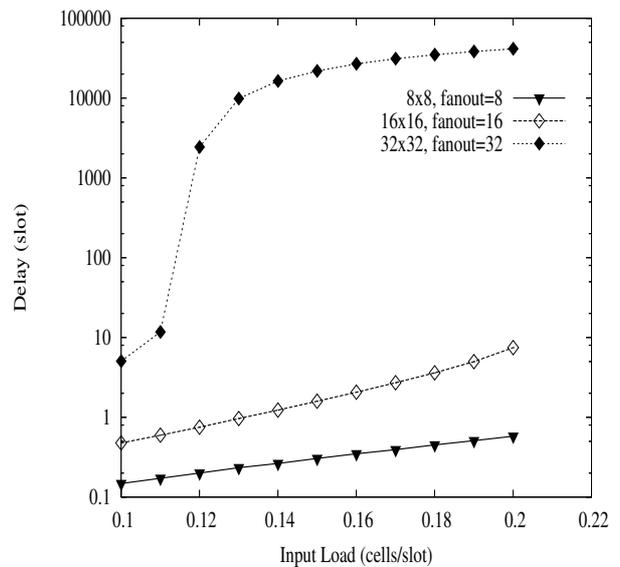


Figure 17: Average delay of unicast cells in different switch sizes; all inputs have uniform i.i.d Bernoulli traffic.

be critical when both unicast and multicast cells are competing for the same output. In SCIA, a tradeoff will have to be made as to which cell type should receive service. Moreover, this decision is not deterministic, as S_m designates a slot to be unicast/multicast probabilistically. Our scheduler, however, bases its decisions on cells' weight. Thereby, as our results showed, both unicast and multicast cells have the same delay.

Secondly, it uses staging buffers to address the multicast cell HOL blocking problem. Unlike existing works that utilize $k < 2^N - 1$ queues, our approach utilizes much less memory. The tradeoff, however, is extra computations involving TCAM lookups. Fortunately, these computations can be pipelined and is not critical to the scheduling process.

Thirdly, it utilizes the crossbar fabric's innate multicast ability opportunistically. Prior works such as [16] and [2] establish multicast matchings without considering the weight of unicast cells. Our scheduler, however, looks at both cell types and only enables the crossbar's multicast capability when multiple outputs deem a multicast cell to be the highest weighted cell in a given round. This is particularly advantageous as it increases a switch's throughput.

Fourthly, it does not use a tuning knob, e.g., S_m . From our results, we see that when inputs have low loads, our scheduler has comparable performance to SCIA [16]. However, we need to take into consideration that SCIA's performance is achieved by adjusting S_m iteratively. Our scheduler, however, does not have this limitation. Hence, it is able to operate with changing traffic conditions.

Lastly, it supports both uniform and non-uniform traffic. Existing approaches, such as iSLIP-Emulate [11] and SCIA [16], are designed for uniform traffic. Our scheduler, however, works well in non-uniform traffic scenarios. Specifically, in each round, it considers cells' weight, thereby allowing it to adapt to input loads that vary over time.

7 Conclusions

We have presented a novel scheduler capable of scheduling both unicast and multicast cells simultaneously. From our extensive simulation studies, our scheduler demonstrates comparable or better performance than existing schemes during low loads, and superior performance during high loads. More importantly, our scheduler is adaptive to changing traffic conditions, thereby making it suitable for both uniform and non-uniform traffic conditions.

References

- [1] Internet 2 multicast applications. <http://multicast.internet2.edu/wg-multicast-applications.shtml>.
- [2] M. Andrews, S. Khanna, and K. Kumaran. Integrated scheduling of unicast and multicast traffic in an input-queued switch. In *IEEE Infocom*, New York, USA, June 1999.
- [3] A. Bianco, P. Giaccone, E. Leonardi, F. Neri, and C. Piglion. On the number of input queues to efficiently support multicast traffic in input queued switches. In *IEEE Workshop on High Performance Switching and Routing*, Torino, Italy, June 2003.
- [4] M. Carol, M. Hluchyj, and S. Morgan. Input versus output queueing on a space division switch. *IEEE Transactions on Communications*, 35:1347–1356, Jan. 1988.
- [5] J. Chao and B. Liu. *High Performance Switches and Routers*. Wiley-Interscience, 2007.
- [6] M. A. Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri. Multicast traffic in input-queued switches: Optimal scheduling and maximum throughput. *IEEE Transactions on Networking*, 11(3):465–477, June 2003.
- [7] N. McKeown. SIM: A fixed length packet simulator. <http://klamath.stanford.edu/tools/SIM/>.
- [8] N. McKeown. *Scheduling Algorithms for Input-Queued Cell Switches*. PhD thesis, Department of Electrical Engineering, University of California Berkeley, may 1995.
- [9] N. McKeown. A fast switched backplane for a gigabit switched router. *Business Communications Review*, 27(12):1020–1030, 1997.
- [10] N. McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE Trans. on Networking*, 7(2):188–198, Apr. 1999.
- [11] C. Minkenberg. Integrating unicast and multicast traffic scheduling in a combined input and output queued packet switching system. In *IEEE ICCCN*, pages 127–134, Las Vegas, USA, Oct. 2000.
- [12] B. Prabhakar, N. McKeown, and R. Ahuja. Multicast scheduling for input-queued switches. *IEEE Journal on Selected Areas in Communications*, 15(5):855–866, June 1997.
- [13] E. Schiattarella and C. Minkenberg. Fair integrated scheduling of unicast and multicast traffic in an input-queued switch. In *IEEE ICC*, Istanbul, Turkey, June 2006.
- [14] K. Schultz and P. Gulak. CAM-based single-chip shared buffer ATM switch. In *IEEE Conference on Communications*, New Orleans, USA, May 1994.
- [15] M. Song and W. Zhu. Throughput analysis for multicast switches with multiple input queues. *IEEE Communications Letters*, 8(7):479–481, 2004.
- [16] W. Zhu and M. Song. Integration of unicast and multicast scheduling in input-queued packet switches. *Elsevier Computer Networks*, 50(8):667–687, Aug. 2006.

A Dynamic, Decentralised Search Algorithm for Efficient Data Retrieval in a Distributed Tuple Space

Alistair Atkinson

Eastern Institute of Technology
Hawke's Bay, New Zealand
Private Bag 1201, Hawke's Bay Mail Centre, Napier 4142
Email: aatkinson@eit.ac.nz

Abstract

This paper presents an algorithm which may be used to efficiently search for and retrieve tuples in a distributed tuple space. The algorithm, a core part of the Tupleware system, is based on the success or failure of previous tuple requests to remote nodes in the system, and this data is used to determine the relative probability of particular remote nodes being able to fulfil subsequent future requests. The logic of this algorithm is distributed and decentralised: each node dynamically calculates its relationship with other nodes at runtime. The behaviour of the algorithm using two applications is analysed, and shows significant improvement in terms of efficiency and performance compared to comparable tuple space implementations.

Keywords: tuple space, data retrieval, locality, distributed computing.

1 Introduction

This paper describes a search algorithm for efficiently retrieving tuples on a cluster-based distributed tuple space. This algorithm forms one of the core parts of the Tupleware cluster middleware, a high-level description of which can be found in [3].

The Tupleware system was implemented with the goal of achieving a scalable platform for the implementation and execution of parallel array-based applications, whilst maintaining the simplicity and transparency of the original tuple space paradigm [7].

In order to achieve scalability, it was decided to use a decentralised approach, and to distribute the tuple space across nodes in the cluster. It followed, then, that it would be necessary to store and search for tuples in the most efficient manner possible, and it is for this reason that the search algorithm presented in this paper was developed.

To evaluate the performance and scalability of Tupleware (and its search algorithm) the performance of two non-trivial applications are presented. Ease of programmability is discussed in the previous paper cited above.

The contribution of this research is that it provides an investigation into a concrete implementation of distributed tuple space using non-trivial data-parallel applications. This area of research, while studied previously as we will see in Section 3, often focusses on theoretical models which lack a concrete implementation, and others seek to provide a more general-purpose platform. This research project has focussed on a particular class of applications in order to exploit their common characteristics,

which has informed the development of the search techniques being presented.

2 Motivation

Tuple spaces, first introduced by Gelernter's Linda coordination language [7], are recognised as offering many advantages over the more common message-passing model as a distributed computing paradigm. These advantages include: a decoupling of the computations and coordination parts of a parallel program, both temporal and geographic distribution, loosely-coupled interaction between processes, and a higher level of abstraction which unburdens the programmer from needing to explicitly deal with lower-level details of inter-process communication.

However, the adaption of the tuple space model to a distributed environment poses some additional challenges compared to its implementation on multiprocessor computers. Namely, the increase in latency and relatively restricted network data transfer rates cause any operations involving network communication to become relatively expensive, and also distributed systems generally need to be able to scale to large numbers of nodes. This means that the available network bandwidth must be used as efficiently as possible, and thus it is necessary to minimise the number of communication events in a given system in order to achieve this.

These limitations were illustrated in [18], which described the scalability of various tuple space implementations under varying loads, and showed the scalability of the included systems to be relatively poor. A study presented in [15] also outlined the issues involved with efficiently implementing a tuple space in a distributed environment, and presented a performance evaluation of an unmodified JavaSpaces system using applications with similar characteristics to those presented in this paper. The results of this evaluation showed the limitations of a centralised tuple space for tightly-coupled applications, and its more commendable performance for loosely-coupled replicated-worker style applications.

These factors are what motivated this particular research, which addresses these issues, and proposes a search algorithm suitable for use in a distributed tuple space which can be utilised for array-based parallel applications.

3 Previous Work

Some of the more notable systems which feature distributed or multiple tuple spaces are briefly described in this section. The systems included can be contrasted in terms of the transparency of their distribution, the logical integration of the tuple space(s), and whether or not the system logic is centralised or decentralised.

Copyright ©2010, Australian Computer Society, Inc. This paper appeared at the Eighth Australasian Symposium on Parallel and Distributed Computing (AusPDC2010), Brisbane, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 107, Jinjun Chen and Rajiv Ranjan, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

3.1 Multiple Tuple Spaces in Linda

Multiple Tuple Space Linda [14] (usually abbreviated to MTS-Linda) was one of the earliest attempts to add multiple tuple spaces to the original Linda model. MTS-Linda incorporates tuple spaces which are treated as first-class objects, and can be manipulated by the programmer to suit an application's requirements. The use of multiple tuple spaces allowed data (represented as passive tuples), and processes (represented as active tuples), to be grouped and manipulated as a whole. As tuple spaces are treated as first class objects, each tuple space is simply conceptualised as a "local data structure within a process" [14], which goes some way towards raising the level of transparency of the system's distribution.

Tuples which reside in other (non-local) tuple spaces may also be accessed, provided they are within the "context tuple space" of the process making the access request. That is, multiple tuple spaces may belong to the same context, and processes may opt to retrieve tuples from either its local tuple space, or from a tuple space contained in the same context. In this way, multiple tuple spaces are added in a hierarchical manner, rather than the flat, or disjoint way they have been incorporated in some other systems.

Another attempt at implementing multiple tuple spaces for Linda was by Rowstron & Wood [16], who adapted the Linda model to networks of heterogeneous workstations. This system did not propose a new way of adding multiple tuple spaces to the system, but simply assumed that they existed. The main contribution this system made was the addition of new tuple space access primitives, namely bulk retrieval operations `collect()` and `copy-collect()`. The former operation moves a set of matching tuples from one tuple space to another, and the latter performs a similar function, except matching tuples are copied from one tuple space to another [17]. This implementation classifies tuple space as either local or remote, the main difference being that tuples stored in a local tuple space are not accessible by remote nodes in the system, whereas those stored in a remote tuple space do not have this restriction. Further, local tuples are stored locally, in the local processes address space, whereas remote tuples are stored on remote *tuple space servers*, which generally reside on separate, dedicated nodes on the network. The decision as to whether a given tuple is classified as being local or remote is performed dynamically at runtime by the system kernel.

The bulk operations allowed the movement of multiple tuples using only a single operation, whereas in the original Linda model this would have required multiple invocations of the tuple spaces access operations. This factor, along with the optimisation of the locality of stored tuples, allowed the system to make more efficient use of the network, and to realise some significant performance gains compared to traditional implementations [16].

3.2 SwarmLinda

In terms of algorithmic approaches to tuple search and retrieval in systems with multiple or distributed tuple spaces, there are a small number of proposed systems at the time of writing that have this as their focus. The most notable of these is arguably the SwarmLinda distributed tuple space described in [11].

SwarmLinda employs a tuple storage and retrieval algorithm inspired by the collective intelligence displayed by swarms of ants. SwarmLinda is characterised by agents (in this case, ants) acting individually, but whose individual actions combine to exhibit a collective intelligence. These agents "act extremely decentralised" and perform their actions "by making purely local decisions and by taking actions that require only a few computations" [5].

The collectively intelligent behaviour displayed by these ant swarms relates to the locality of tuple storage, and efficient tuple retrieval from the network. In short,

when a new tuple is produced, it will be stored by one of the 'ant' agents on a node which stores tuples with the most similar characteristics to the new tuple. Tuples with a particular characteristic emit a 'scent', and this scent is used by the agents when they need to retrieve a tuple. The characteristics of the tuple to be retrieved, along with the scent being emitted, is used to guide the search of the agent. It is argued that a SwarmLinda system will dynamically adapt itself to the characteristics of the tuples being stored, so that tuple retrieval operations will tend towards optimal over time. However, at the time of writing, these ideas have not been implemented in a concrete system, and as such no performance data are available to determine their effectiveness.

3.3 Scope

Scope [12] is a formal model for the addition of multiple tuple spaces to Linda-like systems. It aims to address the scalability problem of Linda, and also to increase the expressiveness of Linda-like operations so as to enable operations such as transactions, and prevent semantic limitations such as the multiple-read problem. Most relevant to the research presented in this paper, however, is the generalised way in which Scope handles the issue of multiple tuple spaces, in particular its idea of "overlapping" tuple spaces.

Multiple spaces have traditionally been added to tuple space systems in one of two ways: by nesting spaces hierarchically, as in MTS-Linda, or by simply adding disjoint spaces which have no logical relationship, as in JavaSpaces [12].

Scope presents a generalised approach to the addition of multiple spaces, introducing the idea of overlapping tuple spaces. This allows some parts of each space to be shared, and other parts to be separate. In concrete terms, tuples are able to belong to more than one space at a time. Essentially, each "portion" of tuple space is represented by a named scope, and these portions can be combined and arranged based on defined scope operations. These operations are based on the set operations union, complement and intersection, and can be used to define tuple membership to one more more scopes. The expressiveness of Scope allows it to implement hierarchical and disjoint tuple spaces in addition to overlapping spaces.

A concrete implementation of Scope is presented in [13]. However, no performance results are available for any Scope-based implementation, and no subsequent research seems to have been done at the time of writing.

3.4 JavaSpaces

JavaSpaces [10] is an implementation of the spaces paradigm from Sun Microsystems. Specifically, it is a service which forms part of the Jini distributed software architecture. It provides a stand-alone object space, called a *JavaSpace*.

The system may have more than one space, however each space is a separate entity and their respective roles in the system are not coordinated. Each application must contain the logic for utilising the available JavaSpaces infrastructure.

Like most derivative implementations of the tuple space model, JavaSpaces is an effective platform for implementing a range of distributed applications and utilising common design patterns. In particular, it has been shown in [2] to be ideally suited to the Master/Worker style of parallelism, particularly coarse-grained parallel applications. For applications which are more fine-grained or tightly-coupled, JavaSpaces can experience scalability problems due to the increased communication demands inherent in these applications (see, for example [18]).

4 Tupleware Overview

Tupleware is a library and runtime system which provides a distributed tuple space platform for computing clusters. It is aimed specifically at array-based numerical and/or scientific applications, which exhibit common characteristics that may be exploited in order to optimise the communication patterns between cluster nodes.

Tupleware has previously been described in [3], so the entire system will not be covered in great detail again here. Instead, the description will be restricted to a high-level overview of the operation of the main components of the system only, in order to inform the discussion of the performance results. For a more detailed description refer to the previous publication referenced above.

4.1 System Architecture

A complete Tupleware system consists of a collection of nodes, each of which hosts its own local partition of the tuple space. These local partitions, combined, constitute the whole (distributed) tuple space.

The main components of the Tupleware architecture consist of the following: a runtime system, a tuple space service, and tuple space stub objects. These components form a layered architecture upon which an application module can execute. An example Tupleware system consisting of two nodes is illustrated in Figure 1.

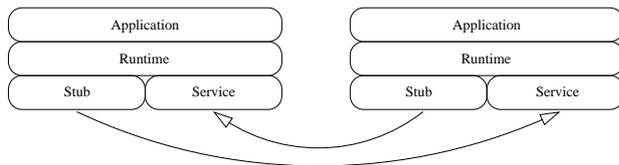


Figure 1: Architecture of a two-node Tupleware system.

4.2 System Components

The main components are briefly described as follows:

4.2.1 Tuples & Templates

The fundamental data object in a tuple space system is a tuple, which are used to encapsulate one or more data objects. A tuple has one or more fields each of which contain a value. Fields should not contain any **null** values, and tuples are treated as immutable objects.

Templates are used to perform content-based associative lookup on tuples. Templates are similar to tuples in that it encapsulates a set of data fields. However, unlike a tuple, some (or all) of these fields may be assigned **null** values, denoting wildcards which may match against any value during associative lookup. Associative lookup involves the use of the `matches()` method, which determines whether a template matches a given tuple. The matching function has the same semantics as the original Linda: a template must be an equivalent length, and its specified values must be equal to a given tuple in order to positively match.

4.2.2 Local Tuple Space

A local tuple space provides the basic functionality required for tuple storage and lookup on a single node. The local space maintains all of the stored tuples on node, and is searchable by a node's service in response to tuple requests from remote nodes.

The local tuple space is thread-safe, and, in most instances, uses the first three fields of a tuple as the key to a hash table which references the tuple data itself. The

reasoning behind this was that in almost all applications relevant to those targeted by Tupleware, the first three elements are always those used to identify and index the array, and, in the case of the ocean model, more than one iteration of previous array values will need to be stored (usually two, and sometimes three).

An example of this arrangement is shown in Figure 2.

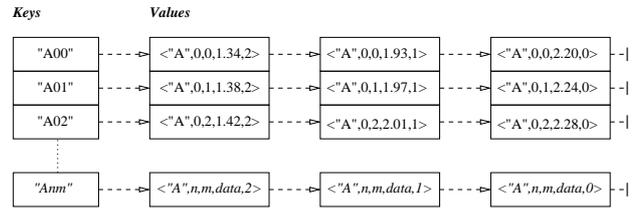


Figure 2: Local tuple storage using a hash table.

4.2.3 Inter-node Communication

Communication between nodes is carried between the stub and service components of the system. All communication instances are initiated by the stub objects, which send requests for tuples across the network to the service running on a remote node. This service provides the means by which remote nodes may search other nodes' local tuple spaces, and always answer queries directly rather than forward unfulfillable requests.

4.2.4 Runtime System

The Tupleware runtime system contains the core system logic, and control the operation of the lower level components such as stub objects and each node's local tuple space. The runtime system initiates and controls the search and retrieval of remote tuples, using the search function presented in the following section. Each nodes' runtime system maintains a collection of stub objects, each of which corresponds to a remote node in the system.

Finally, the runtime system presents an API for use by the top level application layer. It is the only interface into the Tupleware system for an application, and is largely responsible for maintaining the transparency of the tuple space's distribution on the cluster.

4.2.5 Application Processes

At the highest layer, application processes implement the application logic which in turn interfaces with the underlying runtime system. Applications have a reasonably transparent interface to the distributed Tupleware system, and are able to make use of the standard Linda-style operations.

5 Search Algorithm

5.1 Overview

The principle behind the search algorithm is to minimise the number of communication instances required to retrieve a tuple by targeting retrieval requests to those nodes which have the highest probability of being able to satisfy the request, based on the success of previous requests. This technique was adopted due to the nature of the applications at which Tupleware is targeted for use. These are generally array-based applications in which the array is decomposed into individual regions, and each region is processed in parallel.

The characteristics of applications such as these is that any communication between processes is going to tend

to occur between those processes which are processing "neighbouring" regions of the array, whereas nodes processing unrelated regions of an array are going to tend to communicate very rarely, if at all. It is these observed characteristics that we wish to be reflected in the tuple search patterns carried out by the runtime, and the search function provides the platform for achieving this.

5.2 Search Algorithm Operation

An executing Tupleware system consist of individual nodes, each with its own runtime system, each of which will need to communicate with a subset of all nodes in the system almost exclusively, and rarely if at all with all other nodes. These groupings, or clusters, of nodes will emerge quickly during the execution of an application as each individual runtime system dynamically adapts to the patterns of communication instances it is tasked with carrying out.

Underpinning the operation of the search algorithm is a *success factor* which is associated with each tuple space stub object maintained by the runtime system. The success factor is a numerical value between zero and one, and is used to denote the likelihood of the tuple space service associated with a given tuple space stub being able to fulfil a request for a tuple. A higher success factor represents that there is a greater chance of success, and vice versa.

At the beginning of an application's execution, each stub has a success factor of 0.5 as there are no previous requests from which to calculate another value. A value of 0.5 is meant to represent an intermediate chance of success. Due to all stubs starting with an equal success factor, the initial requests made are random, however the success factor will be recalculated based on the success or failure of these requests, and quite quickly a distinct ordering, or ranking, emerges which can be used to prioritise subsequent requests.

The recalculation of the success factor occurs every time a stub is used to perform a request, and is based on the following equation:

$$S = \begin{cases} S + (1 - S) \times A & \text{Success} \\ S - S \times A & \text{Failure} \end{cases}$$

where:

- S is the success factor, and
- A is the adjustment factor.

The adjustment factor is a floating point value between zero and one used to specify by how much the success factor should be adjusted each time it is recalculated. This value will determine how quickly the success factor moves towards either one or zero, or in other words, by how representative a successful request is in terms of the prioritisation of subsequent requests.

This value should be chosen based on the application and the number of processes in a system. If there is a weak relationship between the data being computed on each process, then each process may ultimately end up needing to communicate with a relatively large number of other processes. In cases such as this a small adjustment factor should be used, as one successful request to a remote tuple space does not imply that there is a much greater probability of success for future requests. However, if there is a tight relationship between the data segments being computed by each process, then it follows that these processes will likely communicate very frequently, and that a successful request should have a higher bearing on the probability of success for subsequent requests.

In practice, an adjustment factor of 0.2 was used as it reflects the characteristics of these particular applications. An adjustment factor of greater than 0.5 would reflect a

fairly volatile system with a very weak relationship between processes, where a value of 0.1 or 0.2 would represent a more stable relationship.

An application performs each iteration of its processing, the success factor will be recalculated, and over a relatively small number of iterations, the success factor associated with a node's neighbouring nodes will be greater than non-neighbouring nodes. An example of this is illustrated in Figure 3.

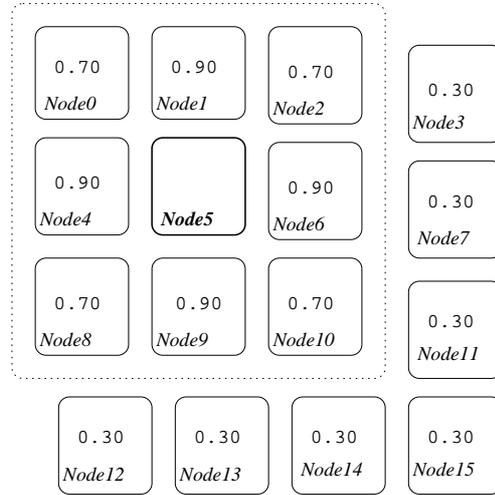


Figure 3: The success factor associated with a node's neighbouring nodes.

5.3 Benefits of the search algorithm

The search algorithm being presented here can be contrasted with the one used previously, which relied on a most-recently-successful approach to order tuple requests to remote nodes. The problem with this approach was that, while it was more effective than a blind search, it took into account only the success of the immediately previous search.

This was a non-optimal solution for the types of application being targeted by this system, especially in the case of the ocean model, as quite often data retrieval will involve several requests to several neighbouring nodes (one for each boundary being updated). In these cases a null response from a remote node does not necessarily mean that the request cannot be fulfilled by the said node in the future, but rather often it is that the required tuple simply has not yet been produced at the current point in time. However, the result would be, using the most-recently-successful approach, for this particular remote node to be treated as though is should be given a much lesser precedence for subsequent searches, which is not necessarily the desired outcome.

On the other hand, the search algorithm being presented here imposes much less drastic modification to search precedence, as it takes into account all historical retrieval requests (with a greater weight to those carried out more recently). The search precedence given by remote nodes' associated success factors provides a more accurate reflection of the actual probability of a successful tuple retrieval.

Another benefit implicit to this search algorithm is its decentralised nature: updates to the success factor associated with each remote node is performed by each individual runtime without requiring any sharing of global state information between nodes. Put simply, each node maintains its own unique "view" of the cluster's tuple storage, based on its own search history. This eliminates any overhead associated with the transmission of global state information.

Finally, the search algorithm executed dynamically and allows for changes and reconfigurations to tuple storage on the cluster. If the storage characteristics change, the groupings illustrated in Figure 3 will alter to reflect this.

5.4 Summary

In this section we have presented a dynamic, decentralised search algorithm which is used to guide searches for tuples stored on remote nodes of a cluster. The algorithm has been contrasted with the one previously used by the Tupleware system, and its benefits discussed.

In the following sections we present the performance characteristics of the system using two non-trivial applications.

6 Applications

6.1 Overview

Two applications were used to test the behaviour of Tupleware: a 2-D ocean modelling application, and a parallel sorting application based on a modified quicksort. These applications were chosen for their contrasting characteristics, namely their different levels of granularity and communication characteristics. However, both of these applications involve processing segments of an array in parallel, and both are able to benefit from the search algorithm being presented in this paper.

Each application is briefly described below.

6.2 Ocean Model

The ocean model is a two-dimensional simulation of an enclosed body of water. The model calculates the water current velocity and surface elevation based on a given wind velocity and bathymetry.

The body of water is represented by the application as a 2-D grid, and each cell in the grid represents a single grid point. Grid points each individually store descriptive data, including the depth of the water at that point, along with the surface elevation and current velocity. Wind velocity is assumed to be constant across the grid. The variables stored in each grid point are staggered in such a way that the u and v variables are associated, respectively, with the x -axis and y -axis edges of each grid point. The eta variable is representative of the centre point of each grid point.

When executed, the model iterates through a specified number of time-steps; and at each time-step, the surface elevation and current velocity values of each grid point are recalculated based on the values stored at neighbouring grid points. This process continues for the specified number of time-steps, at which point the model should be in a steady-state and thus finished.

The model is parallelised through domain decomposition of the grid, which splits the grid into a number separate *panels*, up to the number of nodes available for processing. Each panel is assigned to a specific node, whose responsibility it is to perform the processing on the panel. As each panel represents only part of the complete grid, at each iteration it is necessary for the boundary values of each panel to be retrieved from neighbouring panels. This process is illustrated by Figure 4, which shows a 9x9 grid which has been decomposed into three panels. Each 9x3 panel has a halo region, represented by the shaded cells, whose values are updated after each iteration of the model. The arrows between neighbouring halo region cells represent the communication instances which are involved in each boundary update.

6.3 Quicksort

Quicksort [9] is a widely used sorting algorithm with an average case execution time of $O(n \log n)$. It is an effi-

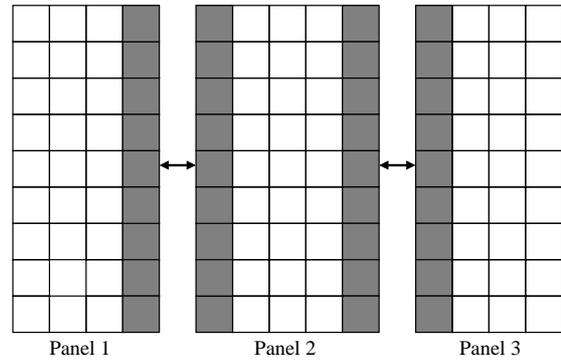


Figure 4: Updating panel boundary values.

cient general-purpose sorting algorithm which rarely exhibits its worst-case execution time.

There are several characteristics of the quicksort algorithm which led to it being used in the performance evaluation which follows. Firstly, parallelisation of quicksort (and modified and implemented here) is reasonably straightforward, and produces processing tasks which are loosely-coupled and have only moderate data dependencies. Secondly, by modifying the quicksort algorithm so that partitioning ends when an array segment length reaches a certain predefined threshold, it is possible to adjust the granularity of the parallelism exhibited by the sorting algorithm. This feature is useful as it allows us to evaluate the performance of the system with various levels of communication frequency.

The algorithm used to evaluate the system in this paper is a modified version of quicksort. As described above, unsorted arrays are repeatedly partitioned until their length is less than or equal to a predetermined threshold value. At this point, partitioning ends and the remaining unsorted array segment is sorted using some other sequential algorithm; in this case, insertion sort [6].

The ability to adjust the granularity of the application in this manner is useful, as it assists in determining the true scalability of the system and at which point the communication requirements begin to outweigh the benefits of the distribution of the application.

7 Performance Evaluation & Analysis

7.1 Metrics

The metrics used to evaluate the behaviour and performance of Tupleware as presented in this paper are the *runtimes*, which are the wall-clock timings of the execution of various part of the application, and from which we can derive the *speedup* delivered by the Tupleware system. Speedup as used here does not differ from its standard usage in this area, that being the ratio of sequential runtime to parallel runtime [4, p. 74].

From these metrics, we can then make assertions regarding the *scalability* of the system as a whole. There are two aspects of scalability which will be outlined: scalability in terms of the number of processors, and scalability in terms of the problem size. The former is directly related to speedup and Amdahl's Law [1]; if a parallel program tends towards a speedup of N when executed on N processes, then it is said to be scalable. The latter aspect, scalability in terms of problem size, is concerned with how effectively the problem can be split amongst the available processors. That is, if a parallel system can execute a problem of size S in a time of T , then if the size of S doubles we wish the execution time to be no greater than $2 \cdot T$. If doubling the problem size results in significantly more than doubling the total runtime, the system would not be scalable in terms of problem size.

7.2 Execution Environment

The performance testing was conducted on a sixteen-node cluster, with each node consisting of a Pentium 4 (3GHz) processor with 1GB of memory, and running Ubuntu Linux 8.04 (kernel 2.6) along with Java 6 update 10. Nodes were connected by a 100Mbps Ethernet network. Performance profiling was carried out using the *Clarkware Profiler* (Clark, 2008), which is able to measure the total and per-iteration runtimes (wall-clock time) between specified points in a program. Each process was executed with the following Java command-line options: `-Xms512M` to set an initial heaps size of 512MB, and `-Xmx2048M` for a maximum heap size of 2GB.

7.3 Ocean Model

The ocean model was tested on a varying number of nodes, from one through to sixteen. When discussing the number of nodes taking part in the system, we are specifying the number of worker nodes. In all Tupleware application these always exists exactly one master process in addition to one or more of the worker processes.

The size of the grid was also varied for experimental purposes, ranging from 1200x1200 through to 2400x2400 in increments of 200x200. This gives a substantial range of grid sizes, keeping in mind that the total number of grid points increases exponentially as the grid grows larger; this is illustrated in Table 1, which also details the amount of raw data stored in each size grid. A 2400x2400 grid was the largest possible for execution before some nodes, particularly the master node, began to use virtual memory, which began to artificially effect the behaviour of the system.

| Grid Size | Grid Points (million) | Data (MB) |
|-----------|-----------------------|-----------|
| 1200x1200 | 1.44 | 87.9 |
| 1400x1400 | 1.96 | 119.6 |
| 1600x1600 | 2.56 | 156.3 |
| 1800x1800 | 3.24 | 197.8 |
| 2000x2000 | 4.00 | 244.1 |
| 2200x2200 | 4.84 | 295.4 |
| 2400x2400 | 5.76 | 351.6 |

Table 1: Total grid points and data size.

The number of timesteps completed by the model remained constant at fifty; this gave the system sufficient parallel execution in order for a rigorous performance evaluation to be performed.

7.3.1 Results

The overall speedup of the ocean model application is shown in Figure 5.

This gives us a high high-level overview of the behaviour of the system, however it would be useful to separate the performance of the systems in terms of its sequential and parallel execution. The runtimes of each of these execution phases are illustrated in Figures 6 and 7.

As we can see, an increase in the number of nodes substantially decreases the time taken for the initial application data to be delivered to each worker node, and for the final processed panels to be returned to the master node. This would most likely be due to the use of a switched network, which would allow data to be sent simultaneously to each worker node over each point-to-point circuit. Increasing the number of worker nodes also reduces the size of data being transferred, as the width of each panel would be smaller.

These figures also show that in the cases of fourteen and sixteen nodes, the decrease in sequential runtime becomes negligible, or in some cases increases. This is

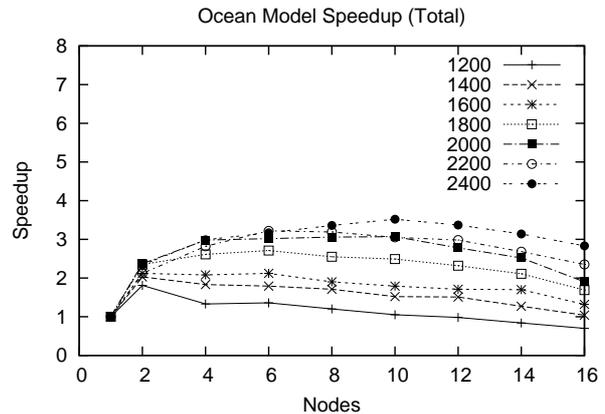


Figure 5: Ocean model's overall speedup.

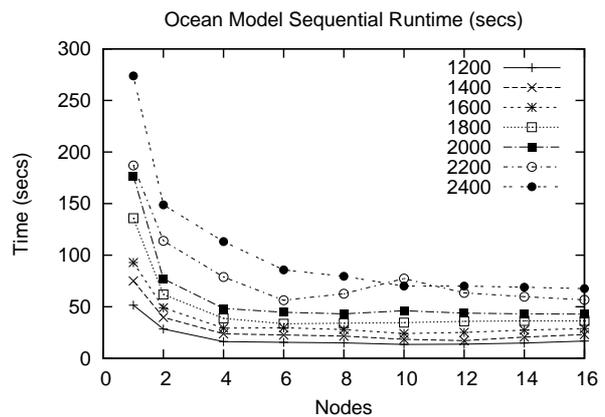


Figure 6: Sequential runtimes (secs) for varying number of nodes in the ocean model.

likely due to the reliance on the master node, which is responsible for either transmitting or receiving all of the data. If we extrapolate these results to a larger number of nodes, then it is likely that these times would continue to slightly increase. However, quite plainly there are significant speedup benefits attained by adding extra nodes to the system in terms of these sequential runtimes.

Following on from the sequential runtime of the application, we can turn our attention to the behaviour during parallel execution. During this phase of execution, the worker nodes behave in a completely decentralised way, and communicate directly in order to share boundary values at each timestep.

The first conclusion which can be drawn from this result is that increasing the number of nodes decreases the width of each panel, resulting in fewer grid points which need to be computed, and hence less time spent performing processing. However, with the size of each panel's boundary region remaining the same, the ratio between computation and communication inevitably becomes smaller.

Secondly, an increase in the number of nodes also increases the likelihood that some required boundary values will not be available between timesteps, resulting in additional time a node must spend searching for or waiting for the values to become available. These factors combine to produce a disappointing level of efficiency during the parallel phase of execution.

However, it can also be seen that an increase in grid size generally results in increased efficiency, something particularly apparent for systems with six or less nodes. This is due to an increased grid size resulting in a linear increase in boundary size along with an exponential

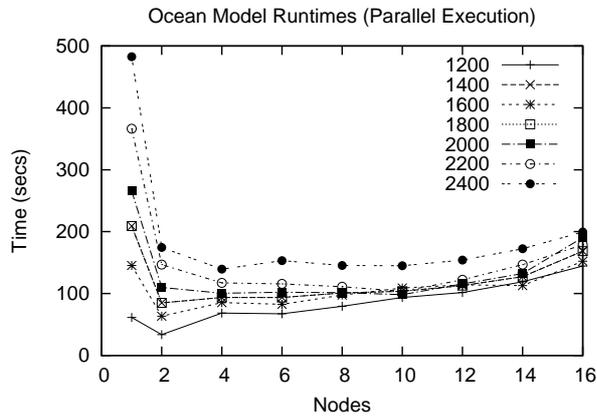


Figure 7: Parallel execution runtimes of the ocean model.

increase in the number of grid points being computed. Whilst it would have been an interesting exercise to experiment with grid sizes greater than 2400x2400, it was at this point that the cluster nodes began to need to use virtual memory, which artificially effected the results.

7.3.2 Ocean Model Summary

The results of the ocean model's performance allow us to conclude the following:

The Tupeware system does provide the application with an overall speedup gain by distributing the application and processing it in parallel. However, the level of speedup is limited, with the best result being experienced with the largest grid size used.

A significant part of this speedup gain is due to the decrease in time taken to perform the beginning and end sequential stages of the application's execution. This is due to the increased efficiency of network data transfer and, as the number of nodes is increased, smaller total panel sizes. Thus, the runtime of this sequential phase is not fixed, despite it being reliant on the single worker node.

The parallel phase of execution also provides a limited level of speedup, and this is due to smaller panels requiring less time to compute. However, the increased time spent on network communications as the number of nodes grows cancels out these benefits.

The overall performance of the ocean model is to be expected given the application's characteristics, in particular its tightly-coupled nature and the fact that each node's execution is synchronised to a high degree with the nodes that are processing neighbouring panels.

Some encouragement can be taken from the fact that scalability tends to increase along with the problem size. Therefore we can conclude that the scalability would likely continue to improve if the grid size were increased to greater than 2400. However, the increase would need to be very significant, as the efficiency of this application clearly showed that the communications time significantly dominated the processing time of the application.

7.4 Modified Quicksort

Much like the ocean model, the sorting application was tested on a varying number of worker nodes, from one through to sixteen, with an additional master node to set up and finalise the application's execution.

7.4.1 Results

The total runtimes of the sorting application are presented in Figure 8.

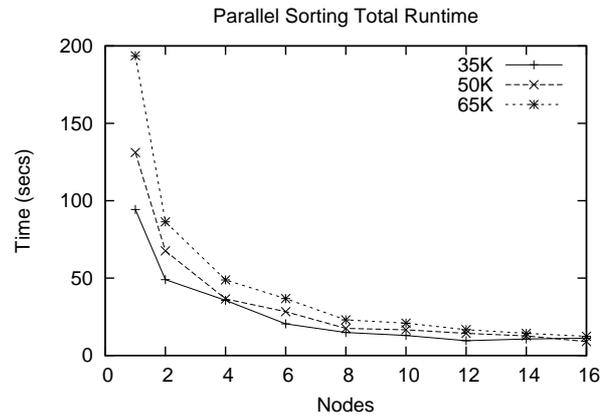


Figure 8: Total runtimes of the parallel sorting application.

As can be seen from the runtime for a single node system, sorting the array with the algorithm being used requires a significant amount of processing. Comparing this against a sixteen node system, we can see that the distribution and parallelisation of the application results in a substantial decrease in the overall runtime.

As a whole, the speedup experienced by the application is very pleasing. These speedup values can be found in Figure 9.

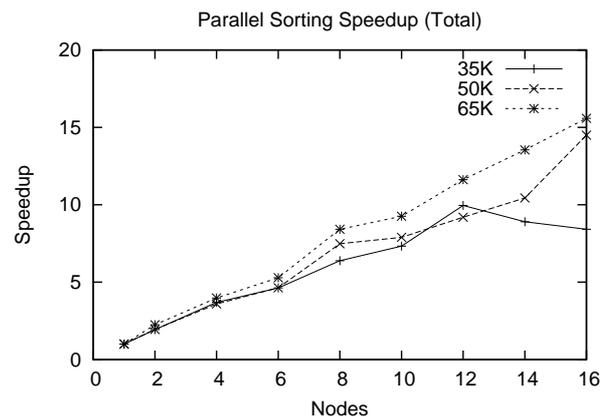


Figure 9: Total speedup of the sorting application.

In the instances of a thirty-five thousand threshold being used, the speedup peaks at 9.95 on twelve nodes before decreasing on fourteen and sixteen node systems. Nonetheless, this still provides a reduction in total runtime from 94.3 seconds to 9.5 seconds, a total decrease of 84.8 seconds. Considering that this threshold size is the smallest used for testing, and entails the greatest amount of network communication relative to other thresholds used, this is a pleasing result.

The two other threshold values used for testing gave a constant speedup up to sixteen nodes. In particular, for a threshold of sixty-five thousand, the speedup is near to optimal, and provides a total reduction in runtime of 181.3 seconds from 193.7 seconds on a single node system to 12.4 when sixteen nodes are used.

Parallel runtime is the sum total of the time spent performing network communications and processing once an initial unsorted array segment has been obtained. Network communications consist of obtaining additional unsorted segments once a worker's own storage in local tuple space has been exhausted, and also transferring sorted segments back to the master process. This will be affected by the threshold size: a smaller threshold requires more frequent communication with the master process, whereas a larger threshold requires less frequent. The speedup in terms of

the parallel phase of execution is illustrated in Figure 10.

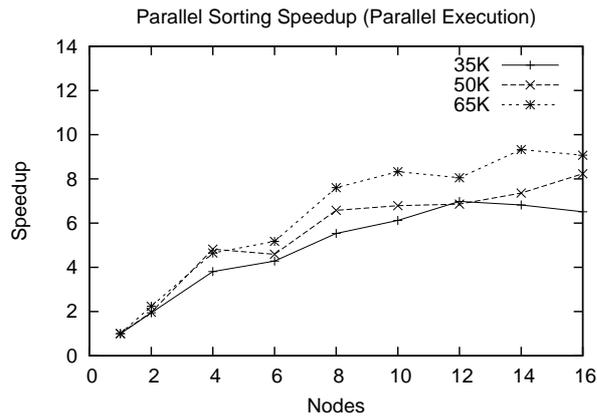


Figure 10: Speedup of parallel phase of sorting application execution.

As this shows, the speedup of the parallel phase of execution closely correlates to the speedup in terms of total runtime. For the smaller threshold value of thirty-five thousand, the speedup peaks at twelve nodes, however the speedup for the fifty thousand threshold continues through to sixteen nodes.

7.4.2 Modified Quicksort Summary

The results of the performance testing of the sorting application in the previous section have shown that the Tupleware system provides the application with significant performance gains and speedup. The speedup is most pronounced when a larger threshold is used. This is to be expected as increasing the threshold increases the granularity of the applications, increasing the ratio computation to communications time.

These results are very pleasing, and demonstrate that the system is able to provide speedup and performance gains for medium-grained applications. Based on Gustafson's Law [8], with the average communication time for each process remaining relatively constant as the number of nodes increases, while the total workload becomes larger, we would expect the application to continue to provide a high level of speedup as the number of nodes increases past sixteen. This prediction is further strengthened when we consider that increasing the problem size (via an increased threshold) actually greatly increases the efficiency of the system.

7.5 Performance Evaluation Summary

This section has presented the performance results of two applications: an ocean model and a parallel sorting application.

The findings of this performance evaluation were pleasing in terms of the sorting application, which displayed a high level of speedup on up to the maximum number of sixteen nodes, and was effective in evenly distributing the processing workload amongst all participating nodes in the system. The performance of this application also clearly illustrated the effect of varying the granularity of each processing task, with the larger threshold size exhibiting a higher degree of speedup than the smaller threshold sizes. This was due to the time each process spent on network communications remaining relatively constant, while the processing performed per process decreased as more nodes were added to the system. This is a result typical of an application such as this, and we can conclude that the Tupleware system has met its aim in this case of providing a scalable platform upon which to develop this style of medium-grained application.

In terms of the ocean model, the results show that the overall speedup gain was limited, and that as the number of nodes increased, the time each node spent performing network communication placed limiting factor to the continued scalability of the application. However, we also found that in increase in problem size, in this case the size of the grid, did not place a disproportionate load on any processes, and so there remains scope for the grid size to be increased further on a cluster with nodes with more than 1GB of memory.

8 Conclusions & Further Work

This paper has presented a dynamic, decentralised search function for the retrieval of tuples in a distributed system which contains multiple or a distributed tuple space. The search function optimises its behaviour based on the historical success or failure of previous tuple requests, which allows an accurate representation of the relative probability of remote nodes being able to fulfil a particular request to be formed by each individual node.

A further benefit is that the search function's logic is decentralised, without any need for sharing request data or global state information between nodes. If this was required, it would further add to the communication requirements of the system, and in turn lower the efficiency and performance of the applications.

Performance testing was carried out in order to determine the effectiveness of the search function, and the results presented in the previous section show that the system can provide performance gains for certain classes of distributed parallel applications, and that it can scale in terms of the number of nodes and also in terms of the problem size. While the performance of the tightly-coupled ocean model is not optimal, this is a common problem with tuple space-based systems, and the performance of Tupleware in this instance is comparatively good. It should be noted that the distribution of the tuple space in Tupleware in itself will introduce a certain amount of overhead, and yet this does not seem to cause the performance of applications running on Tupleware to suffer noticeably.

Further work on the system will entail implementing some sort of mechanism by which to measure the accuracy of tuple requests, so that we may compare the effectiveness of the search function being presented here to other alternatives.

Also some additional work is planned to address dynamic reconfiguration of the system at runtime. At the moment the number of participating nodes in the system must be known at compile-time in order to partition of array being processes and initialise the application. One of the strengths of the tuple space paradigm is its flexibility, and so it would be desirable to implement additional functionality to allow nodes to join and leave the system without interrupting to completion of the application being executed.

References

- [1] Amdahl, G 1967, 'Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities', *AFIPS Conference Proceedings*, (30), pp. 483-485.
- [2] Atkinson, A. and Malhotra, V 2004, 'Coalescing idle workstations as a multiprocessor system using Javaspaces and Java Web Start', *In: Eighth IASTED Intl. Conference on Internet and Multimedia Systems and Applications*, August 16-18, 2004, Kauai, Hawaii, USA.
- [3] Atkinson, A 2008, 'A Distributed Tuple Space for Cluster Computing', *Proceedings of the Ninth In-*

ternational Conference on Parallel and Distributed Computing and Techniques, Dunedin, New Zealand, pp. 121-126.

- [4] Carriero, N & Gelernter, D 1990, *How to Write Parallel Programs*, MIT Press, London.
- [5] Charles, A et al. 2004, 'On the implementation of SwarmLinda'. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pp. 297-298, New York, NY, USA.
- [6] Cormen, T. et al. 1999, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts.
- [7] Gelernter, D 1985, 'Generative Communication in Linda', *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80-112.
- [8] Gustafson, J 1988, 'Reevaluating Amdahl's law'. *Communications of the ACM*, vol 31, no 5, pp. 532-533.
- [9] Hoare, CAR 1961, 'Algorithm 64: Quicksort', *Communications of the ACM*, vol 4, no 7.
- [10] JavaSpaces™ Service Specification, 2003, Sun Microsystems, California.
- [11] Menezes, R and Tolksdorf, R 2003, 'A new approach to scalable Linda-systems based on swarms', *Proceedings of the 18th Symposium on Applied Computing (SAC'03)*, Melbourne, Florida, USA.
- [12] Merrick, I. and A. Wood 2000, 'Coordination with scopes', In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, New York, NY, USA, pp. 210-217.
- [13] Merrick, I 2003, 'Scope-based coordination for open systems', PhD thesis, University of York.
- [14] Nielsen B & Slrensen T, 1994, 'Distributed Programming with Multiple Tuple Space Linda', Masters Thesis, Aalborg University, Denmark.
- [15] Noble, M. S. and Zlateva, S. 2001, *Scientific Computation with Javaspaces*. Lecture Notes in Computer Science 2110, 657-667
- [16] Rowstron, Antony I. T., and Alan Wood. 1996. 'An Efficient Distributed Tuple Space Implementation for Networks of Workstations'. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume 1*, pp. 510-513.
- [17] Rowstron, A 1998, WCL: A Coordination Language to Geographically Distributed Agents, World Wide Web Journal, Volume 1, Issue 3, pp. 167-179.
- [18] Wells, GC et al. 2004, 'Linda implementations in Java for concurrent systems: Research Articles', *Concurrent Computing: Practice and Experience*, vol 16, no 10, pp. 1005-1022.

A Distributed Heuristic Solution using Arbitration for the MMMKP

Md. Mostofa Akbar¹, Eric. G. Manning³, Gholamali C. Shoja², Steven Shelford⁴
Tareque Hossain⁵

¹Department of CSE, BUET, Dhaka, Bangladesh
mostofa@cse.buet.ac.bd

²Department of CS, PANDA Group, UVic, Victoria, BC, Canada
gshoja@csc.uvic.ca

³Department of CS and ECE, PANDA Group, UVic, Victoria, BC, Canada
emanning@csr.uvic.ca

⁴Department of CS, UVic, Victoria, BC, Canada
sshelfor@uvic.ca

⁵Commlink Info Tech Ltd., R&D Group, Dhaka, Bangladesh
tareque@commlinkinfotech.com

Abstract

The Multiple-Choice Multi-Dimension Multi Knapsack Problem (MMMKP) is the distributed version of Multiple-Choice Multi-Dimension Knapsack Problem (MMKP), a variant of the 0-1 classic Knapsack Problem. Algorithms for finding the exact solution of MMKP as well as MMMKP are not suitable for application in real time decision-making applications. This paper presents a new heuristic algorithm, **Arbitrated Heuristic** (A-HEU) for solving MMMKP. A-HEU finds the solution with a few messages at the cost of reduced optimality than that of I-HEU, which is a centralized algorithm. We also discuss practical uses of MMMKP such as distributed Video on Demand service.

Keywords: Heuristic, Knapsack, Distributed.

1 Introduction

The classical 0-1 Knapsack Problem (KP) is to pick up items for a knapsack for maximum total value, so that the total resource required does not exceed the resource constraint R of the knapsack. The 0-1 classical KP and its variants are used in many resource management applications such as cargo loading, industrial production, menu planning and resource allocation in multimedia servers. Let there be n items with values v_1, v_2, \dots, v_n and let the corresponding resources required to pick the items be r_1, r_2, \dots, r_n respectively. The items can represent services and their associated values can be values of revenue earned from that service. In mathematical notation, the 0-1 Knapsack Problem is to find $V = \text{maximize} \sum_{i=1}^n x_i v_i$, subject to the constraint $\sum_{i=1}^n x_i r_i \leq R$ and $x_i \in \{0,1\}$.

The Multidimensional Multiple-choice Knapsack Problem (MMKP) is a variant of the classical 0-1 KP [5][6]. Let there be n groups of items. Group i has l_i items. Each item of the group has a particular value and

it requires m resources. The objective of the MMKP is to pick exactly one item from each group for maximum total value of the collected items, subject to m resource constraints of the knapsack. A resource constraint is the availability of a particular type of resource to pick items for a particular knapsack.

We define a new problem, the Multiple-Choice Multi-Dimension Multi Knapsack Problem (MMMKP) as a distributed version of the MMKP, where the resources are distributed among knapsacks. There is a *solver* associated with each knapsack for picking the items from the group. So, distributed computing techniques will be required for picking items. The following diagram shows an example of the MMMKP.

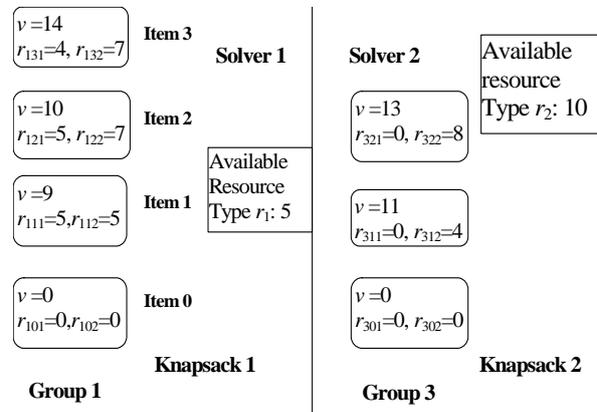


Figure 1 An MMMKP with 2 knapsacks and one resource in each knapsack

To define the MMMKP mathematically we need the following assumptions about the problem.

- There are M knapsacks. M solvers (one for each knapsack) pick items from the groups.
- The dimension of resources in Knapsack s is m_s and it provides resources labelled as μ_s to $\mu_s + m_s - 1$ inclusive. The total set of resources of the s th knapsack is defined by $(R_{\mu_s}, R_{\mu_s+1}, \dots, R_{\mu_s+m_s-1})$.

- Each solver is associated with exactly one knapsack. Only Solver s knows the entire state of Knapsack s and Solver s is solely responsible for allocating the resources of Knapsack s . The state information of a knapsack, such as resources used or available, is completely private to that knapsack and its solver, unless explicitly communicated to another solver by messaging.
- There are n groups of items. The i th group has l_i items. The j th item of the i th group requires r_{ijk} of the k th resource. Each solver knows which resource is served by which knapsack. The value of the j th item of the i th group is v_{ij} . n groups are distributed among M solvers. The number of groups in Solvers 1, 2, ..., M are $n_1, n_2, \dots, n_s, \dots, n_M$ respectively. The resource consumptions and associated values of the items of the n_s local groups of Solver s will not be advertised fully to all the solvers. The *partial resource consumption* of an item for a knapsack is defined by the resource requirement of the item from that knapsack. Thus, partial resource consumption of the j th item of the i th group for the resources of Knapsack s is expressed by the vector $(r_{ij\mu_s}, r_{ij(\mu_s+1)}, \dots, r_{ij(\mu_s+m_s-1)})$. The partial resource consumption of each item for any knapsack is sent to its associated solver. The set of M solvers will jointly execute a suitable distributed algorithm to pick exactly one item from each group, so that the total value of the picked items for the entire set of solvers is maximized subject to the resource constraints of each knapsack.

In mathematical notation, the MMMKP can be described as follows.

$$\text{Maximize } V = \sum_{i=1}^n \sum_{j=1}^{l_i} x_{ij} v_{ij}, \text{ total earned value from the}$$

picked items of the groups of all servers such that the resource constraints $\sum_{i=1}^n \sum_{j=1}^{l_i} x_{ij} r_{ijk} \leq R_k$, and $\sum_{j=1}^{l_i} x_{ij} = 1$ are satisfied.

The subscripts are defined as follows:

- $k = \mu_1, \mu_1+1, \dots, \mu_1+m_1-1, \dots, \mu_s, \mu_s+1, \dots, \mu_s+m_s-1, \dots, \mu_M, \mu_M+1, \dots, \mu_M+m_M-1$
- $x_{ij} \in \{0,1\}$, the picking variables
- $i = 1, 2, \dots, n; j = 1, 2, \dots, l_i$.

For our example in Figure 1 we can express the problem as follows:

$$\text{Maximize } V = \sum_{i=1}^2 \sum_{j=1}^{l_i} x_{ij} v_{ij}, \text{ subject to the resource}$$

$$\text{constraints } \sum_{i=1}^2 \sum_{j=1}^{l_i} x_{ij} r_{ij1} \leq R_1 = 5, \text{ and}$$

$$\sum_{i=1}^2 \sum_{j=1}^{l_i} x_{ij} r_{ij2} \leq R_2 = 10$$

1.1 MMMKP for Solving Multimedia Distribution Problems

MMMKP can be easily applied to revenue maximization problems where we find multiple admission controllers for multimedia session requests under a particular multimedia service provider organization. With MMMKP, the admission controllers may work together sharing multimedia session requests and determine the optimal serving strategy for maximum revenue. The following example demonstrates a viable application of MMMKP in Distributed Multimedia Server System.

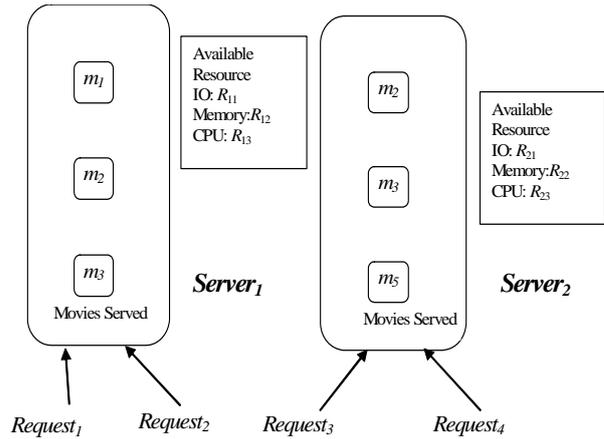


Figure 2 VoD servers serving requests

Consider two “Video-on-Demand” servers each serving two different collections or sets of movies as shown in Figure 2. A subscriber upon authentication may request for a multimedia session to any of the servers. If the movie does not reside in the server attempting to process it or if the server runs out of resources, the server may forward the session request to the other server.

A multimedia session between the server and the subscriber will require allocation of a number of resources on part of the server. These resources may include but are not limited to: Processing power, physical memory and IO capacity. It is allocation of these resources that determine a session’s quality of service. For sake of simplicity we consider only one level of QoS for each of the servers. Real life situations can be more complex with multiple QoS levels, separating subscribers who pay more for high quality audio-visual feed from those who settle for lesser quality. It is worth mentioning that a server has the full authority to allocate and utilize its resource only, which is one of the basic principles of distributed systems. Hence the problem can be defined as that of distributing multimedia session requests between the two servers so that maximum number of requests can be handled under the given resource constraints, thereby maximizing revenue.

From Figure 2 we find that there are 6 resource dimensions as expressed by $(r_{11}, r_{12}, r_{13}, r_{21}, r_{22}, r_{23})$. The first three indicates the resources of $Server_1$ and the remaining three indicates the resources of $Server_2$. Figure

3 shows an example of different choices of serving the requests in further details.

Two solvers are considered representing two servers entertaining requests from the customers. $Request_1$ demands service of movie m_2 . As we can see, both of the servers have the movie. Hence it is possible to serve the request in two different ways, namely $Choice_1$, when served by $Server_1$ & $Choice_2$, when served by $Server_2$. A generic representation for $Choice_1$ would be $(r_{11}, r_{12}, r_{13}, 0, 0, 0)$ and $(0, 0, 0, r_{21}, r_{22}, r_{23})$ for $Choice_2$. Similarly $Request_3$ is for movie m_3 which both $Server_1$ and $Server_2$ offer. Hence we have two choices to have the request satisfied. On the other hand movie m_1 resides only on $Server_1$ and m_5 only on $Server_2$ leaving us with a single choice for servicing requests concerning each of these movies.

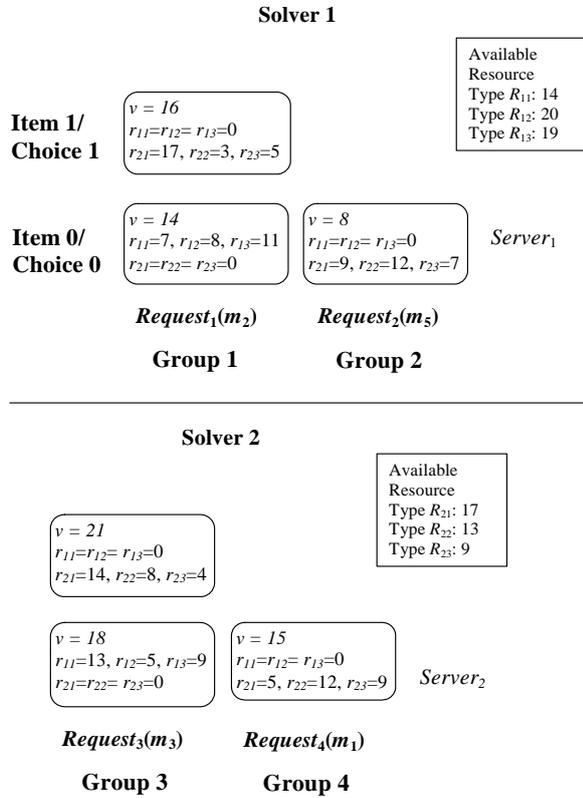


Figure 3 Multimedia distribution system mapped to MMMKP

The goal of MMMKP in such a Distributed Multimedia Server System is to pick exactly one item or QoS from each of the Group representing each request. When working independently, these servers may not choose the combination that is optimal for both of the servers, as already pointed out in the introduction section. MMMKP allows these two servers to share their decisions of resource allocation by passing messages and determines the solution that will yield maximum overall revenue.

2 Related Work on Solving Knapsack Problems

Many practical problems in resource management similar to the one discussed above can be mapped to the MMKP, consequentially their distributed version to MMMKP. But proposed exact solutions for MMKP are so computationally expensive [3][4][8] that they are not feasible for real time applications. In such cases heuristic or approximate algorithms for solving the MMKP and MMMKP play an important role.

Over the years, many heuristics have been proposed with a view to provide real time solution for MMKP. One of the earliest heuristics was HEU, proposed by Khan [7]. Khan has applied the concept of aggregate resource consumption [9] to pick a new candidate item in a group to solve the MMKP. Aggregate resource of the j th item of

the i th group is defined by $a_{ij} = \sum_k r_{ijk} \times C_k / |C|$, where

C_k = amount of the k th resource consumption and $|C| = \sqrt{\sum_k C_k^2}$. His heuristic HEU selects the lowest-

valued items by utility or revenue of each group as an initial solution. It then upgrades the solution by choosing a new *candidate item* from a group, which has the highest positive Δa_{ij} , the change in aggregate consumed resource (the item which gives the best revenue with the least aggregate resource). If no such item is found then an item with the highest $(\Delta v_{ij})/(\Delta a_{ij})$ (maximum value gain per unit aggregate resource expended) is chosen. Here,

$\Delta a_{ij} = \sum_k (r_{i\rho[i]k} - r_{ijk}) \times C_k$, the increase in aggregate consumed resource.

r_{ijk} = amount of the k th resource consumption of the j th item of the i th group.

$\rho[i]$ = index of selected item from the i th group and $\Delta v_{ij} = v_{i\rho[i]} - v_{ij}$, is the gain in total value.

Consequently, Akbar et al. [1] proposed another heuristic using the concept of aggregate resource called M-HEU, a modified version of Khan's HEU. In M-HEU the items in each group of the MMKP are sorted in non-decreasing order according to the value associated with each item. Hence, it can be said that in each group the bottom items are *lower-valued items* than the top ones. The items at the top can be defined as *higher-valued items* than those in the bottom. Picking a higher-valued or lower-valued item than the currently selected item in a group is called an *upgrade* or a *downgrade* respectively. The heuristic focuses on finding an upgrade or downgrade frequently. That is why the items of each group need to be sorted according to the associated values of the items. If a particular pick of items (one from each group) does not satisfy the resource constraints, that solution is defined as *infeasible*. A *feasible solution* is a solution that satisfies the resource constraints. For any resource k , *infeasibility factor* f_k is defined as C_k/R_k . The k th resource is feasible if the infeasibility factor $f_k \leq 1$, otherwise it is infeasible.

If the number of groups in the MMKP is very large then it is not possible to run M-HEU once every few seconds, as a real time system, (for example a multimedia system with 10,000 sessions) might well require. An *incremental* solution is a necessity to achieve better computation speed. By changing the technique of finding feasible solution M-HEU can be used to solve the MMKP *incrementally*, starting from an already solved MMKP. Akbar et al. named this heuristic I-HEU [1]. The proposed arbitrated heuristic A-HEU applies I-HEU in the solvers to select the probable candidate of the selected items. The steps of I-HEU are briefly described as follows:

Finding Feasible Solution (Step 1): In I-HEU a feasible solution is searched by selecting a lower valued item at first. If no feasible solution is found by searching lower valued item then higher valued items are looked up, like M-HEU. In this way most of the solution at hand can be re-used to obtain the new solution with less effort.

Upgrading Feasible Solution (Step 2): This is done by improving the solution value by selecting a feasible higher-valued item from the groups subject to resource constraints, i.e., by feasible upgrades.

Upgrade followed by Downgrades (Step 3): In this step the solution value is improved by one infeasible upgrade followed by one or more downgrades. This is analogous to get rid of local minima in the hill climbing algorithm.

Shahriar [11] presented a scalable solution to run MMKP heuristic using a multiple processor based computing server by distributing the computation among the processing nodes. But the presented algorithm is not intended for running the new problem that we have presented in this article.

In the following section, we present A-HEU, a new arbitrated heuristic, to determine the solution of the MMMKP by arbitrating among the solvers with a lower number of messages. But the total value of the items picked by A-HEU is often less than that of the centralized version.

3 Arbitrated Heuristic (A-HEU) for Solving the MMMKP

This method of solving MMMKP requires a few messages with several rounds of arbitrations; its message passing complexity is $O(M)$. The solver in each knapsack runs I-HEU independently. The candidate for upgrades and downgrades are calculated based on the value of PCAR (Partial Change of Aggregate Resource) defined as follows:

$$\Delta pa_{ijs} = \sum_{k=\mu_s}^{\mu_s+m_s-1} (r_{i\rho[i]k} - r_{ijk}) \times C_k$$

Hence, as a simplifying assumption, the resources in other knapsacks are completely ignored in this calculation. To find a feasible solution we first run Step 1

of I-HEU in each solver. If each solver finds a feasible solution and each solver satisfies the resource constraint of all the selected items from each group, then we find a feasible solution. Now, to find upgrades and downgrades, each solver sends its *proposed list* of selected items, calculated by running Step 2 and 3 of I-HEU, to the other solvers. The proposed list is sorted according to change of value per PCAR. It is worth mentioning that to find the globally selected items according to the values of PCAR the list must be sent along with the associated values of PCAR. The items in this sorted list, which satisfy the resource constraints of all the knapsacks, can be selected. Thus arbitration among the solvers is required to select items from the groups.

The same procedure can then be repeated until we run out of real time, to attempt to obtain better total values. Here we present an arbitration technique to select the items which requires only $O(M)$ message complexity. The following example shows only one arbitration step of A-HEU.

If we run I-HEU in both the solvers of Figure 4 independently, the solution can be shown in Table 1. As the lowest valued items are all zero, we need not find a feasible solution. Picking the j th item of the i th group can be expressed by (i, j) . The proposed lists by Solvers 1 and 2 are $\{(1,3), (2,2)\}$ and $\{(3,2), (4,3)\}$ respectively. These lists are exchanged between the solvers. The sorted global list after merging these proposed lists is $\{(1,3), (4,3), (3,2), (2,2)\}$. Now the feasible picks by Solver 1 are $\{(1,3), (4,3), (3,2)\}$ with $\sum r_1 = 14$. Similarly the feasible picks by Solver 2 are $\{(1,3), (4,3)\}$ with $\sum r_2 = 15$. So Solver 1 and 2 can satisfy the first 3 and 2 picks respectively from the proposed sorted list of selected items. They exchange this information and take the set intersection of their possible solutions; namely, they pick the first 2 items from the proposed list. Hence the solution after the first arbitration is $\{(1,3), (4,3)\}$ with $\sum r_1 = 14$, $\sum r_2 = 15$ and $V = 31$.

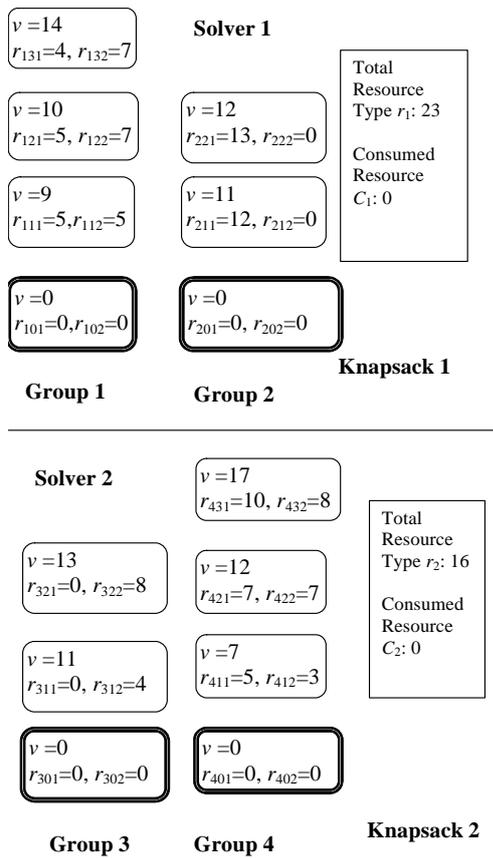


Figure 4 An MMMKP with two knapsacks.

| Groups | Picked Items | Change of value per PCAR | Resource consumption |
|---------|--------------|--------------------------|---|
| Group 1 | 3 | 3.5 | Resource consumption in Solver 1: $\sum r_1 = 17$ |
| Group 2 | 2 | 0.923 | |
| Group 3 | 2 | 1.625 | Resource consumption in Solver 2: $\sum r_2 = 16$ |
| Group 4 | 3 | 2.125 | |

Table 1 Items picked by Solver 1 and 2 by running I-HEU independently

3.1 Format of the Messages

The messages required to run A-HEU are listed and briefly described as follows.

| Message Type | Description of the structure | Monitor counter associated with the message |
|--------------|--|---|
| Groups | This is a list of groups. Each group is defined by (solver number, group number, number of items, partial resource requirements for the items) | $no_of_groups_msgs$ |

| Message Type | Description of the structure | Monitor counter associated with the message |
|-----------------------------|--|---|
| Local Total Value | The vector (solver number, total value) indicates the total value of the items picked by a solver. | $no_of_local_total_value$ l_value $_msgs$ |
| Proposed Selected Item List | The following vector is used to define a proposed selected item (solver number, group number, item number, value per PCAR) | $no_of_local_proposed_list_msgs$ |
| Local Feasibility Index | The vector (s, L) indicates that the first L items from the beginning of the global proposed selected item list are feasible with respect to the resources in Solver s . | $no_of_local_feasibility_msgs$ |
| Solution Not Found | This message indicates that a solver could not find any solution while determining proposed selected items for feasible solution. | Not applicable because the solver terminates if this message is received. |

Table 2 Different Types of Messages used in A-HEU.

3.2 Sequence of Events in A-HEU

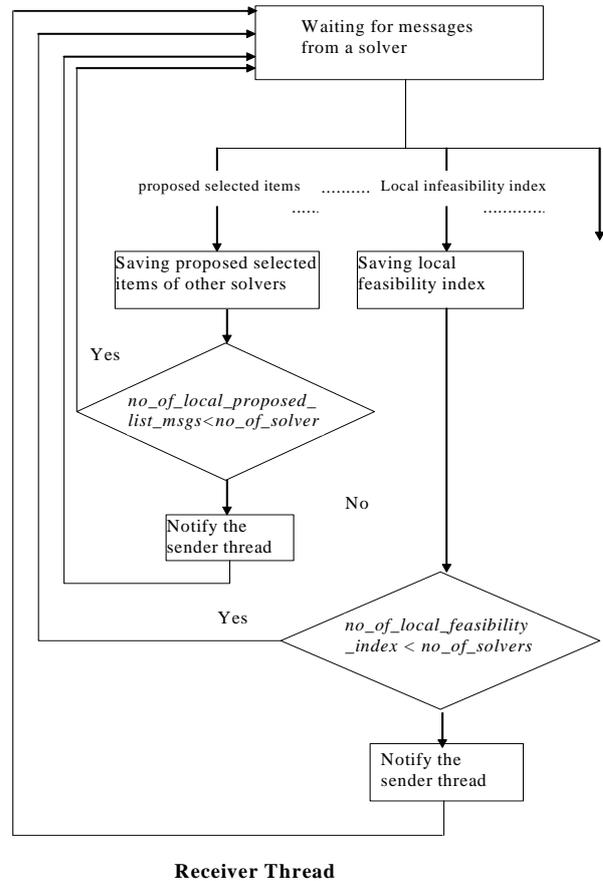


Figure 5 Flow chart of distributed computation by A-HEU (Receiver Thread)

Figure 5 and 6 shows the flow chart of the processes and events in A-HEU during each arbitration. We allocate separate variables for each solver's list of proposed selected items and local feasibility index. Thus, the actions 'saving proposed selected items' and 'saving local feasibility index' executed by the receiver threads need not to be atomic or synchronized. The decision blocks in the flow chart check counters shared by all threads, so, these decision blocks must be synchronized.

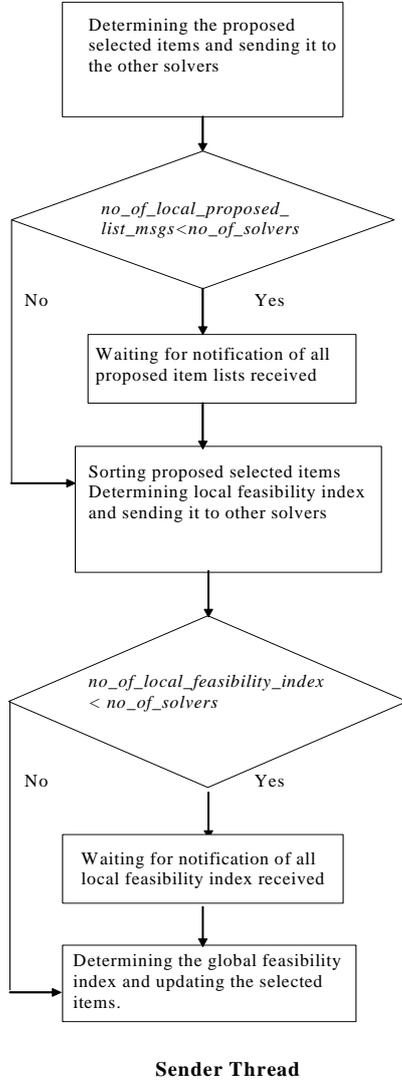


Figure 6 Flow chart of distributed computation by A-HEU (Sender Thread)

3.3 Complexity of A-HEU

Here we present the computational and message passing complexity by one arbitration of A-HEU as the first arbitration yields near optimal total value while the following iterations improve the solution with increased total values.

Computational complexity to run I-HEU on an MMKP with n/M groups and m/M resources is

$\frac{3m}{M} \left(\frac{n}{M}\right)^2 (l-1)^2$ in Step 2 and $\frac{7m}{M} \left(\frac{n}{M}-1\right)^2 (l-1)^2$ to escape from local minima in Step 3.

We require the following messages and computations in each arbitration to find the candidate items for upgrading or downgrading:

$2(M-1)$ messages to send local proposed list of selected items.

$\frac{n(n-1)}{2}$ floating point operations to sort the locally proposed lists to determine the global proposed list.

$2\left(\frac{nm}{M} + M\right)$ comparisons to find the local and global feasibility index. Thus computation and message passing complexities in each solver are $O\left(n^2(l-1)^2 \frac{m}{M^3}\right)$ and $O(M)$ respectively.

4 Experimental Results

In order to study the run-time performance of A-HEU to solve the MMMKP we implemented A-HEU along with I-HEU using Java. For simplicity of the implementation, we assume:

- Each group has the same number of items i.e., $l_1 = l_2 = \dots, \dots = l_n = 5$
- Each knapsack has the same number of resources i.e., $m_1 = m_2 = \dots, \dots = m_M = m = 4$.
- Each solver has the same number of groups i.e., $n_1 = n_2 = \dots, \dots = n_M = n_c$.

The algorithms were tested for an MMMKP with 3 knapsacks. Three different machines were used as three different solvers and a fourth machine was used as a generator of the MMMKP. The generator generates the groups of the MMMKP and sends them to the solvers. The generator machine also runs I-HEU on the transformed MMKP from the generated MMMKP.

4.1 Test Pattern Generation

The total amount of resources in the knapsacks, resource consumption by the items, and the values associated with the items are initialized as follows.

R_c = Maximum amount of a resource consumption by an item

P_c = Maximum cost per unit resource

R_i = Total amount of the i th resource = $n_c \times M \times R_c$.

P_k = Cost of the k th resource = $P_c \times Random(0.0, 1.0)$

$Random(0.0, 1.0)$ = A uniform continuous random number from 0.0 to 1.0.

Item 0 with zero value and zero resource consumption, i.e., $r_{i0k} = 0.0$ and $v_{i0} = 0.0$ is inserted for each group of the data set. Selection of this item indicates rejection of the group which is similar to the rejection of request in

the admission controller. The other items of the groups are initialized by the following random functions:

$$r_{ijk} = \text{The } k\text{th resource of the } j\text{th item of the } i\text{th group} = R_c \times \text{Random}(0.0, 1.0)$$

For initializing item values we use the following functions:

$$v_{ij} = \text{Value of the } j\text{th item of the } i\text{th group} \\ = \sum r_{ijk} \times P_k + \text{Random}(0.0, 1.0) \times \left(m \times M \times \frac{R_c}{10} \times \frac{P_c}{10} \right)$$

4.2 Test Results

The experiment was conducted for different values of n_c , from 100 up to 1000. The following data was collected from the experiments and is presented in Figure 7 to Figure 9.

- Total values of the picked items and time required by I-HEU
- Number of messages required by A-HEU
- Required time, total value of the picked items and number of messages by one, two and three arbitrations of A-HEU

In the experiment we have compared A-HEU with the centralized version I-HEU. There is no algorithm proposed so far in the literature to solve the MMMKP. That is why we present the effectiveness of A-HEU with respect to I-HEU by analyzing experimental results.

| Machine name | CPU speed | RAM | O/S | JDK Versions |
|--------------|-----------|--------|--------------|--------------|
| Solver 1 | 750 MHz | 256 MB | Windows 2000 | JDK 1.2.2 |
| Solver 2 | 700 MHz | 192 MB | Windows 2000 | JDK 1.3.1_03 |
| Solver 3 | 750 MHz | 256 MB | Windows 2000 | JDK 1.2.2 |
| Generator | 700 MHz | 192 MB | Windows 2000 | JDK 1.3.1_03 |

Table 3 Specifications of the solvers and generator of the MMMKP using IBM PC compatible.

4.3 Observations

- The total value of the items picked by A-HEU is approximately 90% of the total value of the items picked by D-HEU, but A-HEU requires a less time compared to D-HEU.
- Figure 9 shows the effect of message passing time in the overall complexity of the algorithm. For smaller data sets computational time is less compared to message passing. For larger data sets quadratic computation complexity of the algorithms dominates over linear message complexity. That is why better performance is observed for the larger data sets.

- We can easily conclude that A-HEU scales better than centralized I-HEU. We observe significant reduction in time requirement using A-HEU as reported by the time requirement data plotted in the exponential scale of the vertical axis.
- An irregular behaviour for the result of the set with 700 groups is observed in the figures showing the experimental results. Our complexity analysis presenting in this article is based on the worst case scenario. The actual computational time mostly depends on the data sets. A particular data set may lead to quick or late convergence to find the solution of the MMMKP showing exceptional behaviour in the result.
- For almost all the MMMKP data sets the total value of the items picked by first arbitration of A-HEU is more than 95% of the total value of the items finally picked by A-HEU.

| Number of groups in each solver | $\frac{V_{A-HEU}^1}{V_{I-HEU}} \times 100$ | $\frac{V_{A-HEU}^2}{V_{I-HEU}} \times 100$ | $\frac{V_{A-HEU}^3}{V_{I-HEU}} \times 100$ | $\frac{V_{A-HEU}}{V_{I-HEU}} \times 100$ |
|---------------------------------|--|--|--|--|
| 100 | 92.63 | 93.07 | 93.07 | 93.07 |
| 200 | 90.71 | 91.15 | 91.15 | 91.15 |
| 300 | 92.27 | 92.56 | 92.56 | 92.56 |
| 400 | 91.43 | 91.52 | 91.52 | 91.52 |
| 500 | 91.35 | 91.35 | 91.35 | 91.35 |
| 600 | 92.43 | 92.43 | 92.43 | 92.43 |
| 700 | 74.40 | 87.80 | 87.91 | 88.12 |
| 800 | 92.05 | 92.05 | 92.05 | 92.05 |
| 900 | 91.64 | 91.71 | 91.71 | 91.71 |
| 1000 | 92.57 | 92.77 | 92.80 | 92.80 |

Table 4 Ratio of total value of the items picked by A-HEU with respect to I-HEU. V_{A-HEU}^i indicates the total value of the items picked by the i th arbitration of A-HEU. V_{A-HEU} and V_{I-HEU} indicates the total value of the items picked by A-HEU and I-HEU.

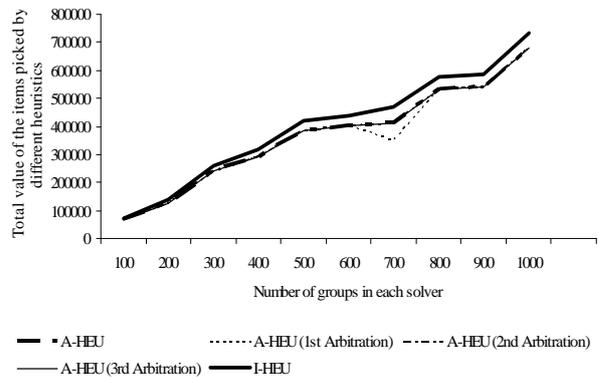


Figure 7 Total value of the items picked by A-HEU, D-HEU and I-HEU

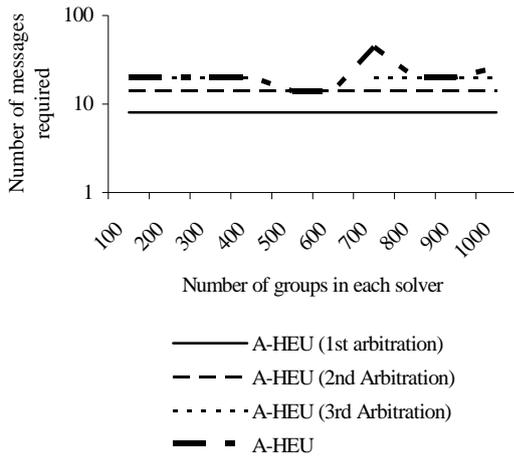


Figure 8 Number of messages required by distributed algorithms to solve the MMMKP.

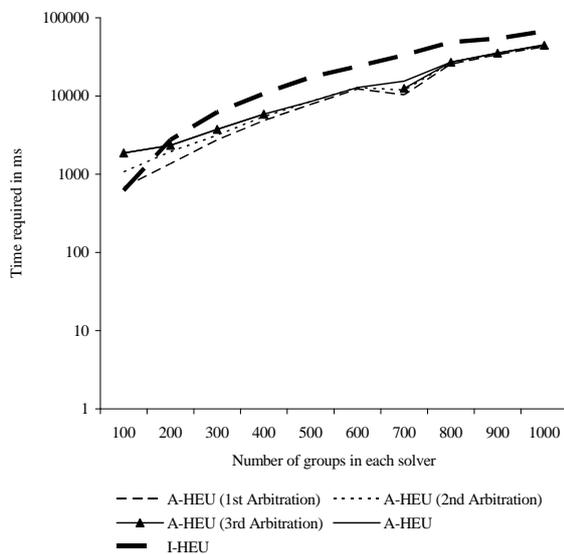


Figure 9 Time required by different algorithms to solve the MMMKP and MMKP

4.4 Discussion of the Performance of A-HEU

- An arbitration in A-HEU requires a few messages compared to any other regular distributed computing algorithms. So an iteration of A-HEU can be easily applicable for on line admission control algorithms. The admission control algorithm can execute further arbitration for better values if it has a more relaxed time constraint. Thus this algorithm is very suitable for an online system that requires quick decisions with a sub optimal total value, with the possibility of using more available time to improve the quality of the computation.
- The main reason for sub optimality in A-HEU lies in not considering all resource requirements in the preliminary selection.

- The arbitration technique for finding global feasibility index is another reason for sub optimality. We give up upgrading from the proposed list of selected items if we get one infeasible upgrade in the list. The very next item might be feasible. A new arbitration technique with $O(nM)$ message complexity might do that. However, if we do the arbitration again, with a newly calculated proposed selected item list for better total value, the competent items will get a chance to be selected. That is why we prefer multiple iterations of the arbitration, to a new arbitration technique with $O(nM)$ message complexity.
- If the items of the groups consume all the resources of all knapsacks uniformly then A-HEU is unlikely to get better total value in the next arbitration, because a particular resource has already been exhausted in the previous arbitration.

In practical cases, such as multimedia servers and Enterprise Networks, a QoS level of a session requires resources of a particular server or a particular network link. So there is a chance to allocate available resources to other selected items of the groups in the next arbitration.

5 Conclusion

In this article we have presented a new variant of knapsack problems which requires distributed algorithm to solve the problem. Our proposed new algorithm A-HEU achieves almost 90% optimality of I-HEU in $\frac{1}{O(M^3)}$ of the computation time required by I-HEU and $O(M)$ message passing where M is the number of solvers in the system. The experimental results show that the message passing time can be ignored for a larger problem set. Thus the new algorithm presents a scalable with the scope of distributed control. This particular problem is very much applicable for different resource sharing system in the distributed real time systems. This algorithm is a potential candidate for admission controlling in distributed real time systems.

6 References

- [1] Akbar, M., Manning, E. G., Shoja, G. C. and Khan, S. (2001): Heuristic solutions for the multiple-choice multi-dimension knapsack problem. *Proceedings of the International Conference on Computational Science* 659–668, San Francisco, Calif, USA, May 2001.
- [2] Akbar, M., Manning, E. G., Shoja, G. C. (2001): Admission Control and QoS adaptation in Distributed Multimedia Server System. *ITCom 2001* Denver, USA, August 2001.
- [3] Armstrong, R., Kung, D., Sinha, P. and Zoltners, A. (1983): A Computational Study of Multiple Choice

- Knapsack Algorithm. *ACM Transaction on Mathematical Software* 9:184-198.
- [4] Koleser, P. (1967): A Branch and Bound Algorithm for Knapsack Problem. *Management Science* 13:723-735.
- [5] Nauss, R. (1978): The 0-1 Knapsack Problem with Multiple Choice Constraints. *European Journal of Operation Research* 2:125-131.
- [6] Magazine, M. and Oguz, O. (1984): A Heuristic Algorithm for Multidimensional Zero-One Knapsack Problem. *European Journal of Operational Research* 16(3):319-326.
- [7] Khan, S., Li, K. F. and Manning, E.G. (1997): The Utility Model for Adaptive Multimedia System. *International Workshop on Multimedia Modeling* 111-126.
- [8] Shih, W. (1979): A Branch and Bound Method for Multiconstraint Knapsack Problem. *Journal of the Operational Research Society* 30:369-378.
- [9] Toyoda, Y. (1975): A Simplified Algorithm for Obtaining Approximate Solution to Zero-one Programming Problems. *Management Science* 21:1417-1427.
- [10] Hifi, M and Michrafy, M. (2006): A reactive local search-based algorithm for the disjunctively constrained knapsack problem. *Journal of the Operational Research Society* 57:718-726.
- [11] Shahriar, M.A.Z, Akbar, M.M., Rahman, M.S. and Newton, M.A.H. (2008): A multiprocessor based heuristic for multi-dimensional multiple-choice knapsack problem. *The Journal of Supercomputing* 43(3): 257-280.
- [12] Hifi, M., Michrafy, M. and Sbihi, A. (2004): Heuristic algorithms for the multiple-choice multidimensional knapsack problem. *Journal of the Operational Research Society* 55:1323–1332.

Object Oriented Parallelisation of Graph Algorithms using Parallel Iterator

Lama Akeila¹, Oliver Sinnen² and Wafaa Humadi³

The Department of Electrical and Computer Engineering
The University of Auckland, New Zealand.

¹ lake003@aucklanduni.ac.nz, ² o.sinnen@auckland.ac.nz, ³ whum003@aucklanduni.ac.nz

Abstract

Multi-core machines are becoming widely used which, as a consequence, forces parallel computing to move from research labs to being adopted everywhere. Due to the fact that developing parallel code is a significantly complex process, the main focus of today's research is to design tools which facilitate the process of parallelising code. The Parallel Iterator (PI) is a tool which was developed to automate the process of parallelising loops in OO applications. Graph algorithms can be represented using objects and hence they are excellent use cases for the PI. This paper discusses using the PI to parallelising graph algorithms such as breadth-first search (BFS), depth-first search (DFS) and minimum spanning tree (MST). Using the PI to parallelise such graph algorithms required adding some adaptations to the current concept of the PI to handle certain graph algorithms. The PI facilitates the process of parallelising graph algorithms in a way which keeps the parallel code readable and maintainable while exhibiting speedup. Java was used as the implementation language since it is one of the most commonly used object oriented languages. The parallelised graph algorithms were tested on different graphs and trees with different densities, granularity and structures. The experimental results show that the parallelised graph algorithms exhibit good speedups.

Keywords: parallel computing, object oriented parallelisation, Parallel Iterator, graph algorithms.

1 Introduction

With the introduction of multi-core processors, the importance of parallel computing has accelerated into the mainstream and hence, parallel computing technologies have been adopted everywhere (Reinders 2007). The expected performance speedup gained by increasing the number of processors depends on the problem to be solved and the algorithm which is used (Rajasekaran & Reif 2007). This signifies that the software has to be designed in a way which takes advantage of the increased number of processors and hence parallelising software applications becomes a necessity. The process of parallelising software applications is not straight forward since the developers are forced to deal with parallelisation details such as synchronisation between processors, locking of shared resources, race conditions and so on. As a consequence,

the main focus of today's research is to develop tools which facilitate the development of parallel applications. Most software applications rely heavily on iterative computations (N.Giacaman & O.Sinnen 2008) (i.e. computations handled by loops). In OO languages loops are handled by Iterator objects. As a consequence, tools for parallelising loops have very significant advantages when developing parallel applications. An example of such a tool is the PI which has been developed and implemented in the ECE department at the University of Auckland (N.Giacaman & O.Sinnen 2008, Akeila 2008).

The Parallel Iterator provides a thread-safe mechanism to iterate through a collection of elements concurrently by multiple threads and hence it eases the process of loop parallelisation in many OO applications. A graph library is an example of an OO application which plays an important role in various fields. Many applications rely on graph algorithms to solve common problems in computer science, chemistry and business (Buckley & Lewinter 2003). Graph libraries can be well implemented with objects. To maintain high productivity, readability and maintainability, the parallelisation should be done in an OO way. As a consequence, an OO tool such as the PI is powerful in terms of producing an OO parallel version of graph algorithms with a readable and maintainable code which exhibits speedup. Graphs are excellent use cases for the PI. Given their special structures, some graph algorithms might need adaptations and improvements to the PI, which is investigated in this paper. These adaptations are encapsulated by the PI (i.e. all the parallelisation details and the new adaptations are implemented internally by the PI) which, as a consequence, requires little or no code restructuring when using the PI to parallelise graph algorithms.

Parallelising three main graph algorithms using the PI is discussed in this paper: Breadth-First Search (BFS), Depth-First Search (DFS) and Minimum Spanning tree (MST). Section 2 introduces the PI. An overview about graph theory is included in section 3. Section 4 discusses the BFS and its parallelised versions while sections 5 and 6 discuss DFS and MST respectively.

2 The Parallel Iterator

Object oriented programming is widely used by software engineers. It allows software designers to develop high quality software solutions that are reusable and easy to implement and maintain (Craig 2001). Most applications heavily rely on iterative computations which are normally encapsulated inside loops (N.Giacaman & O.Sinnen 2008). As a consequence, the main approach in parallelising OO applications is parallelising loops. In object-oriented languages iterative computations such as loops are handled using iterators. Iterators are objects which allow the

Copyright ©2010, Australian Computer Society, Inc. This paper appeared at the 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), Brisbane, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 107, Jinjun Chen and Rajiv Ranjan, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

developer to traverse collections of elements by calling two main methods: 1) *hasNext* which returns a boolean value indicating whether there is any remaining element in the collection. 2) *next* which returns the actual element. However, conflicts occur if multiple threads are accessing the collection of elements concurrently when one element is remaining in the collection and at least two threads call the *hasNext* method simultaneously. Only one thread gets the next element while the other throws an exception. In addition to that, further parallelisation issues need to be taken into account when traversing a collection of elements concurrently such as load balancing and supporting different scheduling mechanism. Due to the insufficiency of using the normal iterator for parallel systems, the need for a mechanism to traverse collections in a thread-safe and parallel appropriate fashion became evident.

The PI concept has been developed and implemented in the ECE department at the University of Auckland. It allows for an efficient parallel traversal of a collection of elements without resulting in any conflicts between the threads which are accessing the collection simultaneously (N.Giacaman & O.Sinnen 2008). Iterations are distributed among the threads according to the specified scheduling policy. Once the thread finishes its allocated iterations, it exits and waits for the other threads to complete their iterations. Such synchronisation between threads is warranted and the program follows sequential semantics.

The PI has two main methods identical to the conventional sequential iterator, *hasNext* and *next*. It follows the typical semantics of an iterator in that the *hasNext* method is always called before calling the *next* method (i.e. it always checks whether there are still elements remaining in the collection before retrieval). The PI supports both random access collections (i.e. elements can be accessed directly in a constant time $O(1)$) and inherently sequential collections (i.e. elements' access time is proportional to the number of elements in the collection $O(n)$). The PI can be used with different scheduling policies and allows for specifying a chunksize as a method parameter. It supports three main scheduling policies as shown below in Figure 1: static (block and cyclic), dynamic and guided scheduling. Figure 1 shows the three different scheduling policies and how the elements are distributed among the threads in each policy with a collection of 9 elements when 3 threads access it simultaneously.

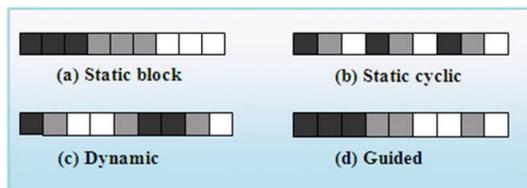


Figure 1: The implemented scheduling policies

The PI simply substitutes the sequential iterator. Little or no code restructuring is required to be added to the application which is being parallelised by the PI. It also can handle breaks and exceptions and it implements important features such as reductions.

3 Graph Theory

A graph can be defined as a mathematical representation of a relationship or set of relationships between elements (Buckley & Lewinter 2003). Any

graph G consists of a nonempty finite list of vertices V and a finite set of edges E that relates the vertices in V . Each edge in the set E consists of two vertices that are related. For example, if $V = \{v_1, v_2, v_3, v_4, \dots, v_n\}$ is the total number of vertices in G and $E = \{e_1, e_2, e_3, e_4, \dots, e_n\}$ is the list of edges in the graph, each edge in E is of the form $\{v_i, v_j\}$ (Buckley & Lewinter 2003). Some graphs have weights $w(v_i, v_j)$ on each edge where w represents the cost of connecting the two vertices, v_i and v_j together.

A path from vertex v_1 to vertex v_4 is the sequence of vertices $\{v_1, v_2, v_3, v_4\}$ which connects vertex v_1 to vertex v_4 . If the starting point vertex of the path is the same as the end point vertex, this path is said to produce a cycle and the graph is said to be cyclic graph. If no such cycles occur in the graph it is called acyclic. A graph is called connected if every pair of vertices is connected by a path

Some of the most commonly used graph algorithms are Breadth-First Search (BFS), Depth-First Search (DFS) and Minimum Spanning Tree (MST).

4 Breadth-First search (BFS)

BFS is one of the simplest algorithms for searching graphs and the idea is used by many other algorithms such as Prim's minimum spanning tree algorithm (Cormen, Leiserson, Rivest & Stein 2001). It is also used in other applications in various fields such as image processing (Silvela & Portillo 2001, Cormen et al. 2001). Given a graph G and a source vertex s , the BFS algorithm explores all the vertices reachable from the source vertex s in stages (i.e. the algorithm discovers the vertices reachable from s at distance k before discovering the vertices reachable from s at distance $(k + 1)$ (Cormen et al. 2001). A vertex in level k indicates that the distance from that vertex to the root is k (Buckley & Lewinter 2003). Figure 2 illustrates the different BFS levels of an example tree.

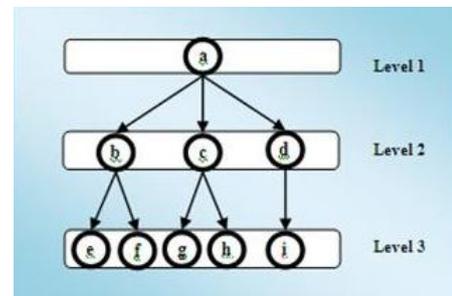


Figure 2: Breadth-First Search Levels

The next section presents the sequential algorithm of BFS and some previous work which was done to parallelise BFS followed by the various approaches which were taken to parallelise the graph algorithm using the PI. The proposed parallel approaches work for any graph. One of those parallel approaches uses the PI directly by passing each level of the tree at a time directly to the PI until no more levels exist. The other parallel approaches are implemented as an extension to the PI concept where the parallel iterator's *hasNext* and *next* methods return the nodes on the fly to the calling processors in a breadth-first order.

4.1 Sequential BFS

The sequential implementation of the BFS relies on processing each level of the tree at a time. It can be implemented using one queue. The following algorithm illustrates the concept where *successors* is

a queue which stores the nodes of the tree or graph. Initially the *successors* queue contains the starting vertex of the graph (i.e. the root in the tree case).

Algorithm 1 Sequential BFS.

```

1: while (successors is not empty){
2:   for (every node n in successors){
3:     process n if it is not processed
4:     add successors of n to successor queue end
5:   }
6: }
7: exit

```

As illustrated in Algorithm 1, the algorithm exits when the *successors* queue is empty (i.e. all nodes have been processed). Processing the nodes level by level ensures the breadth-first order. When BFS is run on trees, node n which is retrieved in line 3 is always processed since every node in a tree has one parent. However in graphs, a node can have more than one parent hence line 3 only processes the node if it was not processed previously.

4.2 Previous Parallel BFS Approaches

Most existing parallel BFS algorithms rely on processing nodes level by level (Yooy, Chowx, Hendersony, McLendonz, Hendrickson & Catalyurek 2005, Zhang & Hansen 2006, Rajasekaran & Reif 2007). All the nodes at a given level can be processed in parallel. A level is usually represented by a certain data structures such as *Queue* which is in turn accessed simultaneously by the different processors (Rajasekaran & Reif 2007). Every time a read or write operations are performed, the queue has to be locked and unlocked to ensure thread-safe behaviour (Rajasekaran & Reif 2007). The parallel algorithm exits when there are no more levels to process and all nodes have been visited.

4.3 Parallelising BFS with PI

In this section, three main approaches to parallelise the BFS algorithm are discussed. The first two approaches are extensions to the current PI as they provide the *hasNext* and *next* method with mechanisms to internally retrieve the next node on the fly in a BFS order. In other words, All the parallelisation details such as synchronizations and communication between threads are encapsulated internally by the iterator's *hasNext* and *next* methods. The third approach uses the conventional PI which was discussed in section 2 directly to parallelise the levels of the tree in a BFS fashion. The first approach uses one concurrent queue, the second uses two sequential queues with a locking mechanism to make the approach thread-safe and the third uses two concurrent queues with the parallel iterator as discussed below.

4.3.1 BFS with One Concurrent Queue

This approach uses one concurrent queue which contains the nodes to be processed. The concurrent queue is a thread-safe Java implementation of *Queue* where all the queuing operations are performed atomically (Sun n.d.). Each node in this approach is associated with a level attribute which indicates the level at which the node is in the tree as shown in Figure 3.

Figure 3 illustrates when this approach is run on the tree shown in figure 2. The level attributes indicate which level each node is in the tree. Threads retrieve nodes to be processed from the head of the queue and add the successors of the retrieved nodes to the end of the queue as show in Figure 3. Nodes

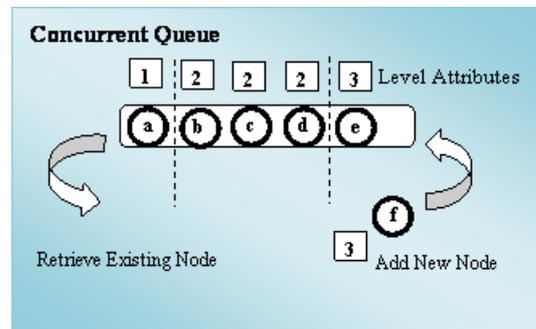


Figure 3: BFS with One Queue

are processed according to their levels in an ascending order. For example, nodes with a level attribute 3 are not processed until all the nodes with a level attribute 2 are processed. To ensure this processing order a global variable, *currentLevel* is shared between all the threads which indicates the current tree level which is currently being processed. It is initialized to 1 (i.e the root level). When all the nodes at a current level has been processed, the global variable *currentLevel* gets incremented to the value of next level. When threads get nodes with a level attribute which is greater than the *currentLevel*, those threads wait until *currentLevel* gets incremented. Algorithm 2 demonstrates the approach which is implemented by the extended PI's *hasNext* and *next* methods. *queue* represents the concurrent queue used in this approach, *id* is the id of each thread which varies from 0 to n where n is the number of processors, *levelsArray* is a 2D array which stores which level is each thread up to and *UpdateCurrentLevel* is a function which updates the value of the global variable *currentLevel* to the minimum positive value stored by *levelsArray*. Every time a thread update its value in *levelsArray*, the function *UpdateCurrentLevel* is called to update the value of the global variable *currentLevel*.

Algorithm 2 Parallel BFS with 1 Queue.

```

1: while (queue is not empty) {
2:   get node d from head of queue
3:   get level attribute, v of node d
4:   add the successors of d to end of queue
5:   levelsArray[id] = v
6:   UpdateCurrentLevel
7:   while(v > currentLevel){
8:     wait until currentLevel gets
       incremented
9:   }
10:  process d
11:}
12: levelsArray[id] = -1
13: UpdateCurrentLevel
14: exit and wait for other threads
   to exit

```

Algorithm 2 calls the *UpdateCurrentLevel* function in lines 6 and 13. This function updates the value of the *currentLevel* by inspecting the level values stored in the *levelsArray* by each thread then setting the global *currentLevel* variable to the minimum positive value. Every time a thread retrieves a different node, the thread records the level value of that node and stores it in the *levelsArray* as shown in Algorithm 2 line 5. When a thread finishes (i.e. line 1 returned false), its recorded level value in the 2D array is set to -1 as shown in line 12 and is ignored by the *UpdateCurrentLevel* function and hence the function will update *currentLevel* to the next positive minimum value when the function gets invoked in line 13. This ensures that the nodes which are stored in the concurrent queue are processed in a level by

level manner (i.e. in a breadth-first order).

4.3.2 BFS with Two Queues and Locking

This approach uses sequential queues (i.e. queues which do not support concurrency) with a locking mechanism to enforce a thread-safe access of elements in the queues. One queue in this approach is processed at a time. Each level of the tree is stored in one queue, say queue 1 which gets processed in parallel while the other queue, queue 2 gets populated with the nodes of the next level. The approach starts by processing the root node from queue 1 and writing the next level (i.e. successors) to queue 2. In the following iteration it processes nodes from queue 2 and stores successors in queue 1. This approach continues to alternate between the two queues until all the nodes have been traversed (i.e. both queues are empty). Since the queues which are used in this approach are not concurrent, threads lock the queues before performing read or write operations then unlock the queues when done with the operations. This approach supports specifying a certain chunksize as a parameter (i.e. number of nodes to be retrieved at once by the thread accessing the queue). Algorithm 3 illustrates the approach which is implemented by the extended PI. *readQ* is the queue which is to be read from, *writeQ* is the queue which is to be written to, *n* is the chunksize and *SwapIfEmpty* is an atomic function which swaps the two queues if the *readQ* is empty.

Algorithm 3 Parallel BFS with 2 Queues and Locks.

```

1: while(true) {
2:   SwapIfEmpty
3:   lock readQ
4:   if (readQ is not empty) {
5:     get top n nodes from readQ
6:     unlock readQ
7:     lock writeQ
8:     add successors of n to writeQ
9:     unlock writeQ
10:    process the n nodes
11:  } else {
12:    unlock readQ
13:    exit and wait for all other
14:    threads to exit
15:  }
16: }

```

4.3.3 BFS with Two Queues and Parallel Iterator

This approach applies the same concept which was discussed in section 4.3.2, however it uses 2 concurrent queues and the PI which was discussed in section 2. Each level of the BFS is passed to the PI to be processed concurrently by the threads until all nodes have been processed. The approach is illustrated in Algorithm 4.

Algorithm 4 Parallel BFS with the Parallel Iterator.

```

1: while(successors is not empty) {
2:   nextLevel = {}
3:   PI = getIterator(successors)
4:   while (PI.hasNext){
5:     node n = PI.next
6:     process n
7:   }
8:   successors = nextLevel
9: }
10: exit

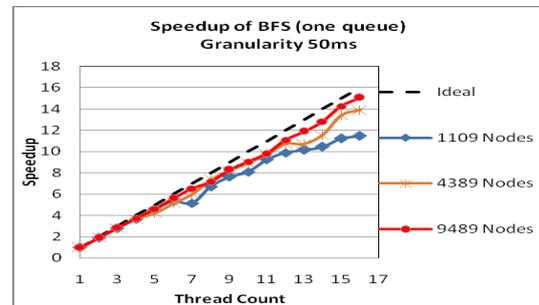
```

Algorithm 4 illustrates how the structure of the parallel BFS approach is similar to any sequential approach. The PI encapsulates the parallelisation details such as synchronization between threads and locking, hence little or no code restructuring is required to parallelise the sequential algorithm. The PI implements a barrier at the end of the iterations loop, hence it is guaranteed that any working thread gets to line 5 of Algorithm 4 when all the other threads have finished executing their iterations (i.e. all nodes in *successors* are processed). Since the PI supports different scheduling schemes and chunksizes, using it to parallelise BFS allows for testing this approach with the different implemented schedule policies to determine which schedule produces the best performance. Performance results are discussed in section 4.4.

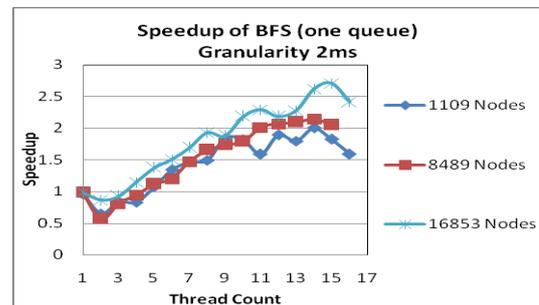
4.4 Parallel BFS Performance

The performance of the three different approaches which were discussed in section 4.3 was evaluated and compared with a large set of experiments using a 16-core machine. The experiments were run on wide trees with different number of nodes and granularity. The granularity was varied by changing the work per node. The speedup of the parallel algorithm is determined by $\frac{SequentialTime}{ParallelTime}$, where *SequentialTime* is the time taken to run the BFS by the sequential algorithm discussed in section 4.1 and *ParallelTime* is the time taken to run the parallel BFS algorithm.

Figure 4 shows the speedup results of the one queue approach discussed in section 4.3.1 with granularity 50 ms and 2 ms per node. The x-axis represents the number of processors (threads) which run the algorithm and the y-axis represents the speedup. A value of 1 on the x-axis represents the parallel code run on 1 processor (i.e. using 1 thread).



(a)



(b)

Figure 4: Speedup Results of BFS with One Queue.

The dashed line shown in Figure 4a represents the ideal speedup. Figure 4 shows that the performance of a coarse-grained parallel BFS is better than the fine-grained one as the speedup lines when the granularity is 50 ms per node are closer to the ideal speedup unlike the speedup lines when the granularity is 2ms which is shown in Figure 4b. Figure

4 also shows that the speedup of the parallel algorithm gets better when the number of node increases. However, the parallel BFS with one queue approach produces fluctuating speedup lines when tested on fine-grained graphs as shown in Figure 4b. This is due to the poor termination detection of the implemented algorithm which causes some threads to exit the algorithm early while some other threads are still writing to the concurrent queue.

The speedup results of the parallel BFS approach with two queues and locking discussed in section 4.3.2 are shown in Figure 5. Figure 5a shows the approach when was tested with different chunk sizes on a wide graph with 16,853 nodes and a granularity of 2 ms per node. It shows how chunk size 3 produces a slightly better speedup than chunk size 1 since increasing the chunksize reduces the communication overhead between processors due to the locking and unlocking of the queues. However, increasing the chunksize to 8 produces worse performance.

The approach was also tested on wide trees with a varying number of nodes and granularity. Since chunksize 3 produced good performance, the approach was tested with chunksize 3. Figures 5b and 5c show some speedup results.

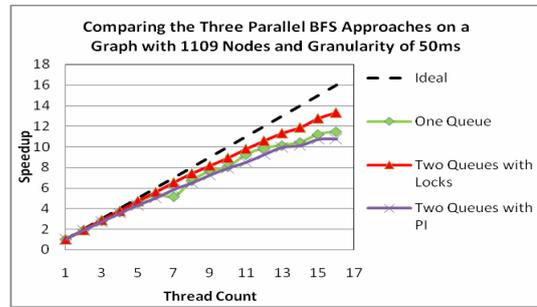
Figures 5b and 5c show how the performance of the parallel BFS gets better as the granularity gets coarser since the speedup lines in Figure 5b are the closest to the ideal speedup. The figure also illustrates that increasing the problem size (i.e. more nodes) enhances the performance of the algorithm. When comparing the 2 ms granularity speedup shown in Figure 5c of this approach with the one queue approach shown in Figure 4b, we find that the fluctuations in the speedup line shown in the one queue approach are not present in this BFS approach hence, the parallel BFS with two queues and locking performs better than the one queue approach when the granularity is small.

The speedup results of the parallel BFS approach with two queues and PI which was discussed in section 4.3.3 are shown in Figure 6. It was tested with a static schedule, dynamic schedule with chunksize 1, 3 and 8 on a wide graph with 16,853 nodes and a granularity of 2 ms as shown in Figure 6a.

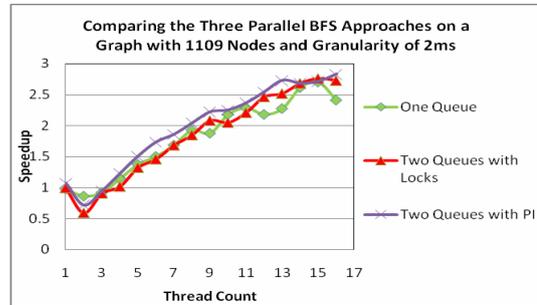
Figure 6a shows that the static schedule produces the worst performance while the dynamic schedule with chunksize 3 produces the best performance which is even better than the results which were produced by the BFS approach with locking shown in Figure 5a.

Similar to the previous approaches, the performance of the parallel BFS which is parallelised by the PI gets better as the granularity gets coarser and increasing the number of nodes enhances the performance of the algorithm. Figure 7 shows the three approaches when tested on a coarse-grained graph (50 ms per node) shown in 7a and a fine-grained graph (2 ms per node) shown in 7b with 1109 nodes.

Figure 7 shows that the parallel BFS with two queues and locks produces a slightly better performance on coarse-grained graphs than the other two approaches. However, the parallel BFS with two queues and PI produces the best performance on fine-grained graphs. For example, the speedup of a fine-grained wide graph with 16,853 nodes using the 2 queues approach with the PI shown in Figure 7b reaches around 2.8 whereas in the one queue approach and the two queues approach with locking, the speedup reaches around 2.4 and 2.7 respectively.



(a)



(b)

Figure 7: Comparing the three parallel BFS approaches on fine-grained graphs.

5 Depth-First Search (DFS)

The depth first search is another searching algorithm which searches deeper in the graph (Buckley & Lewinter 2003). Given a graph G and a source vertex s , nodes are discovered as far as possible along each branch of G before backtracking and discovering the rest of the unvisited edges (Buckley & Lewinter 2003, Cormen et al. 2001). DFS has many applications such as finding strongly connected components, topological sort algorithms and solving puzzles (Cormen et al. 2001, Grama, Gupta, Karypis & Kumar 2003). The order in which nodes are expanded in DFS is illustrated in Figure 8.

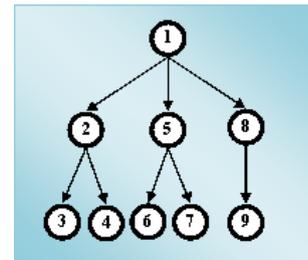


Figure 8: The order in which nodes are expanded in DFS.

The next section discusses the sequential implementation of DFS, some previous approaches which were taken to parallelise the algorithm followed by the various approaches which were taken to parallelise the graph algorithm with the PI.

The sequential DFS is different to the parallel DFS in that a leaf node such as node 3 or 4 in Figure 8 is always discovered before node 5 in the sequential DFS, however the parallel DFS allows node 5 to be discovered before node 4 since multiple threads process the different sub-trees concurrently. In other words, the parallel DFS returns the nodes in topological order.

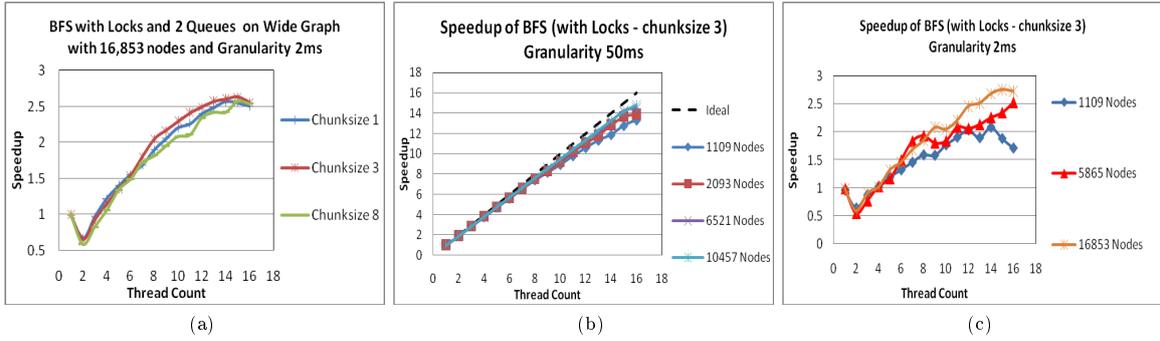


Figure 5: Speedup Results of BFS with two Queues and Locks.

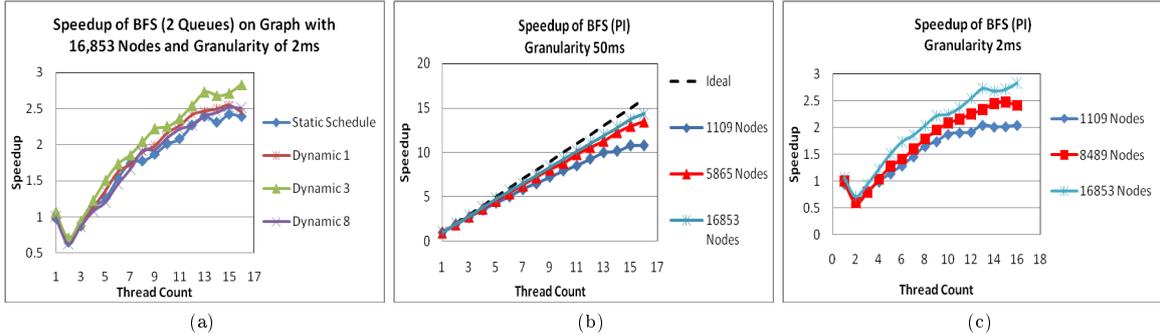


Figure 6: Speedup Results of BFS with Parallel Iterator.

5.1 Sequential DFS

The sequential algorithm uses a stack which is a Last-In-First-Out (LIFO) data structure. The element which is at the top of the stack is processed first. The stack initially contains the root node. As the algorithm proceeds, successors are added to the top of the stack. The nodes in the stack are processed in a LIFO manner until there are no more nodes to be processed. Algorithm 5 illustrates the sequential DFS where *stack* is the LIFO data structure .

Algorithm 5 Sequential DFS.

```

1: while (stack is not empty) {
2:   Get node n from top of stack
3:   Push successors of n
   to stack
4:   process n
5: }
6: exit
    
```

As illustrated by Algorithm 5 , the algorithm exits when the *stack* becomes empty (i.e. all nodes are processed). Processing the nodes in this fashion enforces a depth-first order.

5.2 Previous Parallel DFS Approaches

Many research attempts have approached the problem of developing a parallel DFS algorithm. According to research studies, the most critical characteristic which determines the performance of the DFS is load balancing (i.e. the mechanism of splitting work between the different processors) (Grama et al. 2003). Several researchers have parallelised DFS by dividing the search space into sub-trees (Reinefeld & Schnecke 1994.). Those individual sub-trees are distributed among the processors to search them in depth-first fashion. To balance the load, a processor which has finished its work attempts to get unpro-

cessed sub-tree from another processor (Reinefeld & Schnecke 1994.). This is called work-stealing. In order to keep all processors busy all the time, a dynamical stack-splitting method can be used (Reinefeld & Schnecke 1994.). In such method, every processor works on the node in its own local stack. When a stack is empty, the processor requests work from other processors. The donor processor splits its local stack to donate unprocessed sub-trees to the processor which requested work (Reinefeld & Schnecke 1994.). Determining the donor processor (i.e. the target processors which will donate unprocessed nodes to the idle processor) can be done by several load-balancing schemes such as Asynchronous Round Robin (ARR), Global Round Robin (GRR) or Random Polling (RP) (Grama et al. 2003). In the ARR scheme, each processor stores an independent variable, *target*, locally which represents the donor processor. It gets incremented by each processor each time work is requested. In the GRR scheme, the value of *target* is stored globally and shared between all the processors. The RP scheme is the simplest since it selects a donor processor randomly every time a processor gets idle. The use of a Last In First Out (LIFO) data structure such as *Stack* ensures that the nodes are processed in a depth-first order.

5.3 Parallelising DFS with PI

The concept of the PI was enhanced to support the implementation DFS. The iterator was extended with a *hasNext* method which returns elements to the calling threads in the DFS, i.e. topological, order on the fly. All the parallelisation details such as synchronizations and communication between threads are encapsulated internally by the iterator's *hasNext* and *next* methods. Two main parallel approaches were implemented, one which uses one global stack and the other uses a local stack for each thread and one global stack shared by all threads. The load-balancing is achieved

by the first approach via sharing the nodes (i.e. sub-trees) which are stored in the global stack dynamically. In the second approach, the load-balancing is achieved in two ways: 1) by sharing the node which are stored in the global stack similar to the first approach. 2) by explicitly stealing work from the local stacks of the other working threads when both the global stack and the thread's local stack are empty. The target donor to steal work from is determined randomly as explained in the Random Polling (RP) scheme in section 5.2. The two approaches are explained in the following sections.

5.3.1 Parallel DFS with One Stack Approach

This approach uses one LIFO data structure, a concurrent stack. Threads poll the stack to get a node to process then push successor to the top of the stack. The stack stores nodes which are yet to be processed. The nodes which are stored in the stack are processed dynamically. Once a thread gets idle, it retrieves a new node from the stack. The use of a stack ensures that the nodes are processed in a topological order. Algorithm 6 illustrates the approach where *stack* is the concurrent global stack which is shared between the threads.

Algorithm 6 Parallel DFS with One Stack.

```

1: while (stack is not empty) {
2:   Get node n from top of stack
3:   Push successors of n to stack
4:   process n
5: }
6: exit and wait for all other
   threads to exit

```

Algorithm 6 shows that no explicit locking is performed in this approach since a concurrent stack implementation is used. However, all the locking and unlocking actions are implemented internally by the concurrent stack every time a read or write requests are performed.

5.3.2 Parallel DFS with Local Stacks and Work Stealing

This approach uses one global stack which is shared by all the threads and n local stacks where n is the number of threads. Each thread stores the nodes of its sub-tree in its local stack. Initially the global stack contains the root of the tree and only one thread gets access to it while the other threads are waiting to be woken up by the working thread. The working thread retrieves and processes the root from the global stack, adds one successor to its local stack then pushes the rest of the successors, if they exist, to the global stack to be processed by other threads and wakes up all the waiting threads. Storing only part of successors in the local stack is better than storing all successors at once since it produces better load balancing between threads. It avoids the case where big sub-trees get stored in the local stack of one thread. From this point on, the thread reads from and writes to its local stack until it becomes empty. All the other threads follow the same approach (i.e. when threads get a node from the global stack they push one successor to their local stack and the rest to the global stack then start accessing their local stack until it becomes empty). When the local stack of a thread is empty, the thread retrieves a new node from the global stack. If the global stack is empty, the idle thread tries to get unprocessed nodes from the bottom of a neighboring local stack (i.e. the oldest nodes) using a random

polling scheme and stores them in its local stack. If no extra work is available in the neighboring local stacks, the thread exits and waits for the other threads to exit.

5.4 Parallel DFS Performance

The performance of the two different approaches was evaluated and compared with a large set of experiments using the same 16-core machine as before. The experiments were run on wide trees with different number of nodes and granularity.

Experimental results show that the performance of the two approaches is similar when tested on coarse-grained graphs. Figure 9 shows the speedup results of the approaches when tested on a graph with 5003 nodes with a granularity of 40 ms. increasing the granularity and number of nodes produces better performance.

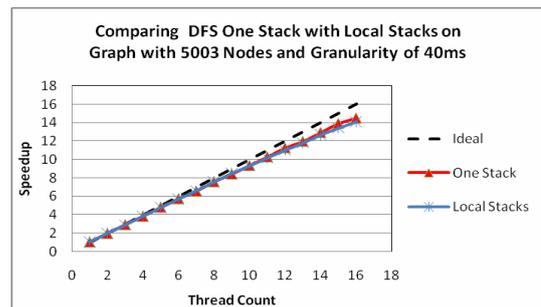


Figure 9: Comparing the two parallel DFS approaches on coarse-grained graph.

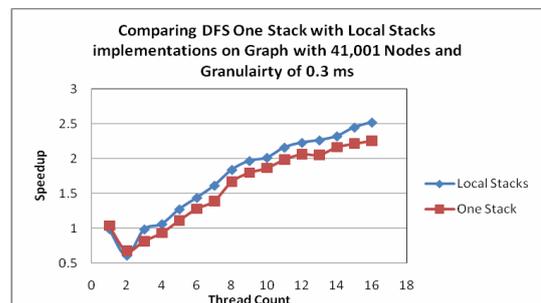


Figure 10: Comparing the 2 parallel DFS approaches on fine-grained graph.

Figure 10 shows a comparison between the two approaches when tested on a fine-grained graphs with 41,001 nodes. The result shows that the local stacks approach performs better than the one stack approach when the granularity is small. This is due to the fact that the overhead produced by the locking and unlocking actions in the local stacks approach is less than the overhead present in the one stack. In the local stacks approach, locking is performed only when threads finish all the nodes in their local stack and attempts to get more nodes from the global stack whereas in the one stack approach locking is always performed when getting nodes since all threads share one global stack.

6 Minimum Spanning Tree (MST)

The Minimum Spanning Tree (MST) is one of the most studied algorithms which has many practical applications in wireless communication, distributed networks (S.Meguerdichian, Koushanfar, Potkonjak & Srivastava 2001) and medical imaging (An, Xiang

& Chavez 2000.). The MST problem determines the minimum-weight path which connects all of the vertices of a given graph G without producing any cycles. The sum of the weights of edges in the path should be the smallest sum possible in the graph G (Cormen et al. 2001). Figure 11 shows the final MST path (in gray) when the algorithm is run on an undirected weighted graph.

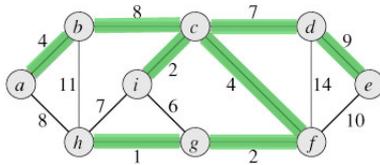


Figure 11: Final MST Path.

6.1 Sequential MST

One of the main algorithms for computing the MST is called Prim's algorithm. It contains a set A that forms the resultant minimum spanning tree. All the edges added to the set A always form a single tree. The safe edge added to A is described as the edge with the minimum weight that is connecting the tree in A to a vertex that is not in the tree (Cormen et al. 2001). The algorithm starts from an arbitrary starting vertex and grows by adding safe edges to the set A until the tree spans all vertices. When the algorithm terminates, set A contains all safe edges that form the minimum spanning tree of graph G . The sequential MST is illustrated in Algorithm 7 where n is the number of vertices in G .

Algorithm 7 Sequential MST.

```

1: Select a vertex  $v$  and let  $V(T)=\{v\}$ 
   and  $E(T)=\{\}$ 
2: while(true){
3:   Iterator through edges of  $v$ 
   and determine the edge  $e=\{v,w\}$  with
   minimum weight which connects  $v$ 
   to another vertex  $w$  where  $w$  is
   not in  $V(T)$ 
4:   Add  $w$  to  $V(T)$  and  $e$  to  $E(T)$ 
5:   If (size of  $E(T) = n - 1$ )
6:     exit algorithm
7:   else
8:      $v = w$ 
9: }
```

6.2 Previous Parallel MST Approaches

One of the solutions to parallelise the MST algorithm is the fast parallel implementation developed by Bader and Cong (D.Bader & Cong 2004.). It allows every processor to start from a different starting vertex and run Prim's algorithm simultaneously (D.Bader & Cong 2004.). In this approach vertices are coloured by the colour of the processor which they were visited by. Every time an edge is to be added to the MST by a particular processor, the processor checks whether the vertex is coloured by its own colour otherwise a collision with another processor occurs. In the case of collisions, the processor stops growing the current sub-tree and exits otherwise it continues until all vertices are visited (D.Bader & Cong 2004.). The final sub trees produced in parallel are merged sequentially in the end to produce the final MST.

Another solution to parallelise the MST algorithm with an adjacency-matrix is to partition the adjacency matrix among the p processors as shown in Figure 12 (Grama et al. 2003).

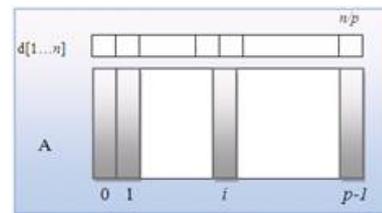


Figure 12: The partitioning of the adjacency matrix and the distance array d .

An array d represents the distance array of length n , where n is the number of vertices in the graph G . Every vertex in the graph has an entry in the distance array d . This entry holds the weight value of the most discovered minimum edge which connects the vertex to another vertex in the graph (Grama et al. 2003). Initially the array entries are initialized to ∞ . As a new vertex is discovered and the minimum edge added to the MST, the distance array d is updated with the weights of the incident edges of the newly added vertex. If the weight of one of those edges is smaller than the weight recorded in the array, the entry value is updated to store the edge with the minimum weight. Figure 12 shows how the distance array and the adjacency matrix are partitioned between p processors.

The set of vertices V is partitioned to subsets with $\frac{n}{p}$ consecutive vertices subsets and each subset V_i is assigned to each processor P_i (Grama et al. 2003). Determining the minimum edge and updating the distance array with the newly added vertex are done in parallel. Each P_i stores the part of the distance array d which includes its set of assigned vertices V_i as shown in Figure 12. Each processor computes the edge with the minimum weight from its part of the adjacency list and then the global minimum is computed and stored in process P_0 using an all-to-one reduction process (Grama et al. 2003). The new vertex that is now stored in P_0 is broadcasted to all other processes using one-to-all broadcast, the new edge is added and then each processor updates its relevant distance array portion with the incident edges of the newly added vertex.

6.3 Parallelising MST with PI

The two approaches taken to parallelise MST are based on the approaches discussed in section 6.2. In the first approach processors start from a different starting vertex of the graph and run the MST algorithm simultaneously. The different produced subtrees are merged in the end sequentially. A book keeping module, which keeps track of which nodes have been traversed by which thread, was integrated with the PI to implement this approach successfully.

The second approach uses a distance array and reductions as discussed in section 6.2 to produce a parallel version of MST. This approach uses the PI directly by passing the distance array to the PI to get updated in parallel and using the PI reductions feature to extract the minimum edge.

6.3.1 Parallel MST with Book-Keeping Module

In this approach, a list of the graph vertices is passed to the extended PI. The extended PI issues only the

starting points (unvisited nodes) to threads by the *hasNext* and *next* methods. The book-keeping module has two main functionalities; it provides an atomic mechanism which ensures that each node in the graph is processed by only one thread and updates the PI with the status of the nodes (i.e. visited or not) during the run time of the MST algorithm as it keeps a record of which nodes are processed by which threads (i.e. colouring mechanism). Every time a vertex is issued, the book-keeping module colours the vertex by the colour of the processor which visited it. From there, threads continue traversing the graph independently by adding safe-edges to the final MST until either the final MST becomes complete or collisions occur between threads (i.e. more than one thread attempts to access the same node). Collisions are detected by the book-keeping module when more than one thread attempt to access a node. When collisions occur, one thread gets the unprocessed node while the others go back to the PI to get a new unvisited starting vertices. The *hasNext* method of the PI always calls the book-keeping atomic methods to ensure that the issued nodes are unvisited. When all nodes are visited, all the sub-trees which are produced by the different threads in parallel are then merged sequentially.

6.3.2 Parallel MST with Distance Array and Reductions

This approach uses the PI presented in section 2, a distance array d and reductions to produce a parallel version of MST. The distance array d has an entry for every vertex of the graph as explained in section 6.2. Each entry in d stores the edge with the minimum weight encountered so far by the algorithm which connects the corresponding vertex (i.e. the key of array d) with some other vertex. Initially all the entries are set to null. In this approach one edge is added to the final MST edges list $E(T)$ at a time after doing two tasks in parallel in every iteration of the algorithm. The first task is updating d after adding a new vertex to the set of visited vertices $V(T)$ and the second is extracting the edge with the minimum weight from d . At the start of the algorithm a starting vertex is assigned and added to $V(T)$. In every iteration of the algorithm, an edge with the minimum weight which connects a visited vertex in $V(T)$ to an unvisited vertex which is not in $V(T)$ is added to $E(T)$. The unvisited vertex is then added to $V(T)$.

Since two tasks are done in parallel, two instances of the PI are created. One PI instance is used to update d in parallel with the weights of the incident edges of the last vertex v which was added to $V(T)$. This PI instance is initialized with the list of the incident edges v . The second PI is used to extract the minimum edge from d in parallel and is initialized with a list of all vertices which are used as keys in d . When extracting the minimum edge, every thread stores the edge with the minimum weight from its corresponding part of d then the global minimum edge is determined using the PI reductions. The algorithm terminates when the number of added edges is equal to $n - 1$ where n is the number of vertices as shown in Algorithm 8.

Lines 4 and 11 of Algorithm 8 are executed in parallel using the PI. In line 18 the reduction feature implemented by the PI is used to retrieve the global minimum edge. Reductions in the PI are performed in an OO fashion using a *Reducible* object shown in line 10 which manages the different copies of the variable to be reduced (Giacaman, Sinnen & Akeila 2008). PI implements reductions in a way which allows the user to customize the type of reduction to be used which makes the feature usable with any type of data

Algorithm 8 Parallel MST with Distance Array and Reductions.

```

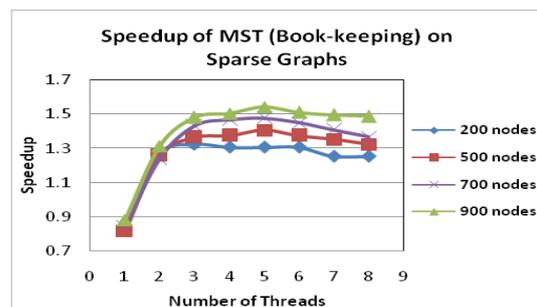
1: Select a vertex  $v$  and let
    $V(T)=\{v\}$  and  $E(T)=\{\}$ 
2: while(true)
3:    $PI = \text{getIterator}(\text{incident edges of } v)$ 
4:   while( $PI.hasNext$ ) {
5:     edge  $i = PI.next$ 
6:     update  $d$  with  $i$  if  $i$  has weight less
       than what is stored in  $d$ 
7:   }
8:    $PI = \text{getIterator}(\text{list of all nodes})$ 
9:    $minimumEdge = \{\}$ 
10:   $Reducible \text{ globalMinEdge} = \{\}$ 
11:  while( $PI.hasNext$ ) {
12:    node  $v = PI.next$ 
13:    edge =  $d[v]$ 
14:    if (weight of edge <  $minimumEdge$ ) {
15:       $minimumEdge = edge$ 
16:    }
17:  }
18:   $FinalMinimumEdge = \text{globalMinEdge.reduce}$ 
19:   $w = \text{vertex of FinalMinimumEdge which is}$ 
   not visited
20:  Add  $w$  to  $V(T)$  and  $FinalMinimumEdge$  to  $E(T)$ 
21:  If ( $\text{size of } E(T)=n-1$ ) {
22:    exit the algorithm
23:  }
24:  else
25:     $v = w$ 
26: }
```

and any kind of reduction (Giacaman et al. 2008). In this approach, the PI reduction was customized in a way which returns an edge with the minimum weight.

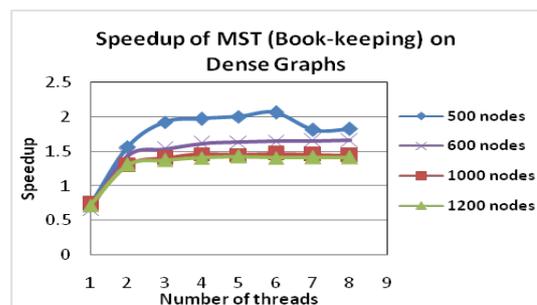
6.4 MST Performance Results

The performance of the two parallel MST approaches which were discussed in sections 6.3.1 and 6.3.2 was evaluated and compared with a large set of experiments on an 8-core machine. The experiments were run on different graphs with number of nodes and densities.

Figure 13 shows the speedup of the parallel MST with the book-keeping approach discussed in section 6.3.1 when tested on sparse and dense graphs.



(a)



(b)

Figure 13: Speedup of MST with Book-keeping.

Figure 13a shows that the performance of the par-

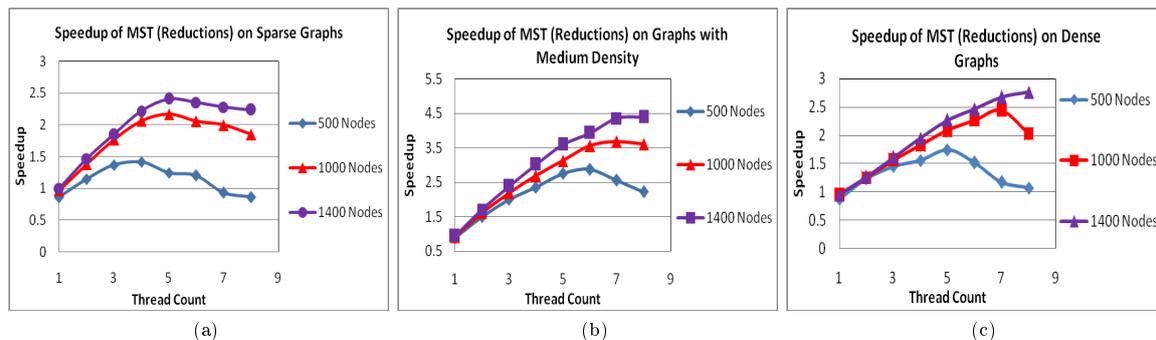


Figure 14: Speedup of MST with Distance Array and Reductions.

allel MST gets better as the number of nodes in the sparse graph increases. However, since the graph is sparse, the produced speedup is not that significant (i.e. around 1.6 max). When the approach is tested on dense graphs as shown in Figure 13b, a slightly better performance is produced (i.e. reaches up to around 2.1). However, Figure 13b shows that as the number of nodes increases, the speedup gets worse when the approach is tested on dense graphs. This is the main limitation of this approach since running the sequential algorithm which connects the sub-trees produced in parallel slows down the run-time of the algorithm and produces bad performance on dense graphs as the number of nodes increases. This is because in dense graphs, the number of edges is equal to V^2 , where V is the number of nodes in the graph. More edges result in more collisions and hence, more sub-trees to be merged sequentially.

Figure 14 shows the speedup results of the parallel MST approach with distance array and reductions discussed in section 6.3.2 when tested on graphs with low density (i.e. Sparse), medium density and high density.

The results in Figure 14 shows that the second parallel MST approach performs significantly better on dense graphs than the first approach illustrated in Figure 13. The approach produces the best performance when was tested on graphs with medium densities as the speedup reaches up to around 4.5.

7 Conclusions

Desktop applications must be parallelised to benefit from the introduction of multi-core processors. However, parallelising applications is considered to be a significantly challenging task. The PI is an OO tool which automates the process of parallelising loops in OO applications. Graphs are excellent use cases for the PI since they are naturally represented by objects. To maintain high productivity, readability and maintainability of OO graph algorithms, the parallelisation should be done in an OO way. Using the PI to parallelise graph algorithms is powerful in terms of producing a readable and maintainable code which exhibits speedup. Three main algorithms were parallelised using the PI: BFS, DFS and MST. Some parallel approaches of those algorithms required some adaptations and improvements to be added to the PI. The improvements include integrating a book-keeping module with the PI to perform concurrent colouring when running the MST and extending the *hasNext* and *next* methods of the PI to return nodes on the fly in breadth-first or depth-first order. Experimental results show that the algorithms which were parallelised by the PI exhibit good speedup while keeping the readability and maintainability of an OO code.

References

- Akeila, L. (2008), Parallel iterator, Technical report, ECE department, University of Auckland.
- An, L., Xiang, Q. & Chavez, S. (2000.), A fast implementation of the minimum spanning tree method for phase unwrapping, in 'Med. Imaging'.
- Buckley, F. & Lewinter, M. (2003), *A Friendly Introduction to Graph Theory*, Prentice Hall/Pearson Ed.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001), *Introduction to Algorithms*, McGraw-Hill.
- Craig, I. (2001), *The Interpretation of Object-oriented Programming Languages*, Springer.
- D.Bader & Cong, G. (2004.), Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs, in 'Parallel and Distributed Processing Symposium'.
- Giacaman, N., Sinnen, O. & Akeila, L. (2008), Object-oriented parallelisation: Improved and extended parallel iterator, in '14th IEEE International Conference on Parallel and Distributed Systems'.
- Grama, A., Gupta, A., Karypis, G. & Kumar, V. (2003), *Introduction to Parallel Computing 2nd edition*, Addison-Wesley.
- N.Giacaman & O.Sinnen (2008), Parallel iterator for parallelising object oriented applications, in '7th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS'08)'.
- Rajasekaran, S. & Reif, J. (2007), *Handbook of Parallel Computing, Models, Algorithms and Applications*, CHAPMAN & HALL.
- Reinders, J. (2007), *Intel Threading Building Blocks: Outfitting C++ for Multi-Core*, O'Reilly.
- Reinefeld, A. & Schnecke, V. (1994.), Work-load balancing in highly parallel depth-first search, in 'Scalable High- Performance Computing Conference'.
- Silvela, J. & Portillo, J. (2001), Breadth-first search and its application to image processing problems, in 'Image Processing', Vol. 10, pp. 1194-1199.
- S.Meguerdichian, Koushanfar, F., Potkonjak, M. & Srivastava, M. (2001), Coverage problems in wireless ad-hoc sensor networks, in 'Proc. INFOCOM'01'.
- Sun, J. (n.d.), 'Java concurrent queue, retrieved from <http://java.sun.com/javase/6/docs/api/>'.
- Yooy, A., Chowx, E., Hendersony, K., McLendon, W., Hendrickson, B. & Catalyurek, U. (2005), A scalable distributed parallel breadth-first search algorithm on bluegene/l.
- Zhang, Y. & Hansen, E. A. (2006), Parallel breadth-first heuristic search on a shared-memory architecture, in 'Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications'.

Experience on the parallelization of the OASIS3 coupler

Italo Epicoco¹Silvia Mocavero²Giovanni Aloisio^{1,2}

¹ Department of Engineering for Innovation
University of Salento,
via per Monteroni, 73100 Lecce, Italy,
Email: {italo.epicoco,giovanni.aloisio}@unisalento.it

² Euro-Mediterranean Centre for Climate Change
Via Augusto Imperatore 16, 73100 Lecce, Italy
Email: silvia.mocavero@cmcc.it

Abstract

This work describes the optimization and parallelization of the OASIS3 coupler. Performance evaluation and profiling have been carried out by means of the CMCC-MED coupled model, developed at the Euro-Mediterranean Centre for Climate Change (CMCC) and currently running on a NEC SX9 cluster. Our experiments highlighted that extrapolation (accomplished by the *extrap* function) and interpolation (implemented from the *scriprmp* function) transformations take the most time. Optimization concerned I/O operations reducing coupling time by 27%. Parallelization of OASIS3 represents a further step towards overall improvement of the whole coupled model. Our proposed parallel approach distributes fields over a pool of available processes. Each process applies coupling transformations to its assigned fields. This approach restricts parallelization level to the number of coupling fields. However, it can be fully combined with a parallelization approach considering the geographical domain distribution. Finally a quantitative comparison of the parallel coupler with the OASIS3 pseudo-parallel version is proposed.

Keywords: OASIS3, climate models, coupled models, performance analysis, parallel modeling

1 Introduction

Climate change models describe complex subsystems such as oceans dynamics; atmospheric, chemical and physical processes; vegetation and land use transformations. Historically, these models have always been stand alone applications. They are not complete enough to describe the complexity of the whole climate system, unless we consider the chance to infer new knowledge from their coupling. A more detailed approach is to model the climate behavior by coupling models each others. In this context a coupler component is a key performance factor of the overall coupled model. The coupler acts as a "collector" amid component models. Its main function is to interpolate, extrapolate, re-grid and, more in general, transform exchanged fields. As for modeling, the coupler should support different parallel approaches in order to be compliant with and portable on heterogeneous parallel architectures. To this aim, the OASIS3 coupler is both OpenMP and MPI parallelized; it is possible to

select a hybrid approach or just one of the available parallelization methods at compile time. Given the nature of the operations performed by the coupler, its execution time cannot overlap with component models one. Thus optimization and parallelization of the coupler have strong impact on the wall clock time of the overall model. Both work and results described in this paper refer to an optimized and parallelized version of OASIS3 adopted at the Euro-Mediterranean Centre for Climate Change (CMCC). In particular, the CMCC-MED (S. Gualdi, E. Scoccimarro et al.) coupled model has been taken into account to test the parallel coupler.

The paper is organized as follows: section 2 describes the OASIS3 coupler; its performance profiling on the target machine (NEC-SX9) and performed optimization are detailed in section 3. We then detail our parallel approach in section 4, performance model and its analysis in section 5. Finally, we give a qualitative comparison of our proposed approach with the OASIS3 pseudo-parallel version in section 6, an evaluation of alternative scheduling solutions in section 7, and draw our conclusions.

2 The OASIS3 coupler

The coupler OASIS3 (Valcke 2006) consists of a set of Fortran 77, Fortran 90 and C routines. At runtime, OASIS3 acts both as a separate single process executable, whose main aim is to interpolate coupling fields exchanged among the component models, and as a library (OASIS3 PSMILe) linked by the component models in order to communicate with the coupler.

OASIS3 provides several transformations and 2D interpolations in order to adapt coupling fields from a source model grid to a target one. For each exchanged field, the user can define a set of required transformations and their order through the *namcouple* configuration file. Available transformations are grouped into five general classes and must be strictly applied following this logical order: time, pre-processing, interpolation, "cooking", and post-processing. This order is also supported by the OASIS3 software internal structure. It is worth noting here that transformations are usually independently performed on each field, thus transformations on a single field can be considered as a separate task. *BLASOLD* and *BLASNEW* transformations represent an exception to this rule, since they introduce functional dependence among fields. They perform, respectively before and after the interpolation phase, a linear combination of the current field with others or with itself. The following steps characterized OASIS3 coupling activity:

- An initialization step, executed once for each

Copyright ©2010, Australian Computer Society, Inc. This paper appeared at 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), Brisbane, Australia. Conferences in Research and Practice in Information Technology, Vol. 107. Editors Jinjun Chen (Swinburne University of Technology) and Rajiv Ranjan (University of New South Wales), Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

run. It includes some preliminary operations such as initialization of internal parameters, definition of logical I/O units descriptors, allocation of data structure for timing purpose, definition of the communication environment among processes and grids, instantiation of variables and opening of required files.

- The execution of a loop over the time steps. For each time step, another loop over the sequencing index (*SEQ*) is performed. *SEQ* defines the order for fields to be transformed. Its aim is to allow overlapping of the coupling time with models computing time. A tag with small values of *SEQ* in the *namcouple* file must be associated to fields sent to the coupler from faster models; that allows overlap of OASIS3 coupling time spent over those fields with computing time of slower models. The loop over sequencing index includes: (i) evaluation of the number of fields to be exchanged among coupled models; (ii) retrieval of fields values from models; (iii) fields transformation through pre-processing, interpolation, cooking and post-processing operations; (iv) delivery of those fields to the target models.
- A finalization step, including arrays deallocation and file closing.

3 OASIS3 profiling

The OASIS3 coupler has been evaluated and profiled within the CMCC-MED model, developed at the CMCC. It is a 3-components coupled model consisting of the Echam5 (Roeckner et al. 2004) T159L31 atmospheric model, the OPA 8.2 (Madec et al. 1998) oceanic global model with a 2° resolution and the Nemo (Madec 1998) Mediterranean sea model with a 1/16° resolution. The atmospheric model provides the coupler with 26 fields defined on a 480x240 spatial grid; 17 are addressed to the ocean global model and the 9 remaining are addressed to the Mediterranean sea model. The ocean global model provides the coupler with 6 fields, defined on a 182x149 spatial grid, to be sent to the atmospheric model. Finally, the Mediterranean sea model provides the coupler with 3 fields, defined on an 871x253 spatial grid, to be addressed to the atmospheric model too. The total amount of managed fields, exchanged among the component models, is 35, with a coupling period of 2h 40' and thus 279 coupling steps in a month. Table 1 lists performed transformation. Extrapolation (over 29 fields) and interpolation (over 35 fields) are the most frequent ones.

The coupled model has been profiled on a NEC SX9 cluster using FTRACE analysis tool in order

Table 1: CMCC-MED namcouple configuration.

| Transformation | # of fields |
|-------------------|-------------|
| Locktrans | 8 |
| Mask | 29 |
| Extrap [ninenn] | 29 |
| Invert | 23 |
| Scripr [distwgt] | 2 |
| Scripr [conserv] | 3 |
| Scripr [bilinear] | 18 |
| Scripr [bicubic] | 12 |
| Conserv [global] | 2 |
| Blasnew | 8 |
| Reverse | 9 |

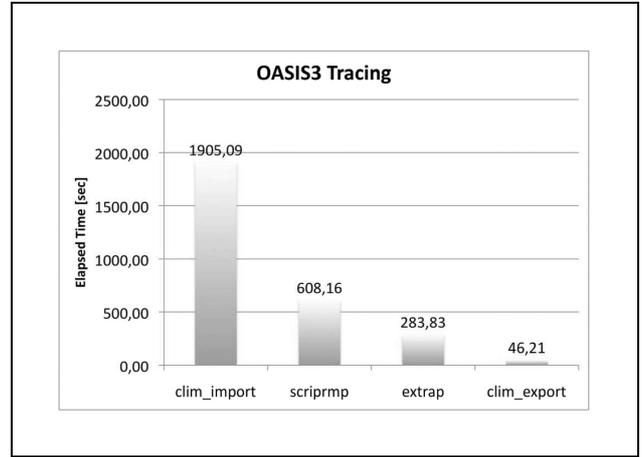


Figure 1: OASIS3 performance tracing.

to identify time-consuming functions. Optimization started from these functions. In particular, a FTRACE region has been defined in the OASIS3 code. The FTRACE output, shown in figure 1, highlights that *clim_import* takes about 1900 seconds followed by *scriprmp* and *extrap*. It is worth noting here that *clim_import* belongs to the CLIM library adopted for the communication among the coupler and the component models; it is devoted to receive fields exchanged among models. The elapsed time spent executing this function is actually an idle time, since the coupler has to wait for the component models to simulate the coupling period. We can thus safely ignore this function since it does not include coupling time.

As table 2 shows, *extrap* and *scriprmp* are the most time consuming transformations; they take about 96% of the total coupling time.

In the following sections we delve into details of the optimization performed on these functions.

3.1 Extrap analysis and optimization

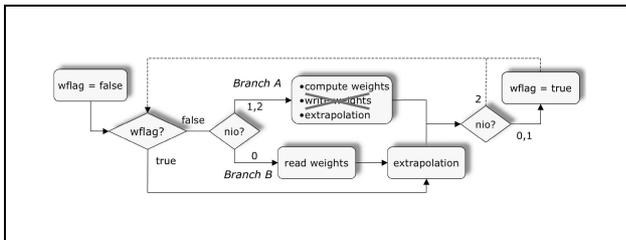
The *extrap* function performs fields extrapolation over their masked points using the source grids. Since the adopted weights depend only on the source grid, it is reasonable to group fields into different datasets, each of these characterized by the same source grid and hence by the same weights. A field is also tagged with a NIO parameter, whose value is 1 if weights must be computed and written to file or 0 in case of reading from file. It is worth noting here that the NIO parameter is taken into account only for the first field of a given dataset and only during the first coupling step, it is ignored otherwise; in these cases weights are always read from memory. The flow chart in figure 2 gives an overview of the original OASIS3 algorithm. *wflag* is a boolean variable used to establish if

Table 2: OASIS3 performance analysis

| | Elapsed Time (sec) | % |
|---------------------|--------------------|-------|
| scriprmp | 608.16 | 64.61 |
| extrap | 283.83 | 30.15 |
| clim_export | 46.21 | 4.91 |
| others | 3.14 | 0.33 |
| Total Coupling Time | 941.35 | |

weights and addresses values for dataset i are available in main memory or not. At the beginning, $wflag$ is initialized to *FALSE* for all of the datasets; when the coupler processes the first field of the dataset i , the instruction control flow depends on NIO value. Following Branch A, both definition of weights and extrapolation of field, are jointly performed; weights are then stored in a file. Branch B is followed when NIO is 0 and extrapolation is performed by means of the stored weights. In both cases, weights are stored in main memory and $wflag$ is asserted. That implies extrapolation to be performed reading weights from the main memory for every field of the same dataset and for each of further coupling steps. Considering the flow chart of figure 2, it is clear that:

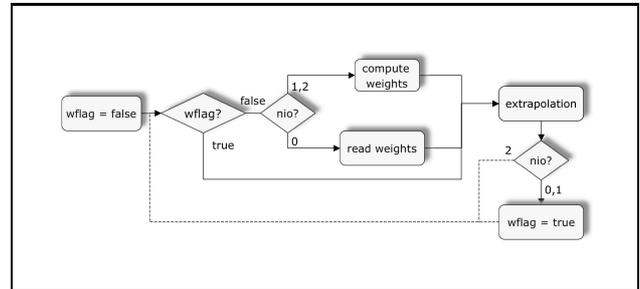
1. weights and addresses values are written in a file only when OASIS3 transforms the first field of a given dataset, during the first coupling step, and its NIO value is equal to 1;
2. weights and addresses values are read from file only when OASIS3 transforms the first field of a given dataset, during the first coupling step and its NIO value is equal to 0;
3. weights and addresses values are read from main memory otherwise.

Figure 2: Flow chart of the *extrap* function.

These assertions reveal that even if weights are written, they are never read. We thus can optimize the *extrap* function skipping the writing procedure. Despite the introduction of this optimization, performance improvement, as shown in table 3, is very poor. This happens because weights writing is performed only during the first coupling step.

Performance analysis of the *extrap* function highlights also some numerical issues owing to the replication of the source code on two different branches (see figure 4). In particular, extrapolation of the first field of each dataset is performed during weights evaluation (Branch A); the others are extrapolated in a different branch (Branch B). Unfortunately, the compiler optimizes the two branches in different way introducing some optimizing transformation of floating point operations. Experiments show that if we change the order of a field in the *namcouple* configuration file, its value, after the extrapolation, is different. In particular, if we swap the position of two fields, the difference on the first field is about of $1.6 \cdot 10^{-14}\%$. This displacement can absolutely be negligible. However,

if we change the order of more than one field belonging to different datasets, this displacement produces a 0.25% difference on the netcdf output files generated by one simulated month. This discrepancy is relevant since it is due only to a different order of the fields in the *namcouple* file. To solve this problem, the code performing weights evaluation and extrapolation has been split. In this way, all the fields, including the first one of each dataset, is extrapolated using the same piece of code. The final solution is represented in figure 3.

Figure 3: Optimized *extrap* transformation.

3.2 Scripr analysis and optimization.

The *scriprmp* routine implements the interpolation techniques offered by the Los Alamos National Laboratory SCRIP1.4 library. In particular, it performs a remapping of the fields using weights and addresses evaluated taking into account the source grid, the target grid, the type of interpolation to be used and the normalization option. For each field, the *scriprmp* function checks whether the file containing remapping weights exists. If not, they are first evaluated and then written to a file for the further coupling steps.

At each coupling step, an access to the file is performed. The main optimization concerns the management of remapping weights into the main memory, in order to reduce the time spent for I/O operations. As detailed in table 4, this optimization reduces elapsed time for the *scriprmp* function of 40%. The overall optimizations of the sequential version of OASIS3, performed on both the *scriprmp* and the *extrap* functions, is shown in table 5. As previously described, the optimizations were mainly focused on reducing the I/O time. The main contribution to the optimization has been gained in the *scriprmp* function. The overall performance improvements is 27% of the whole coupling time.

4 Per field parallelization

In order to further reduce the elapsed time of coupling transformations, a parallel version of the algorithm has been developed. Adopted parallel approach is the master/slaves model. Since computation of each field is independent from the others, slaves do not communicate with each other. The master distributes

Table 3: *extrap* performance evaluation

| | Elapsed Time (sec) | Saved Time (sec) | % |
|-----------|--------------------|------------------|------|
| original | 286.218 | | |
| optimized | 285.032 | 1.186 | 0.41 |

Table 4: *scripr* performance evaluation

| | Elapsed Time (sec) | Saved Time (sec) | % |
|-----------|--------------------|------------------|-------|
| original | 617.129 | | |
| optimized | 367.615 | 249.514 | 40.43 |

```

Branch B
DO ind = 1, count
  ji = list_ji(ind)
  jj = list_jj(ind)
  iind (jj 1) * kxlon + ji
  pwork(ji,jj) = 0.
!cdir unroll=9
DO 250 jl = 1,9
  IF (zweights(knb,jl,iind) .NE. 0.) THEN
    idivi iaddress(knb, jl, iind) / kxlon
    imult idivi * kxlon
    IF (iaddress(knb,jl, iind) .EQ. imult) THEN
      ilat idivi
      ilon kxlon
    ELSE
      ilat idivi / imult
      ilon iaddress(knb,jl, iind)
    END IF
    $
    pwork(ji,jj) = pwork(ji,jj) + pfild(ilon,ilat)
    $
    zweights(knb,jl,iind)
  END IF
  CONTINUE
END DO
250

Branch A
DO 210 jj = 1, kxlon
  DO 221 ji = 1, kxlon, kxlon 1
    .
    .
    .
  IF (inbor .GE. ivoisin) THEN
    C
    C* Some points around P are not masked so we use them to extrapolate
    C* and define the iteration number, weight and address variables
    C
    pwork(ji,jj) = 0.
    iincre(knb, iind) = incre
    DO 243 jl = ideb, ifin
      ilon = ix(jl)
      ilat = iy(jl)
      pwork(ji,jj) = pwork(ji,jj)
        + pfild(ilon,ilat) * zmask(jl)
        / FLOAT(inbor)
      iaddress(knb,jl,iind) (ilat 1) * kxlon ilon
      zweights(knb,jl,iind) zmask(jl) / FLOAT(inbor)
    CONTINUE
  ENDIF
  CONTINUE
CONTINUE
ENDIF
CONTINUE
CONTINUE
210
  
```

Figure 4: *extrap* numerical displacement.

Table 5: OASIS3 optimization

| | Extrap | Script | Others | Coupling | Saved Time (sec) | % |
|-----------|---------|---------|--------|----------|------------------|-------|
| original | 286.218 | 617.130 | 1.008 | 904.360 | | |
| optimized | 285.032 | 367.620 | 1.018 | 653.670 | 250.690 | 27.72 |

fields to slaves and collects them after the execution of coupling operations, using the MPI library, as shown in figure 5. Each OASIS3 process is then in charge of computing all of the foreseen transformations for each assigned fields. The design of the parallel algorithm is driven by two main factors:

1. load balancing among OASIS3 processes;
2. maximum communication reduction.

It is necessary to consider that different fields could require different number and type of transformations; moreover they are also defined on different grids at different resolutions. This implies that coupling time cannot be considered constant for every field. Since computing time of each parallel task is not uniform and not known at compile time, a dynamic scheduling approach should be preferable (Quinn 2004). However, in this case, this choice should introduce an overhead of the same order of magnitude of computing time. For this reason, a static scheduling algorithm to distribute fields to available processes, has been implemented. Fields are allocated to processes taking into account the sequencing index (*SEQ*) and the presence of a correlation among fields. That happens when a field is a linear combination of other fields, using the *BLASNEW* and/or *BLASOLD* transformations. The sequencing index defines an order for fields to be transformed. It has been introduced to allow overlapping coupling time with models computing time. Indeed, fields sent to the coupler from faster models must be tagged, in the *namcouple* file, with smaller values of *SEQ*. This way, the OASIS3 coupling time spent over these fields is overlapped with computing time of slower models. This constraint introduces a temporal dependence among processes: the process performing transformation of a field with a high *SEQ* value should wait for those processes responsible for fields with a smaller sequencing index. To avoid some processes idle time, the scheduling policy must take

into account the *SEQ* value of each field: fields with the same *SEQ* must be uniformly distributed to the available processes. In this case, the maximum number of fields with the same sequencing index gives the maximum level of parallelism. Since the relationship among fields introduced by the use of *SEQ* is not a functional dependence, a field with a high *SEQ* value does not need to know results from those with smaller *SEQ* value; this implies that the sequencing order does not introduce communication among processes.

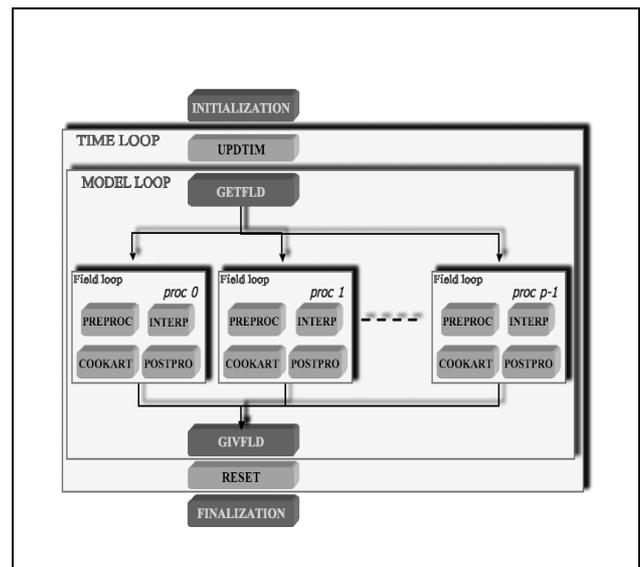


Figure 5: OASIS3 parallel approach.

If a given field *A* is a linear combination of one or more fields *B, C* (*BLASNEW* and *BLASOLD* transformations), the process performing *A* must wait for the transformations of *B* and *C* to be completed and

must communicate with the respective processes. In order to avoid communications, the scheduling algorithm aggregates fields with functional dependence and assigns them to the same process. Since coupling time taken by each field is not known at scheduling time, the algorithm aims at balancing the number of fields assigned to each process. The scheduling algorithm is structured as follow:

1. fields are aggregated in different groups G_{SEQ} according to the SEQ value;
2. within each G_{SEQ} , fields are further grouped in bundles $B_{SEQ,i}$ according to functional dependencies established by *BLASNEW* and *BLASOLD* transformations. Each bundle is ranked with an integer given by the cardinality of the set: $r_i = |B_{SEQ,i}|$;
3. for each G_{SEQ} , the bundles $B_{SEQ,i}$ are sorted by rank in descending order;
4. each MPI process is labeled with an integer l_j representing the number of fields currently assigned to process j . For each G_{SEQ} :
 - (a) l_j is initialized to zero;
 - (b) bundle $B_{SEQ,i}$, with maximum rank and not assigned yet, is associated with the process j having the minimum l_j ;
 - (c) the rank of bundle r_i is added to l_j ;
 - (d) the algorithm iterates from step b for each bundle belonging to current G_{SEQ} .

Unfortunately such an approach cannot guarantee a good load balancing for each configuration, since it assumes that each field takes the same coupling time. Moreover, the balancing is also influenced by the order of fields in the *namcouple* configuration file. More accurate algorithm should take into account the different coupling time requested by each field balancing the load according to it.

The resulting parallel algorithm is then structured as follow:

1. at the beginning of the simulation, the scheduling algorithm defines the sets of fields to be assigned to each available process, according to the actual configuration and taking into account SEQ values, *BLASNEW* and *BLASOLD* transformations;
2. at each coupling step, the master process of OASIS3 gets the fields from the models and scatters them to the slaves, according to the distribution policy established by the scheduling algorithm;
3. each slave process performs coupling transformations on the assigned fields and sends them to the master;
4. master process exports them to the models.

4.1 Parallel model

In this section, we define the analytic model of the execution time of our parallel algorithm. The coupled model elapsed time depends on many factors: number of processes assigned to a single component model; overhead introduced by communications among processes of a model (intra-model communication overhead); coupling transformations and so on. In this paper we focus only on the aspects affecting the coupler behavior. Elapsed time of the CMCC-MED model can be devised as the sum of the following components:

1. initialization of the computing environment;
2. time spent by component models, also including intra-model communications;
3. computing time to perform coupling transformations: it is important to properly evaluate this time, since it could partially overlap with computing time of component models;
4. communication overhead within the coupler: also this time could be partially overlapped with models computing time;
5. finalization of the simulation.

Since we are interested in the coupler parallel behavior, we establish the number of processes assigned to component models and considered time for executing models as intrinsically sequential, constant and independent from the number of processes assigned to the coupler. This choice is also supported by the consideration that the parallelization effort concerned only the coupler and not the whole coupled model. Moreover, a careful analysis leads us to infer that initialization and finalization operations cannot be parallelized. The intrinsically sequential time, T_{seq} , can be expressed as:

$$T_{seq} = T_{init} + T_{models} + T_{end} \quad (1)$$

hence the parallel time is given by:

$$T_{par} = T_{seq} + num_{couple} \cdot (T_{couple} + T_{com}) \quad (2)$$

where num_{couple} represents the total number of coupling steps occurring during the simulation; T_{couple} is the elapsed time required by the slowest process to transform its assigned fields; and T_{com} represents the communication overhead occurring in a coupling step to transfer fields from the OASIS3 master process to slaves and back.

The aforementioned SEQ values can be used to partially overlap coupling time with computing time spent by the component models. Let us define \mathcal{F}_i as the set of fields assigned to the process i . This set may contain fields with different SEQ values; it can also be thought as the union of disjointed subsets $\mathcal{G}_{i,j}$ containing fields assigned to process i with SEQ j . Defining s^* as the maximum SEQ value, we have:

$$\mathcal{F}_i = \bigcup_{j=1}^{s^*} \mathcal{G}_{i,j} \quad (3)$$

It is worth noting here that the coupling time depends only on the set of transformations applied to fields with the maximum value of SEQ ; hence T_{couple} can be expressed as:

$$T_{couple} = max_i \sum_{k \in \mathcal{G}_{i,s^*}} T_{tr_k} \quad (4)$$

where index i represents the process and T_{tr_k} is the elapsed time for coupling transformations applied on field k . Communication overhead has been modeled according to the standard linear communication model (Foster 1995),(Nupairoj et al. 1994). At each coupling step, OASIS3 takes T_{pp} time for point-to-point communications and T_{broad} time for broadcast communications. Thus,

$$\begin{aligned} T_{com} &= T_{pp} + T_{broad} \\ &= T_s(1 + \log 2p) \cdot n^* \\ &\quad + T_b \cdot \sum_{j \in \mathcal{G}_{i,s^*}} (L_{im_j} + L_{ex_j} \cdot \log_2 p) \end{aligned} \quad (5)$$

where T_s and T_b are machine dependent parameters and represent respectively the communication latency and inverse of the effective throughput of the communication channel. L_{im_j} and L_{ex_j} are the lengths (in bytes) of field j , used respectively during import and export operations; p is the number of involved processes. n^* is the highest cardinality (over the processes i) of set containing fields with value of SEQ equal to s^* , given by:

$$n^* = \max_i \{|\mathcal{G}_{i,s^*}|\} \quad (6)$$

5 Parallel performance analysis

The model previously described has been validated performing several tests on the NEC SX9 cluster available at the CMCC Supercomputing Center. Some preliminary tests have been executed in order to experimentally evaluate the latency and throughput of the communication channel. Since the architecture consists of 7 nodes and 16 processors for each node, both intra-node and inter-node communications may take place. However, we can safely generalize considering only inter-node communication. Indeed, the best configuration for the CMCC-MED would map the OASIS3 master process on the same node of the slowest component model master process (in order to minimize the time for communication among models and coupler). OASIS3 slaves processes must be mapped on different nodes. The features of the SX9 node are reported on table 6.

Table 6: NEC-SX9

| NEC SX-9 | |
|----------------------------|-----------------------|
| Performance per CPU | Over 100 GF |
| Machine cycle (clock) | 3.2 GHz |
| Memory bandwidth | 4 TB/s |
| Memory capacity per node | 512 GB |
| CPUs per node | 16 |
| Peak performance per node | 1.6 TF |
| I/O Data rate | 64 GB/s |
| Internode bandwidth (peak) | 128 GB/s x 2 |
| T_s | $3.40 \cdot 10^{-06}$ |
| T_b | $2.30 \cdot 10^{-11}$ |

Table 7: Sequential time

| | |
|-------------|-----------------------|
| Init Time | $2.08 \cdot 10^{+01}$ |
| Models Time | $3.67 \cdot 10^{+03}$ |
| End Time | $3.73 \cdot 10^{-05}$ |

It is worth reminding that performance analysis is mainly focused on the evaluation of the coupler parallelization; then the number of processes assigned to the component models has been pre-defined, changing the number of processes assigned to the coupler. The configuration we used is as follow:

- Ocean global: 1 processor on node A.
- Mediterranean sea: 6 processors on node A.
- Atmosphere: 8 processors on node A.
- Coupler: 1 processor on node A and $(p - 1)/2$ processors on nodes B and C.

Table 8: Parallel time

| # field (k) | T_{tr_k} (sec) | L_{ex_k} (byte) | L_{im_k} (byte) |
|-----------------------|----------------------|----------------------|----------------------|
| 1 | $4.56 \cdot 10^{-2}$ | 921600 | 216944 |
| 2 | $4.15 \cdot 10^{-2}$ | 921600 | 216944 |
| 3 | $4.30 \cdot 10^{-2}$ | 921600 | 216944 |
| 4 | $3.92 \cdot 10^{-2}$ | 921600 | 216944 |
| 5 | $3.93 \cdot 10^{-2}$ | 921600 | 216944 |
| 6 | $4.04 \cdot 10^{-2}$ | 921600 | 216944 |
| 7 | $1.27 \cdot 10^{-1}$ | 921600 | 1762904 |
| 8 | $1.25 \cdot 10^{-1}$ | 921600 | 1762904 |
| 9 | $1.26 \cdot 10^{-1}$ | 921600 | 1762904 |
| 10 | $4.82 \cdot 10^{-2}$ | 216944 | 921600 |
| 11 | $4.63 \cdot 10^{-2}$ | 216944 | 921600 |
| 12 | $4.61 \cdot 10^{-2}$ | 216944 | 921600 |
| 13 | $4.63 \cdot 10^{-2}$ | 216944 | 921600 |
| 14 | $4.84 \cdot 10^{-2}$ | 216944 | 921600 |
| 15 | $4.57 \cdot 10^{-2}$ | 216944 | 921600 |
| 16 | $4.66 \cdot 10^{-2}$ | 216944 | 921600 |
| 17 | $4.62 \cdot 10^{-2}$ | 216944 | 921600 |
| 18 | $2.67 \cdot 10^{-1}$ | 216944 | 921600 |
| 19 | $3.95 \cdot 10^{-2}$ | 216944 | 921600 |
| 20 | $2.64 \cdot 10^{-1}$ | 216944 | 921600 |
| 21 | $3.93 \cdot 10^{-2}$ | 216944 | 921600 |
| 22 | $3.93 \cdot 10^{-2}$ | 216944 | 921600 |
| 23 | $3.93 \cdot 10^{-2}$ | 216944 | 921600 |
| 24 | $4.26 \cdot 10^{-2}$ | 216944 | 921600 |
| 25 | $2.69 \cdot 10^{-2}$ | 216944 | 921600 |
| 26 | $2.68 \cdot 10^{-2}$ | 216944 | 921600 |
| 27 | $8.07 \cdot 10^{-2}$ | 1762904 | 921600 |
| 28 | $7.52 \cdot 10^{-2}$ | 1762904 | 921600 |
| 29 | $8.00 \cdot 10^{-2}$ | 1762904 | 921600 |
| 30 | $7.70 \cdot 10^{-2}$ | 1762904 | 921600 |
| 31 | $5.63 \cdot 10^{-2}$ | 1762904 | 921600 |
| 32 | $5.49 \cdot 10^{-2}$ | 1762904 | 921600 |
| 33 | $5.53 \cdot 10^{-2}$ | 1762904 | 921600 |
| 34 | $4.06 \cdot 10^{-2}$ | 1762904 | 921600 |
| 35 | $4.08 \cdot 10^{-2}$ | 1762904 | 921600 |

With this configuration, T_{seq} time components have been evaluated, as shown in table 7. Table 8 lists coupling time of each field.

In order to have a wide analysis range, we have imposed $SEQ = 1$ for each field, regardless of the speed of the component models; in this way, the number of processors ranges from 1 to 35, that is the total number of fields exchanged through the coupler.

The performance model demonstrated that scalability is heavily limited by the coarse grained parallelization based on both the distribution of the fields among the processors and the different kind and number of transformations performed on the fields. The scalability analysis shows that the algorithm reaches a 50% efficiency with 13 processors, corresponding to a computational load of about 3 fields per process. The developed parallel approach heavily influences the load balancing among processors. The communication overhead takes just almost 2% of the coupling time and it cannot be considered the limitation factor.

Figures 6-8 depict coupling time (on one simulated month $num_{couple} = 279$), speed-up and efficiency of the parallel algorithm with a number of processors ranging from 1 to 35. The analytic performance model approximates the real behavior of the algorithm with a standard deviation of 2.4%, hence

Table 9: Parallel OASIS3 performance evaluation

| # of procs | Execution Time (sec) | Efficiency | Speed up |
|------------|----------------------|------------|----------|
| 1 | 645.13 | 1.00 | 1.00 |
| 2 | 351.80 | 0.92 | 1.83 |
| 3 | 274.86 | 0.78 | 2.35 |
| 5 | 210.83 | 0.61 | 3.06 |
| 7 | 191.12 | 0.48 | 3.38 |
| 9 | 174.17 | 0.41 | 3.70 |
| 11 | 181.22 | 0.32 | 3.56 |
| 13 | 110.77 | 0.45 | 5.82 |
| 15 | 99.71 | 0.43 | 6.47 |
| 17 | 95.28 | 0.40 | 6.77 |
| 26 | 90.01 | 0.28 | 7.16 |
| 33 | 89.59 | 0.22 | 7.20 |

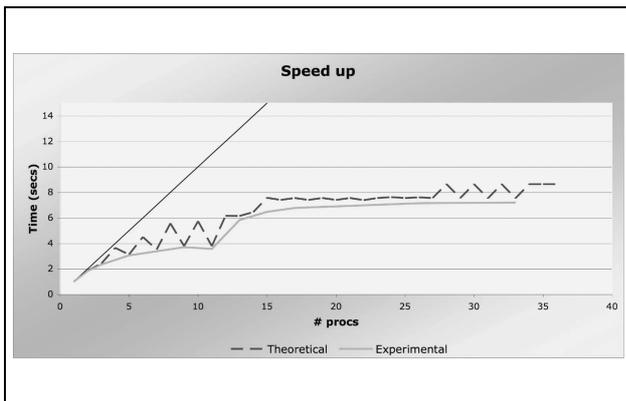


Figure 6: Parallel OASIS3 speedup.

it can be considered reliable. As confirmed by the swing trend of the speed-up and efficiency functions, the coarse grained parallelization produces worst performance when the number of fields is not perfectly divisible by the number of processes, whereas different number and kind of transformations deteriorate performance even if the number of fields is divisible by the number of processes (i.e. $p = 5, 7$). Experimental data obtained analyzing parallel performance is also reported in table 9. As we previously highlighted, a limit of our proposed approach is that the scheduling policy considers the time taken for coupling transformations constant for each field. Better performance could be achieved taking into account the different computational load required by applying transformations on different fields and trying to better balance the load among processors. But a per-field parallelization is still limited by the total number of fields. The highest level of parallelism can be achieved by combining the proposed approach with a parallelization based on a spatial domain decomposition. The timing model can be used for further considerations concerning the suitability of a distributed approach for this problem. In our case, two main reasons restrict the adoption of a distributed approach: (i) the parallelism level of the proposed algorithm is strongly limited by the number of fields to be transformed (it is rare that the number of exchanged fields is greater than 100); (ii) communication overhead in a distributed environment has a stronger impact on parallel performance. Several distributed approaches and frameworks exploit the architectures heterogeneity to improve the parallelization level. In this context different frameworks exist: MapReduce, GRIDSs, Condor are characterized by efficient mechanisms for managing re-

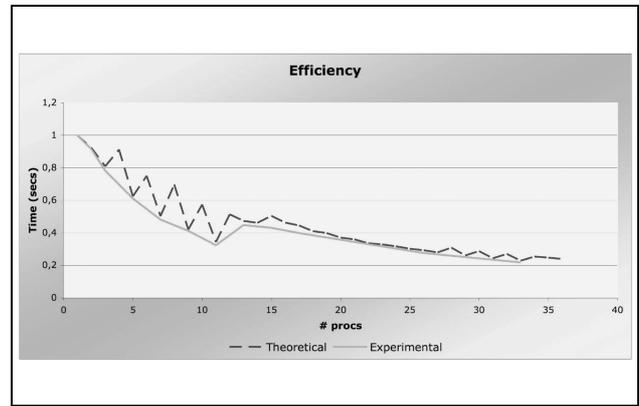


Figure 7: Parallel OASIS3 efficiency.

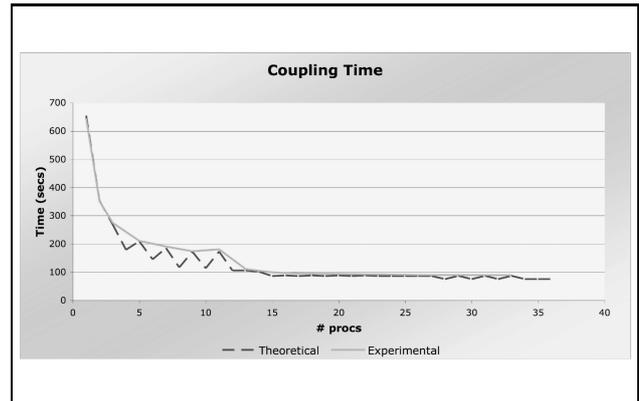


Figure 8: Parallel OASIS3 execution time.

sources, enhancing the fault tolerance and handling node heterogeneity. Generally, these frameworks use I/O operations for communication and hence they are not suitable for coupler parallelization since the communication overhead would exceed the computing time.

5.1 Implementation details

The implementation of the parallel algorithm has been fully integrated in the official version of the OASIS3 coupler, distributed by CERFACS. Code modification has been made minimizing the impact on the structure of the original code. Taking into account that the CLIM libraries, used by the coupler to communicate with the component models, supports both MPI1 and MPI2, the parallel model has been accordingly implemented. More in detail, with MPI1 (Gropp et al. 1996) implementation, a MPMD (Foster et al. 1997) approach is adopted; component models and coupler are executed launching different executables. Only the process itself then knows its "specialization"; an initialization step where the colour of models is exchanged allows each process to know masters and slaves of each model. A communicator for each model, including the coupler, is created using the `MPLComm_split` function.

The MPI2 (Gropp et al. 1998) implementation follows a different approach: with the `mpirun` command, only the coupler processes are instantiated. The executables names and the number of processes to be spawn for each component model are also passed through the command line to the OASIS3 executable. In this case, the OASIS3 communicator is duplicated from the `MPL_COMM_WORLD` at the beginning; other communicators are then created during the spawn of the corresponding processes.

The two implementations differ only on the management of the communicators. Once the coupler communicator has been created, communications are executed within it.

6 Comparison with the pseudo parallel version

A qualitative comparison between the proposed approach and the pseudo-parallel implementation of OASIS3 by CERFACS, has been performed. In the pseudo-parallel approach, each OASIS3 process must have its own *namcouple* file, carefully created by the modeler. Each process is then independent and unaware of the existence of others. It directly communicates with models exchanging fields included into its *namcouple* file. Such an approach implements a distributed communication with models. It avoids the bottleneck represented by a single master process in charge of both the exchange of all fields with the models and the coordination of the slaves. The manual definition of the *namcouple* file allows accurately distributing fields among the processes taking also into account the computational load required by each field. The main disadvantage of a pseudo-parallel approach regards the configuration. Indeed, the user is charged with the burden of creating *namcouple* files, every time the number of OASIS3 processes changes. Moreover, the parallel version of OASIS3 provides both MPI1 and MPI2 CLIM communication techniques, whereas the pseudo-parallel version only supports MPI1.

7 Evaluation of different scheduling policies

Our proposed approach for scheduling and for mapping the fields to the processors, suffers mainly because coupling time, for each field, is not known at compile time. Thus, the algorithm assigns each field the same weight. More efficient algorithms can be taken into consideration in order to reduce the parallel time. A dynamic scheduling algorithm would distribute fields to processes according to a request/response approach. At the beginning, one field for each process is assigned. The generic process i requests a new field to be transformed as soon as it ends the transformation of the current field. This approach generally behaves better with respect to the round-robin algorithm, but it is still influenced by the order of the fields in the *namcouple* configuration file. The best case for this dynamic approach is when fields are ordered from the most time consuming to the less one. In this case, the dynamic allocation of fields behaves exactly as the MaxMin approach (Maheswaran et al. 1999). The worst case happens when fields are sorted in descending order. Figure 9 depicts performance obtained with different scheduling approaches.

The MaxMin algorithm is a static approach, but it assumes that coupling time for each field is already known. Fields are sorted in descending order with respect to the coupling time and each field is assigned to the process with the current minimum computing load. This approach is the best one, but it requires a profiling phase in order to establish the coupling time for each field.

8 Conclusions

In this work, we presented optimization and parallelization of one of the most deployed coupler. Before dealing with parallelization of a code, it is necessary to deeply understand why it badly performs on the target architecture; that involves optimizations. The

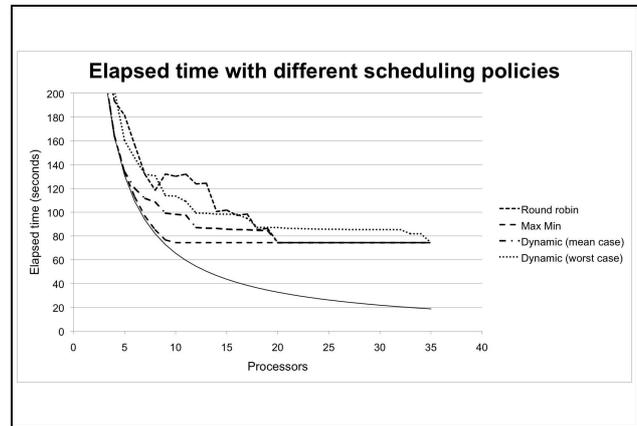


Figure 9: Elapsed time of the OASIS3 using different scheduling policies.

Table 10: Parallel OASIS3 improvements

| | Coupling Time (sec) | Saved Time (sec) | % |
|--------------------|---------------------|------------------|--------------|
| original | 904 | | |
| parallel (13 proc) | 110 | 794 | 87.83 |

profiling phase is mandatory to identify hot-spot functions and to drive optimization. Further level of improvement can be reached with parallelization, after a deep analysis of the algorithm and identification of both data and functional dependencies. In the case here discussed, with just the optimization and elimination of useless I/O operations, the coupling time has been reduced of 27%. Even if the parallelization strategy is coarse grained, it allowed a coupling time reduction up to 80% of the original sequential version, with 13 processors (see table 10).

As we expected, the coarse grained parallel approach cannot guarantee a good load balancing and it limits the level of parallelism. The counterpart is that communication overhead is minimum.

In order to enhance the parallel performance some improvements can be adopted:

- the scheduling algorithm can be modified in order to self adapt to computing requirements and to take into account coupling time of each field, allowing a better load balance. If a scheduling algorithm could know coupling time for each field, it should be able to better distribute load among processes. The scheduler can obtain this information by means of a profiling phase of the coupled model; otherwise, the scheduler could self adapt, keeping track of the time taken by each field to simulate a month and using this information for the scheduling policy of the next month;
- memory bank conflicts (about 40%) (NEC 2006) during OASIS3 execution on the vector machine could be resolved by means of a further optimization step. Bank conflicts occur when two or more processes try to simultaneously access to the same memory bank. The code can be suitably modified avoiding bank conflicts;
- OASIS4 (Valcke et al. 2007) is the new parallel version of the coupler, developed by CERFACS and based on a geographical domain decomposition of fields among processes. Performance evaluation of this new coupler can be performed using the CMCC-MED couple model. These two

parallel approaches can be integrated in a unique solution;

- the CMCC Supercomputing Center has also an IBM supercomputer with 10 power6 nodes for a total number of 960 cores. The performance evaluation of parallel OASIS3 on the scalar architecture can be performed in order to evaluate the behavior of the code on a many core system compared with a vector one;
- the parallel coupler has been validated on a set of available transformations. A complete test of available transformations is needed;
- climate change studies involve several coupled models. They are obtained using different climate models, but also different couplers. Performance comparison of parallel OASIS3 with other couplers such as the NCAR CPL coupler (Bryan et al. 1996) represents a further step to evaluate pros and cons of our approach.

Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J. (1996), *MPI: The Complete Reference*, The MIT Press.

Valcke S. (2006), OASIS3 User Guide, CERFACS.

Valcke S., Redler R. (2007), OASIS4 User Guide, CERFACS.

References

Euro-Mediterranean Centre for Climate Change
<http://www.cmcc.it>

Bryan F.O., Kauman B.G., Large W.G., Gent P.R. (1996), The NCAR CSM FluxCoupler, NCAR Technical Note NCAR/TN-424+STR, National Center for Atmospheric Research.

NEC Corporation (2006), SUPER-UX performance tuning guide.

Foster I., Geisler J., Tuecke S., Kesselman C. (1997), 'Multimethod Communication for High-Performance Metacomputing', *Proceedings of ACM/IEEE Supercomputing*, 1–10.

Foster I.T. (1995), *Designing and Building Parallel Programs : Concepts and Tools for Parallel Software Engineering*, Addison-Wesley.

Snir M., Huss-Lederman S., Lumsdaine A., Lusk E., Nitzberg B., Saphir W., Snir M. (1998), *MPI The complete reference: Volume 2, the MPI-2 extensions*, The MIT Press.

Madec G. (2008), NEMO ocean engine, Institut Pierre-Simon Laplace (IPSL).

Madec G., Delecluse P., Imbard M. & Levy C. (1998), OPA 8.1 Ocean General Circulation Model Reference Manual, Institut Pierre-Simon Laplace (IPSL).

Maheswaran M., Ali S., Siegel H. J., Hensgen D., Freud R. (1999), 'Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems', *8th Heterogeneous Computing Workshop (HCW99)*.

Nupairoj N., Ni L.M. (1994), 'Performance Evaluation of Some MPI Implementations on Workstation Clusters', *Proceedings of the 1994 Scalable Parallel Libraries Conference (SPLC94)*, 98–105.

Quinn M. (2004), *Parallel programming in c with mpi and openmp*, McGraw Hill.

Roeckner E., Brokopf R., Esch M., Giorgetta M., Hagemann S., Kornblueh L., Manzini E., Schlese U. & Schulzweida U. (2004), The atmospheric general circulation model ECHAM5, Max Planck Institute (MPI).

Classification of Malware Using Structured Control Flow

Silvio Cesare and Yang Xiang

School of Management and Information Systems
Centre for Intelligent and Networked Systems
Central Queensland University
Rockhampton, Queensland 4702, Australia

silvio.cesare@gmail.com; y.xiang@cqu.edu.au

Abstract

Malware is a pervasive problem in distributed computer and network systems. Identification of malware variants provides great benefit in early detection. Control flow has been proposed as a characteristic that can be identified across variants, resulting in flowgraph based malware classification. Static analysis is widely used for the classification but can be ineffective if malware undergoes a code packing transformation to hide its real content. This paper proposes a novel algorithm for constructing a control flow graph signature using the decompilation technique of structuring. Similarity between structured graphs can be quickly determined using string edit distances. To reverse the code packing transformation, a fast application level emulator is proposed. To demonstrate the effectiveness of the automated unpacking and flowgraph based classification, we implement a complete system and evaluate it using synthetic and real malware. The evaluation shows our system is highly effective in terms of accuracy in revealing all the hidden code, execution time for unpacking, and accuracy in classification.

Keywords: Network security, malware, structured control flow, unpacking.

1 Introduction

Malware, short for malicious software, means a variety of forms of hostile, intrusive, or annoying software or program code. Malware is a pervasive problem in distributed computer and network systems. Detection of malware is important to a secure distributed computing environment. The predominant technique used in commercial anti-malware systems to detect an instance of malware is through the use of malware signatures. Malware signatures attempt to capture invariant characteristics or patterns in the malware that uniquely identifies it. The patterns used to construct a signature have traditionally derived from the malware's machine code and raw file contents.

Traditional malware signatures ineffectively capture the invariant characteristics common in self-mutating and modified variants in a strain of malware. Static analysis provides alternative characteristics that can be used.

Static analysis incorporating ngrams (Kolter and Maloof 2004, Karim et al. 2005), edit distances (Gheorghescu 2005), and control flow (Carrera and Erdélyi 2004, Dullien and Rolles 2005, Briones and Gomez 2008) have been proposed. A malware's control flow information provides static analysis a characteristic that is identifiable across strains of malware variants. This characteristic is shared because variants of malware often reuse code from earlier strains and versions. This reuse of code can be identified through isomorphic and similar flow graphs.

To hinder static analysis, the malware's real content is frequently hidden using a code transformation known as packing. Packing is not solely used by malware. Packing is also used in software protection schemes and file compression for legitimate software, yet the majority of malware also uses the code packing transformation. In one month during 2007, 79% of identified malware was packed (Panda Research 2007). Additionally, almost 50% of new malware in 2006 were repacked versions of existing malware (Stepan 2006).

Unpacking is a necessary component to perform static analysis and reveal the hidden characteristics of malware. In the problem scope of unpacking, it can be seen that many instances of malware utilise identical or similar packers. Many of these packers are also public, and malware often employs the use of these public packers. Many instances of malware also employ modified versions of public packers. Being able to automatically unpack malware in any of these scenarios, in addition to unpacking novel samples, provides benefit for static analysis to occur.

Automated unpacking relies on typical behaviour seen in the majority of packed malware – hidden code is dynamically generated and then executed. The hidden code is naturally revealed in the process image during normal execution. Monitoring execution for the dynamic generation and execution of the malware's hidden code can be achieved through emulation. Emulation provides a safe and isolated environment for malware analysis.

In this paper we present a system that employs dynamic and static analysis to automatically unpack and classify a malware instance as a variant, based on similarities of control flow graphs.

This paper makes the following contributions. First, we propose a novel algorithm for approximate identification of flowgraphs based on treating decompiled structured flowgraphs as signatures. These signatures can then be used to query a malware database for approximate matches, using the edit distance to indicate similarity. Second, we propose and evaluate automated

unpacking using application level emulation that is fast enough for potential integration into desktop Antivirus. The automated unpacker is capable of unpacking a known samples and also capable of unpacking unknown samples. We also propose a novel algorithm for determining when to stop emulation during unpacking using entropy analysis. Finally, we implement and evaluate our ideas in a prototype system that performs automated unpacking and malware classification.

The structure of this paper is as follows. Section 2 describes related work in automated unpacking and malware classification. Section 3 refines the problem definition and our approach to the proposed complete classification system. Section 4 describes the design and implementation of our prototype system. Section 5 evaluates our prototype using synthetic and real malware samples. Finally, Section 6 summarizes and concludes the paper.

2 Related Work

Automated unpacking employing whole system emulation was proposed in Renovo (Kang et al. 2007) and Pandora's Bochs (Boehne 2008). Whole system emulation has been demonstrated to provide effective results against unknown malware samples, yet is not completely resistant to novel attacks. Renovo and Pandora's Bochs both detect execution of dynamically generated code to determine when unpacking is complete and the hidden code is revealed. An alternative algorithm for detecting when unpacking is complete was proposed using execution histograms in Hump-and-dump (Sun et al. 2008). The Hump-and-dump was proposed as potentially desirable for integration into an emulator. Polyunpack (Royal et al. 2006) proposed a combination of static and dynamic analysis to dynamically detect code at runtime which cannot be identified during an initial static analysis. The main distinction separating our work from previously proposed automated unpackers is our use of application level emulation and an aggressive strategy to determine that unpacking is complete. The advantage of application level emulation over whole system emulation is significantly greater performance. Application level emulation for automated unpacking has had commercial interest (Graf 2005) but has realized few academic publications evaluating its effectiveness and performance.

Dynamic Binary Instrumentation was proposed as an alternative to using an instrumented emulator (Quist and Valsmith 2007) employed by Renovo and Pandora's Bochs. Omnipack (Martignoni et al. 2007) and Saffron (Quist and Valsmith 2007) proposed automated unpacking using native execution and hardware based memory protection features. This results in high performance in comparison to emulation based unpacking. The disadvantage of these approaches is in the use of the unpacking system on E-Mail gateways, which forces the provision of a virtual or emulated sandbox in which to run. A virtual machine approach to unpacking using x86 hardware extensions was proposed in Ether (Dinaburg et al. 2008). The use of such a virtual machine and equally to whole system emulator is the requirement to install a license for each guest operating system. This

restricts desktop adoption which typically has a single license. Virtual machines are also inhibited by slow start-up times, which again are problematic for desktop use. The use of a virtual machine also prevents the system being cross platform as the guest and host CPUs must be the same.

Malware classification has been proposed using a variety of techniques (Kolter and Maloof 2004, Karim et al. 2005, Perdisci et al. 2008). A variation of ngrams, coined nperms has been proposed (Karim et al. 2005) to describe malware characteristics and subsequently used in a classifier. An alternative approach is using basicblocks of unpacked malware, classified using edit distances, inverted indexes and bloom filters (Gheorghescu 2005). The main disadvantage of these approaches is that minor changes to the malware source code can result in significant changes to the resulting bytestream after compilation. This change can significantly impact the classification. Flowgraph based classification is an alternative method that attempts to solve this issue by looking at control flow as a more invariant characteristic between malware variants.

The commercial automated unpacking and structural classification system Vxclass (Zynamics) is most related to our research. Vxclass presents a system for unpacking and malware classification based on similarity of flowgraphs. The algorithm in Vxclass is based on approximately matching flowgraphs by identifying fixed points in the graphs and successively matching neighbouring nodes (Carrera and Erdélyi 2004, Dullien and Rolles 2005, Briones and Gomez 2008). BinHunt (Gao et al. 2008) provides a more thorough test of flowgraph similarity by soundly identifying the maximum common subgraph, but at reduced levels of performance and without application to malware classification. Identifying common subgraphs of fixed sizes can also indicate similarity and has better performance (Kruegel et al. 2006).

Our research differs from previous flowgraph classification research by using a novel approximate control flow graph matching algorithm. Except for Krueger et al., the classification systems in previous research measure similarity in the callgraph and control flow graphs, where as our work relies entirely on the control flow graphs. Also distinguishing our work is the proposed automated unpacking system, which is integrated into the flowgraph based classification system.

3 Problem Definition and Our Approach

The problem of malware classification and variant detection is defined in this section. Additionally, an overview of our approach to design and implement a malware classification system is presented.

3.1 Problem Definition

A malware classification system is assumed to have advance access to a set of known malware. This is for construction of an initial malware database. The database is constructed by identifying invariant characteristics in each malware and generating an associated signature to be stored in the database. After database initialization, normal use of the system commences. The system has as

input a previously unknown binary that is to be classified as being malicious or non malicious. The input binary and the initial malware binaries may have additionally undergone a code packing transformation to hinder static analysis. The classifier calculates similarities between the input binary against each malware in the database. The similarity is measured as a real number between 0 and 1 - 0 indicating not at all similar, 1 indicating an identical match. This similarity is based on the similarity between malware characteristics in the database. If the similarity exceeds a given threshold for any malware in the database, then the input binary is deemed a variant of that malware, and therefore malicious. If identified as a variant, the database may be updated to incorporate the potentially new set of generated signatures associated with that variant.

3.2 Our Approach

Our approach employs both dynamic and static analysis to classify malware. Entropy analysis initially determines if the binary has undergone a code packing transformation. If packed, dynamic analysis employing application level emulation reveals the hidden code using entropy analysis to detect when unpacking is complete. Static analysis then identifies characteristics, building signatures for control flow graphs using the novel application of structuring. Structuring is the process of decompiling unstructured control flow into higher level, source code like, constructs including structured conditions and iteration. Each signature representing the structured control flow is represented as a string. These signatures are then used for querying the database of known malware using an approximate dictionary search. Using the edit distance for approximate matches between each flowgraph, a similarity between flowgraphs can be constructed. The similarity of each control flowgraph in the program is accumulated to construct the final measure of program similarity and variant identification.

4 System Design and Implementation

In this section, the design and implementation of the malware classification system prototype is examined.

4.1 Identifying Packed Binaries Using Entropy Analysis

The malware classification system performs an initial analysis on the input binary to determine if it has undergone a code packing transformation. Entropy analysis (Lyda and Hamrock 2007), is used to identify packed binaries. The entropy of a block of data describes the amount of information it contains. It is calculated as follows:

$$H(x) = - \sum_{i=1}^N \begin{cases} p(i) = 0, & 0 \\ p(i) \neq 0, & p(i) \log_2 p(i) \end{cases}$$

where $p(i)$ is the probability of the i^{th} unit of information in event x 's sequence of N symbols. For malware packing analysis, the unit of information is a byte value, N is 256, and an event is a block of data from the malware. Compressed and encrypted data have relatively high

entropy. Program code and data have much lower entropy. Packed data is typically characterised as being encrypted or compressed, therefore high entropy in the malware can indicate packing.

An analysis most similar to Uncover (Wu et al. 2009) is employed. Identification of packed malware is established if there exists sequential blocks of high entropy data in the input binary.

If the binary is identified as being packed, then the dynamic analysis to perform automated unpacking proceeds. If the binary is not packed, then the static analysis commences immediately.

4.2 Application Level Emulation

Automated unpacking requires malware execution to be simulated so that the malware may reveal its hidden code. The hidden code once revealed is then extracted from the process image.

Application level emulation provides an alternate approach to whole system emulation for automated unpacking. Application level emulation simulates the instruction set architecture and system call interface. In the Windows OS, the officially supported system call interface is the Windows API.

4.2.1 Interpretation

The prototype emulator utilises interpretation to perform simulation. The features of the prototype are described in this section.

x86 Instruction Set Architecture (ISA): Much of the 32-bit x86 ISA has been implemented in the prototype. Extensions to the ISA, including SSE and MMX instructions, have been partially implemented. A partial implementation is adequate for the prototype as the majority of programs do not employ full use of the ISA. FPU, SSE, and MMX instructions are primarily used by malware to evade or detect emulation. Malware may also use the debugging interface component of the ISA, including debug registers and the trap flag, which are primarily used to obfuscate control flow.

Virtual Memory: x86 employs a segmented memory architecture. The Windows OS utilises these segment registers to reference thread specific data. Thread specific data is additionally used by Windows Structured Exception Handling (SEH). SEH is used to gracefully handle abnormal conditions such as division by zero and is routinely used by packers and malware to obfuscate control flow.

Segmented memory is handled in our prototype by maintaining a table of segment descriptions, known in the x86 ISA as the descriptor table. Addressed memory is associated with a segment, known in the ISA as segment selectors, which hold an index into the descriptor table. This enables a translation from segmented addressing to a flat linear addressing.

Virtual memory is maintained by a table of memory regions referenced by their linear address. Each memory region maintains its associated memory contents. Each region also maintains a shadow memory that is utilised by the automated unpacking logic. The shadow memory maintains a flag for each address that is set if that location has been written to or of it has been read.

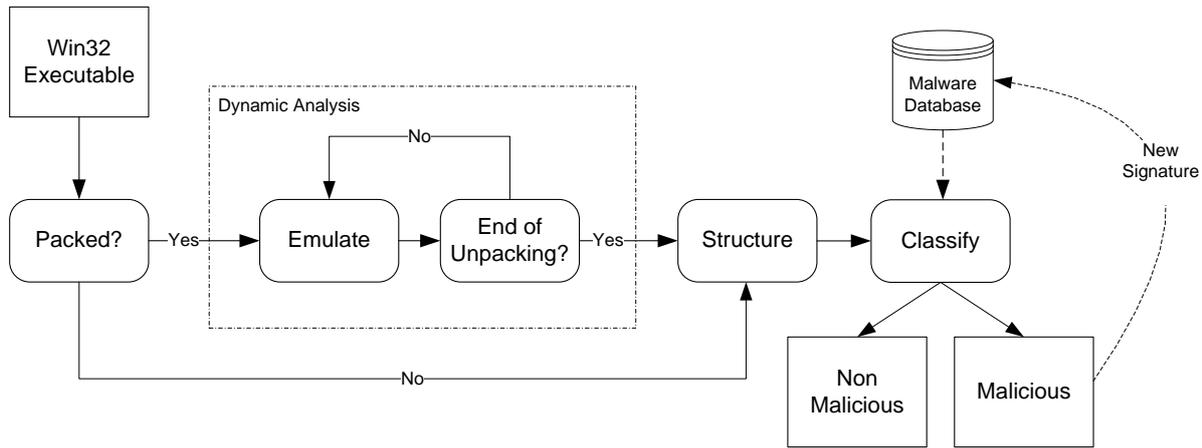


Figure 1: Block diagram of the malware classification system.

Windows API: The Windows API is the official system call interface provided by Windows. Our prototype system detects calls to the Windows API by inspecting the simulated program counter. If the program counter contains the address of a Windows API function, then a handler implementing the functionality of the API is executed.

There are too many windows API functions to fully emulate, so only the most common APIs are implemented. Commonly used APIs include heap management, object management, and file system management.

Linking and Loading: Program loading entails allocating the appropriate virtual memory, loading the program text, data and dynamic libraries and performing any required relocations. OS specific structures and machine state must also be initialized.

The exported functions of a dynamically linked library may be entirely simulated without having access to the native library. Such a system may have benefit when the emulator is cross platform and licensing issues should be avoided. Our implementation performs full dynamic library loading using the native libraries. This is done to provide a more faithful simulation.

Thread and Process Management: Multithreading in applications must be emulated. The prototype implements this using user-level threads - only one thread is running on the host at any particular time and each thread is rescheduled after a specific number of instructions.

Support for emulating multiple processes was not implemented.

OS specific structures: Windows has process and thread specific structures that require initialization such as the Process Environment Block, Thread Environment Block, and Loader Module. These structures are visible to applications and can be used by malware.

4.2.2 Improvements to Emulation

A naive implementation of emulation can result in poor simulation speed. We make a number of improvements to the prototype as follows. We also make additional improvements to enable a mechanism to address anti-emulation code used by malware.

Instruction predecoding (Sharif et al.) is adopted and produces a significant gain in simulation speed. In this technique, the decoding of unique instructions is cached. This results in a performance gain because disassembly in a naive emulator consumes a large amount of processor time. Predecoding can also be used to cache a function pointer directly to the opcode handler. When used in this way, predecoding allows for fast implementation of the x86 debugging ISA including hardware breakpoints and single step execution used by debuggers. In this optimisation, the cache holding a function pointer to the opcode handler is modified on-demand to reflect that it should execute the breakpoint or trap logic. This removes explicit checks for these conditions from the emulator's main loop.

Condition Codes: The x86 condition codes are another point of optimisation and the prototype defers to lazy evaluation of these at the time of their use, similar to QEMU (Bellard 2005).

Emulating Known Sections of Code: Many instances of malware use modified variants of the same packer or share similar code between different packers. Taking advantage of this, it is possible to detect known sections of code during emulation and handle them more specifically, and therefore more efficiently than interpretation (Babar et al. 2009). To implement this it is noted that each stage during unpacking gives access to a layer of hidden code that has been revealed, and the memory in each layer can be searched for sections of known code. These sections of code can then be emulated, in whole, using custom handlers. This approach achieves significantly greater performance than interpreting each individual instruction. Typical code sections to have written handlers include decryption loops, decompression loops and checksum calculations. Handlers can also be written and used to dynamically remove specific anti-emulation code.

The prototype implements handlers for frequently used loops in several well known packers.

4.2.3 Verification of Emulation

An automated approach to testing the correctness of emulation is implemented similar to that of testing whole system emulation (Martignoni et al. 2009). To achieve this, the program being emulated is executed in parallel

on the host machine. The host program is monitored using the Windows debugging API. At the commencement of each instruction, the emulator machine state is compared against the host version and examined for deviant behaviour. This allows the detection of unfaithful simulation.

Faithful emulation is made more difficult, as some instructions and Windows API functions behave differently when debugged. The prototype debugger was modified to rewrite these instructions and functions to emit behaviour consistent to that in a non debugged environment. This enabled testing of packers and malware that employ known techniques to detect and evade debugging.

4.3 Entropy Analysis to Detect Completion of Hidden Code Extraction

Detection of the original entry point (OEP) during emulation identifies the point at which the hidden code is revealed and execution of the original unpacked code begins to take place. Detecting the execution of dynamic code generation by tracking memory writes was used as an estimation of the original entry point in Renovo (Kang et al. 2007). In this approach the emulator executes the malware, and a shadow memory is maintained to track newly written memory. If any newly written memory is executed, then the hidden code in the packed binary being executed will be revealed. To complicate this approach, multiple layers or stages of hidden code may be present, and malware may be packed more than once. This scenario is handled by clearing the shadow memory contents, continuing emulation, and repeating the monitoring process until a timeout expires.

The malware classification prototype extends the concept of identifying the original entry point when unpacking multiple stages by identifying more precisely at which stage to terminate the process, without relying on a timeout. The intuition behind our approach is that if there exists high entropy packed data that has not been used by the packer during execution, then it remains to be unpacked. To determine if a particular stage of unpacking represents the original entry point, the entropy of new or unread memory in the process image is examined. Newly written memory is indicated by the shadow memory for the current stage being unpacked. Unread memory is maintained globally, in a shadow memory for all stages. If the entropy of the analysed data is low, then it is presumed that no more compressed or encrypted data is left to be unpacked. This heuristically indicates completion of unpacking. The prototype also performs the described entropy analysis to detect unpacking completion after a Windows API imposes a significant change to the entropy. This is commonly seen when the packer deallocates large amounts of memory during unpacking. In the remaining case that the original entry point is not identified at any point, an attempt in the emulation to execute an unimplemented Windows API function will have the same effect as having identified the original entry point at this location.

4.4 Flowgraph Based Signature Generation

The static analysis component of the malware classification system proceeds once it receives an unpacked binary. The analysis is used to extract characteristics from the input binary that can be used for classification. The characteristic for each procedure in the input binary is obtained from structuring its control flow into compact representation that is amenable to string matching.

To initiate the static analysis process, the memory image of the binary is disassembled using speculative disassembly (Kruegel et al. 2004). Procedures are identified during this stage. A heuristic is used to eliminate incorrectly identified procedures during speculation of disassembly - the target of a call instruction identifies a procedure, only if the callsite belongs to an existing procedure. Once disassembled the disassembly is translated into an intermediate representation. Using an intermediate representation is not strictly necessary; however the malware classification prototype is built on a more general binary analysis platform which uses the intermediate form. The intermediate representation is used to generate a control flow graph for each identified procedure. The control flow graph is then structured into a signature represented as a character string. The signature is also associated with a weight relative to all sum of all weighted signatures in the program. The weight of procedure x is formally defined as:

$$weight_x = \frac{\text{len}(s_x)}{\sum_i \text{len}(s_i)}$$

where s_i is signature of procedure i in the binary.

4.4.1 Signatures using Structured Control Flow for Approximate Matches

Malware classification using approximate matches of signatures can be performed, and intuitively, using approximate matches of a control flow graph should enable identification a greater number of malware variants. In our approach we use structuring. Structuring is the process of recovering high level structured control flow from a control flow graph. In our system, the control flow graphs in a binary are structured to produce signatures that are amenable to comparison and approximate matching using edit distances.

The intuition behind using structuring as a signature is that similarities between malware variants are reflected by variants sharing similar high level structured control flow. If the source code of the variant is a modified version of the original malware, then this intuition would appear to hold true.

The structuring algorithm implemented in the prototype is a modified algorithm of that proposed in the DCC decompiler (Cifuentes 1994). If the algorithm cannot structure the control flow graph then an unstructured branch is generated. Surprisingly, even when graphs are reducible (a measure of how inherently structured the graph is), the algorithm generates unstructured branches in a small but not insignificant number of cases. Further improvements to this algorithm

to reduce the generation of unstructured branches have been proposed (Moretti et al. 2001, Wei et al. 2007), however these improvements were not implemented.

The result of structuring is output consisting of a string of character tokens representing high level structured constructs that are typical in a structured programming language. Subfunction calls are represented, as are gotos, however the goto and subfunction targets are ignored. The grammar for a resulting signature is defined in figure 3.

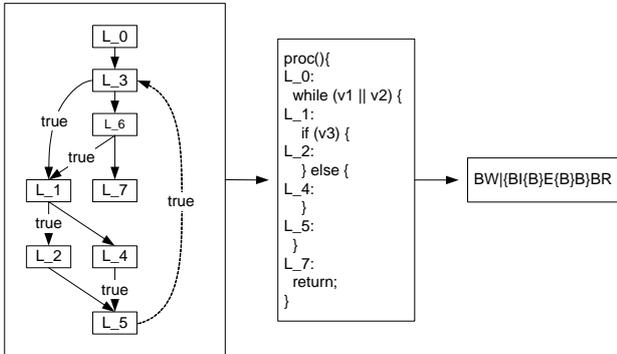


Figure 2: The relationship between a control flow graph, a high level structured graph, and a signature.

| | |
|-------------------|---|
| Procedure | ::= StatementList |
| StatementList | ::= Statement Statement StatementList |
| Statement | ::= Return Break Continue Goto Conditional Loop BasicBlock |
| Goto | ::= 'G' |
| Return | ::= 'R' |
| Break | ::= 'B' |
| Continue | ::= 'C' |
| BasicBlock | ::= 'B' 'B' SubRoutineList |
| SubRoutineList | ::= 'S' 'S' SubRoutineList |
| Condition | ::= ConditionTerm ConditionTerm NextConditionTerm |
| NextConditionTerm | ::= '!' Condition Condition |
| ConditionTerm | ::= '&' ' ' |
| IfThenCondition | ::= Condition '!' Condition |
| Conditional | ::= IfThen IfThenElse |
| IfThen | ::= 'I' IfThenCondition '{' StatementList '}' |
| IfThenElse | ::= 'I' Condition '{' StatementList '}' 'E' '{' StatementList '}' |
| Loop | ::= PreTestedLoop PostTestedLoop EndlessLoop |
| PreTestedLoop | ::= 'W' Condition '{' StatementList '}' |
| PostTestedLoop | ::= 'D' '{' StatementList '}' Condition |
| EndlessLoop | ::= 'F' '{' StatementList '}' |

Figure 3: The grammar to represent a structured control flow graph signature.

4.5 Malware Classification

To classify an input binary, the analysis makes use of a malware database. The database contains the signatures, represented as structured graphs, of known malware. For each procedure in the input binary, the database is queried using an approximate dictionary search. The results are accumulated to give a measure of similarity between the input binary and malware in the database.

The analysis performs more accurately with a greater number of procedures and hence signatures. If the input binary has too few procedures, then classification cannot be performed. The prototype does not perform classification on binaries with less than 10 procedures.

To perform the approximate dictionary search, any signature within an allowable number of errors is identified as a positive match. The edit distance between signatures gives the number of errors, and the search is performed using BK Trees (Baeza-Yates and Navarro 1998). The search algorithm is faster than an exhaustive comparison of each signature in the dictionary. In our prototype, we maintain a separate database for each malware instance and perform classification against all instances one at a time. This has resulted in greater performance than using a global database of all malware.

The similarity ratio (Gheorghescu 2005) was proposed to measure the similarity between basic blocks. It is used in our research to establish the number of allowable errors between flowgraph signatures in a dictionary search. For two signatures or structured graphs represented as strings x and y , the similarity ratio is defined as:

$$w_{ed} = 1 - \frac{ed(x,y)}{\max(len(x), len(y))}$$

where $ed(x,y)$ is the edit distance. Our prototype defines the edit distance as the Levenshtein distance – the number of insertions, deletions, and substitutions to convert one string to another. Signatures that have a similarity ratio equal or exceeding a threshold t ($t=0.9$) are identified as positive matches. This figure was derived empirically through a pilot study. Using the similarity ratio s as a threshold, the number of allowable errors in signature x is defined as $len(x)(1 - s)$.

For a particular malware, once a matching graph is found, this graph is ignored for subsequent searches of the remaining graphs in the input binary. If a graph has multiple matches in a particular malware and it is uncertain which procedure should be selected as a match, the greedy solution is taken. The graph that is weighted the most is selected. Two weights are possible for each procedure from the input binary during dictionary queries, as the procedure may be considered belonging to either the input binary, or the malware binary in the database.

For each malware that has matching signatures, the similarity ratios of those signatures are accumulated proportional to their weights. As two weights are possible, this results in calculating two asymmetric similarities: a similarity that identifies how much of the input binary is approximately found in the database malware, and a similarity to show how much of the database malware is approximately found in the input binary. Formally, the asymmetric similarity is:

$$S_x = \sum_i \begin{cases} 0, & w_{ed_i} < t \\ w_{ed_i} weight_{x_i}, & w_{ed_i} \geq t \end{cases}$$

where t is the empirical threshold value of 0.9, w_{ed_i} is the similarity ratio between the i^{th} control flow graph of

the input binary and the matching graph in the malware database, and $weight_{x_i}$ is the weight of the cfg where x is either the input binary or the malware binary in the database.

The program similarity is the final measure of similarity used for classification and is the product of the asymmetric similarities. The program similarity is defined as:

$$S(i, d) = S_i S_d$$

where i is the input binary, d is the database malware instance, S_i and S_d are the asymmetric symmetries.

Program similarity is not symmetric and therefore $S(i, d)$ and $S(d, i)$ are not guaranteed the same result, although in practice they are generally identical. This difference is due to the use of heuristics, as a control flow graph signature can potentially be found to have multiple matches when calculating the similarity between two binaries. The heuristic used is to select a single signature. A greedy selection is taken, based on the weights of the matching signatures. In $S(i, d)$, the input binary defines the weights used in the greedy selection. Alternatively, in $S(d, i)$, the database malware defines the weights which will be used. Because the weights and therefore greedily selected signatures can be different, the resulting program similarities may be different.

If the program similarity of the examined program to any malware in the database equals or exceeds a threshold of 0.6, then it is deemed to be a variant. As the database contains only malicious software, the binary of unknown status is also deemed malicious. The threshold of 0.6 was chosen empirically through a pilot study. If the binary is identified as malicious, and not deemed as excessively similar to an existing malware in the database, the new set of malware signatures can be stored in the database as part of an automatic system.

4.6 Discussion

It is possible to generate a signature using a faster and simpler method than structuring (Carrera and Erdélyi 2004). This approach takes note that if the signatures of two graphs are not the same, then the graphs are not isomorphic. The converse is used to indicate matching graphs. To generate a signature, the algorithm orders the nodes in the control flow graph, e.g. using a depth first order. A signature subsequently consists of a list of graph edges using the ordered nodes as node labels. The potential advantage of this method is that classification using exact matches of signatures can be performed very efficiently.

Another potential design change is using an alternative method to establish the program similarity. It may be desirable to identify only that malware is approximately found as a subset in another binary, in which case a single asymmetric similarity may be used. This has use in virus detection.

Automated unpacking can potentially be thwarted to result in malware that cannot be unpacked. Application level emulation presents inherent deficiencies when implemented to emulate the Windows operating system. The Windows API is a large set of APIs that requires significant effort to faithfully emulate. Complete

emulation of the API has not been achieved in the prototype and faithful emulation of undocumented side effects may be near impossible. Malware that circumvents usual calling mechanisms and malware that employs the use of uncommon APIs may result in incomplete emulation. Malware is reportedly more frequently using the technique of uncommon APIs to evade Antivirus emulation.

An alternative approach is to emulate the Native API which is used by the Windows API implementation. However, the only complete and official documentation for system call interfaces is the Windows API. The Windows API is a library interface, but malware may employ the use of the Native API to interface directly with the kernel. There does exist reported malware that employ the Native API to evade Antivirus software.

Another problem that exists is early termination of unpacking due to time constraints. Due to real-time constraints of desktop Antivirus, unpacking may be terminated if too much time is consumed during emulation. Malware may employ the use of code which purposely consumes time for the purpose of causing early termination of unpacking. Dynamic binary translation may provide some relief through faster emulation. Additionally, individual cases of anti-emulation code may be treated using custom handlers to perform the simulation where anti-emulation code is detected.

Application level emulation performs optimally against variations of known packers, or unknown packers that do not introduce significantly novel anti-emulation techniques. Many newly discovered malware fulfil these criteria.

Malware classification has inherent problems also, and may fail to perform correctly. Performing static disassembly, identifying procedures and generating control flow graphs is, in the general case, undecidable. Malware may specifically craft itself to make static analysis hard. In practice, the majority of malware is compiled from a high level language and obfuscated as a post-processing stage. The primary method of obfuscation is the code packing transformation. Due to these considerations, static analysis generally performs well in practice.

5 Evaluation

In this section we describe experiments that were used to evaluate automated unpacking and flowgraph based classification using our prototype.

5.1 Unpacking Synthetic Samples

OEP Detection: To verify our system correctly performs hidden code extraction, we tested the prototype against 14 public packing tools. These tools perform various techniques in the resulting code packing transformation including compression, encryption, code obfuscation, debugger detection and virtual machine detection. The samples chosen to undergo the packing transformation were the Microsoft Windows XP system binaries `hostname.exe` and `calc.exe`. `hostname.exe` is 7680 bytes in size, and `calc.exe` is 114688 bytes.

The original entry point identified by the unpacking system was compared against what was identified as the

real OEP. To identify the real OEP, the program counter was inspected during emulation and the memory at that location examined. If the program counter was found to have the same entry point as the original binary, and the 10 bytes of memory at that location the same as the original binary, then that address was designated the real OEP.

The results of the OEP detection evaluation are in table 1. The revealed code column in the tabulated results identifies the size of the dynamically generated code and data. The number of unpacking stages to reach the real OEP is also tabulated, as is the number of stages actually unpacked using entropy based OEP detection. Finally, the percentage of instructions that were unpacked, compared to the number of instructions that were executed to reach the real OEP is also shown. This last metric is not a definitive metric by itself, as the result of the unaccounted for instructions may not affect the revelation of hidden code – the instructions could be only used for debugger evasion for example. Entries where the OEP was not identified are marked with err. Binaries that failed to pack correctly are marked as fail. The closer the results in column 3 and 4 the better. The higher the result in column 5 the better.

| Name | Revealed code and data | Number of stages to real OEP | Stages unpacked | % of instr. to real OEP unpacked |
|------------------|------------------------|------------------------------|-----------------|----------------------------------|
| upx | 13107 | 1 | 1 | 100.00 |
| rlpack | 6947 | 1 | 1 | 100.00 |
| mew | 4808 | 1 | 1 | 100.00 |
| fsg | 12348 | 1 | 1 | 100.00 |
| npack | 10890 | 1 | 1 | 100.00 |
| expressor | 59212 | 1 | 1 | 100.00 |
| packman | 10313 | 2 | 1 | 99.99 |
| pe compact | 18039 | 4 | 3 | 99.98 |
| acprotect | 99900 | 46 | 39 | 98.81 |
| winupack | 41250 | 2 | 1 | 98.80 |
| telock | 3177 | 19 | 15 | 93.45 |
| yoda's protector | 3492 | 6 | 2 | 85.81 |
| aspack | 2453 | 6 | 1 | 43.41 |
| pespin | err | 23 | err | err |

hostname.exe

| Name | Revealed code and data | Number of stages to real OEP | Stages unpacked | % of instr. to real OEP unpacked |
|------------------|------------------------|------------------------------|-----------------|----------------------------------|
| upx | 125308 | 1 | 1 | 100.00 |
| rlpack | 114395 | 1 | 1 | 100.00 |
| mew | 152822 | 2 | 2 | 100.00 |
| fsg | 122936 | 1 | 1 | 100.00 |
| npack | 169581 | 1 | 1 | 100.00 |
| expressor | fail | fail | fail | fail |
| packman | 188657 | 2 | 1 | 99.99 |
| pe compact | 145239 | 4 | 3 | 99.99 |
| acprotect | 251152 | 209 | 159 | 96.51 |
| winupack | 143477 | 2 | 1 | 95.84 |
| telock | fail | fail | fail | fail |
| yoda's protector | 112673 | 6 | 3 | 95.82 |
| aspack | 227751 | 4 | 2 | 99.90 |
| pespin | err | 23 | err | err |

calc.exe

Table 1: Metrics on identifying the original entry point in packed samples.

The results show that unpacking most of the samples reveals some or most of the hidden code as expected. The OEP of pespin was not identified, possibly due to unused

encrypted data remaining in the process image, which would raise the entropy and affect the heuristic OEP detection. The OEP in the packed calc.exe samples was more accurately identified, relative to the metrics, than in the hostname.exe samples. This may be due to fixed size stages during unpacking that were not executed due to incorrect OEP detection. Interestingly, in many cases, the revealed code was greater than the size of the original unpacked sample. This is because the metric for hidden code is all the code and data that is dynamically generated. Use of the heap, and the dynamic generation of internally used hidden code will increase the resultant amount.

The worst result was in hostname.exe using aspack. 43% of the instructions to the real OEP were not executed, yet nearly 2.5K of hidden code and data was revealed, which is around a third of the original sample size. While some of this may be heap usage and the result not ideal, it may still potentially result in enough revealed procedures to use for the classification system in the static analysis phase.

Performance: The system used to evaluate the performance of the unpacking prototype was a modern desktop - a 2.4 GHz Quad core computer, with 4G of memory, running 32bit Windows Vista Home Premium with Service Pack 1. The performance of the unpacking system, shown in table 2. The running time is total time minus start-up time of 0.60s. Binaries that failed to pack correctly are marked as fail.

| Name | Time (s) | Num. Instr. |
|------------------|----------|-------------|
| mew | 0.13 | 56042 |
| fsg | 0.13 | 58138 |
| upx | 0.11 | 61654 |
| packman | 0.13 | 123959 |
| npack | 0.14 | 129021 |
| aspack | 0.15 | 161183 |
| pe compact | 0.14 | 179664 |
| expressor | 0.20 | 620932 |
| winupack | 0.20 | 632056 |
| yoda's protector | 0.15 | 659401 |
| rlpack | 0.18 | 916590 |
| telock | 0.20 | 1304163 |
| acprotect | 0.67 | 3347105 |
| pespin | 0.64 | 10482466 |

hostname.exe

| Name | Time (s) | Num. Instr. |
|------------------|----------|-------------|
| mew | 1.21 | 12691633 |
| fsg | 0.23 | 964168 |
| upx | 0.19 | 1008720 |
| packman | 0.28 | 1999109 |
| npack | 0.40 | 2604589 |
| aspack | 0.51 | 4078540 |
| pe compact | 0.83 | 7691741 |
| expressor | fail | fail |
| winupack | 0.93 | 7889344 |
| yoda's protector | 0.24 | 2620100 |
| rlpack | 0.56 | 7632460 |
| telock | fail | fail |
| acprotect | 0.53 | 5364283 |
| pespin | 1.60 | 27583453 |

calc.exe

Table 2: Running time to perform unpacking.

The results demonstrate the system is fast enough for integration into a desktop anti-malware system. In this evaluation full interpretation of every instruction is performed.

5.2 Flowgraph Based Malware Classification

The malware classification prototype was evaluated using real malware samples. The samples were chosen to mimic a selection of the malware and evaluation metrics in previous research (Carrera and Erdélyi 2004). Netsky, Klez, and Roron malware variants were obtained through public databases. A number of the malware samples were packed. The classification system automatically identifies and unpacks such malware as necessary. Table 3 evaluates the complete system. Highlighted cells identify a malware variant, defined as having a similarity equal or exceeding 0.60. A flowgraph is classed as being a variant of another flowgraph if the similarity ratio is equal or in excess of 0.9.

| | a | b | c | d | g | h |
|---|------|------|------|------|------|------|
| a | | 0.84 | 1.00 | 0.76 | 0.47 | 0.47 |
| b | 0.84 | | 0.84 | 0.87 | 0.46 | 0.46 |
| c | 1.00 | 0.84 | | 0.76 | 0.47 | 0.47 |
| d | 0.76 | 0.87 | 0.76 | | 0.46 | 0.45 |
| g | 0.47 | 0.46 | 0.47 | 0.46 | | 0.83 |
| h | 0.47 | 0.46 | 0.47 | 0.45 | 0.83 | |

The Klez family of malware.

| | aa | ac | f | j | p | t | x | y |
|----|------|------|------|------|------|------|------|------|
| aa | | 0.78 | 0.61 | 0.70 | 0.47 | 0.67 | 0.44 | 0.81 |
| ac | 0.78 | | 0.66 | 0.75 | 0.41 | 0.53 | 0.35 | 0.64 |
| f | 0.61 | 0.66 | | 0.86 | 0.46 | 0.59 | 0.39 | 0.72 |
| j | 0.70 | 0.75 | 0.86 | | 0.52 | 0.67 | 0.44 | 0.83 |
| p | 0.47 | 0.41 | 0.46 | 0.52 | | 0.61 | 0.79 | 0.56 |
| t | 0.67 | 0.53 | 0.59 | 0.67 | 0.61 | | 0.61 | 0.79 |
| x | 0.44 | 0.35 | 0.39 | 0.44 | 0.79 | 0.61 | | 0.49 |
| y | 0.81 | 0.64 | 0.72 | 0.83 | 0.56 | 0.79 | 0.49 | |

The Netsky family of malware.

Table 3: The similarity matrix for two malware families (e.g. Klez.a).

| | ao | b | d | e | g | k | m | q | a |
|----|------|------|------|------|------|------|------|------|------|
| ao | | 0.41 | 0.27 | 0.27 | 0.27 | 0.46 | 0.41 | 0.41 | 0.44 |
| b | 0.41 | | 0.27 | 0.26 | 0.27 | 0.48 | 1.00 | 1.00 | 0.56 |
| d | 0.27 | 0.27 | | 0.44 | 0.50 | 0.27 | 0.27 | 0.27 | 0.27 |
| e | 0.27 | 0.26 | 0.44 | | 0.56 | 0.26 | 0.26 | 0.26 | 0.26 |
| g | 0.27 | 0.27 | 0.50 | 0.56 | | 0.26 | 0.27 | 0.27 | 0.26 |
| k | 0.46 | 0.48 | 0.27 | 0.26 | 0.26 | | 0.48 | 0.48 | 0.73 |
| m | 0.41 | 1.00 | 0.27 | 0.26 | 0.27 | 0.48 | | 1.00 | 0.56 |
| q | 0.41 | 1.00 | 0.27 | 0.26 | 0.27 | 0.48 | 1.00 | | 0.56 |
| a | 0.44 | 0.56 | 0.27 | 0.26 | 0.26 | 0.73 | 0.56 | 0.56 | |

Similarity ratio threshold of 1.0.

| | ao | b | d | e | g | k | m | q | a |
|----|------|------|------|------|------|------|------|------|------|
| ao | | 0.70 | 0.28 | 0.28 | 0.27 | 0.75 | 0.70 | 0.70 | 0.75 |
| b | 0.74 | | 0.31 | 0.34 | 0.33 | 0.82 | 1.00 | 1.00 | 0.87 |
| d | 0.28 | 0.29 | | 0.50 | 0.74 | 0.29 | 0.29 | 0.29 | 0.29 |
| e | 0.31 | 0.34 | 0.50 | | 0.64 | 0.32 | 0.34 | 0.34 | 0.33 |
| g | 0.27 | 0.33 | 0.74 | 0.64 | | 0.29 | 0.33 | 0.33 | 0.30 |
| k | 0.75 | 0.82 | 0.29 | 0.30 | 0.29 | | 0.82 | 0.82 | 0.96 |
| m | 0.74 | 1.00 | 0.31 | 0.34 | 0.33 | 0.82 | | 1.00 | 0.87 |
| q | 0.74 | 1.00 | 0.31 | 0.34 | 0.33 | 0.82 | 1.00 | | 0.87 |
| a | 0.75 | 0.87 | 0.30 | 0.31 | 0.30 | 0.96 | 0.87 | 0.87 | |

Default similarity ratio threshold of 0.9.

Table 4: The similarity matrices for the Roron family of malware (e.g. Roron.ao).

| | cmd.exe | calc.exe | netsky.aa | klez.a | roron.ao |
|-----------|---------|----------|-----------|--------|----------|
| cmd.exe | | | 0.00 | 0.00 | 0.00 |
| calc.exe | 0.00 | | 0.00 | 0.00 | 0.00 |
| netsky.aa | 0.00 | 0.00 | | 0.19 | 0.08 |
| klez.a | 0.00 | 0.00 | 0.19 | | 0.15 |
| roron.ao | 0.00 | 0.00 | 0.08 | 0.15 | |

Table 5: The similarity matrix for non similar malware and programs.

The results demonstrate that the system finds high similarities between samples. Table 4 shows the difference in the similarity matrix when the threshold for the similarity ratio is increased to 1.0. Differences between 5% and 30% were noted across the variety of malware variants using the two similarity ratio thresholds.

To evaluate the generation of false positives in the classification system table 5 shows classification among non similar binaries. Table 6 shows a more thorough evaluation of false positive generation by comparing each executable binary to every other binary in the Windows Vista system directory. The histogram groups binaries that shares similarity in buckets grouped in intervals of 0.1. There results show there exist similarities between some of the binaries, but for the majority of comparisons the similarity is less than 0.1. This seems a reasonable result as most binaries will be unrelated.

| Similarity | Matches |
|------------|---------|
| 0.0 | 105497 |
| 0.1 | 2268 |
| 0.2 | 637 |
| 0.3 | 342 |
| 0.4 | 199 |
| 0.5 | 121 |
| 0.6 | 44 |
| 0.7 | 72 |
| 0.8 | 24 |
| 0.9 | 20 |
| 1.0 | 6 |

Table 6: Histogram of similarities between executable files in Windows system directory.

6 Conclusion

Malware can be classified according to similarity in its flowgraphs. This analysis is made more challenging by packed malware. In this paper we proposed fast algorithms to unpack malware using application level emulation, and perform malware classification using the edit distance between structured control flow graphs. We implemented and evaluated a prototype. It was demonstrated that the automated unpacking system was fast enough for desktop integration. The automated unpacking was also demonstrated to work against a promising number of synthetic samples using known packing tools, with high speed. To detect the completion of unpacking, we proposed and evaluated the use of entropy analysis. It is shown that our system can successfully identify variants of malware.

7 References

Babar, K., Khalid, F. & Pakistan, P. (2009): Generic Unpacking Techniques. *International Conference On Computer, Control and Communication*.

Baeza-Yates, R. & Navarro, G. (1998): Fast approximate string matching in a dictionary. *South American Symposium on String Processing and Information Retrieval (SPIR'98)*, 14-22.

Bellard, F. (2005): QEMU, a fast and portable dynamic translator. *USENIX Annual Technical Conference*, 41-46.

- Boehne, L. (2008): Pandora's Bochs: Automatic Unpacking of Malware. University of Mannheim.
- Briones, I. & Gomez, A. (2008): Graphs, Entropy and Grid Computing: Automatic Comparison of Malware. *Virus Bulletin Conference*, 1-12.
- Carrera, E. & Erdélyi, G. (2004): Digital genome mapping—advanced binary malware analysis. *Virus Bulletin Conference*, 187-197.
- Cifuentes, C. (1994): Reverse compilation techniques. Queensland University of Technology.
- Dinaburg, A., Royal, P., Sharif, M. & Lee, W. (2008): Ether: Malware analysis via hardware virtualization extensions. *Proceedings of the 15th ACM conference on Computer and communications security*, 51-62, ACM New York, NY, USA.
- Dullien, T. & Rolles, R. (2005): Graph-based comparison of Executable Objects (English Version). *SSTIC*.
- Gao, D., Reiter, M. K. & Song, D. (2008): Binhunt: Automatically finding semantic differences in binary programs. *Information and Communications Security*, **5308**:238–255, Springer.
- Gheorghescu, M. (2005): An automated virus classification system. *Virus Bulletin Conference*, 294-300.
- Graf, T. (2005): Generic unpacking: How to handle modified or unknown PE compression engines. *Virus Bulletin Conference*.
- Kang, M. G., Poosankam, P. & Yin, H. (2007): Renovo: A hidden code extractor for packed executables. *Workshop on Recurring Malcode*, 46-53.
- Karim, M. E., Walenstein, A., Lakhoria, A. & Parida, L. (2005) Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, **1** (1):13-23.
- Kolter, J. Z. & Maloof, M. A. (2004): Learning to detect malicious executables in the wild. *International Conference on Knowledge Discovery and Data Mining*, 470-478.
- Kruegel, C., Kirda, E., Mutz, D., Robertson, W. & Vigna, G. (2006) Polymorphic worm detection using structural information of executables. *Lecture notes in computer science*, **3858**:207.
- Kruegel, C., Robertson, W., Valeur, F. & Vigna, G. (2004): Static disassembly of obfuscated binaries. *USENIX Security Symposium*, **13**:18-18.
- Lyda, R. & Hamrock, J. (2007) Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, **5** (2):40.
- Martignoni, L., Christodorescu, M. & Jha, S. (2007): Omniunpack: Fast, generic, and safe unpacking of malware. *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 431-441.
- Martignoni, L., Paleari, R., Roglia, G. F. & Bruschi, D. (2009): Testing CPU emulators. *Proceedings of the eighteenth international symposium on Software testing and analysis*, Chicago, IL, USA, 261-272, ACM.
- Moretti, E., Chanteperdrix, G. & Osorio, A. (2001): New algorithms for control-flow graph structuring. *Software Maintenance and Reengineering*, 184.
- Mal(ware)formation statistics - Panda Research Blog: Panda Research, http://research.pandasecurity.com/archive/Mal_2800_ware_2900_formation-statistics.aspx. 19 August 2009.
- Perdisci, R., Lanzi, A. & Lee, W. (2008): McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables. *Proceedings of the 2008 Annual Computer Security Applications Conference*, 301-310, IEEE Computer Society Washington, DC, USA.
- Quist, D. & Valsmith (2007): Covert Debugging Circumventing Software Armoring Techniques. *Black Hat Briefings USA*.
- Royal, P., Halpin, M., Dagon, D., Edmonds, R. & Lee, W. (2006): Polyunpack: Automating the hidden-code extraction of unpack-executing malware. *Computer Security Applications Conference*, 289-300.
- Sharif, M., Lanzi, A., Giffin, J. & Lee, W. Rotalume: A Tool for Automatic Reverse Engineering of Malware Emulators. INC., I. G.
- Stepan, A. (2006): Improving proactive detection of packed malware. *Virus Bulletin Conference*, **1**.
- Sun, L., Ebringer, T. & Boztas, S. (2008): Hump-and-dump: efficient generic unpacking using an ordered address execution histogram. *International Computer Anti-Virus Researchers Organization (CARO) Workshop*.
- Wei, T., Mao, J., Zou, W. & Chen, Y. (2007): Structuring 2-way branches in binary executables. *International Computer Software and Applications Conference*, **01**: 115-118.
- Wu, Y., Chiueh, T. & Zhao, C. (2009): Efficient and Automatic Instrumentation for Packed Binaries. *International Conference and Workshops on Advances in Information Security and Assurance*, 307-316.
- VxClass: Zynamics, <http://www.zynamics.com/vxclass.html>.

Lazy Evaluation of PDE Coefficients in the EScript System

Joel Fenwick

Lutz Gross

Earth Systems Science Computational Centre (ESSCC),
The University of Queensland,
St Lucia, QLD 4072, Australia.
Email: JoelFenwick@uq.edu.au

Abstract

EScript is an extension to Python for solving partial differential equations on parallel computers. It is parallelised for both MPI and shared memory, multi-core systems using OpenMP. In this paper, we discuss *lazy evaluation* as a strategy to reduce the cost of evaluating the coefficients of PDEs prior to solving. We show that our implementation provides significant memory and time savings for a problem involving complex expressions.

1 Introduction

EScript is a Python extension for solving general linear, steady state, second order partial differential equations (PDEs) (Gross et al. 2007). It supports parallel execution for OpenMP, MPI or both (source code is available on launchpad (lpe 2009)). The goal of *escript* is to provide users, who are primarily modellers, with a means to construct and run simulations on parallel computers without needing expertise in parallel programming or lower level languages such as *C++*. With this in mind, the work described in this paper had three competing objectives:

1. reduced run time.
2. reduced peak memory usage.
3. minimal disruption to the existing interface and minimal work on the part of users to apply new features.

EScript and the simulations built upon it, contain a significant amount of Python code. Any changes we make, must be able to work in an interactive (and interpreted) Python session if required. This interpreted aspect, while making experimentation and scripting easier for non-programmers, does impose some constraints on the methods used to improve performance. In particular, this limits techniques which require preprocessing or precompilation of scripts.

To describe a PDE in *escript*, functions must be constructed (or loaded) to form its coefficients. The coefficients may depend on a PDE solution from a previous iteration or timestep. They may also depend

This work is supported by the AuScope National Collaborative Research Infrastructure Strategy by the Australian Commonwealth, the Queensland State Government and The University of Queensland.

Copyright ©2010, Australian Computer Society, Inc. This paper appeared at the 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), Brisbane, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 107, Jinjun Chen and Rajiv Ranjan, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

on the solutions to a different PDE in the case of a coupled problem. In this paper, we discuss the representation and evaluation of these coefficient functions.

By default, functions are stored explicitly. That is, the representation stores a number of floating point values proportional to the complexity of the domain. The more operations required to construct a function and the more complex the objects involved, the greater the amount of memory required to store intermediate functions. As a very simple example, consider the expression

$$y = ax + b.$$

In order to evaluate y , an intermediate result

$$\tau = ax$$

is required. If a and x are simple values then storing τ won't present much of a problem. If they are more complex objects though, storing their product could require non-trivial amounts of memory. Such intermediate functions are typically not required once the final function is computed. For large domains and intermediate functions involving tensors, this memory cost can be significant. This can limit the problem sizes which can be handled by the system. We investigate the use of *lazy evaluation* to reduce the burden on the system. Lazy evaluation means that "arguments to a function are evaluated only when needed for computation" (Pandey 2008).

After a brief comment on OpenMP, we will discuss data representation followed by threading. Section 5 contrasts evaluation in functional environments to *escript*. Section 6 describes two implementations of the resolve operation. Performance experiments follow in Section 7.

2 OpenMP

The OpenMP (omp 2009a) model of shared-memory parallelism uses a team of threads. Code executes on a single master thread until it reaches a parallel region. All threads in the team execute the section. At the end of the section, execution continues on the master thread (omp 2009b). In the case of *escript*, OpenMP is used primarily to parallelise *for* loops.

3 Representation

EScript can be compiled to use MPI, OpenMP or both (hybrid mode). In the case of MPI, function information is distributed among the MPI processes at creation time and processed independently. There may be an aggregation step at the end for operations such as integration. Since MPI does not present threading issues, we will ignore it for the purposes of this discussion.

Functions are represented in two parts (input and output). The input to the function is determined by a Domain object (a mesh) and a point selection strategy (called a FunctionSpace in *escript* terminology). For example, the values of the function could be derived from the nodes bounding an element or from the points in the interior. Storing this information is the responsibility of the Domain object. Note that *escript* itself does not impose any meaning on particular FunctionSpace IDs; these are determined by the Domain in use.

Interpolation between different FunctionSpaces (where possible), is performed by the Domain object. Regardless of the FunctionSpace in use, the collection of points representing an element is termed a *sample*.

It should be noted that while the ordering of samples and points must be consistent, *escript* does not assume a global ordering of points.¹

Data objects represent the values (outputs) of functions and are linked to a particular Domain and FunctionSpace. Values can be scalars (rank²0) through to 4-Tensors (rank 4) of various shapes. Mathematical operations including basic arithmetic, matrix operations and tensor products are defined on Data objects. The majority of computation in *escript* itself acts on Data objects and evaluating expressions involving Data is the focus of this paper.

The final component is the solver. In the *escript* distribution, the PDE is converted to a sparse matrix representation and passed to the *Paso* sparse matrix solver. This paper will not directly discuss the performance of Paso either.

Operations on Data objects could be thought of as batch operations in that the same operation is performed on each value of the collection. Since the values in the Data object (function output) must correspond to points in the Domain, it might be tempting to view Data as an array. But this analogy breaks down when the Domain is distributed across multiple processes.

Internally, Data objects act as proxies for instances of the DataAbstract class which actually store the values. This allows flexibility in switching representations without disturbing the rest of the system. For the rest of the paper, we will use “node” to refer to instances of DataAbstract rather than to parts of a mesh.

For the purposes of this paper, Data objects reference one of two types of nodes: Ready and Lazy. *Ready* (or *non-lazy*) nodes store their values explicitly. *Lazy* nodes represent maths operations and link to other lazy nodes in a *directed acyclic graph* (DAG) to represent expressions. Special *identity* nodes are used to wrap ready nodes so they can be added to the graph. The acyclic property of the graph is guaranteed because a lazy node can only be formed using existing nodes. For example, if three Data variables P , Q , R which store their values in nodes $V1$, $V2$, $V3$ (Figure 1(a)), then the expressions:

$$S = P + Q$$

$$T = S/R$$

result in the DAG shown in Figure 1(b).

Once in the DAG, a node is never modified except when a sub-expression is replaced by a ready node. The *root* of an expression is the node directly referenced by the Data object (which might not be a root in the underlying DAG). The roots in Figure 1 are the nodes pointed to by dotted lines.

¹This is particularly true when MPI is involved and the same sampleID may represent different entities on different machines.

²We follow the terminology from Python where the rank of an object is the number of indices required to identify a component. The shape of an object is the range of values for each index.

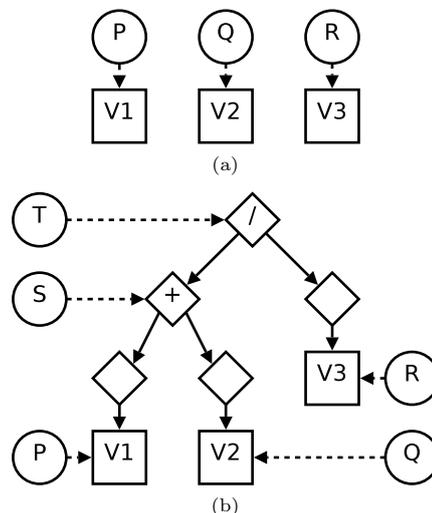


Figure 1: A DAG representing an expression. Circles represent Data objects, squares represent Ready nodes and diamonds represent lazy nodes. Empty diamonds represent identity operations. Solid lines are DAG edges, dotted lines show ownership of nodes.

The lifetimes of interrelated Domain, FunctionSpace and DataAbstract nodes are managed using shared pointers from *Boost* (boo 2009). So for example, Data have a shared pointer to their DataAbstract node while FunctionSpaces hold a shared pointer to their Domain. (*Boost* also allows us to make these wrappers transparent to Python.)

Sharing DataAbstract nodes between Data objects makes for efficient copy and return operations but it introduces complications.³ When nodes are shared or incorporated into a lazy expression, they must keep their values from that point in time. Users should not need to be aware that a particular Data object uses shared values, so *escript* implements transparent copy on write (COW)(Glass & Ables 2003). Any Data method which modifies values, checks to see if it is the sole owner of the node. If not, it makes a copy of the node, transfers ownership to a it and modifies the copy instead.

4 Threading

EScript employs the following threading model. The Python layer (both user scripts and *escript* modules) is assumed to be single threaded. Some objects and methods are implemented in C++/C and these are parallelised using OpenMP and MPI.

No multithreading apart from OpenMP is used. Hence, there are no threading issues to consider apart from those within individual parallel regions. Because some of the memory allocation is based on the number of threads in use, we assume that the number of threads available to OpenMP does not change. For this reason we also do not use nested OpenMP parallel regions.

In theory, the sole owner requirement for COW could be checked using the `.unique()` method on *Boost* pointers but threading issues prevented this. Specifically, it proved difficult to differentiate between a node being passed from one owner to another and a node with two owners. Instead, we maintain a list of which Data objects own which nodes. To simplify

³“Copies” here refers to objects created at the C++ level (possibly indirectly by calling Python methods), not the Python assignment statement (which leads to two references to the same object).

matters involving large DAGs which encode a number of expressions, once a node becomes part of a lazy expression it is assumed to always be shared.

5 Lazy Evaluation

Lazy evaluation is “popular in functional programming languages (those with no effects) and rarely found elsewhere” (Friedman & Wand 2008). Hudak’s 1989 survey (Hudak 1989) lists lazy evaluation as one of the distinguishing features of modern functional languages. His definition includes the idea that a particular expression be evaluated at most once.

The Haskell functional language (Thompson 1999) uses this form of laziness (outermost function application evaluated first and expressions only evaluated once) for its expression processing. For functions defined by conditionals, Haskell only performs enough evaluation to determine which branch of the condition to take. That is, it performs short circuit evaluation.

In the case of *escript*, there is a single conditional type operation (a masked copy) and it does not currently permit short circuit or lazy evaluation (although it could be simply modified to do so).

The main benefits of lazy evaluation for functional languages are (Thompson 1999)(Hudak 1989)(Pandey 2008):

1. Expressions need not be evaluated at all if they are not required.
2. Since lists do not always *need* to be represented explicitly, programs can be written to deal with conceptually infinite lists.

Our requirements differ from the functional setting. We do not need to process infinite objects. We do allow large objects to be examined a chunk at a time though, and our operations make use of this capability. However, user scripts must request this operation explicitly for their own use.

Our expressions only include values and operations from a predefined set. As such, they cannot contain arbitrary user defined functions. Our implementation links directly to values, so there are no names (functions or variables) which require forming a closure. We also differ from the functional version in that expressions may (depending on implementation strategy) be evaluated more than once. This is because caching the complete result is something we wish to avoid.

6 Resolution

Values of a function represented as lazy data can be resolved in two ways. Either the function can be queried one sample at a time or it can be *completely resolved*, that is all samples are evaluated and the result stored as a new function. Because of the extra memory required, we wish to avoid complete resolution as much as possible. Note that the process for assembling a sparse matrix to solve, only requires one sample at a time.

In the current build, Lazy data will be completely resolved when one of the following occurs:

- `.resolve()` is called on the expression.
- An attempt is made to set the value of the function at a point.⁴
- A masked copy is attempted.
- An operation is performed which depends on all points for its value, such as `integrate()`.

⁴This operation is permitted in *escript* but not encouraged.

- The expression becomes too large.

Apart from the first point, these may change in the future. The size constraint refers to either the total number of descendants or the height of the expression root (or both).

Some care must be taken when acting on lazy or shared data because *escript* may need to resolve or duplicate the data. In both cases memory may be allocated, deallocated and ownership may change. For this reason, we require that issues of resolution and node creation be settled before entering parallel sections. This restriction is only of concern to implementors.

We have implemented two strategies for evaluating samples in Lazy expressions. In both cases, the expression is evaluated using a recursive post-order traversal.

6.1 Temporary Buffer (Method 1)

A buffer is created and passed into the evaluation to function somewhat like a stack. To evaluate each node, space is reserved for the result and the remaining space is available to evaluate the node’s children. Evaluation returns a pointer to the buffer (and an offset within the buffer) containing the result. This is to avoid copying data from non-lazy nodes under the Identity operation. The size required for the buffer is computed using a modified Sethi-Ullman register labelling algorithm (Appel & Palsberg 2002). Once the result has been retrieved, the buffer can be disposed of.

6.2 Per-Node Cache (Method 2)

Each node has a buffer big enough to store a single sample for each OpenMP thread. When the value of a sample for a lazy node is computed, it is stored in this buffer. Again we make an exception for identity nodes which wrap non-lazy values. The final value of the sample can be retrieved from the root node of the expression. The size limits on expressions described above were introduced primarily for Method 1 and should be relaxed or removed for this method.

7 Performance

The experiments for this paper were carried out on a single compute node⁵ of an SGI ICE 8200 EX. The code was compiled with support for OpenMP threading but not MPI. Please note that understanding the applications from which these tests are derived is not necessary to understand the performance changes.

7.1 Experiment 1 — Power Law

Here a script to compute power law values (Muhlhaus & Regenauer-Lieb 2005) was run for a single OpenMP thread⁶. See Appendix A for details of the script. Both lazy resolution methods were tested. Figure 2 shows the peak memory and real time use for runs with various numbers of elements. The values plotted are averaged over ten runs. The memory use for the two lazy methods is reasonably similar, although Method 1 (temporary buffer) is slightly smaller. The run time for Method 1 was significantly higher than non-lazy while Method 2 (node cache) was only slightly higher.

⁵32 GB RAM and two quad-core 2.8Ghz Xeon processors. The version of *escript* was repository revision 2532.

⁶That is, `OMP_NUM_THREADS=1`.

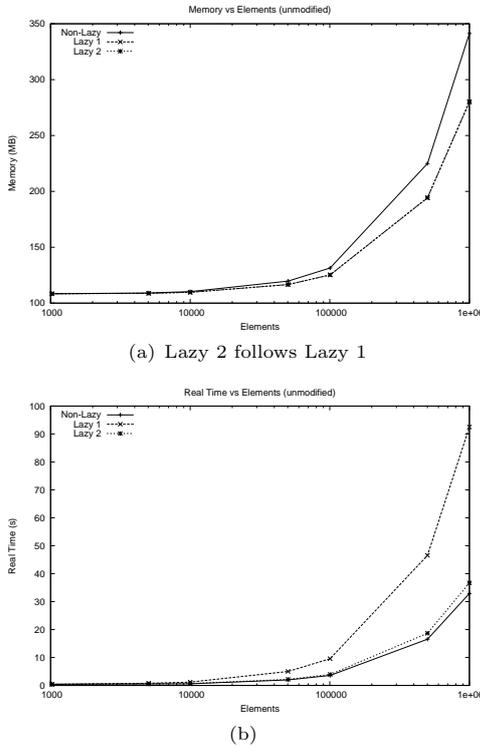


Figure 2: Unmodified Powerlaw

7.2 Repeated Sub-Expressions

The temporary buffer method uses less memory than the second method because the same storage can be used to evaluate multiple nodes. In memory intensive problems though, both methods use less memory than the non-lazy method.

The run time for Method 1 can be reduced by forcing nodes which appear multiple times to be resolved first before resolving the main expression. This means that the values can be retrieved directly instead of being recomputed. This has two drawbacks:

1. More memory is required to hold the extra resolved expressions. This is particularly significant if the results are of high rank. For example the Drucker-Prager tests in Section 7.3 contain rank 4 tensors.
2. The user must examine their expressions to determine suitable variables for early resolution and the order in which to do so. For example if f is a function of g (and both are repeated expressions), then g should be resolved first. If not, the work to evaluate g will be done twice.

Next we make two changes to the computation. Computing the power law values builds an expression $\eta(\theta + \omega)/(\eta\theta^2 + \omega)$. We resolve η , θ , ω before building the main expression.

Secondly, we rearrange calls to the L^∞ -norm so that sub-expressions are evaluated first. The performance for this modified version can be seen in Figure 3. Memory use for lazy resolution is still below non-lazy resolution. The time spent for Method 1 now approximates non-lazy time, while Method 2 is lower. Note that the memory usage in Figure 3(a) is worse than in Figure 2(a).

7.3 Experiment 2 — Drucker-Prager

A script (see Appendix B) to compute coefficients for Drucker-Prager flow (Gross et al. 2008) was run for a

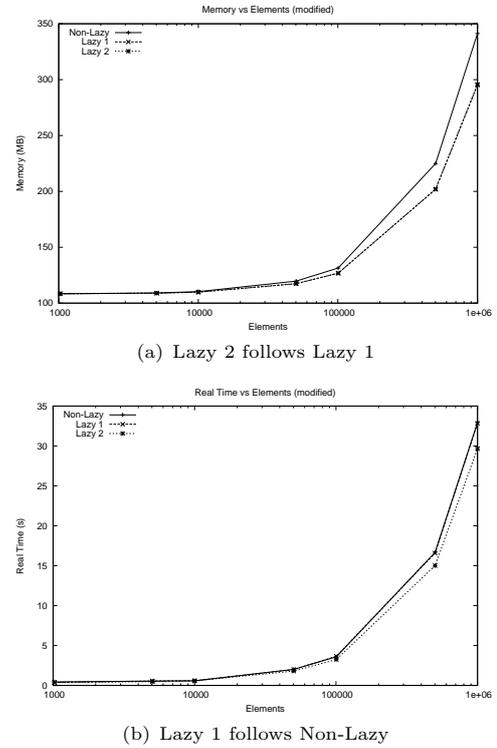


Figure 3: Modified Powerlaw

single OpenMP thread. Only Method 2 (node cache) was used for lazy testing. See Figures 4, 5 for results (values shown are averaged over ten runs). In both 2D and 3D domains, Method 2 shows significantly lower costs in both time and space. In Figure 5(a) the non-lazy version could not complete the 512,000 element test (presumably due to exhausting available memory).

8 Discussion

When the two lazy evaluation methods are assessed against our goals (memory, time and ease of use), we have some success. For less complex expressions such as those in the power law tests, there is some reduction in memory use. There is a time penalty involved however, so employing lazy evaluation for this problem would only be of benefit in cases where memory consumption is close to system limits.

The presence of repeated subexpressions seems to severely limit the usefulness of Method 1 (temporary buffer). As we showed in Section 7.2, these problems can be reduced by strategic calls to `resolve()` and reordering of expressions but this comes at the price of ease of use. Adding additional resolves also increases memory usage (contrast Figure 2(a) with Figure 3(a)). Deciding on the size limits for expressions also requires some care.

On the other hand when used on Drucker-Prager, Method 2 (node cache) showed significant reductions in both memory and time for large instances. In fact it rendered larger instances accessible.

Currently, lazy evaluation in *escript* can be switched on or off at runtime. Doing this with Method 2 seems to be the best policy.

Further work in this area should be directed to making automatic resolution more intelligent and providing more specific guidance for users as to when a problem is “large enough” for lazy evaluation to provide significant benefits.

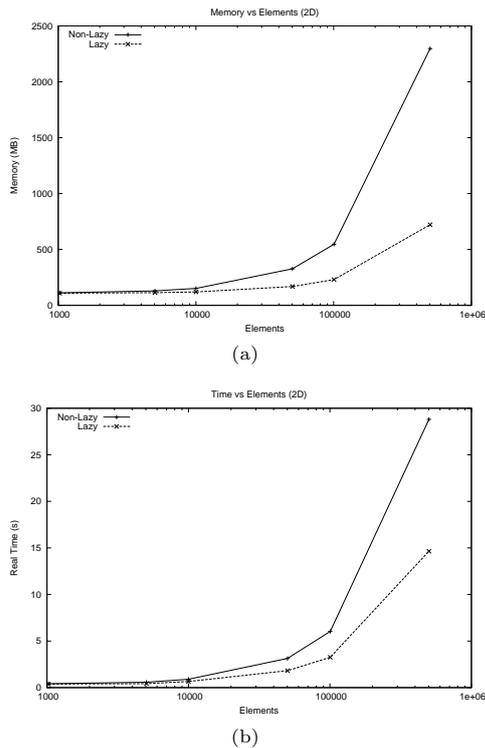


Figure 4: Drucker-Prager 2D

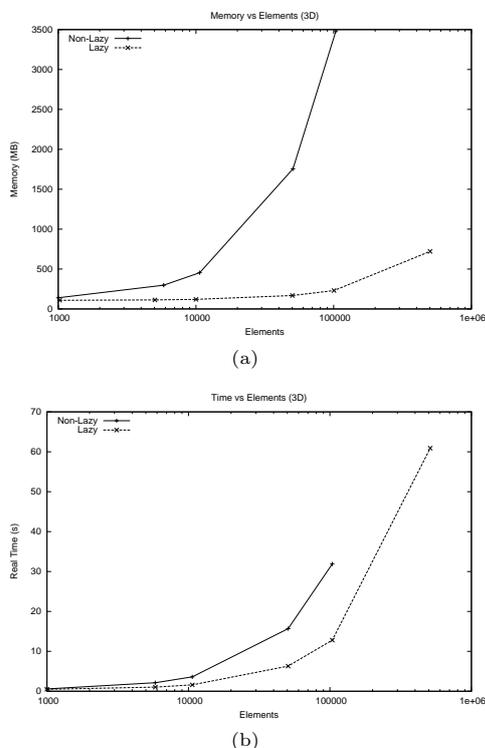


Figure 5: Drucker-Prager 3D

References

Appel, A. W. & Palsberg, J. (2002), *Modern compiler implementation in Java*, second edn, Cambridge University Press.

boo (2009), 'Boost C++ libraries', <http://www.boost.org/>.
URL: <http://www.boost.org>

Friedman, D. P. & Wand, M. (2008), *Essentials of Programming Languages*, third edn, MIT Press.

Glass, G. & Ables, K. (2003), *Unix for Programmers and Users*, third edn, Pearson Education.

Gross, L., Cumming, B., Steube, K. & Weatherley, D. (2007), 'A python module for pde-based numerical modelling', *PARA* **4699**, 270–279.

Gross, L., Muhlhaus, H., Thorne, E. & Steube, K. (2008), *Earthquakes : Simulations, Sources and Tsunamis*, Pageoph Topical Volumes, Birkhäuser Basel, chapter A New Design of Scientific Software Using Python and XML, pp. 653–670.

Hudak, P. (1989), 'Conception, evolution, and application of functional programming languages', *ACM Comput. Surv.* **21**(3), 359–411.

lpe (2009), 'escript-finley', <https://launchpad.net/escript-finley>.

Muhlhaus, H.-B. & Regenauer-Lieb, K. (2005), 'Towards a self-consistent plate mantle model that includes elasticity: simple benchmarks and application to basic modes of convection', *Geophysical Journal International* **163**(2), 788–800.

omp (2009a), 'OpenMP specification', <http://openmp.org/>.

omp (2009b), 'OpenMP tutorial', <https://computing.llnl.gov/tutorials/openMP/>.

Pandey, A. K. (2008), *Programming Languages, Principles and Paradigms*, Alpha Science.

Thompson, S. (1999), *Haskell, The craft of Functional Programming*, second edn, Addison-Wesley.

A Power Law script

This is the script used in the power law tests (some boilerplate and irrelevant lines removed). The NE variable should be set to the number of elements required.

```
SIDE=int(math.ceil(float(NE)**(1./2)))
setEscriptParamInt("TOO_MANY_LEVELS",15)
setEscriptParamInt("TOO_MANY_NODES",500)
```

```
d=Rectangle(SIDE,SIDE).getX()+1,1)
pl=PowerLaw(numMaterials=3, verbose=False)
pl.setDruckerPragerLaw(tau_Y=100.)
pl.setPowerLaws(eta_N=[2.,0.01,25./4.], \
    tau_t=[1, 25.,64.], power=[1,2,3])
pl.setEtaTolerance(1.e-8)
z=pl.getEtaEff(length(d))
z.resolve()
```

B Drucker Prager script

This is the script used in the drucker prager tests (some boilerplate and irrelevant lines removed). The domain variable should be a Rectangle (or Brick for 3D) with the required number of elements.

```
setEscriptParamInt("TOO_MANY_NODES",10000)
setEscriptParamInt("TOO_MANY_LEVELS",70)
```

```
G=10.
K=12
alpha=0.2
beta=0.03
h=1.
deps_th=0.1
```

```
stress=Tensor(1.,Function(domain))
tau_Y_safe=Scalar(13.,Function(domain))
tau_Y=Scalar(13.,Function(domain))
du=domain.getX()
plastic_stress=Scalar(0.,Function(domain))
```

```
d=domain.getDim()
abs_tol=1.e-15
SAFTY_FACTOR=1.e-8
k3=kroncker(Function(domain))
# elastic trial stress:
g=grad(du)
D=symmetric(g)
W=nonsymmetric(g)
s_e=stress+K*deps_th*k3+2*G*D+(K-2./3 \
    *G)*trace(D)*k3+2*symmetric( \
    matrix_mult(W,stress))
p_e=-1./d*trace(s_e)
s_e_dev=s_e+p_e*k3
tau_e=sqrt(1./2*inner(s_e_dev,s_e_dev))
```

```
F=tau_e-alpha*p_e-tau_Y
chi=whereNonNegative(F+SAFTY_FACTOR*tau_Y)
l=chi*F/(h+G+beta*K)
tau=tau_e-G*l
stress=tau/(tau_e+abs_tol* \
    whereZero(tau_e,abs_tol)) \
    *s_e_dev-(p_e+l*beta*K)*k3
plastic_stress=plastic_stress+l
```

```
hardening=(tau_Y-tau_Y_safe)/(1+abs_tol* \
    whereZero(1))
sXk3=outer(stress,k3)
k3Xk3=outer(k3,k3)
s_dev=stress-trace(stress)*(k3/d)
tmp=G*s_dev/(tau+abs_tol* \
    whereZero(tau,abs_tol))
```

```
S=G*(swap_axes(k3Xk3,0,3)+ \
    swap_axes(k3Xk3,1,3)) + (K-2./3*G) \
    *k3Xk3 + (sXk3-swap_axes( \
    swap_axes(sXk3,1,2),2,3)) \
    + 1./2*(swap_axes(swap_axes(sXk3, \
    0,2),2,3) \
    -swap_axes(swap_axes(sXk3,0,3),2,3)\
    -swap_axes(sXk3,1,2) \
    +swap_axes(sXk3,1,3)) \
    - outer(chi/(hardening+G+alpha*beta \
    *K)*(tmp+beta*K*k3),tmp+alpha*K*k3)
```

```
S.resolve()
tau.resolve()
stress.resolve()
plastic_stress.resolve()
```

Author Index

Akbar, Md. Mostofa, 31
Akeila, Lama, 41
Aloisio, Giovanni, 51
Atkinson, Alistair, 21

Brock, Michael, 3

Cesare, Silvio, 61
Chen, Jinjun, iii
Chin, Kwan-Wu, 13

Epicoco, Italo, 51

Fenwick, Joel, 71

Goscinski, Andrzej, 3

Gross, Lutz, 71

Hossain, Tareque, 31
Humadi, Wafaa, 41

Manning, Eric. G., 31
Mocavero, Silvia, 51

Ranjan, Rajiv, iii

Shelford, Steven, 31
Shoja, Gholamali C., 31
Sinnen, Oliver, 41

Xiang, Yang, 61

Recent Volumes in the CRPIT Series

ISSN 1445-1336

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

- Volume 84 - Artificial Intelligence and Data Mining 2007**
 Edited by Kok-Leong Ong, Deakin University, Australia, Wenyuan Li, University of Texas at Dallas, USA and Junbin Gao, Charles Sturt University, Australia. December, 2007. 978-1-920682-65-1. Contains the proceedings of the 2nd International Workshop on Integrating AI and Data Mining (AIDM 2007), Gold Coast, Australia. December 2007.
- Volume 85 - Advances in Ontologies 2007**
 Edited by Thomas Meyer, Meraka Institute, South Africa and Abhaya Nayak, Macquarie University, Australia. December, 2007. 978-1-920682-66-8. Contains the proceedings of the 3rd Australasian Ontology Workshop (AOW 2007), Gold Coast, Queensland, Australia.
- Volume 86 - Safety Critical Systems and Software 2007**
 Edited by Tony Cant, Defence Science and Technology Organisation, Australia. December, 2007. 978-1-920682-67-5. Contains the proceedings of the 12th Australian Conference on Safety Critical Systems and Software, August 2007, Adelaide, Australia.
- Volume 87 - Data Mining and Analytics 2008**
 Edited by John F. Roddick, Jiuyong Li, Peter Christen and Paul Kennedy. November, 2008. 978-1-920682-68-2. Contains the proceedings of the 7th Australasian Data Mining Conference (AusDM 2008), Adelaide, Australia. December 2008.
- Volume 88 - Koli Calling 2007**
 Edited by Raymond Lister University of Technology, Sydney and Simon University of Newcastle. November, 2007. 978-1-920682-69-9. Contains the proceedings of the 7th Baltic Sea Conference on Computing Education Research.
- Volume 89 - Australian Video**
 Edited by Heng Tao Shen and Michael Frater. October, 2008. 978-1-920682-70-5. Contains the proceedings of the 1st Australian Video Conference.
- Volume 90 - Advances in Ontologies**
 Edited by Thomas Meyer, Meraka Institute, South Africa and Mehmet Orgun, Macquarie University, Australia. September, 2008. 978-1-920682-71-2. Contains the proceedings of the Knowledge Representation Ontology Workshop (KROW 2008), Sydney, September 2008.
- Volume 91 - Computer Science 2009**
 Edited by Bernard Mans Macquarie University. January, 2009. 978-1-920682-72-9. Contains the proceedings of the Thirty-Second Australasian Computer Science Conference (ACSC2009), Wellington, New Zealand, January 2009.
- Volume 92 - Database Technologies 2009**
 Edited by Xuemin Lin, University of New South Wales and Athman Bouguettaya, CSIRO. January, 2009. 978-1-920682-73-6. Contains the proceedings of the Twentieth Australasian Database Conference (ADC2009), Wellington, New Zealand, January 2009.
- Volume 93 - User Interfaces 2009**
 Edited by Paul Calder Flinders University and Gerald Weber University of Auckland. January, 2009. 978-1-920682-74-3. Contains the proceedings of the Tenth Australasian User Interface Conference (AUIC2009), Wellington, New Zealand, January 2009.
- Volume 94 - Theory of Computing 2009**
 Edited by Prabhhu Manyem, University of Ballarat and Rod Downey, Victoria University of Wellington. January, 2009. 978-1-920682-75-0. Contains the proceedings of the Fifteenth Computing: The Australasian Theory Symposium (CATS2009), Wellington, New Zealand, January 2009.
- Volume 95 - Computing Education 2009**
 Edited by Margaret Hamilton, RMIT University and Tony Clear, Auckland University of Technology. January, 2009. 978-1-920682-76-7. Contains the proceedings of the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009.
- Volume 96 - Conceptual Modelling 2009**
 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. Contains the proceedings of the Fifth Asia-Pacific Conference on Conceptual Modelling (APCCM2008), Wollongong, NSW, Australia, January 2008.
- Volume 97 - Health Data and Knowledge Management 2009**
 Edited by James R. Warren, University of Auckland. January, 2009. 978-1-920682-78-1. Contains the proceedings of the Third Australasian Workshop on Health Data and Knowledge Management (HDKM 2009), Wellington, New Zealand, January 2009.
- Volume 98 - Information Security 2009**
 Edited by Ljiljana Brankovic, University of Newcastle and Willy Susilo, University of Wollongong. January, 2009. 978-1-920682-79-8. Contains the proceedings of the Australasian Information Security Conference (AISC 2009), Wellington, New Zealand, January 2009.
- Volume 99 - Grid Computing and e-Research 2009**
 Edited by Paul Roe and Wayne Kelly, QUT. January, 2009. 978-1-920682-80-4. Contains the proceedings of the Australasian Workshop on Grid Computing and e-Research (AusGrid 2009), Wellington, New Zealand, January 2009.
- Volume 100 - Safety Critical Systems and Software 2007**
 Edited by Tony Cant, Defence Science and Technology Organisation, Australia. December, 2008. 978-1-920682-81-1. Contains the proceedings of the 13th Australian Conference on Safety Critical Systems and Software, Canberra Australia.
- Volume 101 - Data Mining and Analytics 2009**
 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. Contains the proceedings of the 8th Australasian Data Mining Conference (AusDM 2009), Melbourne Australia.